Python®

Notes for Professionals

700+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/PythonBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Python Language

Python 3.x

Version	Release Date
3.7	2018-06-27
3.6	2016-12-23
3.5	2015-09-13
3.4	2014-03-17
3.3	2012-09-29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

Version	Release Date
2.7	2010-07-03
2.6	2008-10-02
2.5	2006-09-19
2.4	2004-11-30
2.3	2003-07-29
2.2	2001-12-21
2.1	2001-04-15
2.0	2000-10-16

Section 1.1: Getting Started

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Two major versions of Python are currently in active use:

- Python 3.x is the current version and is under active development.
- Python 2.x is the legacy version and will receive only security updates until 2020. No new features will be implemented. Note that many projects still use Python 2, although migrating to Python 3 is getting easier.

You can download and install either version of Python here. See Python 3 vs. Python 2 for a comparison between them. In addition, some third-parties offer re-packaged versions of Python that add commonly used libraries and other features to ease setup for common use cases, such as math, data analysis or scientific use. See the list at the official site.

Verify if Python is installed

To confirm that Python was installed correctly, you can verify that by running the following command in your favorite terminal (If you are using Windows OS, you need to add path of python to the environment variable before using it in command prompt):

```
$ python --version
```

Python 3.x `Version ≥ 3.0`

If you have *Python 3* installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
Python 3.6.0
```

Python 2.x `Version ≤ 2.7`

If you have *Python 2* installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
Python 2.7.13
```

If you have installed Python 3, but `$ python --version` outputs a Python 2 version, you also have Python 2 installed. This is often the case on MacOS, and many Linux distributions. Use `$ python3` instead to explicitly use the Python 3 interpreter.

Hello, World in Python using IDLE

IDLE is a simple editor for Python, that comes bundled with Python.

How to create Hello, World program in IDLE

- Open IDLE on your system of choice.
  - In older versions of Windows, it can be found at `All Programs` under the Windows menu.
  - In Windows 8+, search for `IDLE` or find it in the apps that are present in your system.
  - On Unix-based (including Mac) systems you can open it from the shell by typing `$ idle python_file.py`.
- It will open a shell with options along the top.

In the shell, there is a prompt of three right angle brackets:

```
>>>
```

Now write the following code in the prompt:

```
>>> print("Hello, World")
```

Hit `Enter`.

```
>>> print("Hello, World")
Hello, World
```

Hello World Python file

Create a new file `hello.py` that contains the following line:

Python 3.x `Version ≥ 3.0`

```
print('Hello, World')
```

Python 2.x `Version ≥ 2.6`

You can use the Python 3 `print` function in Python 2 with the following `import` statement:

```
from __future__ import print_function
```

Python 2 has a number of functionalities that can be optionally imported from Python 3 using the `__future__` module, as discussed here.

Python 2.x `Version ≤ 2.7`

If using Python 2, you may also type the line below. Note that this is not valid in Python 3 and thus not recommended because it reduces cross-version code compatibility.

```
print 'Hello, World'
```

In your terminal, navigate to the directory containing the file `hello.py`.

Type `python hello.py`, then hit the `Enter` key.

```
$ python hello.py
Hello, World
```

You should see `Hello, World` printed to the console.

You can also substitute `hello.py` with the path to your file. For example, if you have the file in your home directory and your user is "user" on Linux, you can type `python /home/user/hello.py`.

Launch an interactive Python shell

By executing (running) the `python` command in your terminal, you are presented with an interactive Python shell. This is also known as the Python Interpreter or a REPL (for 'Read Evaluate Print Loop').

```
$ python
Python 2.7.12 (default, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, World'
Hello, World
>>>
```

If you want to run Python 3 from your terminal, execute the command `python3`.

```
$ python3
Python 3.6.0 (default, Jan 13 2017, 00:00:00)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, World')
Hello, World
>>>
```

Alternatively, start the interactive prompt and load file with `python -i <file.py>`.

In command line, run:

```
$ python -i hello.py
"Hello World"
>>>
```

There are multiple ways to close the Python shell:

```
>>> exit()
```

or

```
>>> quit()
```

Alternatively, `CTRL + D` will close the shell and put you back on your terminal's command line.

If you want to cancel a command you're in the middle of typing and get back to a clean command prompt, while staying inside the Interpreter shell, use `CTRL + C`.

Try an interactive Python shell online.

Other Online Shells

Various websites provide online access to Python shells.

Online shells may be useful for the following purposes:

- Run a small code snippet from a machine which lacks python installation(smartphones, tablets etc).
- Learn or teach basic Python.
- Solve online judge problems.

Examples:

> Disclaimer: documentation author(s) are not affiliated with any resources listed below.

- https://www.python.org/shell/ - The online Python shell hosted by the official Python website.
- https://ideone.com/ - Widely used on the Net to illustrate code snippet behavior.
- https://repl.it/languages/python3 - Powerful and simple online compiler, IDE and interpreter. Code, compile, and run code in Python.
- https://www.tutorialspoint.com/execute_python_online.php - Full-featured UNIX shell, and a user-friendly project explorer.
- http://rextester.com/l/python3_online_compiler - Simple and easy to use IDE which shows execution time

Run commands as a string

Python can be passed arbitrary code as a string in the shell:

```
$ python -c 'print("Hello, World")'
Hello, World
```

This can be useful when concatenating the results of scripts together in the shell.

Shells and Beyond

Package Management - The PyPA recommended tool for installing Python packages is PIP. To install, on your command line execute `pip install <the package name>`. For instance, `pip install numpy`. (Note: On windows you must add pip to your PATH environment variables. To avoid this, use `python -m pip install <the package name>`)

Shells - So far, we have discussed different ways to run code using Python's native interactive shell. Shells use Python's interpretive power for experimenting with code real-time. Alternative shells include IDLE - a pre-bundled

GUI, IPython - known for extending the interactive experience, etc.

Programs - For long-term storage you can save content to .py files and edit/execute them as scripts or programs with external tools e.g. shell, IDEs (such as PyCharm), Jupyter notebooks, etc. Intermediate users may use these tools; however, the methods discussed here are sufficient for getting started.

Python tutor allows you to step through Python code so you can visualize how the program will flow, and helps you to understand where your program went wrong.

PEP8 defines guidelines for formatting Python code. Formatting code well is important so you can quickly read what the code does.

Section 1.2: Creating variables and assigning values

To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it.

```
<variable name> = <value>
```

Python uses `=` to assign values to variables. There's no need to declare a variable in advance (or to assign a data type to it), assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.

```
# Integer
a = 2
print(a)
# Output: 2

# Integer
b = 9223372036854775807
print(b)
# Output: 9223372036854775807

# Floating point
pi = 3.14
print(pi)
# Output: 3.14

# String
c = 'A'
print(c)
# Output: A

# String
name = 'John Doe'
print(name)
# Output: John Doe

# Boolean
q = True
print(q)
# Output: True

# Empty value or null data type
x = None
print(x)
# Output: None
```

Variable assignment works from left to right. So the following will give you an syntax error.

```
0 = x
=> Output: SyntaxError: can't assign to literal
```

You can not use python's keywords as a valid variable name. You can see the list of keyword by:

```
import keyword
print(keyword.kwlist)
```

Rules for variable naming:

1. Variables names must start with a letter or an underscore.

```
x  = True   # valid
_y = True   # valid

9x = False  # starts with numeral
=> SyntaxError: invalid syntax

$y = False #  starts with symbol
=> SyntaxError: invalid syntax
```

2. The remainder of your variable name may consist of letters, numbers and underscores.

```
has_0_in_it = "Still Valid"
```

3. Names are case sensitive.

```
x = 9
y = X*5
=>NameError: name 'X' is not defined
```

Even though there's no need to specify a data type when declaring a variable in Python, while allocating the necessary area in memory for the variable, the Python interpreter automatically picks the most suitable built-in type for it:

```
a = 2
print(type(a))
# Output: <type 'int'>

b = 9223372036854775807
print(type(b))
# Output: <type 'int'>

pi = 3.14
print(type(pi))
# Output: <type 'float'>

c = 'A'
print(type(c))
# Output: <type 'str'>

name = 'John Doe'
print(type(name))
# Output: <type 'str'>

q = True
print(type(q))
# Output: <type 'bool'>
```

```
x = None
print(type(x))
# Output: <type 'NoneType'>
```

Now you know the basics of assignment, let's get this subtlety about assignment in python out of the way.

When you use `=` to do an assignment operation, what's on the left of `=` is a ***name*** for the ***object*** on the right. Finally, what `=` does is assign the ***reference*** of the object on the right to the ***name*** on the left.

That is:

```
a_name = an_object  # "a_name" is now a name for the reference to the object "an_object"
```

So, from many assignment examples above, if we pick `pi = 3.14`, then `pi` is **a** name (not **the** name, since an object can have multiple names) for the object `3.14`. If you don't understand something below, come back to this point and read this again! Also, you can take a look at this for a better understanding.

You can assign multiple values to multiple variables in one line. Note that there must be the same number of arguments on the right and left sides of the `=` operator:

```
a, b, c = 1, 2, 3
print(a, b, c)
# Output: 1 2 3

a, b, c = 1, 2
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack

a, b = 1, 2, 3
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b = 1, 2, 3
=> ValueError: too many values to unpack
```

The error in last example can be obviated by assigning remaining values to equal number of arbitrary variables. This dummy variable can have any name, but it is conventional to use the underscore (_) for assigning unwanted values:

```
a, b, _ = 1, 2, 3
print(a, b)
# Output: 1, 2
```

Note that the number of _ and number of remaining values must be equal. Otherwise 'too many values to unpack error' is thrown as above:

```
a, b, _ = 1,2,3,4
=>Traceback (most recent call last):
=>File "name.py", line N, in <module>
=>a, b, _ = 1,2,3,4
=>ValueError: too many values to unpack (expected 3)
```

You can also assign a single value to several variables simultaneously.

```
a = b = c = 1
print(a, b, c)
```

```
# Output: 1 1 1
```

When using such cascading assignment, it is important to note that all three variables `a`, `b` and `c` refer to the *same object* in memory, an `int` object with the value of 1. In other words, `a`, `b` and `c` are three different names given to the same int object. Assigning a different object to one of them afterwards doesn't change the others, just as expected:

```
a = b = c = 1    # all three names a, b and c refer to same int object with value 1
print(a, b, c)
# Output: 1 1 1
b = 2            # b now refers to another int object, one with a value of 2
print(a, b, c)
# Output: 1 2 1  # so output is as expected.
```

The above is also true for mutable types (like `list`, `dict`, etc.) just as it is true for immutable types (like `int`, `string`, `tuple`, etc.):

```
x = y = [7, 8, 9]   # x and y refer to the same list object just created, [7, 8, 9]
x = [13, 8, 9]      # x now refers to a different list object just created, [13, 8, 9]
print(y)            # y still refers to the list it was first assigned
# Output: [7, 8, 9]
```

So far so good. Things are a bit different when it comes to *modifying* the object (in contrast to *assigning* the name to a different object, which we did above) when the cascading assignment is used for mutable types. Take a look below, and you will see it first hand:

```
x = y = [7, 8, 9]     # x and y are two different names for the same list object just created, [7,
8, 9]
x[0] = 13             # we are updating the value of the list [7, 8, 9] through one of its names, x
in this case
print(y)              # printing the value of the list using its other name
# Output: [13, 8, 9]  # hence, naturally the change is reflected
```

Nested lists are also valid in python. This means that a list can contain another list as an element.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list
print x[2]
# Output: [3, 4, 5]
print x[2][1]
# Output: 4
```

Lastly, variables in Python do not have to stay the same type as which they were first defined -- you can simply use `=` to assign a new value to a variable, even if that value is of a different type.

```
a = 2
print(a)
# Output: 2

a = "New value"
print(a)
# Output: New value
```

If this bothers you, think about the fact that what's on the left of `=` is just a name for an object. First you call the `int` object with value 2 `a`, then you change your mind and decide to give the name `a` to a `string` object, having value 'New value'. Simple, right?

Section 1.3: Block Indentation

Python uses indentation to define control and loop constructs. This contributes to Python's readability, however, it requires the programmer to pay close attention to the use of whitespace. Thus, editor miscalibration could result in code that behaves in unexpected ways.

Python uses the colon symbol (`:`) and indentation for showing where blocks of code begin and end (If you come from another language, do not confuse this with somehow being related to the ternary operator). That is, blocks in Python, such as functions, loops, `if` clauses and other constructs, have no ending identifiers. All blocks start with a colon and then contain the indented lines below it.

For example:

```
def my_function():    # This is a function definition. Note the colon (:)
    a = 2             # This line belongs to the function because it's indented
    return a          # This line also belongs to the same function
print(my_function())  # This line is OUTSIDE the function block
```

or

```
if a > b:             # If block starts here
    print(a)          # This is part of the if block
else:                 # else must be at the same level as if
    print(b)          # This line is part of the else block
```

Blocks that contain exactly one single-line statement may be put on the same line, though this form is generally not considered good style:

```
if a > b: print(a)
else: print(b)
```

Attempting to do this with more than a single statement will *not* work:

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

if x > y: while y != z: y -= 1  # SyntaxError: invalid syntax
```

An empty block causes an `IndentationError`. Use `pass` (a command that does nothing) when you have a block with no content:

```
def will_be_implemented_later():
    pass
```

Spaces vs. Tabs

In short: **always** use 4 spaces for indentation.

Using tabs exclusively is possible but PEP 8, the style guide for Python code, states that spaces are preferred.

Python 3.x `Version ≥ 3.0`

Python 3 disallows mixing the use of tabs and spaces for indentation. In such case a compile-time error is generated: `Inconsistent use of tabs and spaces in indentation` and the program will not run.

Python 2.x `Version ≤ 2.7`

Python 2 allows mixing tabs and spaces in indentation; this is strongly discouraged. The tab character completes the previous indentation to be a multiple of **8** spaces. Since it is common that editors are configured to show tabs as multiple of **4** spaces, this can cause subtle bugs.

Citing PEP 8:

> When invoking the Python 2 command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt` these warnings become errors. These options are highly recommended!

Many editors have "tabs to spaces" configuration. When configuring the editor, one should differentiate between the tab *character* ('\t') and the Tab key.

- The tab *character* should be configured to show 8 spaces, to match the language semantics - at least in cases when (accidental) mixed indentation is possible. Editors can also automatically convert the tab character to spaces.
- However, it might be helpful to configure the editor so that pressing the Tab key will insert 4 spaces, instead of inserting a tab character.

Python source code written with a mix of tabs and spaces, or with non-standard number of indentation spaces can be made pep8-conformant using autopep8. (A less powerful alternative comes with most Python installations: reindent.py)

Section 1.4: Datatypes

Built-in Types
Booleans

`bool`: A boolean value of either `True` or `False`. Logical operations like `and`, `or`, `not` can be performed on booleans.

```
x or y    # if x is False then y otherwise x
x and y   # if x is False then x otherwise y
not x     # if x is True then False, otherwise True
```

In Python 2.x and in Python 3.x, a boolean is also an `int`. The `bool` type is a subclass of the `int` type and `True` and `False` are its only instances:

```
issubclass(bool, int) # True

isinstance(True, bool) # True
isinstance(False, bool) # True
```

If boolean values are used in arithmetic operations, their integer values (`1` and `0` for `True` and `False`) will be used to return an integer result:

```
True + False == 1 # 1 + 0 == 1
True * True  == 1 # 1 * 1 == 1
```

Numbers

- `int`: Integer number

```
a = 2
b = 100
c = 123456789
d = 38563846326424324
```

Integers in Python are of arbitrary sizes.

Note: in older versions of Python, a `long` type was available and this was distinct from `int`. The two have been unified.

- `float`: Floating point number; precision depends on the implementation and system architecture, for CPython the `float` datatype corresponds to a C double.

```
a = 2.0
b = 100.e0
c = 123456789.e1
```

- `complex`: Complex numbers

```
a = 2 + 1j
b = 100 + 10j
```

The `<`, `<=`, `>` and `>=` operators will raise a `TypeError` exception when any operand is a complex number.

Strings

Python 3.x `Version ≥ 3.0`

- `str`: a **unicode string**. The type of `'hello'`
- `bytes`: a **byte string**. The type of `b'hello'`

Python 2.x `Version ≤ 2.7`

- `str`: a **byte string**. The type of `'hello'`
- `bytes`: synonym for `str`
- `unicode`: a **unicode string**. The type of `u'hello'`

Sequences and collections

Python differentiates between ordered sequences and unordered collections (such as `set` and `dict`).

- strings (`str`, `bytes`, `unicode`) are sequences
- `reversed`: A reversed order of `str` with `reversed` function

```
a = reversed('hello')
```

- `tuple`: An ordered collection of `n` values of any type (`n >= 0`).

```
a = (1, 2, 3)
b = ('a', 1, 'python', (1, 2))
b[2] = 'something else' # returns a TypeError
```

Supports indexing; immutable; hashable if all its members are hashable

- `list`: An ordered collection of `n` values (`n >= 0`)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # allowed
```

Not hashable; mutable.

- `set`: An unordered collection of unique values. Items must be hashable.

```
a = {1, 2, 'a'}
```

- `dict`: An unordered collection of unique key-value pairs; keys must be hashable.

```
a = {1: 'one',
     2: 'two'}

b = {'a': [1, 2, 3],
     'b': 'a string'}
```

> An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equality must have the same hash value.

Built-in constants

In conjunction with the built-in datatypes there are a small number of built-in constants in the built-in namespace:

- `True`: The true value of the built-in type `bool`
- `False`: The false value of the built-in type `bool`
- `None`: A singleton object used to signal that a value is absent.
- `Ellipsis` or `...`: used in core Python3+ anywhere and limited usage in Python2.7+ as part of array notation. `numpy` and related packages use this as a 'include everything' reference in arrays.
- `NotImplemented`: a singleton used to indicate to Python that a special method doesn't support the specific arguments, and Python will try alternatives if available.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x `Version ≥ 3.0`

`None` doesn't have any natural ordering. Using ordering comparison operators (`<`, `<=`, `>=`, `>`) isn't supported anymore and will raise a `TypeError`.

Python 2.x `Version ≤ 2.7`

`None` is always less than any number (`None < -32` evaluates to `True`).

Testing the type of variables

In python, we can check the datatype of an object using the built-in function `type`.

```
a = '123'
print(type(a))
```

```
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

In conditional statements it is possible to test the datatype with `isinstance`. However, it is usually not encouraged to rely on the type of the variable.

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

For information on the differences between `type()` and `isinstance()` read: Differences between isinstance and type in Python

To test if something is of `NoneType`:

```
x = None
if x is None:
    print('Not a surprise, I just defined x as None.')
```

Converting between datatypes

You can perform explicit datatype conversion.

For example, '123' is of `str` type and it can be converted to integer using `int` function.

```
a = '123'
b = int(a)
```

Converting from a float string such as '123.456' can be done using `float` function.

```
a = '123.456'
b = float(a)
c = int(a)    # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)    # 123
```

You can also convert sequence or collection types

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)   # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

Explicit string type at definition of literals

With one letter labels just in front of the quotes you can tell what type of string you want to define.

- `b'foo bar'`: results `bytes` in Python 3, `str` in Python 2
- `u'foo bar'`: results `str` in Python 3, `unicode` in Python 2
- `'foo bar'`: results `str`
- `r'foo bar'`: results so called raw string, where escaping special characters is not necessary, everything is taken verbatim as you typed

```
normal  = 'foo\nbar'   # foo
```

```
                        # bar
escaped = 'foo\\nbar'   # foo\nbar
raw     = r'foo\nbar'   # foo\nbar
```

Mutable and Immutable Data Types

An object is called *mutable* if it can be changed. For example, when you pass a list to some function, the list can be changed:

```
def f(m):
    m.append(3)  # adds a number to the list. This is a mutation.

x = [1, 2]
f(x)
x == [1, 2]  # False now, since an item was added to the list
```

An object is called *immutable* if it cannot be changed in any way. For example, integers are immutable, since there's no way to change them:

```
def bar():
    x = (1, 2)
    g(x)
    x == (1, 2)  # Will always be True, since no function can change the object (1, 2)
```

Note that **variables** themselves are mutable, so we can reassign the *variable* `x`, but this does not change the object that `x` had previously pointed to. It only made `x` point to a new object.

Data types whose instances are mutable are called *mutable data types*, and similarly for immutable objects and datatypes.

Examples of immutable Data Types:

- `int`, `long`, `float`, `complex`
- `str`
- `bytes`
- `tuple`
- `frozenset`

Examples of mutable Data Types:

- `bytearray`
- `list`
- `set`
- `dict`

Section 1.5: Collection Types

There are a number of collection types in Python. While types such as `int` and `str` hold a single value, collection types hold multiple values.

Lists

The `list` type is probably the most commonly used collection type in Python. Despite its name, a list is more like an array in other languages, mostly JavaScript. In Python, a list is merely an ordered collection of valid Python values. A list can be created by enclosing values, separated by commas, in square brackets:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

A list can be empty:

```
empty_list = []
```

The elements of a list are not restricted to a single data type, which makes sense given that Python is a dynamic language:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

A list can contain another list as its element:

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

The elements of a list can be accessed via an *index*, or numeric representation of their position. Lists in Python are *zero-indexed* meaning that the first element in the list is at index 0, the second element is at index 1 and so on:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

Indices can also be negative which means counting from the end of the list (`-1` being the index of the last element). So, using the list from the above example:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Lists are mutable, so you can change the values in a list:

```
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

Besides, it is possible to add and/or remove elements from a list:

Append object to end of list with `L.append(object)`, returns `None`.

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Add a new element to list at a specific index. `L.insert(index, object)`

```
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Remove the first occurrence of a value with `L.remove(value)`, returns `None`

```
names.remove("Bob")
print(names) # Outputs ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

Get the index in the list of the first item whose value is x. It will show an error if there is no such item.

```
name.index("Alice")
0
```

Count length of list

```
len(names)
6
```

count occurrence of any item in list

```
a = [1, 1, 1, 2, 3, 4]
a.count(1)
3
```

Reverse the list

```
a.reverse()
[4, 3, 2, 1, 1, 1]
# or
a[::-1]
[4, 3, 2, 1, 1, 1]
```

Remove and return item at index (defaults to the last item) with `L.pop([index])`, returns the item

```
names.pop() # Outputs 'Sia'
```

You can iterate over the list elements like below:

```
for element in my_list:
    print (element)
```

Tuples

A `tuple` is similar to a list except that it is fixed-length and immutable. So the values in the tuple cannot be changed nor the values be added to or removed from the tuple. Tuples are commonly used for small collections of values that will not need to change, such as an IP address and port. Tuples are represented with parentheses instead of square brackets:

```
ip_address = ('10.20.30.40', 8080)
```

The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid.

A tuple with only one member must be defined (note the comma) this way:

```
one_member_tuple = ('Only member',)
```

or

```
one_member_tuple = 'Only member',    # No brackets
```

or just using `tuple` syntax

```
one_member_tuple = tuple(['Only member'])
```

Dictionaries

A `dictionary` in Python is a collection of key-value pairs. The dictionary is surrounded by curly braces. Each pair is separated by a comma and the key and value are separated by a colon. Here is an example:

```
state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

To get a value, refer to it by its key:

```
ca_capital = state_capitals['California']
```

You can also get all of the keys in a dictionary and then iterate over them:

```
for k in state_capitals.keys():
    print('{} is the capital of {}'.format(state_capitals[k], k))
```

Dictionaries strongly resemble JSON syntax. The native `json` module in the Python standard library can be used to convert between JSON and dictionaries.

set

A `set` is a collection of elements with no repeats and without insertion order but sorted order. They are used in situations where it is only important that some things are grouped together, and not what order they were included. For large groups of data, it is much faster to check whether or not an element is in a `set` than it is to do the same for a `list`.

Defining a `set` is very similar to defining a `dictionary`:

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Or you can build a `set` using an existing `list`:

```
my_list = [1,2,3]
my_set = set(my_list)
```

Check membership of the `set` using `in`:

```
if name in first_names:
    print(name)
```

You can iterate over a `set` exactly like a list, but remember: the values will be in an arbitrary, implementation-defined order.

defaultdict

A `defaultdict` is a dictionary with a default value for keys, so that keys for which no value has been explicitly defined can be accessed without errors. `defaultdict` is especially useful when the values in the dictionary are collections (lists, dicts, etc) in the sense that it does not need to be initialized every time when a new key is used.

A `defaultdict` will never raise a KeyError. Any key that does not exist gets the default value returned.

For example, consider the following dictionary

```
>>> state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

If we try to access a non-existent key, python returns us an error as follows

```
>>> state_capitals['Alabama']
Traceback (most recent call last):

  File "<ipython-input-61-236329695e6f>", line 1, in <module>
    state_capitals['Alabama']

KeyError: 'Alabama'
```

Let us try with a `defaultdict`. It can be found in the collections module.

```
>>> from collections import defaultdict
>>> state_capitals = defaultdict(lambda: 'Boston')
```

What we did here is to set a default value (**Boston**) in case the give key does not exist. Now populate the dict as before:

```
>>> state_capitals['Arkansas'] = 'Little Rock'
>>> state_capitals['California'] = 'Sacramento'
>>> state_capitals['Colorado'] = 'Denver'
>>> state_capitals['Georgia'] = 'Atlanta'
```

If we try to access the dict with a non-existent key, python will return us the default value i.e. Boston

```
>>> state_capitals['Alabama']
'Boston'
```

and returns the created values for existing key just like a normal `dictionary`

```
>>> state_capitals['Arkansas']
'Little Rock'
```

Section 1.6: IDLE - Python GUI

IDLE is Python's Integrated Development and Learning Environment and is an alternative to the command line. As the name may imply, IDLE is very useful for developing new code or learning python. On Windows this comes with the Python interpreter, but in other operating systems you may need to install it through your package manager.

The main purposes of IDLE are:

- Multi-window text editor with syntax highlighting, autocompletion, and smart indent
- Python shell with syntax highlighting
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility
- Automatic indentation (useful for beginners learning about Python's indentation)

- Saving the Python program as .py files and run them and edit them later at any them using IDLE.

In IDLE, hit `F5` or `run Python Shell` to launch an interpreter. Using IDLE can be a better learning experience for new users because code is interpreted as the user writes.

Note that there are lots of alternatives, see for example this discussion or this list.

Troubleshooting

- **Windows**

  If you're on Windows, the default command is `python`. If you receive a `"'python' is not recognized"` error, the most likely cause is that Python's location is not in your system's `PATH` environment variable. This can be accessed by right-clicking on 'My Computer' and selecting 'Properties' or by navigating to 'System' through 'Control Panel'. Click on 'Advanced system settings' and then 'Environment Variables...'. Edit the `PATH` variable to include the directory of your Python installation, as well as the Script folder (usually `C:\Python27;C:\Python27\Scripts`). This requires administrative privileges and may require a restart.

  When using multiple versions of Python on the same machine, a possible solution is to rename one of the `python.exe` files. For example, naming one version `python27.exe` would cause `python27` to become the Python command for that version.

  You can also use the Python Launcher for Windows, which is available through the installer and comes by default. It allows you to select the version of Python to run by using `py -[x.y]` instead of `python[x.y]`. You can use the latest version of Python 2 by running scripts with `py -2` and the latest version of Python 3 by running scripts with `py -3`.

- **Debian/Ubuntu/MacOS**

  This section assumes that the location of the `python` executable has been added to the `PATH` environment variable.

  If you're on Debian/Ubuntu/MacOS, open the terminal and type `python` for Python 2.x or `python3` for Python 3.x.

  Type `which python` to see which Python interpreter will be used.

- **Arch Linux**

  The default Python on Arch Linux (and descendants) is Python 3, so use `python` or `python3` for Python 3.x and `python2` for Python 2.x.

- **Other systems**

  Python 3 is sometimes bound to `python` instead of `python3`. To use Python 2 on these systems where it is installed, you can use `python2`.

Section 1.7: User Input

Interactive input

To get input from the user, use the `input` function (**note**: in Python 2.x, the function is called `raw_input` instead, although Python 2.x has its own version of `input` that is completely different):

Python 2.x `Version ≥ 2.3`

```
name = raw_input("What is your name? ")
# Out: What is your name? _
```

> **Security Remark** Do not use `input()` in Python2 - the entered text will be evaluated as if it were a Python expression (equivalent to `eval(input())` in Python3), which might easily become a vulnerability. See this article for further information on the risks of using this function.

Python 3.x `Version ≥ 3.0`

```
name = input("What is your name? ")
# Out: What is your name? _
```

The remainder of this example will be using Python 3 syntax.

The function takes a string argument, which displays it as a prompt and returns a string. The above code provides a prompt, waiting for the user to input.

```
name = input("What is your name? ")
# Out: What is your name?
```

If the user types "Bob" and hits enter, the variable `name` will be assigned to the string `"Bob"`:

```
name = input("What is your name? ")
# Out: What is your name? Bob
print(name)
# Out: Bob
```

Note that the `input` is always of type `str`, which is important if you want the user to enter numbers. Therefore, you need to convert the `str` before trying to use it as a number:

```
x = input("Write a number:")
# Out: Write a number: 10
x / 2
# Out: TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Out: 5.0
```

NB: It's recommended to use `try`/`except` blocks to catch exceptions when dealing with user inputs. For instance, if your code wants to cast a `raw_input` into an `int`, and what the user writes is uncastable, it raises a `ValueError`.

Section 1.8: Built in Modules and Functions

A module is a file containing Python definitions and statements. Function is a piece of code which execute some logic.

```
>>> pow(2,3)     #8
```

To check the built in function in python we can use `dir()`. If called without an argument, return the names in the current scope. Else, return an alphabetized list of names comprising (some of) the attribute of the given object, and of attributes reachable from it.

```
>>> dir(__builtins__)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
    '__doc__',
```

```
    '__import__',
    '__name__',
    '__package__',
    'abs',
    'all',
    'any',
    'apply',
    'basestring',
    'bin',
    'bool',
    'buffer',
    'bytearray',
    'bytes',
    'callable',
    'chr',
    'classmethod',
    'cmp',
    'coerce',
    'compile',
    'complex',
    'copyright',
    'credits',
    'delattr',
    'dict',
    'dir',
    'divmod',
    'enumerate',
    'eval',
    'execfile',
    'exit',
    'file',
    'filter',
    'float',
    'format',
    'frozenset',
    'getattr',
    'globals',
    'hasattr',
    'hash',
    'help',
    'hex',
    'id',
    'input',
    'int',
    'intern',
    'isinstance',
    'issubclass',
    'iter',
    'len',
    'license',
    'list',
    'locals',
    'long',
    'map',
    'max',
    'memoryview',
    'min',
    'next',
    'object',
    'oct',
    'open',
    'ord',
```

```
    'pow',
    'print',
    'property',
    'quit',
    'range',
    'raw_input',
    'reduce',
    'reload',
    'repr',
    'reversed',
    'round',
    'set',
    'setattr',
    'slice',
    'sorted',
    'staticmethod',
    'str',
    'sum',
    'super',
    'tuple',
    'type',
    'unichr',
    'unicode',
    'vars',
    'xrange',
    'zip'
]
```

To know the functionality of any function, we can use built in function `help` .

```
>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
```

Built in modules contains extra functionalities. For example to get square root of a number we need to include `math` module.

```
>>> import math
>>> math.sqrt(16) # 4.0
```

To know all the functions in a module we can assign the functions list to a variable, and then print the variable.

```
>>> import math
>>> dir(math)

   ['__doc__', '__name__', '__package__', 'acos', 'acosh',
   'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
   'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
   'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
   'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
   'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
   'tan', 'tanh', 'trunc']
```

it seems `__doc__` is useful to provide some documentation in, say, functions

```
>>> math.__doc__
'This module is always available.  It provides access to the\nmathematical
 functions defined by the C standard.'
```

In addition to functions, documentation can also be provided in modules. So, if you have a file named `helloWorld.py` like this:

```
"""This is the module docstring."""

def sayHello():
    """This is the function docstring."""
    return 'Hello World'
```

You can access its docstrings like this:

```
>>> import helloWorld
>>> helloWorld.__doc__
'This is the module docstring.'
>>> helloWorld.sayHello.__doc__
'This is the function docstring.'
```

- For any user defined type, its attributes, its class's attributes, and recursively the attributes of its class's base classes can be retrieved using dir()

```
>>> class MyClassObject(object):
...     pass
...
>>> dir(MyClassObject)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Any data type can be simply converted to string using a builtin function called `str`. This function is called by default when a data type is passed to `print`

```
>>> str(123)    # "123"
```

Section 1.9: Creating a module

A module is an importable file containing definitions and statements.

A module can be created by creating a `.py` file.

```
# hello.py
def say_hello():
    print("Hello!")
```

Functions in a module can be used by importing the module.

For modules that you have made, they will need to be in the same directory as the file that you are importing them into. (However, you can also put them into the Python lib directory with the pre-included modules, but should be avoided if possible.)

```
$ python
>>> import hello
>>> hello.say_hello()
```

```
=> "Hello!"
```

Modules can be imported by other modules.

```
# greet.py
import hello
hello.say_hello()
```

Specific functions of a module can be imported.

```
# greet.py
from hello import say_hello
say_hello()
```

Modules can be aliased.

```
# greet.py
import hello as ai
ai.say_hello()
```

A module can be stand-alone runnable script.

```
# run_hello.py
if __name__ == '__main__':
    from hello import say_hello
    say_hello()
```

Run it!

```
$ python run_hello.py
=> "Hello!"
```

If the module is inside a directory and needs to be detected by python, the directory should contain a file named `__init__.py`.

Section 1.10: Installation of Python 2.7.x and 3.x

> **Note**: Following instructions are written for Python 2.7 (unless specified): instructions for Python 3.x are similar.

Windows

First, download the latest version of Python 2.7 from the official Website (https://www.python.org/downloads/). Version is provided as an MSI package. To install it manually, just double-click the file.

By default, Python installs to a directory:

```
C:\Python27\
```

Warning: installation does not automatically modify the PATH environment variable.

Assuming that your Python installation is in C:\Python27, add this to your PATH:

```
C:\Python27\;C:\Python27\Scripts\
```

Now to check if Python installation is valid write in cmd:

```
python --version
```

Python 2.x and 3.x Side-By-Side

To install and use both Python 2.x and 3.x side-by-side on a Windows machine:

1. Install Python 2.x using the MSI installer.
    - Ensure Python is installed for all users.
    - Optional: add Python to `PATH` to make Python 2.x callable from the command-line using `python`.
2. Install Python 3.x using its respective installer.
    - Again, ensure Python is installed for all users.
    - Optional: add Python to `PATH` to make Python 3.x callable from the command-line using `python`. This may override Python 2.x `PATH` settings, so double-check your `PATH` and ensure it's configured to your preferences.
    - Make sure to install the `py launcher` for all users.

Python 3 will install the Python launcher which can be used to launch Python 2.x and Python 3.x interchangeably from the command-line:

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To use the corresponding version of `pip` for a specific Python version, use:

```
C:\>py -3 -m pip -V
pip 9.0.1 from C:\Python36\lib\site-packages (python 3.6)

C:\>py -2 -m pip -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)
```

Linux

The latest versions of CentOS, Fedora, Red Hat Enterprise (RHEL) and Ubuntu come with Python 2.7.

To install Python 2.7 on linux manually, just do the following in terminal:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
```

```
sudo make install
```

Also add the path of new python in PATH environment variable. If new python is in `/root/python-2.7.X` then run `export PATH = $PATH:/root/python-2.7.X`

Now to check if Python installation is valid write in terminal:

```
python --version
```

Ubuntu (From Source)

If you need Python 3.6 you can install it from source as shown below (Ubuntu 16.10 and 17.04 have 3.6 version in the universal repository). Below steps have to be followed for Ubuntu 16.04 and lower versions:

```
sudo apt install build-essential checkinstall
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz
tar xvf Python-3.6.1.tar.xz
cd Python-3.6.1/
./configure --enable-optimizations
sudo make altinstall
```

macOS

As we speak, macOS comes installed with Python 2.7.10, but this version is outdated and slightly modified from the regular Python.

> The version of Python that ships with OS X is great for learning but it's not good for development. The version shipped with OS X may be out of date from the official current Python release, which is considered the stable production version. (source)

Install Homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install Python 2.7:

```
brew install python
```

For Python 3.x, use the command `brew install python3` instead.

Section 1.11: String function - str() and repr()

There are two functions that can be used to obtain a readable representation of an object.

`repr(x)` calls `x.__repr__()`: a representation of `x`. `eval` will usually convert the result of this function back to the original object.

`str(x)` calls `x.__str__()`: a human-readable string that describes the object. This may elide some technical detail.

repr()

For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`. Otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object along with additional information. This often includes the name and address of the object.

str()

For strings, this returns the string itself. The difference between this and `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to `eval()`. Rather, its goal is to return a printable or 'human readable' string. If no argument is given, this returns the empty string, `''`.

Example 1:

```
s = """w'o"w"""
repr(s) # Output: '\'w\\\'o"w\''
str(s)  # Output: 'w\'o"w'
eval(str(s)) == s  # Gives a SyntaxError
eval(repr(s)) == s # Output: True
```

Example 2:

```
import datetime
today = datetime.datetime.now()
str(today)  # Output: '2016-09-15 06:58:46.915000'
repr(today) # Output: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

When writing a class, you can override these methods to do whatever you want:

```
class Represent(object):

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y=\"{}\")".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

Using the above class we can see the results:

```
r = Represent(1, "Hopper")
print(r)  # prints __str__
print(r.__repr__)  # prints __repr__: '<bound method Represent.__repr__ of
Represent(x=1,y="Hopper")>'
rep = r.__repr__()  # sets the execution of __repr__ to a new variable
print(rep)  # prints 'Represent(x=1,y="Hopper")'
r2 = eval(rep) # evaluates rep
print(r2)  # prints __str__ from new object
print(r2 == r)  # prints 'False' because they are different objects
```

Section 1.12: Installing external modules using pip

`pip` is your friend when you need to install any package from the plethora of choices available at the python package index (PyPI). `pip` is already installed if you're using Python 2 >= 2.7.9 or Python 3 >= 3.4 downloaded from python.org. For computers running Linux or another *nix with a native package manager, `pip` must often be manually installed.

On instances with both Python 2 and Python 3 installed, `pip` often refers to Python 2 and `pip3` to Python 3. Using `pip` will only install packages for Python 2 and `pip3` will only install packages for Python 3.

Finding / installing a package

Searching for a package is as simple as typing

```
$ pip search <query>
# Searches for packages whose name or summary contains <query>
```

Installing a package is as simple as typing *(in a terminal / command-prompt, not in the Python interpreter)*

```
$ pip install [package_name]           # latest version of the package

$ pip install [package_name]==x.x.x    # specific version of the package

$ pip install '[package_name]>=x.x.x'  # minimum version of the package
```

where `x.x.x` is the version number of the package you want to install.

When your server is behind proxy, you can install package by using below command:

```
$ pip --proxy http://<server address>:<port> install
```

Upgrading installed packages

When new versions of installed packages appear they are not automatically installed to your system. To get an overview of which of your installed packages have become outdated, run:

```
$ pip list --outdated
```

To upgrade a specific package use

```
$ pip install [package_name] --upgrade
```

Updating all outdated packages is not a standard functionality of `pip`.

Upgrading pip

You can upgrade your existing pip installation by using the following commands

- On Linux or macOS X:

  ```
  $ pip install -U pip
  ```

  You may need to use `sudo` with pip on some Linux Systems

- On Windows:

  ```
  py -m pip install -U pip
  ```

  or

  ```
  python -m pip install -U pip
  ```

For more information regarding pip do read here.

Section 1.13: Help Utility

Python has several functions built into the interpreter. If you want to get information of keywords, built-in functions, modules or topics open a Python console and enter:

```
>>> help()
```

You will receive information by entering keywords directly:

```
>>> help(help)
```

or within the utility:

```
help> help
```

which will show an explanation:

```
Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
| Define the builtin 'help'.
|
| This is a wrapper around pydoc.help that provides a helpful message
| when 'help' is typed at the Python interactive prompt.
|
| Calling help() at the Python prompt starts an interactive help session.
| Calling help(thing) prints help for the python object 'thing'.
|
| Methods defined here:
|
| __call__(self, *args, **kwds)
|
| __repr__(self)
|
| ----------------------------------------------------------------------
| Data descriptors defined here:
|
| __dict__
| dictionary for instance variables (if defined)
|
| __weakref__
| list of weak references to the object (if defined)
```

You can also request subclasses of modules:

```
help(pymysql.connections)
```

You can use help to access the docstrings of the different modules you have imported, e.g., try the following:

```
>>> help(math)
```

and you'll get an error

```
>>> import math
```

```
>>> help(math)
```

And now you will get a list of the available methods in the module, but only AFTER you have imported it.

Close the helper with `quit`

Chapter 2: Python Data Types

Data types are nothing but variables you use to reserve some space in memory. Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

Section 2.1: String Data Type

String are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Strings are immutable sequence data type, i.e each time one makes any changes to a string, completely new string object is created.

```
a_str = 'Hello World'
print(a_str)    #output will be whole string. Hello World
print(a_str[0])    #output will be first character. H
print(a_str[0:5])    #output will be first five characters. Hello
```

Section 2.2: Set Data Types

Sets are unordered collections of unique objects, there are two types of set:

1. Sets - They are mutable and new elements can be added once sets are defined

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)            # duplicates will be removed
> {'orange', 'banana', 'pear', 'apple'}
a = set('abracadabra')
print(a)                 # unique letters in a
> {'a', 'r', 'b', 'c', 'd'}
a.add('z')
print(a)
> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2. Frozen Sets - They are immutable and new elements cannot added after its defined.

```
b = frozenset('asdfagsa')
print(b)
> frozenset({'f', 'g', 'd', 'a', 's'})
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

Section 2.3: Numbers data type

Numbers have four types in Python. Int, float, complex, and long.

```
int_num = 10    #int value
float_num = 10.2    #float value
complex_num = 3.14j    #complex value
long_num = 1234567L    #long value
```

Section 2.4: List Data Type

A list contains items separated by commas and enclosed within square brackets [].lists are almost similar to arrays in C. One difference is that all the items belonging to a list can be of different data type.

```
list = [123,'abcd',10.2,'d']    #can be an array of any data type or single data type.
list1 = ['hello','world']
print(list)    #will output whole list. [123,'abcd',10.2,'d']
print(list[0:2])    #will output first two element of list. [123,'abcd']
print(list1 * 2)    #will gave list1 two times. ['hello','world','hello','world']
print(list + list1)    #will gave concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

Section 2.5: Dictionary Data Type

Dictionary consists of key-value pairs. It is enclosed by curly braces {} and values can be assigned and accessed using square brackets[].

```
dic={'name':'red','age':10}
print(dic)    #will output all the key-value pairs. {'name':'red','age':10}
print(dic['name'])    #will output only value with 'name' key. 'red'
print(dic.values())    #will output list of values in dic. ['red',10]
print(dic.keys())    #will output list of keys. ['name','age']
```

Section 2.6: Tuple Data Type

Lists are enclosed in brackets [ ] and their elements and size can be changed, while tuples are enclosed in parentheses ( ) and cannot be updated. Tuples are immutable.

```
tuple = (123,'hello')
tuple1 = ('world')
print(tuple)    #will output whole tuple. (123,'hello')
print(tuple[0])    #will output first value. (123)
print(tuple + tuple1)    #will output (123,'hello','world')
tuple[1]='update'    #this will give you error.
```

Chapter 3: Indentation

Section 3.1: Simple example

For Python, Guido van Rossum based the grouping of statements on indentation. The reasons for this are explained in the first section of the "Design and History Python FAQ". Colons, `:`, are used to declare an indented code block, such as the following example:

```
class ExampleClass:
    #Every function belonging to a class must be indented equally
    def __init__(self):
        name = "example"

    def someFunction(self, a):
        #Notice everything belonging to a function must be indented
        if a > 5:
            return True
        else:
            return False

#If a function is not indented to the same level it will not be considers as part of the parent class
def separateFunction(b):
    for i in b:
    #Loops are also indented and nested conditions start a new indentation
        if i == 1:
            return True
    return False

separateFunction([2,3,5,6,1])
```

Spaces or Tabs?

The recommended indentation is 4 spaces but tabs or spaces can be used so long as they are consistent. ***Do not mix tabs and spaces in Python*** as this will cause an error in Python 3 and can causes errors in Python 2.

Section 3.2: How Indentation is Parsed

Whitespace is handled by the lexical analyzer before being parsed.

The lexical analyzer uses a stack to store indentation levels. At the beginning, the stack contains just the value 0, which is the leftmost position. Whenever a nested block begins, the new indentation level is pushed on the stack, and an "INDENT" token is inserted into the token stream which is passed to the parser. There can never be more than one "INDENT" token in a row (`IndentationError`).

When a line is encountered with a smaller indentation level, values are popped from the stack until a value is on top which is equal to the new indentation level (if none is found, a syntax error occurs). For each value popped, a "DEDENT" token is generated. Obviously, there can be multiple "DEDENT" tokens in a row.

The lexical analyzer skips empty lines (those containing only whitespace and possibly comments), and will never generate either "INDENT" or "DEDENT" tokens for them.

At the end of the source code, "DEDENT" tokens are generated for each indentation level left on the stack, until just the 0 is left.

For example:

```
if foo:
    if bar:
        x = 42
else:
    print foo
```

is analyzed as:

```
<if> <foo> <:>                  [0]
<INDENT> <if> <bar> <:>         [0, 4]
<INDENT> <x> <=> <42>           [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>    [0]
<INDENT> <print> <foo>          [0, 2]
<DEDENT>
```

The parser than handles the "INDENT" and "DEDENT" tokens as block delimiters.

Section 3.3: Indentation Errors

The spacing should be even and uniform throughout. Improper indentation can cause an `IndentationError` or cause the program to do something unexpected. The following example raises an `IndentationError`:

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

Or if the line following a colon is not indented, an `IndentationError` will also be raised:

```
if True:
print "true"
```

If you add indentation where it doesn't belong, an `IndentationError` will be raised:

```
if  True:
    a = 6
        b = 5
```

If you forget to un-indent functionality could be lost. In this example `None` is returned instead of the expected `False`:

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

Chapter 4: Comments and Documentation

Section 4.1: Single line, inline and multiline comments

Comments are used to explain code when the basic code itself isn't clear.

Python ignores comments, and so will not execute code in there, or raise syntax errors for plain English sentences.

Single-line comments begin with the hash character (#) and are terminated by the end of line.

- Single line comment:

```
# This is a single line comment in Python
```

- Inline comment:

```
print("Hello World")  # This line prints "Hello World"
```

- Comments spanning multiple lines have `"""` or `'''` on either end. This is the same as a multiline string, but they can be used as comments:

```
"""
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
"""
```

Section 4.2: Programmatically accessing docstrings

Docstrings are - unlike regular comments - stored as an attribute of the function they document, meaning that you can access them programmatically.

An example function

```
def func():
    """This is a function that does nothing at all"""
    return
```

The docstring can be accessed using the `__doc__` attribute:

```
print(func.__doc__)
```

> This is a function that does nothing at all

```
help(func)
```

> Help on function `func` in module `__main__`:
>
> `func()`
>
> This is a function that does nothing at all

Another example function

`function.__doc__` is just the actual docstring as a string, while the `help` function provides general information about a function, including the docstring. Here's a more helpful example:

```
def greet(name, greeting="Hello"):
    """Print a greeting to the user `name`

    Optional parameter `greeting` can change what they're greeted with."""

    print("{} {}".format(greeting, name))
```

```
help(greet)
```

> Help on function `greet` in module `__main__`:
>
> `greet(name, greeting='Hello')`
>
> Print a greeting to the user `name`
> Optional parameter `greeting` can change what they're greeted with.

Advantages of docstrings over regular comments

Just putting no docstring or a regular comment in a function makes it a lot less helpful.

```
def greet(name, greeting="Hello"):
    # Print a greeting to the user `name`
    # Optional parameter `greeting` can change what they're greeted with.

    print("{} {}".format(greeting, name))
```

```
print(greet.__doc__)
```

> None

```
help(greet)
```

> Help on function greet in module **main**:
>
> `greet(name, greeting='Hello')`

Section 4.3: Write documentation using docstrings

A docstring is a multi-line comment used to document modules, classes, functions and methods. It has to be the first statement of the component it describes.

```
def hello(name):
    """Greet someone.

    Print a greeting ("Hello") for the person with the given name.
    """

    print("Hello "+name)
```

```
class Greeter:
    """An object used to greet people.
```

```
    It contains multiple greeting functions for several languages
    and times of the  day.
    """
```

The value of the docstring can be accessed within the program and is - for example - used by the `help` command.

Syntax conventions
PEP 257

PEP 257 defines a syntax standard for docstring comments. It basically allows two types:

- One-line Docstrings:

According to PEP 257, they should be used with short and simple functions. Everything is placed in one line, e.g:

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

The docstring shall end with a period, the verb should be in the imperative form.

- Multi-line Docstrings:

Multi-line docstring should be used for longer, more complex functions, modules or classes.

```
def hello(name, language="en"):
    """Say hello to a person.

    Arguments:
    name: the name of the person
    language: the language in which the person should be greeted
    """

    print(greeting[language]+" "+name)
```

They start with a short summary (equivalent to the content of a one-line docstring) which can be on the same line as the quotation marks or on the next line, give additional detail and list parameters and return values.

Note PEP 257 defines what information should be given within a docstring, it doesn't define in which format it should be given. This was the reason for other parties and documentation parsing tools to specify their own standards for documentation, some of which are listed below and in this question.

Sphinx

Sphinx is a tool to generate HTML based documentation for Python projects based on docstrings. Its markup language used is reStructuredText. They define their own standards for documentation, pythonhosted.org hosts a very good description of them. The Sphinx format is for example used by the pyCharm IDE.

A function would be documented like this using the Sphinx/reStructuredText format:

```
def hello(name, language="en"):
    """Say hello to a person.

    :param name: the name of the person
    :type name: str
    :param language: the language in which the person should be greeted
    :type language: str
```

```
    :return: a number
    :rtype: int
    """

    print(greeting[language]+" "+name)
    return 4
```

Google Python Style Guide

Google has published Google Python Style Guide which defines coding conventions for Python, including documentation comments. In comparison to the Sphinx/reST many people say that documentation according to Google's guidelines is better human-readable.

The pythonhosted.org page mentioned above also provides some examples for good documentation according to the Google Style Guide.

Using the Napoleon plugin, Sphinx can also parse documentation in the Google Style Guide-compliant format.

A function would be documented like this using the Google Style Guide format:

```
def hello(name, language="en"):
    """Say hello to a person.

    Args:
        name: the name of the person as string
        language: the language code string

    Returns:
        A number.
    """

    print(greeting[language]+" "+name)
    return 4
```

Chapter 5: Date and Time

Section 5.1: Parsing a string into a timezone aware datetime object

Python 3.2+ has support for `%z` format when parsing a string into a `datetime` object.

> UTC offset in the form `+HHMM` or `-HHMM` (empty string if the object is naive).

Python 3.x `Version ≥ 3.2`

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

For other versions of Python, you can use an external library such as `dateutil`, which makes parsing a string with timezone into a `datetime` object is quick.

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

The `dt` variable is now a `datetime` object with the following value:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

Section 5.2: Constructing timezone-aware datetimes

By default all `datetime` objects are naive. To make them timezone-aware, you must attach a `tzinfo` object, which provides the UTC offset and timezone abbreviation as a function of date and time.

Fixed Offset Time Zones

For time zones that are a fixed offset from UTC, in Python 3.2+, the `datetime` module provides the `timezone` class, a concrete implementation of `tzinfo`, which takes a `timedelta` and an (optional) name parameter:

Python 3.x `Version ≥ 3.2`

```
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname)
# 'JST'
```

For Python versions before 3.2, it is necessary to use a third party library, such as `dateutil`. `dateutil` provides an equivalent class, `tzoffset`, which (as of version 2.5.3) takes arguments of the form `dateutil.tz.tzoffset(tzname, offset)`, where `offset` is specified in seconds:

Python 3.x `Version < 3.2`

Python 2.x `Version < 2.7`

```
from datetime import datetime, timedelta
from dateutil import tz

JST = tz.tzoffset('JST', 9 * 3600) # 3600 seconds per hour
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname)
# 'JST'
```

Zones with daylight savings time

For zones with daylight savings time, python standard libraries do not provide a standard class, so it is necessary to use a third party library. `pytz` and `dateutil` are popular libraries providing time zone classes.

In addition to static time zones, `dateutil` provides time zone classes that use daylight savings time (see the documentation for the `tz` module). You can use the `tz.gettz()` method to get a time zone object, which can then be passed directly to the `datetime` constructor:

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # Local time
PT = tz.gettz('US/Pacific') # Pacific time

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # I am in EST
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # DST is handled automatically
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

CAUTION: As of version 2.5.3, `dateutil` does not handle ambiguous datetimes correctly, and will always default to the *later* date. There is no way to construct an object with a `dateutil` timezone representing, for example `2015-11-01 1:30 EDT-4`, since this is *during* a daylight savings time transition.

All edge cases are handled properly when using `pytz`, but `pytz` time zones should *not* be directly attached to time zones through the constructor. Instead, a `pytz` time zone should be attached using the time zone's `localize` method:

```
from datetime import datetime, timedelta
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

Be aware that if you perform datetime arithmetic on a `pytz`-aware time zone, you must either perform the calculations in UTC (if you want absolute elapsed time), or you must call `normalize()` on the result:

```
dt_new = dt_pdt + timedelta(hours=3) # This should be 2:30 AM PST
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00
```

Section 5.3: Computing time differences

the `timedelta` module comes in handy to compute differences between times:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)      # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

Specifying time is optional when creating a new `datetime` object

```
delta = now-then
```

`delta` is of type `timedelta`

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

To get n day's after and n day's before date we could use:

n day's after date:

```
def get_n_days_after_date(date_format="%d %B %Y", add_days=120):

    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)
```

n day's before date:

```
 def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):

        date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
        return date_n_days_ago.strftime(date_format)
```

Section 5.4: Basic datetime objects usage

The datetime module contains three primary types of objects - date, time, and datetime.

```
import datetime

# Date object
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# Time object
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# Current datetime
now = datetime.datetime.now()
```

```
# Datetime object
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

Arithmetic operations for these objects are only supported within same datatype and performing simple arithmetic with instances of different types will result in a TypeError.

```
# subtraction of noon from today
noon-today
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.date'
However, it is straightforward to convert between types.

# Do this instead
print('Time since the millenium at midnight: ',
      datetime.datetime(today.year, today.month, today.day) - millenium_turn)

# Or this
print('Time since the millenium at noon: ',
      datetime.datetime.combine(today, noon) - millenium_turn)
```

Section 5.5: Switching between time zones

To switch between time zones, you need datetime objects that are timezone-aware.

```
from datetime import datetime
from dateutil import tz

utc = tz.tzutc()
local = tz.tzlocal()

utc_now = datetime.utcnow()
utc_now # Not timezone-aware.

utc_now = utc_now.replace(tzinfo=utc)
utc_now # Timezone-aware.

local_now = utc_now.astimezone(local)
local_now # Converted to local time.
```

Section 5.6: Simple date arithmetic

Dates don't exist in isolation. It is common that you will need to find the amount of time between dates or determine what the date will be tomorrow. This can be accomplished using `timedelta` objects

```
import datetime

today = datetime.date.today()
print('Today:', today)

yesterday = today - datetime.timedelta(days=1)
print('Yesterday:', yesterday)

tomorrow = today + datetime.timedelta(days=1)
print('Tomorrow:', tomorrow)

print('Time between tomorrow and yesterday:', tomorrow - yesterday)
```

This will produce results similar to:

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

Section 5.7: Converting timestamp to datetime

The `datetime` module can convert a POSIX `timestamp` to a ITC `datetime` object.

The Epoch is January 1st, 1970 midnight.

```
import time
from datetime import datetime
seconds_since_epoch=time.time()  #1469182681.709

utc_date=datetime.utcfromtimestamp(seconds_since_epoch) #datetime.datetime(2016, 7, 22, 10, 18, 1,
709000)
```

Section 5.8: Subtracting months from a date accurately

Using the `calendar` module

```
import calendar
from datetime import date

def monthdelta(date, delta):
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12
    if not m: m = 12
    d = min(date.day, calendar.monthrange(y, m)[1])
    return date.replace(day=d,month=m, year=y)

next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

Using the `dateutils` module

```
import datetime
import dateutil.relativedelta

d = datetime.datetime.strptime("2013-03-31", "%Y-%m-%d")
d2 = d - dateutil.relativedelta.relativedelta(months=1)  #datetime.datetime(2013, 2, 28, 0, 0)
```

Section 5.9: Parsing an arbitrary ISO 8601 timestamp with minimal libraries

Python has only limited support for parsing ISO 8601 timestamps. For `strptime` you need to know exactly what format it is in. As a complication the stringification of a `datetime` is an ISO 8601 timestamp, with space as a separator and 6 digit fraction:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

but if the fraction is 0, no fractional part is output

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

But these 2 forms need a *different* format for `strptime`. Furthermore, `strptime' does not support at all parsing minute timezones that have a:in it, thus2016-07-22 09:25:59+0300can be parsed, but the standard format2016-07-22 09:25:59+03:00` cannot.

There is a single-file library called `iso8601` which properly parses ISO 8601 timestamps and only them.

It supports fractions and timezones, and the `T` separator all with a single function:

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

If no timezone is set, `iso8601.parse_date` defaults to UTC. The default zone can be changed with `default_zone` keyword argument. Notably, if this is `None` instead of the default, then those timestamps that do not have an explicit timezone are returned as naive datetimes instead:

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

Section 5.10: Get an ISO 8601 timestamp

Without timezone, with microseconds

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

With timezone, with microseconds

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

With timezone, without microseconds

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

See ISO 8601 for more information about the ISO 8601 format.

Section 5.11: Parsing a string with a short time zone name into

a timezone aware datetime object

Using the `dateutil` library as in the previous example on parsing timezone-aware timestamps, it is also possible to parse timestamps with a specified "short" time zone name.

For dates formatted with short time zone names or abbreviations, which are generally ambiguous (e.g. CST, which could be Central Standard Time, China Standard Time, Cuba Standard Time, etc - more can be found here) or not necessarily available in a standard database, it is necessary to specify a mapping between time zone abbreviation and `tzinfo` object.

```
from dateutil import tz
from dateutil.parser import parse

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
              'EST': ET, 'EDT': ET,
              'MST': MT, 'MDT': MT,
              'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)
```

After running this:

```
dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
dt_pst
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))
```

It is worth noting that if using a `pytz` time zone with this method, it will *not* be properly localized:

```
from dateutil.parser import parse
import pytz

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})
```

This simply attaches the `pytz` time zone to the datetime:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

If using this method, you should probably re-`localize` the naive portion of the datetime after parsing:

```
dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # Now it's EST.
# <DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>)
```

Section 5.12: Fuzzy datetime parsing (extracting datetime out of a text)

It is possible to extract a date out of a text using the dateutil parser in a "fuzzy" mode, where components of the

string not recognized as being part of a date are ignored.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` is now a *`datetime` object* and you would see `datetime.datetime(2047, 1, 1, 8, 21)` printed.

Section 5.13: Iterate over dates

Sometimes you want to iterate over a range of dates from a start date to some end date. You can do it using `datetime` library and `timedelta` object:

```
import datetime

# The size of each step in days
day_delta = datetime.timedelta(days=1)

start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
```

Which produces:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Chapter 6: Date Formatting

Section 6.1: Time between two date-times

```
from datetime import datetime

a = datetime(2016,10,06,0,0,0)
b = datetime(2016,10,01,23,59,59)

a-b
# datetime.timedelta(4, 1)

(a-b).days
# 4
(a-b).total_seconds()
# 518399.0
```

Section 6.2: Outputting datetime object to string

Uses C standard format codes.

```
from datetime import datetime
datetime_for_string = datetime(2016,10,1,0,0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strftime(datetime_for_string,datetime_string_format)
# Oct 01 2016, 00:00:00
```

Section 6.3: Parsing string to datetime object

Uses C standard format codes.

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

Chapter 7: Enum

Section 7.1: Creating an enum (Python 2.4 through 3.3)

Enums have been backported from Python 3.4 to Python 2.4 through Python 3.3. You can get this the enum34 backport from PyPI.

```
pip install enum34
```

Creation of an enum is identical to how it works in Python 3.4+

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red)  # Color.red
print(Color(1))  # Color.red
print(Color['red'])  # Color.red
```

Section 7.2: Iteration

Enums are iterable:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color]  # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

Chapter 8: Set

Section 8.1: Operations on sets

with other sets

```
# Intersection
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6})  # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6}              # {3, 4, 5}

# Union
{1, 2, 3, 4, 5}.union({3, 4, 5, 6})  # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6}       # {1, 2, 3, 4, 5, 6}

# Difference
{1, 2, 3, 4}.difference({2, 3, 5})  # {1, 4}
{1, 2, 3, 4} - {2, 3, 5}            # {1, 4}

# Symmetric difference with
{1, 2, 3, 4}.symmetric_difference({2, 3, 5})  # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5}                      # {1, 4, 5}

# Superset check
{1, 2}.issuperset({1, 2, 3})  # False
{1, 2} >= {1, 2, 3}           # False

# Subset check
{1, 2}.issubset({1, 2, 3})  # True
{1, 2} <= {1, 2, 3}         # True

# Disjoint check
{1, 2}.isdisjoint({3, 4})  # True
{1, 2}.isdisjoint({1, 4})  # False
```

with single elements

```
# Existence check
2 in {1,2,3}      # True
4 in {1,2,3}      # False
4 not in {1,2,3}  # True

# Add and Remove
s = {1,2,3}
s.add(4)        # s == {1,2,3,4}

s.discard(3)    # s == {1,2,4}
s.discard(5)    # s == {1,2,4}

s.remove(2)     # s == {1,4}
s.remove(2)     # KeyError!
```

Set operations return new sets, but have the corresponding in-place versions:

method	in-place operation	in-place method
union	s |= t	update
intersection	s &= t	intersection_update
difference	s -= t	difference_update

symmetric_difference	s ^= t	symmetric_difference_update

For example:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Section 8.2: Get the unique elements of a list

Let's say you've got a list of restaurants -- maybe you read it from a file. You care about the *unique* restaurants in the list. The best way to get the unique elements from a list is to turn it into a set:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Note that the set is not in the same order as the original list; that is because sets are *unordered*, just like `dict`s.

This can easily be transformed back into a `List` with Python's built in `list` function, giving another list that is the same list as the original but without duplicates:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

It's also common to see this as one line:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Now any operations that could be performed on the original list can be done again.

Section 8.3: Set of Sets

```
{{1,2}, {3,4}}
```

leads to:

```
TypeError: unhashable type: 'set'
```

Instead, use `frozenset`:

```
{frozenset({1, 2}), frozenset({3, 4})}
```

Section 8.4: Set Operations using Methods and Builtins

We define two sets `a` and `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> NOTE: `{1}` creates a set of one element, but `{}` creates an empty `dict`. The correct way to create an empty set is `set()`.

Intersection

`a.intersection(b)` returns a new set with elements present in both `a` and `b`

```
>>> a.intersection(b)
{3, 4}
```

Union

`a.union(b)` returns a new set with elements present in either `a` and `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Difference

`a.difference(b)` returns a new set with elements present in `a` but not in `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

Symmetric Difference

`a.symmetric_difference(b)` returns a new set with elements present in either `a` or `b` but not in both

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

NOTE: `a.symmetric_difference(b) == b.symmetric_difference(a)`

Subset and superset

`c.issubset(a)` tests whether each element of `c` is in `a`.

`a.issuperset(c)` tests whether each element of `c` is in `a`.

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

The latter operations have equivalent operators as shown below:

Method	Operator
`a.intersection(b)`	`a & b`
`a.union(b)`	`a|b`
`a.difference(b)`	`a - b`
`a.symmetric_difference(b)`	`a ^ b`
`a.issubset(b)`	`a <= b`
`a.issuperset(b)`	`a >= b`

Disjoint sets

Sets `a` and `d` are disjoint if no element in `a` is also in `d` and vice versa.

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} are in both sets
False
>>> a.isdisjoint(d)
True

# This is an equivalent check, but less efficient
>>> len(a & d) == 0
True

# This is even less efficient
>>> a & d == set()
True
```

Testing membership

The builtin `in` keyword searches for occurances

```
>>> 1 in a
True
>>> 6 in a
False
```

Length

The builtin `len()` function returns the number of elements in the set

```
>>> len(a)
4
>>> len(b)
3
```

Section 8.5: Sets versus multisets

Sets are unordered collections of distinct elements. But sometimes we want to work with unordered collections of elements that are not necessarily distinct and keep track of the elements' multiplicities.

Consider this example:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

By saving the strings `'a'`, `'b'`, `'b'`, `'c'` into a set data structure we've lost the information on the fact that `'b'` occurs twice. Of course saving the elements to a list would retain this information

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

but a list data structure introduces an extra unneeded ordering that will slow down our computations.

For implementing multisets Python provides the `Counter` class from the `collections` module (starting from version 2.7):

Python 2.x `Version ≥ 2.7`

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
Counter({'b': 2, 'a': 1, 'c': 1})
```

`Counter` is a dictionary where where elements are stored as dictionary keys and their counts are stored as dictionary values. And as all dictionaries, it is an unordered collection.

Chapter 9: Simple Mathematical Operators

Numerical types and their metaclasses

The `numbers` module contains the abstract metaclasses for the numerical types:

subclasses	numbers.Number	numbers.Integral	numbers.Rational	numbers.Real	numbers.Complex
bool	✓	✓	✓	✓	✓
int	✓	✓	✓	✓	✓
fractions.Fraction	✓	–	✓	✓	✓
float	✓	–	–	✓	✓
complex	✓	–	–	–	✓
decimal.Decimal	✓	–	–	–	–

Python does common mathematical operators on its own, including integer and float division, multiplication, exponentiation, addition, and subtraction. The math module (included in all standard Python versions) offers expanded functionality like trigonometric functions, root operations, logarithms, and many more.

Section 9.1: Division

Python does integer division when both operands are integers. The behavior of Python's division operators have changed from Python 2.x and 3.x (see also Integer Division).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x `Version ≤ 2.7`

In Python 2 the result of the ' / ' operator depends on the type of the numerator and denominator.

```
a / b                # = 1

a / c                # = 1.5

d / b                # = -2

b / a                # = 0

d / e                # = -1
```

Note that because both `a` and `b` are `int`s, the result is an `int`.

The result is always rounded down (floored).

Because `c` is a float, the result of `a / c` is a `float`.

You can also use the operator module:

```
import operator         # the operator module provides 2-argument arithmetic functions
operator.div(a, b)      # = 1
operator.__div__(a, b) # = 1
```

Python 2.x `Version ≥ 2.2`

What if you want float division:

Recommended:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                   # = 1.5
a // b                  # = 1
```

Okay (if you don't want to apply to the whole module):

```
a / (b * 1.0)           # = 1.5
1.0 * a / b             # = 1.5
a / b * 1.0             # = 1.0    (careful with order of operations)

from operator import truediv
truediv(a, b)           # = 1.5
```

Not recommended (may raise TypeError, eg if argument is complex):

```
float(a) / b            # = 1.5
a / float(b)            # = 1.5
```

Python 2.x `Version ≥ 2.2`

The ' // ' operator in Python 2 forces floored division regardless of type.

```
a // b                # = 1
a // c                # = 1.0
```

Python 3.x `Version ≥ 3.0`

In Python 3 the `/` operator performs 'true' division regardless of types. The `//` operator performs floor division and maintains type.

```
a / b                   # = 1.5
e / b                   # = 5.0
a // b                  # = 1
a // c                  # = 1.0

import operator             # the operator module provides 2-argument arithmetic functions
operator.truediv(a, b)      # = 1.5
operator.floordiv(a, b)     # = 1
operator.floordiv(a, c)     # = 1.0
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int` in Python 2 and a `float` in Python 3)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

See PEP 238 for more information.

Section 9.2: Addition

```
a, b = 1, 2

# Using the "+" operator:
a + b                   # = 3
```

```
# Using the "in-place" "+=" operator to add and assign:
a += b                  # a = 3 (equivalent to a = a + b)

import operator         # contains 2 argument arithmetic functions for the examples

operator.add(a, b)      # = 5  since a is set to 3 right before this line

# The "+=" operator is equivalent to:
a = operator.iadd(a, b)    # a = 5 since a is set to 3 right before this line
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Note: the `+` operator is also used for concatenating strings, lists and tuples:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Section 9.3: Exponentiation

```
a, b = 2, 3

(a ** b)                # = 8
pow(a, b)               # = 8

import math
math.pow(a, b)          # = 8.0 (always float; does not allow complex results)

import operator
operator.pow(a, b)      # = 8
```

Another difference between the built-in `pow` and `math.pow` is that the built-in `pow` can accept three arguments:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)            # 0, calculates (2 ** 3) % 2, but as per Python docs,
                        #    does so more efficiently
```

Special functions

The function `math.sqrt(x)` calculates the square root of `x`.

```
import math
import cmath
c = 4
math.sqrt(c)            # = 2.0 (always float; does not allow complex results)
cmath.sqrt(c)           # = (2+0j) (always complex)
```

To compute other roots, such as a cube root, raise the number to the reciprocal of the degree of the root. This could be done with any of the exponential functions or operator.

```
import math
x = 8
math.pow(x, 1/3) # evaluates to 2.0
x**(1/3) # evaluates to 2.0
```

The function `math.exp(x)` computes `e ** x`.

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

The function `math.expm1(x)` computes `e ** x - 1`. When x is small, this gives significantly better precision than `math.exp(x) - 1`.

```
math.expm1(0)       # 0.0

math.exp(1e-6) - 1  # 1.0000004999621837e-06
math.expm1(1e-6)    # 1.0000005000001665e-06
# exact result      # 1.0000005000001666667083333416666...
```

Section 9.4: Trigonometric Functions

```
a, b = 1, 2

import math

math.sin(a)  # returns the sine of 'a' in radians
# Out: 0.8414709848078965

math.cosh(b)  # returns the inverse hyperbolic cosine of 'b' in radians
# Out: 3.7621956910836314

math.atan(math.pi)  # returns the arc tangent of 'pi' in radians
# Out: 1.2626272556789115

math.hypot(a, b) # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
# Out: 2.23606797749979
```

> Note that `math.hypot(x, y)` is also the length of the vector (or Euclidean distance) from the origin `(0, 0)` to the point `(x, y)`.
>
> To compute the Euclidean distance between two points `(x1, y1)` & `(x2, y2)` you can use `math.hypot` as follows
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

To convert from radians -> degrees and degrees -> radians respectively use `math.degrees` and `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

Section 9.5: Inplace Operations

It is common within applications to need to have code like this:

```
a = a + 1
```

or

```
a = a * 2
```

There is an effective shortcut for these in place operations:

```
a += 1
# and
a *= 2
```

Any mathematic operator can be used before the '=' character to make an inplace operation:

- `-=` decrement the variable in place
- `+=` increment the variable in place
- `*=` multiply the variable in place
- `/=` divide the variable in place
- `//=` floor divide the variable in place # Python 3
- `%=` return the modulus of the variable in place
- `**=` raise to a power in place

Other in place operators exist for the bitwise operators (`^`, `|` etc)

Section 9.6: Subtraction

```
a, b = 1, 2

# Using the "-" operator:
b - a                   # = 1


import operator         # contains 2 argument arithmetic functions
operator.sub(b, a)      # = 1
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Section 9.7: Multiplication

```
a, b = 2, 3

a * b                   # = 6

import operator
```

```
operator.mul(a, b)     # = 6
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Note: The `*` operator is also used for repeated concatenation of strings, lists, and tuples:

```
3 * 'ab'  # = 'abababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

Section 9.8: Logarithms

By default, the `math.log` function calculates the logarithm of a number, base e. You can optionally specify a base as the second argument.

```
import math
import cmath

math.log(5)         # = 1.6094379124341003
# optional base argument. Default is math.e
math.log(5, math.e) # = 1.6094379124341003
cmath.log(5)        # = (1.6094379124341003+0j)
math.log(1000, 10)   # 3.0 (always returns float)
cmath.log(1000, 10)  # (3+0j)
```

Special variations of the `math.log` function exist for different bases.

```
# Logarithm base e - 1 (higher precision for low values)
math.log1p(5)       # = 1.791759469228055

# Logarithm base 2
math.log2(8)        # = 3.0

# Logarithm base 10
math.log10(100)     # = 2.0
cmath.log10(100)    # = (2+0j)
```

Section 9.9: Modulus

Like in many other languages, Python uses the `%` operator for calculating modulus.

```
3 % 4     # 3
10 % 2    # 0
6 % 4     # 2
```

Or by using the `operator` module:

```
import operator

operator.mod(3 , 4)     # 3
```

```
operator.mod(10 , 2)    # 0
operator.mod(6 , 4)     # 2
```

You can also use negative numbers.

```
-9 % 7      # 5
9 % -7      # -5
-9 % -7     # -2
```

If you need to find the result of integer division and modulus, you can use the `divmod` function as a shortcut:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Chapter 10: Bitwise Operators

Bitwise operations alter binary strings at the bit level. These operations are incredibly basic and are directly supported by the processor. These few operations are necessary in working with device drivers, low-level graphics, cryptography, and network communications. This section provides useful knowledge and examples of Python's bitwise operators.

Section 10.1: Bitwise NOT

The `~` operator will flip all of the bits in the number. Since computers use signed number representations — most notably, the two's complement notation to encode negative binary numbers where negative numbers are written with a leading one (1) instead of a leading zero (0).

This means that if you were using 8 bits to represent your two's-complement numbers, you would treat patterns from `0000 0000` to `0111 1111` to represent numbers from 0 to 127 and reserve `1xxx xxxx` to represent negative numbers.

Eight-bit two's-complement numbers

Bits	Unsigned Value	Two's-complement Value
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

In essence, this means that whereas `1010 0110` has an unsigned value of 166 (arrived at by adding `(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)`), it has a two's-complement value of -90 (arrived at by adding `(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)`, and complementing the value).

In this way, negative numbers range down to -128 (`1000 0000`). Zero (0) is represented as `0000 0000`, and minus one (-1) as `1111 1111`.

In general, though, this means `~n = -n - 1`.

```
# 0 = 0b0000 0000
~0
# Out: -1
# -1 = 0b1111 1111

# 1 = 0b0000 0001
~1
# Out: -2
# -2 = 1111 1110

# 2 = 0b0000 0010
~2
```

```
# Out: -3
# -3 = 0b1111 1101

# 123 = 0b0111 1011
~123
# Out: -124
# -124 = 0b1000 0100
```

Note, the overall effect of this operation when applied to positive numbers can be summarized:

> `~n -> -|n+1|`

And then, when applied to negative numbers, the corresponding effect is:

> `~-n -> |n-1|`

The following examples illustrate this last rule...

```
# -0 = 0b0000 0000
~-0
# Out: -1
# -1 = 0b1111 1111
# 0 is the obvious exception to this rule, as -0 == 0 always

# -1 = 0b1000 0001
~-1
# Out: 0
# 0 = 0b0000 0000

# -2 = 0b1111 1110
~-2
# Out: 1
# 1 = 0b0000 0001

# -123 = 0b1111 1011
~-123
# Out: 122
# 122 = 0b0111 1010
```

Section 10.2: Bitwise XOR (Exclusive OR)

The `^` operator will perform a binary **XOR** in which a binary `1` is copied if and only if it is the value of exactly **one** operand. Another way of stating this is that the result is `1` only if the operands are different. Examples include:

```
# 0 ^ 0 = 0
# 0 ^ 1 = 1
# 1 ^ 0 = 1
# 1 ^ 1 = 0

# 60 = 0b111100
# 30 = 0b011110
60 ^ 30
# Out: 34
# 34 = 0b100010

bin(60 ^ 30)
```

```
# Out: 0b100010
```

Section 10.3: Bitwise AND

The `&` operator will perform a binary **AND**, where a bit is copied if it exists in **both** operands. That means:

```
# 0 & 0 = 0
# 0 & 1 = 0
# 1 & 0 = 0
# 1 & 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 & 30
# Out: 28
# 28 = 0b11100

bin(60 & 30)
# Out: 0b11100
```

Section 10.4: Bitwise OR

The `|` operator will perform a binary "or," where a bit is copied if it exists in either operand. That means:

```
# 0 | 0 = 0
# 0 | 1 = 1
# 1 | 0 = 1
# 1 | 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 | 30
# Out: 62
# 62 = 0b111110

bin(60 | 30)
# Out: 0b111110
```

Section 10.5: Bitwise Left Shift

The `<<` operator will perform a bitwise "left shift," where the left operand's value is moved left by the number of bits given by the right operand.

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

Performing a left bit shift of `1` is equivalent to multiplication by `2`:

```
7 << 1
# Out: 14
```

Performing a left bit shift of `n` is equivalent to multiplication by `2**n`:

```
3 << 4
# Out: 48
```

Section 10.6: Bitwise Right Shift

The `>>` operator will perform a bitwise "right shift," where the left operand's value is moved right by the number of bits given by the right operand.

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

Performing a right bit shift of `1` is equivalent to integer division by `2`:

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

Performing a right bit shift of `n` is equivalent to integer division by `2**n`:

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

Section 10.7: Inplace Operations

All of the Bitwise operators (except ~) have their own in place versions

```
a = 0b001
a &= 0b010
# a = 0b000

a = 0b001
a |= 0b010
# a = 0b011

a = 0b001
a <<= 2
# a = 0b100

a = 0b100
a >>= 2
# a = 0b001

a = 0b101
a ^= 0b011
# a = 0b110
```

Chapter 11: Boolean Operators

Section 11.1: `and` and `or` are not guaranteed to return a boolean

When you use `or`, it will either return the first value in the expression if it's true, else it will blindly return the second value. I.e. `or` is equivalent to:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

For `and`, it will return its first value if it's false, else it returns the last value:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

Section 11.2: A simple example

In Python you can compare a single element using two binary operators--one on either side:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

In many (most?) programming languages, this would be evaluated in a way contrary to regular math: `(3.14 < x) < 3.142`, but in Python it is treated like `3.14 < x and x < 3.142`, just like most non-programmers would expect.

Section 11.3: Short-circuit evaluation

Python minimally evaluates Boolean expressions.

```
>>> def true_func():
...     print("true_func()")
...     return True
...
>>> def false_func():
...     print("false_func()")
...     return False
...
>>> true_func() or false_func()
true_func()
True
>>> false_func() or true_func()
false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
```

```
false_func()
False
```

Section 11.4: and

Evaluates to the second argument if and only if both of the arguments are truthy. Otherwise evaluates to the first falsey argument.

```
x = True
y = True
z = x and y # z = True

x = True
y = False
z = x and y # z = False

x = False
y = True
z = x and y # z = False

x = False
y = False
z = x and y # z = False

x = 1
y = 1
z = x and y # z = y, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 0
y = 1
z = x and y # z = x, so z = 0 (see above)

x = 1
y = 0
z = x and y # z = y, so z = 0 (see above)

x = 0
y = 0
z = x and y # z = x, so z = 0 (see above)
```

The `1`'s in the above example can be changed to any truthy value, and the `0`'s can be changed to any falsey value.

Section 11.5: or

Evaluates to the first truthy argument if either one of the arguments is truthy. If both arguments are falsey, evaluates to the second argument.

```
x = True
y = True
z = x or y # z = True

x = True
y = False
z = x or y # z = True

x = False
y = True
z = x or y # z = True
```

```
x = False
y = False
z = x or y # z = False

x = 1
y = 1
z = x or y # z = x, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 1
y = 0
z = x or y # z = x, so z = 1 (see above)

x = 0
y = 1
z = x or y # z = y, so z = 1 (see above)

x = 0
y = 0
z = x or y # z = y, so z = 0 (see above)
```

The `1`'s in the above example can be changed to any truthy value, and the `0`'s can be changed to any falsey value.

Section 11.6: not

It returns the opposite of the following statement:

```
x = True
y = not x # y = False

x = False
y = not x # y = True
```

Chapter 12: Operator Precedence

Python operators have a set **order of precedence**, which determines what operators are evaluated first in a potentially ambiguous expression. For instance, in the expression 3 * 2 + 7, first 3 is multiplied by 2, and then the result is added to 7, yielding 13. The expression is not evaluated the other way around, because * has a higher precedence than +.

Below is a list of operators by precedence, and a brief description of what they (usually) do.

Section 12.1: Simple Operator Precedence Examples in python

Python follows PEMDAS rule. PEMDAS stands for Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction.

Example:

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c)  # parentheses
256
>>> a * b ** c  # exponent: same as `a * (b ** c)`
7776
>>> a + b * c / d  # multiplication / division: same as `a + (b * c / d)`
4.142857142857142
```

Extras: mathematical rules hold, but not always:

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Chapter 13: Variable Scope and Binding

Section 13.1: Nonlocal Variables

Python 3.x `Version ≥ 3.0`

Python 3 added a new keyword called **nonlocal**. The nonlocal keyword adds a scope override to the inner scope. You can read all about it in PEP 3104. This is best illustrated with a couple of code examples. One of the most common examples is to create function that can increment:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

If you try running this code, you will receive an **UnboundLocalError** because the **num** variable is referenced before it is assigned in the innermost function. Let's add nonlocal to the mix:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Basically `nonlocal` will allow you to assign to variables in an outer scope, but not a global scope. So you can't use `nonlocal` in our `counter` function because then it would try to assign to a global scope. Give it a try and you will quickly get a `SyntaxError`. Instead you must use `nonlocal` in a nested function.

(Note that the functionality presented here is better implemented using generators.)

Section 13.2: Global Variables

In Python, variables inside functions are considered local if and only if they appear in the left side of an assignment statement, or some other binding occurrence; otherwise such a binding is looked up in enclosing functions, up to the global scope. This is true even if the assignment statement is never executed.

```
x = 'Hi'

def read_x():
    print(x)   # x is just referenced, therefore assumed global

read_x()       # prints Hi

def read_y():
    print(y)   # here y is just referenced, therefore assumed global
```

```
read_y()       # NameError: global name 'y' is not defined

def read_y():
    y = 'Hey'  # y appears in an assignment, therefore it's local
    print(y)   # will find the local y

read_y()       # prints Hey

def read_x_local_fail():
    if False:
        x = 'Hey'  # x appears in an assignment, therefore it's local
    print(x)   # will look for the _local_ z, which is not assigned, and will not be found

read_x_local_fail()   # UnboundLocalError: local variable 'x' referenced before assignment
```

Normally, an assignment inside a scope will shadow any outer variables of the same name:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Declaring a name `global` means that, for the rest of the scope, any assignments to the name will happen at the module's top level:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

The `global` keyword means that assignments will happen at the module's top level, not at the program's top level. Other modules will still need the usual dotted access to variables within the module.

To summarize: in order to know whether a variable x is local to a function, you should read the *entire* function:

1. if you've found `global x`, then x is a **global** variable
2. If you've found `nonlocal x`, then x belongs to an enclosing function, and is neither local nor global
3. If you've found `x = 5` or `for x in range(3)` or some other binding, then x is a **local** variable
4. Otherwise x belongs to some enclosing scope (function scope, global scope, or builtins)

Section 13.3: Local Variables

If a name is *bound* inside a function, it is by default accessible only within the function:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

Control flow constructs have no impact on the scope (with the exception of `except`), but accessing variable that was not assigned yet is an error:

```
def foo():
    if True:
        a = 5
    print(a) # ok

b = 3
def bar():
    if False:
        b = 5
    print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

Common binding operations are assignments, `for` loops, and augmented assignments such as `a += 5`

Section 13.4: The del command

This command has several related yet distinct forms.

`del v`

If `v` is a variable, the command `del v` removes the variable from its scope. For example:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Note that `del` is a *binding occurrence*, which means that unless explicitly stated otherwise (using `nonlocal` or `global`), `del v` will make `v` local to the current scope. If you intend to delete `v` in an outer scope, use `nonlocal v` or `global v` in the same scope of the `del v` statement.

In all the following, the intention of a command is a default behavior but is not enforced by the language. A class might be written in a way that invalidates this intention.

`del v.name`

This command triggers a call to `v.__delattr__(name)`.

The intention is to make the attribute `name` unavailable. For example:

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # out: 7
del a.x
print(a.x) # error: AttributeError: 'A' object has no attribute 'x'
```

`del v[item]`

This command triggers a call to `v.__delitem__(item)`.

The intention is that `item` will not belong in the mapping implemented by the object `v`. For example:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

`del v[a:b]`

This actually calls `v.__delslice__(a, b)`.

The intention is similar to the one described above, but with slices - ranges of items instead of a single item. For example:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

See also Garbage Collection#The del command.

Section 13.5: Functions skip class scope when looking up names

Classes have a local scope during definition, but functions inside the class do not use that scope when looking up names. Because lambdas are functions, and comprehensions are implemented using function scope, this can lead to some surprising behavior.

```
a = 'global'

class Fred:
    a = 'class'  # class scope
    b = (a for i in range(10))  # function scope
    c = [a for i in range(10)]  # function scope
    d = a  # class scope
    e = lambda: a  # function scope
    f = lambda a=a: a  # default argument uses class scope

    @staticmethod  # or @classmethod, or regular instance method
    def g():  # function scope
        return a

print(Fred.a)  # class
print(next(Fred.b))  # global
print(Fred.c[0])  # class in Python 2, global in Python 3
print(Fred.d)  # class
print(Fred.e())  # global
print(Fred.f())  # class
print(Fred.g()) # global
```

Users unfamiliar with how this scope works might expect `b`, `c`, and `e` to print `class`.

From PEP 227:

> Names in class scope are not accessible. Names are resolved in the innermost enclosing function scope. If a class definition occurs in a chain of nested scopes, the resolution process skips class definitions.

From Python's documentation on naming and binding:

> The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes comprehensions and generator expressions since they are implemented using a function scope. This means that the following will fail:
>
> ```
> class A:
>     a = 42
>     b = list(a + i for i in range(10))
> ```

This example uses references from this answer by Martijn Pieters, which contains more in depth analysis of this behavior.

Section 13.6: Local vs Global Scope

What are local and global scope?

All Python variables which are accessible at some point in code are either in *local scope* or in *global scope*.

The explanation is that local scope includes all variables defined in the current function and global scope includes variables defined outside of the current function.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

One can inspect which variables are in which scope. Built-in functions `locals()` and `globals()` return the whole scopes as dictionaries.

```
foo = 1

def func():
    bar = 2
    print(globals().keys())  # prints all variable names in global scope
    print(locals().keys())  # prints all variable names in local scope
```

What happens with name clashes?

```
foo = 1

def func():
    foo = 2  # creates a new variable foo in local scope, global foo is not affected

    print(foo)  # prints 2

    # global variable foo still exists, unchanged:
    print(globals()['foo'])  # prints 1
    print(locals()['foo'])  # prints 2
```

To modify a global variable, use keyword `global`:

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

The scope is defined for the whole body of the function!

What it means is that a variable will never be global for a half of the function and local afterwards, or vice-versa.

```
foo = 1

def func():
    # This function has a local variable foo, because it is defined down below.
    # So, foo is local from this point. Global foo is hidden.

    print(foo) # raises UnboundLocalError, because local foo is not yet initialized
    foo = 7
    print(foo)
```

Likewise, the opposite:

```
foo = 1

def func():
    # In this function, foo is a global variable from the beginning

    foo = 7  # global foo is modified

    print(foo)  # 7
    print(globals()['foo'])  # 7

    global foo  # this could be anywhere within the function
    print(foo)  # 7
```

Functions within functions

There may be many levels of functions nested within functions, but within any one function there is only one local scope for that function and the global scope. There are no intermediate scopes.

```
foo = 1

def f1():
    bar = 1

    def f2():
        baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys())  # ['baz']
        print('bar' in locals())  # False
        print('bar' in globals())  # False

    def f3():
        baz = 3
        print(bar)  # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False

    def f4():
        bar = 4  # a new local bar which hides bar from local scope of f1
        baz = 4
        print(bar)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
```

```
        print('bar' in globals())  # False
```

`global` vs `nonlocal` (Python 3 only)

Both these keywords are used to gain write access to variables which are not local to the current functions.

The `global` keyword declares that a name should be treated as a global variable.

```
foo = 0  # global foo

def f1():
    foo = 1  # a new foo local in f1

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            foo = 3  # a new foo local in f3
            print(foo)  # 3
            foo = 30  # modifies local foo in f3 only

        def f4():
            global foo
            print(foo)  # 0
            foo = 100  # modifies global foo
```

On the other hand, `nonlocal` (see Nonlocal Variables), available in Python 3, takes a *local* variable from an enclosing scope into the local scope of current function.

From the Python documentation on `nonlocal`:

> The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

Python 3.x `Version ≥ 3.0`

```
def f1():

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            nonlocal foo  # foo from f2, which is the nearest enclosing scope
            print(foo)  # 2
            foo = 20  # modifies foo from f2!
```

Section 13.7: Binding Occurrence

```
x = 5
x += 7
for x in iterable: pass
```

Each of the above statements is a *binding occurrence* - `x` become bound to the object denoted by `5`. If this statement appears inside a function, then `x` will be function-local by default. See the "Syntax" section for a list of binding statements.

Chapter 14: Conditionals

Conditional expressions, involving keywords such as if, elif, and else, provide Python programs with the ability to perform different actions depending on a boolean condition: True or False. This section covers the use of Python conditionals, boolean logic, and ternary statements.

Section 14.1: Conditional Expression (or "The Ternary Operator")

The ternary operator is used for inline conditional expressions. It is best used in simple, concise operations that are easily read.

- The order of the arguments is different from many other languages (such as C, Ruby, Java, etc.), which may lead to bugs when people unfamiliar with Python's "surprising" behaviour use it (they may reverse the order).
- Some find it "unwieldy", since it goes contrary to the normal flow of thought (thinking of the condition first and then the effects).

```
n = 5

"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

The result of this expression will be as it is read in English - if the conditional expression is True, then it will evaluate to the expression on the left side, otherwise, the right side.

Ternary operations can also be nested, as here:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

They also provide a method of including conditionals in lambda functions.

Section 14.2: if, elif, and else

In Python you can define a series of conditionals using `if` for the first one, `elif` for the rest, up until the final (optional) `else` for anything not caught by the other conditionals.

```
number = 5

if number > 2:
    print("Number is bigger than 2.")
elif number < 2:  # Optional clause (you can have multiple elifs)
    print("Number is smaller than 2.")
else:  # Optional clause (you can only have one else)
    print("Number is 2.")
```

Outputs `Number is bigger than 2`

Using `else if` instead of `elif` will trigger a syntax error and is not allowed.

Section 14.3: Truth Values

The following values are considered falsey, in that they evaluate to `False` when applied to a boolean operator.

- None
- False
- `0`, or any numerical value equivalent to zero, for example `0L`, `0.0`, `0j`
- Empty sequences: `''`, `""`, `()`, `[]`
- Empty mappings: `{}`
- User-defined types where the `__bool__` or `__len__` methods return 0 or `False`

All other values in Python evaluate to `True`.

Note: A common mistake is to simply check for the Falseness of an operation which returns different Falsey values where the difference matters. For example, using `if foo()` rather than the more explicit `if foo() is None`

Section 14.4: Boolean Logic Expressions

Boolean logic expressions, in addition to evaluating to `True` or `False`, return the *value* that was interpreted as `True` or `False`. It is Pythonic way to represent logic that might otherwise require an if-else test.

And operator

The `and` operator evaluates all expressions and returns the last expression if all expressions evaluate to `True`. Otherwise it returns the first value that evaluates to `False`:

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

Or operator

The `or` operator evaluates the expressions left to right and returns the first value that evaluates to `True` or the last value (if none are `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

Lazy evaluation

When you use this approach, remember that the evaluation is lazy. Expressions that are not required to be evaluated to determine the result are not evaluated. For example:

```
>>> def print_me():
        print('I am here!')
>>> 0 and print_me()
0
```

In the above example, `print_me` is never executed because Python can determine the entire expression is `False` when it encounters the `0` (`False`). Keep this in mind if `print_me` needs to execute to serve your program logic.

Testing for multiple conditions

A common mistake when checking for multiple conditions is to apply the logic incorrectly.

This example is trying to check if two variables are each greater than 2. The statement is evaluated as - `if (a) and (b > 2)`. This produces an unexpected result because `bool(a)` evaluates as `True` when `a` is not zero.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Each variable needs to be compared separately.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Another, similar, mistake is made when checking if a variable is one of multiple values. The statement in this example is evaluated as - `if (a == 3) or (4) or (6)`. This produces an unexpected result because `bool(4)` and `bool(6)` each evaluate to `True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')

yes
```

Again each comparison must be made separately

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Using the in operator is the canonical way to write this.

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

Section 14.5: Using the cmp function to get the comparison result of two objects

Python 2 includes a `cmp` function which allows you to determine if one object is less than, equal to, or greater than another object. This function can be used to pick a choice out of a list based on one of those three options.

Suppose you need to print `'greater than'` if `x > y`, `'less than'` if `x < y` and `'equal'` if `x == y`.

```
['equal', 'greater than', 'less than', ][cmp(x,y)]

# x,y = 1,1 output: 'equal'
# x,y = 1,2 output: 'less than'
# x,y = 2,1 output: 'greater than'
```

`cmp(x,y)` returns the following values

Comparison	Result
x < y	-1
x == y	0
x > y	1

This function is removed on Python 3. You can use the `cmp_to_key(func)` helper function located in `functools` in Python 3 to convert old comparison functions to key functions.

Section 14.6: Else statement

```
if condition:
    body
else:
    body
```

The else statement will execute it's body only if preceding conditional statements all evaluate to False.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."

# Output: It is true!

if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

Section 14.7: Testing if an object is None and assigning it

You'll often want to assign something to an object if it is `None`, indicating it has not been assigned. We'll use `aDate`.

The simplest way to do this is to use the `is None` test.

```
if aDate is None:
    aDate=datetime.date.today()
```

(Note that it is more Pythonic to say `is None` instead of `== None`.)

But this can be optimized slightly by exploiting the notion that `not None` will evaluate to `True` in a boolean expression. The following code is equivalent:

```
if not aDate:
    aDate=datetime.date.today()
```

But there is a more Pythonic way. The following code is also equivalent:

```
aDate=aDate or datetime.date.today()
```

This does a Short Circuit evaluation. If `aDate` is initialized and is `not None`, then it gets assigned to itself with no net effect. If it `is None`, then the `datetime.date.today()` gets assigned to `aDate`.

Section 14.8: If statement

```
if condition:
    body
```

The `if` statements checks the condition. If it evaluates to `True`, it executes the body of the `if` statement. If it evaluates to `False`, it skips the body.

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

The condition can be any valid expression:

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Chapter 15: Comparisons

Parameter	Details
x	First item to be compared
y	Second item to be compared

Section 15.1: Chain Comparisons

You can compare multiple items with multiple comparison operators with chain comparison. For example

```
x > y > z
```

is just a short form of:

```
x > y and y > z
```

This will evaluate to `True` only if both comparisons are `True`.

The general form is

```
a OP b OP c OP d ...
```

Where `OP` represents one of the multiple comparison operations you can use, and the letters represent arbitrary valid expressions.

> Note that `0 != 1 != 0` evaluates to `True`, even though `0 != 0` is `False`. Unlike the common mathematical notation in which `x != y != z` means that `x`, `y` and `z` have different values. Chaining `==` operations has the natural meaning in most cases, since equality is generally transitive.

Style

There is no theoretical limit on how many items and comparison operations you use as long you have proper syntax:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

The above returns `True` if each comparison returns `True`. However, using convoluted chaining is not a good style. A good chaining will be "directional", not more complicated than

```
1 > x > -4 > y != 8
```

Side effects

As soon as one comparison returns `False`, the expression evaluates immediately to `False`, skipping all remaining comparisons.

Note that the expression `exp` in `a > exp > b` will be evaluated only once, whereas in the case of

```
a > exp and exp > b
```

`exp` will be computed twice if `a > exp` is true.

Section 15.2: Comparison by `is` vs `==`

A common pitfall is confusing the equality comparison operators `is` and `==`.

`a == b` compares the value of `a` and `b`.

`a is b` will compare the *identities* of `a` and `b`.

To illustrate:

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # returns True
a is b # returns False

a = [1, 2, 3, 4, 5]
b = a      # b references a
a == b     # True
a is b     # True
b = a[:]   # b now references a copy of a
a == b     # True
a is b     # False [!!]
```

Basically, `is` can be thought of as shorthand for `id(a) == id(b)`.

Beyond this, there are quirks of the run-time environment that further complicate things. Short strings and small integers will return `True` when compared with `is`, due to the Python machine attempting to use less memory for identical objects.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

But longer strings and larger integers will be stored separately.

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

You should use `is` to test for `None`:

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

A use of `is` is to test for a “sentinel” (i.e. a unique object).

```
sentinel = object()
def myfunc(var=sentinel):
```

```
    if var is sentinel:
        # value wasn't provided
        pass
    else:
        # value was provided
        pass
```

Section 15.3: Greater than or less than

```
x > y
x < y
```

These operators compare two types of values, they're the less than and greater than operators. For numbers this simply compares the numerical values to see which is larger:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

For strings they will compare lexicographically, which is similar to alphabetical order but not quite the same.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

In these comparisons, lowercase letters are considered 'greater than' uppercase, which is why `"gamma" < "OMEGA"` is false. If they were all uppercase it would return the expected alphabetical ordering result:

```
"GAMMA" < "OMEGA"
# True
```

Each type defines it's calculation with the `<` and `>` operators differently, so you should investigate what the operators mean with a given type before using it.

Section 15.4: Not equal to

```
x != y
```

This returns `True` if `x` and `y` are not equal and otherwise returns `False`.

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Section 15.5: Equal To

```
x == y
```

This expression evaluates if `x` and `y` are the same value and returns the result as a boolean value. Generally both type and value need to match, so the int `12` is not the same as the string `'12'`.

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
'12' == 12
# False
```

Note that each type has to define a function that will be used to evaluate if two values are the same. For builtin types these functions behave as you'd expect, and just evaluate things based on being the same value. However custom types could define equality testing as whatever they'd like, including always returning `True` or always returning `False`.

Section 15.6: Comparing Objects

In order to compare the equality of custom classes, you can override `==` and `!=` by defining `__eq__` and `__ne__` methods. You can also override `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`), and `__ge__` (`>`). Note that you only need to override two comparison methods, and Python can handle the rest (`==` is the same as `not <` and `not >`, etc.)

```
class Foo(object):
    def __init__(self, item):
        self.my_item = item
    def __eq__(self, other):
        return self.my_item == other.my_item

a = Foo(5)
b = Foo(5)
a == b     # True
a != b     # False
a is b     # False
```

Note that this simple comparison assumes that `other` (the object being compared to) is the same object type. Comparing to another type will throw an error:

```
class Bar(object):
    def __init__(self, item):
        self.other_item = item
    def __eq__(self, other):
        return self.other_item == other.other_item
    def __ne__(self, other):
        return self.other_item != other.other_item

c = Bar(5)
a == c    # throws AttributeError: 'Foo' object has no attribute 'other_item'
```

Checking `isinstance()` or similar will help prevent this (if desired).

Chapter 16: Loops

Parameter	Details
boolean expression	expression that can be evaluated in a boolean context, e.g. `x < 10`
variable	variable name for the current element from the `iterable`
iterable	anything that implements iterations

As one of the most basic functions in programming, loops are an important piece to nearly every programming language. Loops enable developers to set certain portions of their code to repeat through a number of loops which are referred to as iterations. This topic covers using multiple types of loops and applications of loops in Python.

Section 16.1: Break and Continue in Loops

`break` statement

When a `break` statement executes inside a loop, control flow "breaks" out of the loop immediately:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

The loop conditional will not be evaluated after the `break` statement is executed. Note that `break` statements are only allowed *inside loops*, syntactically. A `break` statement inside a function cannot be used to terminate loops that called that function.

Executing the following prints every digit until number `4` when the `break` statement is met and the loop stops:

```
0
1
2
3
4
Breaking from loop
```

`break` statements can also be used inside `for` loops, the other looping construct provided by Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

Executing this loop now prints:

```
0
1
2
```

Note that 3 and 4 are not printed since the loop has ended.

If a loop has an `else` clause, it does not execute when the loop is terminated through a `break` statement.

`continue` statement

A `continue` statement will skip to the next iteration of the loop bypassing the rest of the current block but continuing the loop. As with `break`, `continue` can only appear inside loops:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)

0
1
3
5
```

Note that `2` and `4` aren't printed, this is because `continue` goes to the next iteration instead of continuing on to `print(i)` when `i == 2` or `i == 4`.

Nested Loops

`break` and `continue` only operate on a single level of loop. The following example will only break out of the inner `for` loop, not the outer `while` loop:

```
while True:
    for i in range(1,5):
        if i == 2:
            break    # Will only break out of the inner loop!
```

Python doesn't have the ability to break out of multiple levels of loop at once -- if this behavior is desired, refactoring one or more loops into a function and replacing `break` with `return` may be the way to go.

Use `return` from within a function as a `break`

The `return` statement exits from a function, without executing the code that comes after it.

If you have a loop inside a function, using `return` from inside that loop is equivalent to having a `break` as the rest of the code of the loop is not executed (*note that any code after the loop is not executed either*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

If you have nested loops, the `return` statement will break all loops:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

will output:

```
1 # 1*1
2 # 1*2
3 # 1*3
4 # 1*4
2 # 2*1
4 # 2*2
# return because 2*3 = 6, the remaining iterations of both loops are not executed
```

Section 16.2: For loops

`for` loops iterate over a collection of items, such as `list` or `dict`, and run a block of code with each element from the collection.

```
for i in [0, 1, 2, 3, 4]:
    print(i)
```

The above `for` loop iterates over a list of numbers.

Each iteration sets the value of `i` to the next element of the list. So first it will be `0`, then `1`, then `2`, etc. The output will be as follow:

```
0
1
2
3
4
```

`range` is a function that returns a series of numbers under an iterable form, thus it can be used in `for` loops:

```
for i in range(5):
    print(i)
```

gives the exact same result as the first `for` loop. Note that `5` is not printed as the range here is the first five numbers counting from `0`.

Iterable objects and iterators

`for` loop can iterate on any iterable object which is an object which defines a `__getitem__` or a `__iter__` function. The `__iter__` function returns an iterator, which is an object with a `next` function that is used to access the next element of the iterable.

Section 16.3: Iterating over lists

To iterate through a list you can use `for`:

```
for x in ['one', 'two', 'three', 'four']:
    print(x)
```

This will print out the elements of the list:

```
one
two
three
four
```

The `range` function generates numbers which are also often used in a for loop.

```
for x in range(1, 6):
    print(x)
```

The result will be a special range sequence type in python >=3 and a list in python <=2. Both can be looped through using the for loop.

```
1
2
3
4
5
```

If you want to loop though both the elements of a list *and* have an index for the elements as well, you can use Python's `enumerate` function:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

`enumerate` will generate tuples, which are unpacked into `index` (an integer) and `item` (the actual value from the list). The above loop will print

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

Iterate over a list with value manipulation using `map` and `lambda`, i.e. apply lambda function on each element in the list:

```
x = map(lambda e :  e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Output:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: in Python 3.x `map` returns an iterator instead of a list so you in case you need a list you have to cast the result `print(list(x))`

Section 16.4: Loops with an "else" clause

The `for` and `while` compound statements (loops) can optionally have an `else` clause (in practice, this usage is fairly rare).

The `else` clause only executes after a `for` loop terminates by iterating to completion, or after a `while` loop terminates by its conditional expression becoming false.

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
```

```
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

output:

```
0
1
2
done
```

The `else` clause does *not* execute if the loop terminates some other way (through a `break` statement or by raising an exception):

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

output:

```
0
1
```

Most other programming languages lack this optional `else` clause of loops. The use of the keyword `else` in particular is often considered confusing.

The original concept for such a clause dates back to Donald Knuth and the meaning of the `else` keyword becomes clear if we rewrite a loop in terms of `if` statements and `goto` statements from earlier days before structured programming or from a lower-level assembly language.

For example:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
```

is equivalent to:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
```

```
<<end>>:
```

These remain equivalent if we attach an `else` clause to each of them.

For example:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

is equivalent to:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

A `for` loop with an `else` clause can be understood the same way. Conceptually, there is a loop condition that remains True as long as the iterable object or sequence still has some remaining elements.

Why would one use this strange construct?

The main use case for the `for...else` construct is a concise implementation of search as for instance:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

To make the `else` in this construct less confusing one can think of it as "*if not break*" or "*if not found*".

Some discussions on this can be found in [Python-ideas] Summary of for...else threads, Why does python use 'else' after for and while loops? , and Else Clauses on Loop Statements

Section 16.5: The Pass Statement

`pass` is a null statement for when a statement is required by Python syntax (such as within the body of a `for` or `while` loop), but no action is required or desired by the programmer. This can be useful as a placeholder for code that is yet to be written.

```
for x in range(10):
```

```
    pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

In this example, nothing will happen. The `for` loop will complete without error, but no commands or code will be actioned. `pass` allows us to run our code successfully without having all commands and action fully implemented.

Similarly, `pass` can be used in `while` loops, as well as in selections and function definitions etc.

```
while x == y:
    pass
```

Section 16.6: Iterating over dictionaries

Considering the following dictionary:

```
d = {"a": 1, "b": 2, "c": 3}
```

To iterate through its keys, you can use:

```
for key in d:
    print(key)
```

Output:

```
"a"
"b"
"c"
```

This is equivalent to:

```
for key in d.keys():
    print(key)
```

or in Python 2:

```
for key in d.iterkeys():
    print(key)
```

To iterate through its values, use:

```
for value in d.values():
    print(value)
```

Output:

```
1
2
3
```

To iterate through its keys and values, use:

```
for key, value in d.items():
    print(key, "::", value)
```

Output:

```
a :: 1
b :: 2
c :: 3
```

Note that in Python 2, `.keys()`, `.values()` and `.items()` return a `list` object. If you simply need to iterate through the result, you can use the equivalent `.iterkeys()`, `.itervalues()` and `.iteritems()`.

The difference between `.keys()` and `.iterkeys()`, `.values()` and `.itervalues()`, `.items()` and `.iteritems()` is that the `iter*` methods are generators. Thus, the elements within the dictionary are yielded one by one as they are evaluated. When a `list` object is returned, all of the elements are packed into a list and then returned for further evaluation.

> Note also that in Python 3, Order of items printed in the above manner does not follow any order.

Section 16.7: The "half loop" do-while

Unlike other languages, Python doesn't have a do-until or a do-while construct (this will allow code to be executed once before the condition is tested). However, you can combine a `while True` with a `break` to achieve the same purpose.

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

This will print:

```
9
8
7
6
Done.
```

Section 16.8: Looping and Unpacking

If you want to loop over a list of tuples for example:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

instead of doing something like this:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

or something like this:

```
for item in collection:
```

```
    i1, i2, i3 = item
    # logic
```

You can simply do this:

```
for i1, i2, i3 in collection:
    # logic
```

This will also work for *most* types of iterables, not just tuples.

Section 16.9: Iterating different portion of a list with different step size

Suppose you have a long list of elements and you are only interested in every other element of the list. Perhaps you only want to examine the first or last elements, or a specific range of entries in your list. Python has strong indexing built-in capabilities. Here are some examples of how to achieve these scenarios.

Here's a simple list that will be used throughout the examples:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

Iteration over the whole list

To iterate over each element in the list, a `for` loop like below can be used:

```
for s in lst:
    print s[:1] # print the first letter
```

The `for` loop assigns s for each element of `lst`. This will print:

```
a
b
c
d
e
```

Often you need both the element and the index of that element. The `enumerate` keyword performs that task.

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

The index `idx` will start with zero and increment for each iteration, while the `s` will contain the element being processed. The previous snippet will output:

```
alpha has an index of 0
bravo has an index of 1
charlie has an index of 2
delta has an index of 3
echo has an index of 4
```

Iterate over sub-list

If we want to iterate over a range (remembering that Python uses zero-based indexing), use the `range` keyword.

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

This would output:

```
lst at 2 contains charlie
lst at 3 contains delta
```

The list may also be sliced. The following slice notation goes from element at index 1 to the end with a step of 2. The two `for` loops give the same result.

```
for s in lst[1::2]:
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

The above snippet outputs:

```
bravo
delta
```

Indexing and slicing is a topic of its own.

Section 16.10: While Loop

A `while` loop will cause the loop statements to be executed until the loop condition is falsey. The following code will execute the loop statements a total of 4 times.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

While the above loop can easily be translated into a more elegant `for` loop, `while` loops are useful for checking if some condition has been met. The following loop will continue to execute until `myObject` is ready.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

`while` loops can also run without a condition by using numbers (complex or real) or `True`:

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # You can also replace complex_num with any number, True or a value of any
type
    print(complex_num)   # Prints 1j forever
```

If the condition is always true the while loop will run forever (infinite loop) if it is not terminated by a break or return statement or an exception.

```
while True:
    print "Infinite loop"
```

```
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

Chapter 17: Arrays

Parameter	Details
`b`	Represents signed integer of size 1 byte
`B`	Represents unsigned integer of size 1 byte
`c`	Represents character of size 1 byte
`u`	Represents unicode character of size 2 bytes
`h`	Represents signed integer of size 2 bytes
`H`	Represents unsigned integer of size 2 bytes
`i`	Represents signed integer of size 2 bytes
`I`	Represents unsigned integer of size 2 bytes
`w`	Represents unicode character of size 4 bytes
`l`	Represents signed integer of size 4 bytes
`L`	Represents unsigned integer of size 4 bytes
`f`	Represents floating point of size 4 bytes
`d`	Represents floating point of size 8 bytes

"Arrays" in Python are not the arrays in conventional programming languages like C and Java, but closer to lists. A list can be a collection of either homogeneous or heterogeneous elements, and may contain ints, strings or other lists.

Section 17.1: Access individual elements through indexes

Individual elements can be accessed through indexes. Python arrays are zero-indexed. Here is an example:

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

Section 17.2: Basic Introduction to Arrays

An array is a data structure that stores values of same data type. In Python, this is the main difference between arrays and lists.

While python lists can contain values corresponding to different data types, arrays in python can only contain values corresponding to same data type. In this tutorial, we will understand the Python arrays with few examples.

If you are new to Python, get started with the Python Introduction article.

To use arrays in python language, you need to import the standard `array` module. This is because array is not a fundamental data type like strings, integer etc. Here is how you can import `array` module in python :

```
from array import *
```

Once you have imported the `array` module, you can declare an array. Here is how you do it:

```
arrayIdentifierName = array(typecode, [Initializers])
```

In the declaration above, `arrayIdentifierName` is the name of array, `typecode` lets python know the type of array and `Initializers` are the values with which array is initialized.

Typecodes are the codes that are used to define the type of array values or the type of array. The table in the parameters section shows the possible values you can use when declaring an array and it's type.

Here is a real world example of python array declaration :

```
my_array = array('i',[1,2,3,4])
```

In the example above, typecode used is `i`. This typecode represents signed integer whose size is 2 bytes.

Here is a simple example of an array containing 5 integers

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

Section 17.3: Append any value to the array using append() method

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Note that the value `6` was appended to the existing array values.

Section 17.4: Insert value in an array using insert() method

We can use the `insert()` method to insert a value at any index of the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

In the above example, the value 0 was inserted at index 0. Note that the first argument is the index while second argument is the value.

Section 17.5: Extend python array using extend() method

A python array can be extended with more than one value using `extend()` method. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

We see that the array my_array was extended with values from `my_extnd_array`.

Section 17.6: Add items from list into array using fromlist() method

Here is an example:

```
my_array = array('i', [1,2,3,4,5])
c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

So we see that the values 11,12 and 13 were added from list `c` to `my_array`.

Section 17.7: Remove any array element using remove() method

Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

We see that the element 4 was removed from the array.

Section 17.8: Remove last array element using pop() method

`pop` removes the last element from the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

So we see that the last element (5) was popped out of array.

Section 17.9: Fetch any element through its index using index() method

`index()` returns first index of the matching value. Remember that arrays are zero-indexed.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

Note in that second example that only one index was returned, even though the value exists twice in the array

Section 17.10: Reverse a python array using reverse() method

The `reverse()` method does what the name says it will do - reverses the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Section 17.11: Get array buffer information through buffer_info() method

This method provides you the array buffer start address in memory and number of elements in array. Here is an example:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

Section 17.12: Check for number of occurrences of an element using count() method

`count()` will return the number of times and element appears in an array. In the following example we see that the value `3` occurs twice.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

Section 17.13: Convert array to string using tostring() method

`tostring()` converts the array to a string.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

Section 17.14: Convert array to a python list with same elements using tolist() method

When you need a Python `list` object, you can utilize the `tolist()` method to convert your array to a list.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

Section 17.15: Append a string to char array using fromstring() method

You are able to append a string to a character array using `fromstring()`

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Chapter 18: Multidimensional arrays

Section 18.1: Lists in lists

A good way to visualize a 2d array is as a list of lists. Something like this:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

here the outer list `lst` has three things in it. each of those things is another list: The first one is: `[1,2,3]`, the second one is: `[4,5,6]` and the third one is: `[7,8,9]`. You can access these lists the same way you would access another other element of a list, like this:

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

You can then access the different elements in each of those lists the same way:

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

Here the first number inside the `[]` brackets means get the list in that position. In the above example we used the number `0` to mean get the list in the 0th position which is `[1,2,3]`. The second set of `[]` brackets means get the item in that position from the inner list. In this case we used both `0` and `1` the 0th position in the list we got is the number `1` and in the 1st position it is `2`

You can also set values inside these lists the same way:

```
lst[0]=[10,11,12]
```

Now the list is `[[10,11,12],[4,5,6],[7,8,9]]`. In this example we changed the whole first list to be a completely new list.

```
lst[1][2]=15
```

Now the list is `[[10,11,12],[4,5,15],[7,8,9]]`. In this example we changed a single element inside of one of the inner lists. First we went into the list at position 1 and changed the element within it at position 2, which was 6 now it's 15.

Section 18.2: Lists in lists in lists in..

This behaviour can be extended. Here is a 3-dimensional array:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,
313],[321,322,323],[331,332,333]]]
```

As is probably obvious, this gets a bit hard to read. Use backslashes to break up the different dimensions:

```
[[[111,112,113],[121,122,123],[131,132,133]],\
 [[211,212,213],[221,222,223],[231,232,233]],\
 [[311,312,313],[321,322,323],[331,332,333]]]
```

By nesting the lists like this, you can extend to arbitrarily high dimensions.

Accessing is similar to 2D arrays:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

And editing is also similar:

```
myarray[1]=new_n-1_d_list
myarray[2][1]=new_n-2_d_list
myarray[1][0][2]=new_n-3_d_list #or a single number if you're dealing with 3D arrays
etc.
```

Chapter 19: Dictionary

Parameter	Details
key	The desired key to lookup
value	The value to set or return

Section 19.1: Introduction to Dictionary

A dictionary is an example of a *key value store* also known as *Mapping* in Python. It allows you to store and retrieve elements by referencing a key. As dictionaries are referenced by key, they have very fast lookups. As they are primarily used for referencing items by key, they are not sorted.

creating a dict

Dictionaries can be initiated in many ways:

literal syntax

```
d = {}                          # empty dict
d = {'key': 'value'}            # dict with initial values
```

Python 3.x `Version ≥ 3.5`

```
# Also unpacking one or multiple dictionaries with the literal syntax is possible

# makes a shallow copy of otherdict
d = {**otherdict}
# also updates the shallow copy with the contents of the yetanotherdict.
d = {**otherdict, **yetanotherdict}
```

dict comprehension

```
d = {k:v for k,v in [('key', 'value',)]}
```

see also: Comprehensions

built-in class: `dict()`

```
d = dict()                      # empty dict
d = dict(key='value')           # explicit keyword arguments
d = dict([('key', 'value')])  # passing in a list of key/value pairs
# make a shallow copy of another dict (only possible if keys are only strings!)
d = dict(**otherdict)
```

modifying a dict

To add items to a dictionary, simply create a new key with a value:

```
d['newkey'] = 42
```

It also possible to add `list` and `dictionary` as value:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

To delete an item, delete the key from the dictionary:

```
del d['newkey']
```

Section 19.2: Avoiding KeyError Exceptions

One common pitfall when using dictionaries is to access a non-existent key. This typically results in a `KeyError` exception

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

One way to avoid key errors is to use the `dict.get` method, which allows you to specify a default value to return in the case of an absent key.

```
value = mydict.get(key, default_value)
```

Which returns `mydict[key]` if it exists, but otherwise returns `default_value`. Note that this doesn't add `key` to `mydict`. So if you want to retain that key value pair, you should use `mydict.setdefault(key, default_value)`, which *does* store the key value pair.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

An alternative way to deal with the problem is catching the exception

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

You could also check if the key is `in` the dictionary.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Do note, however, that in multi-threaded environments it is possible for the key to be removed from the dictionary after you check, creating a race condition where the exception can still be thrown.

Another option is to use a subclass of dict, collections.defaultdict, that has a default_factory to create new entries in the dict when given a new_key.

Section 19.3: Iterating Over a Dictionary

If you use a dictionary as an iterator (e.g. in a `for` statement), it traverses the **keys** of the dictionary. For example:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

The same is true when used in a comprehension

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x `Version ≥ 3.0`

The `items()` method can be used to loop over both the **key** and **value** simultaneously:

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

While the `values()` method can be used to iterate over only the values, as would be expected:

```
for key, value in d.values():
    print(key, value)
    # 3
    # 2
    # 1
```

Python 2.x `Version ≥ 2.2`

Here, the methods `keys()`, `values()` and `items()` return lists, and there are the three extra methods `iterkeys()` `itervalues()` and `iteritems()` to return iterators.

Section 19.4: Dictionary with default values

Available in the standard library as `defaultdict`

```
from collections import defaultdict

d = defaultdict(int)
d['key']                        # 0
d['key'] = 5
d['key']                        # 5

d = defaultdict(lambda: 'empty')
d['key']                        # 'empty'
d['key'] = 'full'
d['key']                        # 'full'
```

[*] Alternatively, if you must use the built-in `dict` class, `using dict.setdefault()` will allow you to create a default whenever you access a key that did not exist before:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
```

```
{'Another_key': ['This worked!']}
```

Keep in mind that if you have many values to add, `dict.setdefault()` will create a new instance of the initial value (in this example a [ ]) every time it's called - which may create unnecessary workloads.

[*] *Python Cookbook, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.*

Section 19.5: Merging dictionaries

Consider the following dictionaries:

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
```

Python 3.5+

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

As this example demonstrates, duplicate keys map to their lattermost value (for example "Clifford" overrides "Nemo").

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

With this technique the foremost value takes precedence for a given key rather than the last ("Clifford" is thrown out in favor of "Nemo").

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

This uses the lattermost value, as with the `**`-based technique for merging ("Clifford" overrides "Nemo").

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` uses the latter dict to overwrite the previous one.

Section 19.6: Accessing keys and values

When working with dictionaries, it's often necessary to access all the keys and values in the dictionary, either in a `for` loop, a list comprehension, or just as a plain list.

Given a dictionary like:

```
mydict = {
    'a': '1',
```

```
    'b': '2'
}
```

You can get a list of keys using the `keys()` method:

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

If instead you want a list of values, use the `values()` method:

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

If you want to work with both the key and its corresponding value, you can use the `items()` method:

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

NOTE: Because a `dict` is unsorted, `keys()`, `values()`, and `items()` have no sort order. Use `sort()`, `sorted()`, or an `OrderedDict` if you care about the order that these methods return.

Python 2/3 Difference: In Python 3, these methods return special iterable objects, not lists, and are the equivalent of the Python 2 `iterkeys()`, `itervalues()`, and `iteritems()` methods. These objects can be used like lists for the most part, though there are some differences. See PEP 3106 for more details.

Section 19.7: Accessing values of a dictionary

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

The above code will print `1234`.

The string `"Hello"` in this example is called a *key*. It is used to lookup a value in the `dict` by placing the key in square brackets.

The number `1234` is seen after the respective colon in the `dict` definition. This is called the *value* that `"Hello"` *maps to* in this `dict`.

Looking up a value like this with a key that does not exist will raise a `KeyError` exception, halting execution if uncaught. If we want to access a value without risking a `KeyError`, we can use the `dictionary.get` method. By default if the key does not exist, the method will return `None`. We can pass it a second value to return instead of `None` in the event of a failed lookup.

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

In this example `w` will get the value `None` and `x` will get the value `"nuh-uh"`.

Section 19.8: Creating a dictionary

Rules for creating a dictionary:

- Every key must be **unique** (otherwise it will be overridden)
- Every key must be **hashable** (can use the `hash` function to hash it; otherwise `TypeError` will be thrown)
- There is no particular order for the keys.

```
# Creating and populating it with values
stock = {'eggs': 5, 'milk': 2}

# Or creating an empty dictionary
dictionary = {}

# And populating it after
dictionary['eggs'] = 5
dictionary['milk'] = 2

# Values can also be lists
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}

# Use list.append() method to add new elements to the values list
mydict['a'].append(4)   # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three']}
mydict['b'].append('four')  # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three', 'four']}

# We can also create a dictionary using a list of two-items tuples
iterable = [('eggs', 5), ('milk', 2)]
dictionary = dict(iterables)

# Or using keyword argument:
dictionary = dict(eggs=5, milk=2)

# Another way will be to use the dict.fromkeys:
dictionary = dict.fromkeys((milk, eggs))  # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5))  # => {'milk': 2, 'eggs': 5}
```

Section 19.9: Creating an ordered dictionary

You can create an ordered dictionary which will follow a determined order when iterating over the keys in the dictionary.

Use `OrderedDict` from the `collections` module. This will always return the dictionary elements in the original insertion order when iterated over.

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# Outputs "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

Section 19.10: Unpacking dictionaries using the ** operator

You can use the `**` keyword argument unpacking operator to deliver the key-value pairs in a dictionary into a function's arguments. A simplified example from the official documentation:

```
>>>
```

```
>>> def parrot(voltage, state, action):
...     print("This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)

This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

As of Python 3.5 you can also use this syntax to merge an arbitrary number of `dict` objects.

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
>>> fishdog = {**fish, **dog}
>>> fishdog

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

As this example demonstrates, duplicate keys map to their lattermost value (for example "Clifford" overrides "Nemo").

Section 19.11: The trailing comma

Like lists and tuples, you can include a trailing comma in your dictionary.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 dictates that you should leave a space between the trailing comma and the closing brace.

Section 19.12: The dict() constructor

The `dict()` constructor can be used to create dictionaries from keyword arguments, or from a single iterable of key-value pairs, or from a single dictionary and keyword arguments.

```
dict(a=1, b=2, c=3)                   # {'a': 1, 'b': 2, 'c': 3}
dict([('d', 4), ('e', 5), ('f', 6)])  # {'d': 4, 'e': 5, 'f': 6}
dict([('a', 1)], b=2, c=3)            # {'a': 1, 'b': 2, 'c': 3}
dict({'a' : 1, 'b' : 2}, c=3)         # {'a': 1, 'b': 2, 'c': 3}
```

Section 19.13: Dictionaries Example

Dictionaries map keys to values.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Dictionary values can be accessed by their keys.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

Dictionaries can also be created in a JSON style:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Dictionary values can be iterated over:

```
for key in car:
  print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

Section 19.14: All combinations of dictionary values

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Given a dictionary such as the one shown above, where there is a list representing a set of values to explore for the corresponding key. Suppose you want to explore `"x"="a"` with `"y"=10`, then `"x"="a"` with`"y"=10`, and so on until you have explored all possible combinations.

You can create a list that returns all such combinations of values using the following code.

```
import itertools

options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]}

keys = options.keys()
values = (options[key] for key in keys)
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]
print combinations
```

This gives us the following list stored in the variable `combinations`:

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

Chapter 20: List

The Python **List** is a general data structure widely used in Python programs. They are found in other languages, often referred to as *dynamic arrays*. They are both *mutable* and a *sequence* data type that allows them to be *indexed* and *sliced*. The list can contain different types of objects, including other list objects.

Section 20.1: List methods and supported operators

Starting with a given list `a`:

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` – appends a new element to the end of the list.

```
# Append values 6, 7, and 7 to the list
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]

# Append another list
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

# Append an element of a different type, as list elements do not need to have the same type
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

Note that the `append()` method only appends one new element to the end of the list. If you append a list to another list, the list that you append becomes a single element at the end of the first list.

```
# Appending a list to another list
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# Returns: [8,9]
```

2. `extend(enumerable)` – extends the list by appending elements from another enumerable.

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

# Extend list by appending all elements from b
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

# Extend list with elements from a non-list enumerable:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

Lists can also be concatenated with the `+` operator. Note that this does not modify any of the original lists:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. `index(value, [startIndex])` – gets the index of the first occurrence of the input value. If the input value is not in the list a `ValueError` exception is raised. If a second argument is provided, the search is started at that specified index.

```
a.index(7)
# Returns: 6

a.index(49) # ValueError, because 49 is not in a.

a.index(7, 7)
# Returns: 7

a.index(7, 8) # ValueError, because there is no 7 starting at index 8
```

4. `insert(index, value)` – inserts `value` just before the specified `index`. Thus after the insertion the new element occupies position `index`.

```
a.insert(0, 0)  # insert 0 at position 0
a.insert(2, 5)  # insert 5 at position 2
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. `pop([index])` – removes and returns the item at `index`. With no argument it removes and returns the last element of the list.

```
a.pop(2)
# Returns: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Returns: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# With no argument:
a.pop()
# Returns: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. `remove(value)` – removes the first occurrence of the specified value. If the provided value cannot be found, a `ValueError` is raised.

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
# ValueError, because 10 is not in a
```

7. `reverse()` – reverses the list in-place and returns `None`.

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

   There are also other ways of reversing a list.

8. `count(value)` – counts the number of occurrences of some value in the list.

```
a.count(7)
# Returns: 2
```

9. `sort()` – sorts the list in numerical and lexicographical order and returns `None`.

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Lists can also be reversed when sorted using the `reverse=True` flag in the `sort()` method.

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

If you want to sort by attributes of items, you can use the `key` keyword argument:

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

In case of list of dicts the concept is the same:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'height': 175},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'height': 180},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'height': 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Sort by sub dict:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'size': {'height': 175,
'weight': 100}},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'size' : {'height': 180,
'weight': 90}},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'size': {'height': 185,
'weight': 110}}]

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Better way to sort using `attrgetter` and `itemgetter`

Lists can also be sorted using `attrgetter` and `itemgetter` functions from the operator module. These can help improve readability and reusability. Here are some examples,

```
from operator import itemgetter,attrgetter

people = [{'name':'chandan','age':20,'salary':2000},
          {'name':'chetan','age':18,'salary':5000},
          {'name':'guru','age':30,'salary':3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #in-place sorting by age
people.sort(key=by_salary) #in-place sorting by salary
```

`itemgetter` can also be given an index. This is helpful if you want to sort based on indices of a tuple.

```
list_of_tuples = [(1,2), (3,4), (5,0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[(5, 0), (1, 2), (3, 4)]
```

Use the `attrgetter` if you want to sort by attributes of an object,

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175),
           Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
           Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #reusing Person class from above
example

person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```

10. `clear()` – removes all items from the list

```
a.clear()
# a = []
```

11. **Replication** – multiplying an existing list by an integer will produce a larger list consisting of that many copies of the original. This can be useful for example for list initialization:

```
b = ["blah"] * 3
# b = ["blah", "blah", "blah"]
```

```
b = [1, 3, 5] * 5
# [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Take care doing this if your list contains references to objects (eg a list of lists), see Common Pitfalls - List multiplication and common references.

12. **Element deletion** – it is possible to delete multiple elements in the list using the `del` keyword and slice notation:

```
a = list(range(10))
del a[::2]
# a = [1, 3, 5, 7, 9]
del a[-1]
# a = [1, 3, 5, 7]
del a[:]
# a = []
```

13. **Copying**

The default assignment "=" assigns a reference of the original list to the new name. That is, the original name and new name are both pointing to the same list object. Changes made through any of them will be reflected in another. This is often not what you intended.

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

If you want to create a copy of the list you have below options.

You can slice it:

```
new_list = old_list[:]
```

You can use the built in list() function:

```
new_list = list(old_list)
```

You can use generic copy.copy():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

This is a little slower than list() because it has to find out the datatype of old_list first.

If the list contains objects and you want to copy them as well, use generic copy.deepcopy():

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Obviously the slowest and most memory-needing method, but sometimes unavoidable.

Python 3.x `Version ≥ 3.0`

`copy()` – Returns a shallow copy of the list

```
aa = a.copy()
# aa = [1, 2, 3, 4, 5]
```

Section 20.2: Accessing list values

Python lists are zero-indexed, and act like arrays in other languages.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

Attempting to access an index outside the bounds of the list will raise an `IndexError`.

```
lst[4]  # IndexError: list index out of range
```

Negative indices are interpreted as counting from the *end* of the list.

```
lst[-1]  # 4
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

This is functionally equivalent to

```
lst[len(lst)-1]  # 4
```

Lists allow to use *slice notation* as `lst[start:end:step]`. The output of the slice notation is a new list containing elements from index `start` to `end-1`. If options are omitted `start` defaults to beginning of list, `end` to end of list and `step` to 1:

```
lst[1:]      # [2, 3, 4]
lst[:3]      # [1, 2, 3]
lst[::2]     # [1, 3]
lst[::-1]    # [4, 3, 2, 1]
lst[-1:0:-1] # [4, 3, 2]
lst[5:8]     # [] since starting index is greater than length of lst, returns empty list
lst[1:10]    # [2, 3, 4] same as omitting ending index
```

With this in mind, you can print a reversed version of the list by calling

```
lst[::-1]    # [4, 3, 2, 1]
```

When using step lengths of negative amounts, the starting index has to be greater than the ending index otherwise the result will be an empty list.

```
lst[3:1:-1] # [4, 3]
```

Using negative step indices are equivalent to the following code:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

The indices used are 1 less than those used in negative indexing and are reversed.

Advanced slicing

When lists are sliced the `__getitem__()` method of the list object is called, with a `slice` object. Python has a builtin slice method to generate slice objects. We can use this to *store* a slice and reuse it later like so,

```
data = 'chandan purohit    22 2000'  #assuming data fields of fixed length
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#now we can have more readable slices
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

This can be of great use by providing slicing functionality to our objects by overriding `__getitem__` in our class.

Section 20.3: Checking if list is empty

The emptiness of a list is associated to the boolean `False`, so you don't have to check `len(lst) == 0`, but just `lst` or `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

Section 20.4: Iterating over a list

Python supports using a `for` loop directly on a list:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

You can also get the position of each item at the same time:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))

# Output: The item in position 0 is: foo
# Output: The item in position 1 is: bar
# Output: The item in position 2 is: baz
```

The other way of iterating a list based on the index value:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
```

```
baz
```

Note that changing items in a list while iterating on it may have unexpected results:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

In this last example, we deleted the first item at the first iteration, but that caused `bar` to be skipped.

Section 20.5: Checking whether an item is in a list

Python makes it very simple to check whether an item is in a list. Simply use the `in` operator.

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

> Note: the `in` operator on sets is asymptotically faster than on lists. If you need to use it many times on potentially large lists, you may want to convert your `list` to a `set`, and test the presence of elements on the `set`.

```
slst = set(lst)
'test' in slst
# Out: True
```

Section 20.6: Any and All

You can use `all()` to determine if all the values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Likewise, `any()` determines if one or more values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

While this example uses a list, it is important to note these built-ins work with any iterable, including generators.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Section 20.7: Reversing list elements

You can use the `reversed` function which returns an iterator to the reversed list:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Note that the list "numbers" remains unchanged by this operation, and remains in the same order it was originally.

To reverse in place, you can also use the `reverse` method.

You can also reverse a list (actually obtaining a copy, the original list is unaffected) by using the slicing syntax, setting the third argument (the step) as -1:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Section 20.8: Concatenate and Merge lists

1. **The simplest way to concatenate** `list1` and `list2`:

   ```
   merged = list1 + list2
   ```

2. `zip` **returns a list of tuples**, where the i-th tuple contains the i-th element from each of the argument sequences or iterables:

   ```
   alist = ['a1', 'a2', 'a3']
   blist = ['b1', 'b2', 'b3']

   for a, b in zip(alist, blist):
       print(a, b)

   # Output:
   # a1 b1
   # a2 b2
   # a3 b3
   ```

   If the lists have different lengths then the result will include only as many elements as the shortest one:

   ```
   alist = ['a1', 'a2', 'a3']
   blist = ['b1', 'b2', 'b3', 'b4']
   for a, b in zip(alist, blist):
       print(a, b)

   # Output:
   # a1 b1
   ```

```
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# Output:
# 0
```

For padding lists of unequal length to the longest one with `None`s use `itertools.zip_longest` (`itertools.izip_longest` in Python 2)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# Output:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

3. **Insert to a specific index values:**

```
alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("Final List :", alist)
```

Output:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Section 20.9: Length of a list

Use `len()` to get the one-dimensional length of a list.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

`len()` also works on strings, dictionaries, and other data structures similar to lists.

Note that `len()` is a built-in function, not a method of a list object.

Also note that the cost of `len()` is `0(1)`, meaning it will take the same amount of time to get the length of a list regardless of its length.

Section 20.10: Remove duplicate values in list

Removing duplicate values in a list can be done by converting the list to a `set` (that is an unordered collection of distinct objects). If a `list` data structure is needed, then the set can be converted back to a list using the function `list()`:

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Note that by converting a list to a set the original ordering is lost.

To preserve the order of the list one can use an `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Section 20.11: Comparison of lists

It's possible to compare lists and other sequences lexicographically using comparison operators. Both operands must be of the same type.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

If one of the lists is contained at the start of the other, the shortest list wins.

```
[1, 10] < [1, 10, 100]
# True
```

Section 20.12: Accessing values in nested list

Starting with a three-dimensional list:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Accessing items in the list:

```
print(alist[0][0][1])
#2
#Accesses second element in the first list in the first list

print(alist[1][1][2])
#10
#Accesses the third element in the second list in the second list
```

Performing support operations:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

Using nested for loops to print the list:

```
for row in alist: #One way to loop through nested lists
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Note that this operation can be used in a list comprehension or even as a generator to produce efficiencies, e.g.:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Not all items in the outer lists have to be lists themselves:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Another way to use nested for loops. The other way is better but I've needed to use this on occasion:

```
for row in range(len(alist)): #A less Pythonic way to loop through lists
    for col in range(len(alist[row])):
       print(alist[row][col])

#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

Using slices in nested list:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

The final list:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Section 20.13: Initializing a List to a Fixed Number of Elements

For **immutable** elements (e.g. `None`, string literals etc.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

For **mutable** elements, the same construct will result in all elements of the list referring to the same object, for example, for a set:

```
>>> my_list=[{1}] * 10
```

```
>>> print(my_list)
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
>>> my_list[0].add(2)
>>> print(my_list)
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}]
```

Instead, to initialize the list with a fixed number of **different mutable** objects, use:

```
my_list=[{1} for _ in range(10)]
```

Chapter 21: List comprehensions

List comprehensions in Python are concise, syntactic constructs. They can be utilized to generate lists from other lists by applying functions to each element in the list. The following section explains and demonstrates the use of these expressions.

Section 21.1: List Comprehensions

A list comprehension creates a new `list` by applying an expression to each element of an iterable. The most basic form is:

```
[ <expression> for <element> in <iterable> ]
```

There's also an optional 'if' condition:

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Each **`<element>`** in the **`<iterable>`** is plugged in to the **`<expression>`** if the (optional) **`<condition>`** evaluates to true. All results are returned at once in the new list. Generator expressions are evaluated lazily, but list comprehensions evaluate the entire iterator immediately - consuming memory proportional to the iterator's length.

To create a `list` of squared integers:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

The `for` expression sets `x` to each value in turn from `(1, 2, 3, 4)`. The result of the expression `x * x` is appended to an internal `list`. The internal `list` is assigned to the variable `squares` when completed.

Besides a speed increase (as explained here), a list comprehension is roughly equivalent to the following for-loop:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

The expression applied to each element can be as complex as needed:

```
# Get a list of uppercase characters from a string
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']

# Strip off any commas from the end of strings in a list
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']

# Organize letters in words more reasonably - in an alphabetical order
sentence = "Beautiful is better than ugly"
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

else

`else` can be used in List comprehension constructs, but be careful regarding the syntax. The if/else clauses should

be used before `for` loop, not after:

```
# create a list of characters in apple, replacing non vowels with '*'
# Ex - 'apple' --> ['a', '*', '*', '*' ,'e']

[x for x in 'apple' if x in 'aeiou' else '*']
#SyntaxError: invalid syntax

# When using if/else together use them before the loop
[x if x in 'aeiou' else '*' for x in 'apple']
#['a', '*', '*', '*', 'e']
```

Note this uses a different language construct, a conditional expression, which itself is not part of the comprehension syntax. Whereas the `if` after the `for`...`in` *is* a part of list comprehensions and used to *filter* elements from the source iterable.

Double Iteration

Order of double iteration `[... for x in ... for y in ...]` is either natural or counter-intuitive. The rule of thumb is to follow an equivalent `for` loop:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

This becomes:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

This can be compressed into one line as `[str(x) for i in range(3) for x in foo(i)]`

In-place Mutation and Other Side Effects

Before using list comprehension, understand the difference between functions called for their side effects (*mutating*, or *in-place* functions) which usually return `None`, and functions that return an interesting value.

Many functions (especially *pure* functions) simply take an object and return some object. An *in-place* function modifies the existing object, which is called a *side effect*. Other examples include input and output operations such as printing.

`list.sort()` sorts a list *in-place* (meaning that it modifies the original list) and returns the value `None`. Therefore, it won't work as expected in a list comprehension:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Instead, `sorted()` returns a sorted `list` rather than sorting in-place:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

Using comprehensions for side-effects is possible, such as I/O or in-place functions. Yet a for loop is usually more readable. While this works in Python 3:

```
[print(x) for x in (1, 2, 3)]
```

Instead use:

```
for x in (1, 2, 3):
    print(x)
```

In some situations, side effect functions *are* suitable for list comprehension. `random.randrange()` has the side effect of changing the state of the random number generator, but it also returns an interesting value. Additionally, `next()` can be called on an iterator.

The following random value generator is not pure, yet makes sense as the random generator is reset every time the expression is evaluated:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Whitespace in list comprehensions

More complicated list comprehensions can reach an undesired length, or become less readable. Although less common in examples, it is possible to break a list comprehension into multiple lines like so:

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

Section 21.2: Conditional List Comprehensions

Given a list comprehension you can append one or more `if` conditions to filter values.

```
[<expression> for <element> in <iterable> if <condition>]
```

For each **`<element>`** in **`<iterable>`**; if **`<condition>`** evaluates to `True`, add **`<expression>`** (usually a function of **`<element>`**) to the returned list.

For example, this can be used to extract only even numbers from a sequence of integers:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Live demo

The above code is equivalent to:

```
even_numbers = []
```

```
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

Also, a conditional list comprehension of the form `[e for x in y if c]` (where `e` and `c` are expressions in terms of `x`) is equivalent to `list(filter(lambda x: c, map(lambda x: e, y)))`.

Despite providing the same result, pay attention to the fact that the former example is almost 2x faster than the latter one. For those who are curious, this is a nice explanation of the reason why.

Note that this is quite different from the `... if ... else ...` conditional expression (sometimes known as a ternary expression) that you can use for the **`<expression>`** part of the list comprehension. Consider the following example:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Live demo

Here the conditional expression isn't a filter, but rather an operator determining the value to be used for the list items:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

This becomes more obvious if you combine it with other operators:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Live demo

If you are using Python 2.7, `xrange` may be better than `range` for several reasons as described in the `xrange` documentation.

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

The above code is equivalent to:

```
numbers = []
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
        temp = -1
    numbers.append(2 * temp + 1)
print(numbers)
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

One can combine ternary expressions and `if` conditions. The ternary operator works on the filtered result:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

The same couldn't have been achieved just by ternary operator only:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

See also: Filters, which often provide a sufficient alternative to conditional list comprehensions.

Section 21.3: Avoid repetitive and expensive operations using conditional clause

Consider the below list comprehension:

```
>>> def f(x):
...     import time
...     time.sleep(.1)       # Simulate expensive function
...     return x**2

>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

This results in two calls to `f(x)` for 1,000 values of `x`: one call for generating the value and the other for checking the `if` condition. If `f(x)` is a particularly expensive operation, this can have significant performance implications. Worse, if calling `f()` has side effects, it can have surprising results.

Instead, you should evaluate the expensive operation only once for each value of `x` by generating an intermediate iterable (generator expression) as follows:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Or, using the builtin map equivalent:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Another way that could result in a more readable code is to put the partial result (`v` in the previous example) in an iterable (such as a list or a tuple) and then iterate over it. Since `v` will be the only element in the iterable, the result is that we now have a reference to the output of our slow function computed only once:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

However, in practice, the logic of code can be more complicated and it's important to keep it readable. In general, a separate generator function is recommended over a complex one-liner:

```
>>> def process_prime_numbers(iterable):
...     for x in iterable:
...         if is_prime(x):
...             yield f(x)
...
>>> [x for x in process_prime_numbers(range(1000)) if x > 10]
[11, 13, 17, 19, ...]
```

Another way to prevent computing `f(x)` multiple times is to use the `@functools.lru_cache()`(Python 3.2+) decorator on `f(x)`. This way since the output of `f` for the input `x` has already been computed once, the second

function invocation of the original list comprehension will be as fast as a dictionary lookup. This approach uses memoization to improve efficiency, which is comparable to using generator expressions.

Say you have to flatten a list

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Some of the methods could be:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

However list comprehension would provide the best time complexity.

```
[item for sublist in l for item in sublist]
```

The shortcuts based on + (including the implied use in sum) are, of necessity, O(L^2) when there are L sublists -- as the intermediate result list keeps getting longer, at each step a new intermediate result list object gets allocated, and all the items in the previous intermediate result must be copied over (as well as a few new ones added at the end). So (for simplicity and without actual loss of generality) say you have L sublists of I items each: the first I items are copied back and forth L-1 times, the second I items L-2 times, and so on; total number of copies is I times the sum of x for x from 1 to L excluded, i.e., I * (L**2)/2.

The list comprehension just generates one list, once, and copies each item over (from its original place of residence to the result list) also exactly once.

Section 21.4: Dictionary Comprehensions

A dictionary comprehension is similar to a list comprehension except that it produces a dictionary object instead of a list.

A basic example:

Python 2.x Version ≥ 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

which is just another way of writing:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

As with a list comprehension, we can use a conditional statement inside the dict comprehension to produce only the dict elements meeting some criterion.

Python 2.x Version ≥ 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

Or, rewritten using a generator expression.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

Starting with a dictionary and using dictionary comprehension as a key-value pair filter

Python 2.x Version ≥ 2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Out: {'x': 1}
```

Switching key and value of dictionary (invert dictionary)

If you have a dict containing simple *hashable* values (duplicate values may have unexpected results):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

and you wanted to swap the keys and values you can take several approaches depending on your coding style:

- `swapped = {v: k for k, v in my_dict.items()}`
- `swapped = dict((v, k) for k, v in my_dict.iteritems())`
- `swapped = dict(zip(my_dict.values(), my_dict))`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(map(reversed, my_dict.items()))`

```
print(swapped)
# Out: {a: 1, b: 2, c: 3}
```

Python 2.x Version ≥ 2.3

If your dictionary is large, consider *importing* *itertools* and utilize `izip` or `imap`.

Merging Dictionaries

Combine dictionaries and optionally override old values with a nested dictionary comprehension.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

However, dictionary unpacking (PEP 448) may be a preferred.

Python 3.x Version ≥ 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Note: dictionary comprehensions were added in Python 3.0 and backported to 2.7+, unlike list comprehensions, which were added in 2.0. Versions < 2.7 can use generator expressions and the `dict()` builtin to simulate the behavior of dictionary comprehensions.

Section 21.5: List Comprehensions with Nested Loops

List Comprehensions can use nested `for` loops. You can code any number of nested for loops within a list comprehension, and each `for` loop may have an optional associated `if` test. When doing so, the order of the `for`

constructs is the same order as when writing a series of nested `for` statements. The general structure of list comprehensions looks like this:

```
[ expression for target1 in iterable1 [if condition1]
             for target2 in iterable2 [if condition2]...
             for targetN in iterableN [if conditionN] ]
```

For example, the following code flattening a list of lists using multiple `for` statements:

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

can be equivalently written as a list comprehension with multiple `for` constructs:

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

Live Demo

In both the expanded form and the list comprehension, the outer loop (first for statement) comes first.

In addition to being more compact, the nested comprehension is also significantly faster.

```
In [1]: data = [[1,2],[3,4],[5,6]]
In [2]: def f():
   ...:     output=[]
   ...:     for each_list in data:
   ...:         for element in each_list:
   ...:             output.append(element)
   ...:     return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

The overhead for the function call above is about *140ns*.

Inline `if`s are nested similarly, and may occur in any position after the first `for`:

```
data = [[1], [2, 3], [4, 5]]
output = [element for each_list in data
                  if len(each_list) == 2
                  for element in each_list
                  if element != 5]
print(output)
# Out: [2, 3, 4]
```

Live Demo

For the sake of readability, however, you should consider using traditional *for-loops*. This is especially true when nesting is more than 2 levels deep, and/or the logic of the comprehension is too complex. multiple nested loop list

comprehension could be error prone or it gives unexpected result.

Section 21.6: Generator Expressions

Generator expressions are very similar to list comprehensions. The main difference is that it does not create a full set of results at once; it creates a generator object which can then be iterated over.

For instance, see the difference in the following code:

```
# list comprehension
[x**2 for x in range(10)]
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python 2.x `Version ≥ 2.4`

```
# generator comprehension
(x**2 for x in xrange(10))
# Output: <generator object <genexpr> at 0x11b4b7c80>
```

These are two very different objects:

- the list comprehension returns a `list` object whereas the generator comprehension returns a `generator`.
- `generator` objects cannot be indexed and makes use of the `next` function to get items in order.

Note: We use `xrange` since it too creates a generator object. If we would use range, a list would be created. Also, `xrange` exists only in later version of python 2. In python 3, `range` just returns a generator. For more information, see the *Differences between range and xrange functions* example.

Python 2.x `Version ≥ 2.4`

```
g = (x**2 for x in xrange(10))
print(g[0])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next()  # 0
g.next()  # 1
g.next()  # 4
...
g.next()  # 81

g.next()  # Throws StopIteration Exception
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python 3.x `Version ≥ 3.0`

> NOTE: The function `g.next()` should be substituted by `next(g)` and `xrange` with `range` since `Iterator.next()` and `xrange()` do not exist in Python 3.

Although both of these can be iterated in a similar way:

```
for i in [x**2 for x in range(10)]:
    print(i)

"""
Out:
0
1
4
...
81
"""
```

Python 2.x `Version ≥ 2.4`

```
for i in (x**2 for x in xrange(10)):
    print(i)

"""
Out:
0
1
4
.
.
.
81
"""
```

Use cases

Generator expressions are lazily evaluated, which means that they generate and return each value only when the generator is iterated. This is often useful when iterating through large datasets, avoiding the need to create a duplicate of the dataset in memory:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Another common use case is to avoid iterating over an entire iterable if doing so is not necessary. In this example, an item is retrieved from a remote API with each iteration of `get_objects()`. Thousands of objects may exist, must be retrieved one-by-one, and we only need to know if an object matching a pattern exists. By using a generator expression, when we encounter an object matching the pattern.

```
def get_objects():
    """Gets objects from an API one by one"""
    while True:
        yield get_next_item()

def object_matches_pattern(obj):
    # perform potentially complex calculation
    return matches_pattern

def right_item_exists():
    items = (object_matched_pattern(each) for each in get_objects())
    for item in items:
        if item.is_the_right_one:


            return True
    return False
```

Section 21.7: Set Comprehensions

Set comprehension is similar to list and dictionary comprehension, but it produces a set, which is an unordered collection of unique elements.

Python 2.x Version ≥ 2.7

```
# A set containing every value in range(5):
{x for x in range(5)}
# Out: {0, 1, 2, 3, 4}

# A set of even numbers between 1 and 10:
{x for x in range(1, 11) if x % 2 == 0}
# Out: {2, 4, 6, 8, 10}

# Unique alphabetic characters in a string of text:
text = "When in the Course of human events it becomes necessary for one people..."
{ch.lower() for ch in text if ch.isalpha()}
# Out: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

Live Demo

Keep in mind that sets are unordered. This means that the order of the results in the set may differ from the one presented in the above examples.

Note: Set comprehension is available since python 2.7+, unlike list comprehensions, which were added in 2.0. In Python 2.2 to Python 2.6, the `set()` function can be used with a generator expression to produce the same result:

Python 2.x Version ≥ 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Section 21.8: Refactoring filter and map to list comprehensions

The `filter` or `map` functions should often be replaced by list comprehensions. Guido Van Rossum describes this well in an open letter in 2005:

> `filter(P, S)` is almost always written clearer as `[x for x in S if P(x)]`, and this has the huge advantage that the most common usages involve predicates that are comparisons, e.g. `x==42`, and defining a lambda for that just requires much more effort for the reader (plus the lambda is slower than the list comprehension). Even more so for `map(F, S)` which becomes `[F(x) for x in S]`. Of course, in many cases you'd be able to use generator expressions instead.

The following lines of code are considered "*not pythonic*" and will raise errors in many python linters.

```
filter(lambda x: x % 2 == 0, range(10)) # even numbers < 10
map(lambda x: 2*x, range(10)) # multiply each number by two
reduce(lambda x,y: x+y, range(10)) # sum of all elements in list
```

Taking what we have learned from the previous quote, we can break down these `filter` and `map` expressions into their equivalent *list comprehensions*; also removing the *lambda* functions from each - making the code more readable in the process.

```
# Filter:
# P(x) = x % 2 == 0
# S = range(10)
[x for x in range(10) if x % 2 == 0]

# Map
# F(x) = 2*x
# S = range(10)
[2*x for x in range(10)]
```

Readability becomes even more apparent when dealing with chaining functions. Where due to readability, the results of one map or filter function should be passed as a result to the next; with simple cases, these can be replaced with a single list comprehension. Further, we can easily tell from the list comprehension what the outcome of our process is, where there is more cognitive load when reasoning about the chained Map & Filter process.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Refactoring - Quick Reference

- **Map**

```
map(F, S) == [F(x) for x in S]
```

- **Filter**

```
filter(P, S) == [x for x in S if P(x)]
```

where `F` and `P` are functions which respectively transform input values and return a `bool`

Section 21.9: Comprehensions involving tuples

The `for` clause of a list comprehension can specify more than one variable:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [3, 7, 11]

[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]
# Out: [3, 7, 11]
```

This is just like regular `for` loops:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Note however, if the expression that begins the comprehension is a tuple then it must be parenthesized:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
```

```
# SyntaxError: invalid syntax

[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [(1, 2), (3, 4), (5, 6)]
```

Section 21.10: Counting Occurrences Using Comprehension

When we want to count the number of items in an iterable, that meet some condition, we can use comprehension to produce an idiomatic syntax:

```
# Count the numbers in `range(1000)` that are even and contain the digit `9`:
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# Out: 95
```

The basic concept can be summarized as:

1. Iterate over the elements in `range(1000)`.
2. Concatenate all the needed `if` conditions.
3. Use 1 as *expression* to return a 1 for each item that meets the conditions.
4. Sum up all the `1`s to determine number of items that meet the conditions.

Note: Here we are not collecting the `1`s in a list (note the absence of square brackets), but we are passing the ones directly to the `sum` function that is summing them up. This is called a *generator expression*, which is similar to a Comprehension.

Section 21.11: Changing Types in a List

Quantitative data is often read in as strings that must be converted to numeric types before processing. The types of all list items can be converted with either a List Comprehension or the `map()` function.

```
# Convert a list of strings to integers.
items = ["1","2","3","4"]
[int(item) for item in items]
# Out: [1, 2, 3, 4]

# Convert a list of strings to float.
items = ["1","2","3","4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

Section 21.12: Nested List Comprehensions

Nested list comprehensions, unlike list comprehensions with nested loops, are List comprehensions within a list comprehension. The initial expression can be any arbitrary expression, including another list comprehension.

```
#List Comprehension with nested loop
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]

#Nested List Comprehension
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#Out: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

The Nested example is equivalent to

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

One example where a nested comprehension can be used it to transpose a matrix.

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]

[[row[i] for row in matrix] for i in range(len(matrix))]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Like nested `for` loops, there is no limit to how deep comprehensions can be nested.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Section 21.13: Iterate two or more list simultaneously within list comprehension

For iterating more than two lists simultaneously within *list comprehension*, one may use `zip()` as:

```
>>> list_1 = [1, 2, 3 , 4]
>>> list_2 = ['a', 'b', 'c', 'd']
>>> list_3 = ['6', '7', '8', '9']

# Two lists
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Three lists
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]

# so on ...
```

Chapter 22: List slicing (selecting parts of lists)

Section 22.1: Using the third "step" argument

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Section 22.2: Selecting a sublist from a list

```
lst = ['a', 'b', 'c', 'd', 'e']

lst[2:4]
# Output: ['c', 'd']

lst[2:]
# Output: ['c', 'd', 'e']

lst[:4]
# Output: ['a', 'b', 'c', 'd']
```

Section 22.3: Reversing a list with slicing

```
a = [1, 2, 3, 4, 5]

# steps through the list backwards (step=-1)
b = a[::-1]

# built-in list method to reverse 'a'
a.reverse()

if a = b:
    print(True)

print(b)

# Output:
# True
# [5, 4, 3, 2, 1]
```

Section 22.4: Shifting a list using slicing

```
def shift_list(array, s):
    """Shifts the elements of a list to the left or right.

    Args:
        array - the list to shift
        s - the amount to shift the list ('+': right-shift, '-': left-shift)

    Returns:
        shifted_array - the shifted list
```

```
    """
    # calculate actual shift amount (e.g., 11 --> 1 if length of the array is 5)
    s %= len(array)

    # reverse the shift direction to be more intuitive
    s *= -1

    # shift array with list slicing
    shifted_array = array[s:] + array[:s]

    return shifted_array

my_array = [1, 2, 3, 4, 5]

# negative numbers
shift_list(my_array, -7)
>>> [3, 4, 5, 1, 2]

# no shift on numbers equal to the size of the array
shift_list(my_array, 5)
>>> [1, 2, 3, 4, 5]

# works on positive numbers
shift_list(my_array, 3)
>>> [3, 4, 5, 1, 2]
```

Chapter 23: groupby()

Parameter	Details
iterable	Any python iterable
key	Function(criteria) on which to group the iterable

In Python, the `itertools.groupby()` method allows developers to group values of an iterable class based on a specified property into another iterable set of values.

Section 23.1: Example 4

In this example we see what happens when we use different types of iterable.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"), \
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results in

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

This example below is essentially the same as the one above it. The only difference is that I have changed all the tuples to lists.

```
things = [["animal", "bear"], ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"], \
          ["vehicle", "speed boat"], ["vehicle", "school bus"]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
  ['vehicle', 'speed boat'],
  ['vehicle', 'school bus']]}
```

Section 23.2: Example 2

This example illustrates how the default key is chosen if we do not specify any

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
```

```
    dic[k] = list(v)
dic
```

Results in

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Notice here that the tuple as a whole counts as one key in this list

Section 23.3: Example 3

Notice in this example that mulato and camel don't show up in our result. Only the last element with the specified key shows up. The last result for c actually wipes out two previous results. But watch the new version where I have the data sorted first on same key.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Sorted Version

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons', 'man',
'woman'), 'wombat']
```

```
{'c': ['cow', 'cat', 'camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mulato', 'mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Chapter 24: Linked lists

A linked list is a collection of nodes, each made up of a reference and a value. Nodes are strung together into a sequence using their references. Linked lists can be used to implement more complex data structures like lists, stacks, queues, and associative arrays.

Section 24.1: Single linked list example

This example implements a linked list with many of the same methods as that of the built-in list object.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty"""
        return self.head is None

    def add(self, item):
        """Add the item to the list"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """Return the length/size of the list"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
            current = current.getNext()
        return count

    def search(self, item):
        """Search for item in list. If found, return True. If not found, return False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
```

```
        return found

    def remove(self, item):
        """Remove item from list. If item is not found in list, raise ValueError"""
        current = self.head
        previous = None
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if found:
            if previous is None:
                self.head = current.getNext()
            else:
                previous.setNext(current.getNext())
        else:
            raise ValueError
            print 'Value not found.'

    def insert(self, position, item):
        """
        Insert item at position specified. If position specified is
        out of bounds, raise IndexError
        """
        if position > self.size() - 1:
            raise IndexError
            print "Index out of bounds."
        current = self.head
        previous = None
        pos = 0
        if position is 0:
            self.add(item)
        else:
            new_node = Node(item)
            while pos < position:
                pos += 1
                previous = current
                current = current.getNext()
            previous.setNext(new_node)
            new_node.setNext(current)

    def index(self, item):
        """
        Return the index where item is found.
        If item is not found, return None.
        """
        current = self.head
        pos = 0
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
                pos += 1
        if found:
            pass
        else:
            pos = None
```

```
        return pos

    def pop(self, position = None):
        """
        If no argument is provided, return and remove the item at the head.
        If position is provided, return and remove the item at that position.
        If index is out of bounds, raise IndexError
        """
        if position > self.size():
            print 'Index out of bounds'
            raise IndexError

        current = self.head
        if position is None:
            ret = current.getData()
            self.head = current.getNext()
        else:
            pos = 0
            previous = None
            while pos < position:
                previous = current
                current = current.getNext()
                pos += 1
                ret = current.getData()
            previous.setNext(current.getNext())
        print ret
        return ret

    def append(self, item):
        """Append item to the end of the list"""
        current = self.head
        previous = None
        pos = 0
        length = self.size()
        while pos < length:
            previous = current
            current = current.getNext()
            pos += 1
        new_node = Node(item)
        if previous is None:
            new_node.setNext(current)
            self.head = new_node
        else:
            previous.setNext(new_node)

    def printList(self):
        """Print the list"""
        current = self.head
        while current is not None:
            print current.getData()
            current = current.getNext()
```

Usage functions much like that of the built-in list.

```
ll = LinkedList()
ll.add('l')
ll.add('H')
ll.insert(1,'e')
ll.append('l')
ll.append('o')
ll.printList()
```

```
H
e
l
l
o
```

Chapter 25: Linked List Node

Section 25.1: Write a simple Linked List Node in python

A linked list is either:

- the empty list, represented by None, or
- a node that contains a cargo object and a reference to a linked list.

```
#! /usr/bin/env python

class Node:
      def __init__(self, cargo=None, next=None):
          self.car = cargo
          self.cdr = next
      def __str__(self):
          return str(self.car)


      def display(lst):
          if lst:
             w("%s " % lst)
             display(lst.cdr)
          else:
             w("nil\n")
```

Chapter 26: Filter

Parameter	Details
function	*callable* that determines the condition or `None` then use the identity function for filtering (*positional-only*)
iterable	iterable that will be filtered (*positional-only*)

Section 26.1: Basic use of filter

To `filter` discards elements of a sequence based on some criteria:

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5
```

Python 2.x `Version ≥ 2.0`

```
filter(long_name, names)
# Out: ['Barney']

[name for name in names if len(name) > 5] # equivalent list comprehension
# Out: ['Barney']


from itertools import ifilter
ifilter(long_name, names)       # as generator (similar to python 3.x filter builtin)
# Out: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # equivalent to filter with lists
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x0000000003FD5D38>
```

Python 2.x `Version ≥ 2.6`

```
# Besides the options for older python 2.x versions there is a future_builtin function:
from future_builtins import filter
filter(long_name, names)       # identical to itertools.ifilter
# Out: <itertools.ifilter at 0x3eb0ba8>
```

Python 3.x `Version ≥ 3.0`

```
filter(long_name, names)        # returns a generator
# Out: <filter at 0x1fc6e443470>
list(filter(long_name, names))  # cast to list
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

Section 26.2: Filter without function

If the function parameter is `None`, then the identity function will be used:

```
list(filter(None, [1, 0, 2, [], '', 'a']))  # discards 0, [] and ''
# Out: [1, 2, 'a']
```

Python 2.x `Version ≥ 2.0.1`

```
[i for i in [1, 0, 2, [], '', 'a'] if i] # equivalent list comprehension
```

Python 3.x `Version ≥ 3.0.0`

```
(i for i in [1, 0, 2, [], '', 'a'] if i) # equivalent generator expression
```

Section 26.3: Filter as short-circuit check

`filter` (python 3.x) and `ifilter` (python 2.x) return a generator so they can be very handy when creating a short-circuit test like `or` or `and`:

Python 2.x `Version ≥ 2.0.1`

```
 # not recommended in real use but keeps the example short:
from itertools import ifilter as filter
```

Python 2.x `Version ≥ 2.6.1`

```
 from future_builtins import filter
```

To find the first element that is smaller than 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filter(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
#        Check rectangular tire, 80$
# Out: ('rectangular tire', 80)
```

The `next`-function gives the next (in this case first) element of and is therefore the reason why it's short-circuit.

Section 26.4: Complementary function: filterfalse, ifilterfalse

There is a complementary function for `filter` in the `itertools`-module:

Python 2.x `Version ≥ 2.0.1`

```
 # not recommended in real use but keeps the example valid for python 2.x and python 3.x
from itertools import ifilterfalse as filterfalse
```

Python 3.x `Version ≥ 3.0.0`

```
from itertools import filterfalse
```

which works exactly like the *generator* `filter` but keeps only the elements that are `False`:

```
# Usage without function (None):
list(filterfalse(None, [1, 0, 2, [], '', 'a']))  # discards 1, 2, 'a'
# Out: [0, [], '']
```

```
# Usage with function
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

list(filterfalse(long_name, names))
# Out: ['Fred', 'Wilma']
```

```
# Short-circuit usage with next:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
```

```
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
# Out: ('Toyota', 1000)
```

```
# Using an equivalent generator:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

Chapter 27: Heapq

Section 27.1: Largest and smallest items in a collection

To find the largest items in a collection, `heapq` module has a function called `nlargest`, we pass it two arguments, the first one is the number of items that we want to retrieve, the second one is the collection name:

```
import heapq


numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers))  # [200, 150, 100, 50]
```

Similarly, to find the smallest items in a collection, we use `nsmallest` function:

```
print(heapq.nsmallest(4, numbers))  # [1, 2, 4, 8]
```

Both `nlargest` and `nsmallest` functions take an optional argument (key parameter) for complicated data structures. The following example shows the use of `age` property to retrieve the oldest and the youngest people from `people` dictionary:

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Output: [{'firstname': 'John', 'age': 45, 'lastname': 'Roe'}, {'firstname': 'John', 'age': 30,
'lastname': 'Doe'}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Output: [{'firstname': 'Janie', 'age': 10, 'lastname': 'Doe'}, {'firstname': 'Johnny', 'age': 12,
'lastname': 'Doe'}]
```

Section 27.2: Smallest item in a collection

The most interesting property of a `heap` is that its smallest element is always the first element: `heap[0]`

```
import heapq


numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Output: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers)  # 2
print(numbers)
# Output: [4, 8, 10, 100, 20, 50, 32, 200, 150]
```

```
heapq.heappop(numbers)  # 4
print(numbers)
# Output:  [8, 20, 10, 100, 150, 50, 32, 200]
```

Chapter 28: Tuple

A tuple is an immutable list of values. Tuples are one of Python's simplest and most common collection types, and can be created with the comma operator (`value = 1, 2, 3`).

Section 28.1: Tuple

Syntactically, a tuple is a comma-separated list of values:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Although not necessary, it is common to enclose tuples in parentheses:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Create an empty tuple with parentheses:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

To create a tuple with a single element, you have to include a final comma:

```
t1 = 'a',
type(t1)            # <type 'tuple'>
```

Note that a single value in parentheses is not a tuple:

```
t2 = ('a')
type(t2)            # <type 'str'>
```

To create a singleton tuple it is necessary to have a trailing comma.

```
t2 = ('a',)
type(t2)            # <type 'tuple'>
```

Note that for singleton tuples it's recommended (see PEP8 on trailing commas) to use parentheses. Also, no white space after the trailing comma (see PEP8 on whitespaces)

```
t2 = ('a',)         # PEP8-compliant
t2 = 'a',           # this notation is not recommended by PEP8
t2 = ('a', )        # this notation is not recommended by PEP8
```

Another way to create a tuple is the built-in function `tuple`.

```
t = tuple('lupins')
print(t)            # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)            # (0, 1, 2)
```

These examples are based on material from the book **Think Python** by Allen B. Downey.

Section 28.2: Tuples are immutable

One of the main differences between `lists` and `tuples` in Python is that tuples are immutable, that is, one cannot add or modify items once the tuple is initialized. For example:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Similarly, tuples don't have `.append` and `.extend` methods as `list` does. Using `+=` is possible, but it changes the binding of the variable, and not the tuple itself:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Be careful when placing mutable objects, such as `lists`, inside tuples. This may lead to very confusing outcomes when changing them. For example:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Will ***both*** raise an error and change the contents of the list within the tuple:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

You can use the `+=` operator to "append" to a tuple - this works by creating a new tuple with the new element you "appended" and assign it to its current variable; the old tuple is not changed, but replaced!

This avoids converting to and from a list, but this is slow and is a bad practice, especially if you're going to append multiple times.

Section 28.3: Packing and Unpacking Tuples

Tuples in Python are values separated by commas. Enclosing parentheses for inputting tuples are optional, so the two assignments

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

and

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

are equivalent. The assignment `a = 1, 2, 3` is also called *packing* because it packs values together in a tuple.

Note that a one-value tuple is also a tuple. To tell Python that a variable is a tuple and not a single value you can use

a trailing comma

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

A comma is needed also if you use parentheses

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

To unpack values from a tuple and do multiple assignments use

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

The symbol _ can be used as a disposable variable name if one only needs some elements of a tuple, acting as a placeholder:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Single element tuples:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

In Python 3 a target variable with a * prefix can be used as a *catch-all* variable (see Unpacking Iterables):

Python 3.x `Version ≥ 3.0`

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Section 28.4: Built-in Tuple Functions

Tuples support the following build-in functions

Comparison

If elements are of the same type, python performs the comparison and returns the result. If elements are different types, it checks whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1','2','3')
```

```
tuple3 = ('a', 'b', 'c', 'd', 'e')

cmp(tuple1, tuple2)
Out: 1

cmp(tuple2, tuple1)
Out: -1

cmp(tuple1, tuple3)
Out: 0
```

Tuple Length

The function `len` returns the total length of the tuple

```
len(tuple1)
Out: 5
```

Max of a tuple

The function `max` returns item from the tuple with the max value

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Min of a tuple

The function min returns the item from the tuple with the min value

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Convert a list into tuple

The built-in function `tuple` converts a list into a tuple.

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

Tuple concatenation

Use `+` to concatenate two tuples

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Section 28.5: Tuple Are Element-wise Hashable and Equatable

```
hash( (1, 2) )  # ok
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Thus a tuple can be put inside a `set` or as a key in a `dict` only if each of its elements can.

```
{ (1, 2) } #  ok
```

```
{ ([], {"hello"}) ) # not ok
```

Section 28.6: Indexing Tuples

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

Indexing with negative numbers will start from the last element as -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indexing a range of elements

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Section 28.7: Reversing Elements

Reverse elements within a tuple

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Or using reversed (reversed gives an iterable which is converted to a tuple):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Chapter 29: Basic Input and Output

Section 29.1: Using the print function

Python 3.x `Version ≥ 3.0`

In Python 3, print functionality is in the form of a function:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x `Version ≥ 2.3`

In Python 2, print was originally a statement, as shown below.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Note: using `from __future__ import print_function` in Python 2 will allow users to use the `print()` function the same as Python 3 code. This is only available in Python 2.6 and above.

Section 29.2: Input from a File

Input can also be read from files. Files can be opened using the built-in function `open`. Using a `with <command> as <name>` syntax (called a 'Context Manager') makes using `open` and getting a handle for the file super easy:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

This ensures that when code execution leaves the block the file is automatically closed.

Files can be opened in different modes. In the above example the file is opened as read-only. To open an existing file for reading only use `r`. If you want to read that file as bytes use `rb`. To append data to an existing file use `a`. Use `w` to create a file or overwrite any existing files of the same name. You can use `r+` to open a file for both reading and writing. The first argument of `open()` is the filename, the second is the mode. If mode is left blank, it will default to `r`.

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
```

```
    # here we read the whole content into one string:
    content = fileobj.read()
    # get a list of lines, just like int the previous example:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

If the size of the file is tiny, it is safe to read the whole file contents into memory. If the file is very large it is often better to read line-by-line or by chunks, and process the input in the same loop. To do that:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

When reading files, be aware of the operating system-specific line-break characters. Although `for line in fileobj` automatically strips them off, it is always safe to call `strip()` on the lines read, as it is shown above.

Opened files (`fileobj` in the above examples) always point to a specific location in the file. When they are first opened the file handle points to the very beginning of the file, which is the position `0`. The file handle can display its current position with `tell`:

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

Upon reading all the content, the file handler's position will be pointed at the end of the file:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

The file handler position can be set to whatever is needed:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

You can also read any length from the file content during a given call. To do this pass an argument for `read()`. When `read()` is called with no argument it will read until the end of the file. If you pass an argument it will read that number of bytes or characters, depending on the mode (`rb` and `r` respectively):

```
# reads the next 4 characters
# starting at the current position
next4 = fileobj.read(4)
# what we got?
print(next4) # 'cucu'
# where we are now?
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.

fileobj.close()
```

To demonstrate the difference between characters and bytes:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

Section 29.3: Read from stdin

Python programs can read from unix pipelines. Here is a simple example how to read from `stdin`:

```
import sys

for line in sys.stdin:
    print(line)
```

Be aware that `sys.stdin` is a stream. It means that the for-loop will only terminate when the stream has ended.

You can now pipe the output of another program into your python program as follows:

```
$ cat myfile | python myprogram.py
```

In this example `cat myfile` can be any unix command that outputs to `stdout`.

Alternatively, using the fileinput module can come in handy:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Section 29.4: Using input() and raw_input()

Python 2.x `Version ≥ 2.3`

`raw_input` will wait for the user to enter text and then return the result as a string.

```
foo = raw_input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

Python 3.x `Version ≥ 3.0`

`input` will wait for the user to enter text and then return the result as a string.

```
foo = input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

Section 29.5: Function to prompt user for a number

```
def input_number(msg, err_msg=None):
    while True:
        try:
```

```
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)


def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

And to use it:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Or, if you do not want an "error message":

```
user_number = input_number("input a number: ")
```

Section 29.6: Printing a string without a newline at the end

Python 2.x `Version ≥ 2.3`

In Python 2.x, to continue a line with `print`, end the `print` statement with a comma. It will automatically add a space.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x `Version ≥ 3.0`

In Python 3.x, the `print` function has an optional `end` parameter that is what it prints at the end of the given string. By default it's a newline character, so equivalent to this:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

But you could pass in other strings

```
print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

If you want more control over the output, you can use `sys.stdout.write`:

```python
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Chapter 30: Files & Folders I/O

Parameter	Details
filename	the path to your file or, if the file is in the working directory, the filename of your file
access_mode	a string value that determines how the file is opened
buffering	an integer value used for optional line buffering

When it comes to storing, reading, or communicating data, working with the files of an operating system is both necessary and easy with Python. Unlike other languages where file input and output requires complex reading and writing objects, Python simplifies the process only needing commands to open, read/write and close the file. This topic explains how Python can interface with files on the operating system.

Section 30.1: File modes

There are different modes you can open a file with, specified by the `mode` parameter. These include:

- `'r'` - reading mode. The default. It allows you only to read the file, not to modify it. When using this mode the file must exist.
- `'w'` - writing mode. It will create a new file if it does not exist, otherwise will erase the file and allow you to write to it.
- `'a'` - append mode. It will write data to the end of the file. It does not erase the file, and the file must exist for this mode.
- `'rb'` - reading mode in binary. This is similar to `r` except that the reading is forced in binary mode. This is also a default choice.
- `'r+'` - reading mode plus writing mode at the same time. This allows you to read and write into files at the same time without having to use `r` and `w`.
- `'rb+'` - reading and writing mode in binary. The same as `r+` except the data is in binary
- `'wb'` - writing mode in binary. The same as `w` except the data is in binary.
- `'w+'` - writing and reading mode. The exact same as `r+` but if the file does not exist, a new one is made. Otherwise, the file is overwritten.
- `'wb+'` - writing and reading mode in binary mode. The same as `w+` but the data is in binary.
- `'ab'` - appending in binary mode. Similar to `a` except that the data is in binary.
- `'a+'` - appending and reading mode. Similar to `w+` as it will create a new file if the file does not exist. Otherwise, the file pointer is at the end of the file if it exists.
- `'ab+'` - appending and reading mode in binary. The same as `a+` except that the data is in binary.

```
with open(filename, 'r') as f:
    f.read()
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write('\\n' + newdata)
```

	r	r+	w	w+	a	a+
Read	✓	✓	✗	✓	✗	✓

Write	✗	✓	✓	✓	✓	✓
Creates file	✗	✗	✓	✓	✓	✓
Erases file	✗	✗	✓	✓	✗	✗
Initial position	Start	Start	Start	Start	End	End

Python 3 added a new mode for `exclusive creation` so that you will not accidentally truncate or overwrite and existing file.

- `'x'` - open for exclusive creation, will raise `FileExistsError` if the file already exists
- `'xb'` - open for exclusive creation writing mode in binary. The same as `x` except the data is in binary.
- `'x+'` - reading and writing mode. Similar to `w+` as it will create a new file if the file does not exist. Otherwise, will raise `FileExistsError`.
- `'xb+'` - writing and reading mode. The exact same as `x+` but the data is binary

	x	x+
Read	✗	✓
Write	✓	✓
Creates file	✓	✓
Erases file	✗	✗
Initial position	Start	Start

Allow one to write your file open code in a more pythonic manner:

Python 3.x `Version ≥ 3.3`

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

In Python 2 you would have done something like

Python 2.x `Version ≥ 2.0`

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # Work with your open file
else:
    # Your error handling goes here
```

Section 30.2: Reading a file line-by-line

The simplest way to iterate over a file line-by-line:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` allows for more granular control over line-by-line iteration. The example below is equivalent to the one above:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
```

```
        # If the result is an empty string
        if cur_line == '':
            # We have reached the end of the file
            break
        print(cur_line)
```

Using the for loop iterator and readline() together is considered bad practice.

More commonly, the `readlines()` method is used to store an iterable collection of the file's lines:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

This would print the following:

> Line 0: hello
>
> Line 1: world

Section 30.3: Iterate files (recursively)

To iterate all files, including in sub directories, use os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir can be "." to start from current directory, or any other path to start from.

Python 3.x `Version ≥ 3.5`

If you also wish to get information about the file, you may use the more efficient method os.scandir like so:

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Section 30.4: Getting the full contents of a file

The preferred method of file i/o is to use the `with` keyword. This will ensure the file handle is closed once the reading or writing has been completed.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

or, to handle closing the file manually, you can forgo `with` and simply call `close` yourself:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
```

```
in_file.close()
```

Keep in mind that without using a `with` statement, you might accidentally keep the file open in case an unexpected exception arises like so:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

Section 30.5: Writing to a file

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

If you open `myfile.txt`, you will see that its contents are:

> Line 1Line 2Line 3Line 4

Python doesn't automatically add line breaks, you need to do that manually:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

> Line 1
> Line 2
> Line 3
> Line 4

Do not use `os.linesep` as a line terminator when writing files opened in text mode (the default); use `\n` instead.

If you want to specify an encoding, you simply add the `encoding` parameter to the `open` function:

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

It is also possible to use the print statement to write to a file. The mechanics are different in Python 2 vs Python 3, but the concept is the same in that you can take the output that would have gone to the screen and send it to a file instead.

Python 3.x `Version ≥ 3.0`

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile

    #Note: it is possible to specify the file parameter AND write to the screen
    #by making sure file ends up with a None value either directly or via a variable
```

```
    myfile = None
    print(s, file = myfile) # writes to stdout
    print(s, file = None)   # writes to stdout
```

In Python 2 you would have done something like

Python 2.x Version ≥ 2.0

```
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s   # writes to stdout
print >> outfile, s   # writes to outfile
```

Unlike using the write function, the print function does automatically add line breaks.

Section 30.6: Check whether a file or path exists

Employ the EAFP coding style and `try` to open it.

```
import errno

try:
    with open(path) as f:
        # File exists
except IOError as e:
    # Raise the exception if it is not ENOENT (No such file or directory)
    if e.errno != errno.ENOENT:
        raise
    # No such file or directory
```

This will also avoid race-conditions if another process deleted the file between the check and when it is used. This race condition could happen in the following cases:

- Using the `os` module:

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x Version ≥ 3.4

- Using `pathlib`:

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

To check whether a given path exists or not, you can follow the above EAFP procedure, or explicitly check the path:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

Section 30.7: Random File Access Using mmap

Using the `mmap` module allows the user to randomly access locations in a file by mapping the file into memory. This is an alternative to using normal file operations.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)

    # print characters at indices 5 through 10
    print mm[5:10]

    # print the line starting from mm's current position
    print mm.readline()

    # write a character to the 5th index
    mm[5] = 'a'

    # return mm's position to the beginning of the file
    mm.seek(0)

    # close the mmap object
    mm.close()
```

Section 30.8: Replacing text in a file

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end='')
```

Section 30.9: Checking if a file is empty

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

or

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

However, both will throw an exception if the file does not exist. To avoid having to catch such an error, do this:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

which will return a `bool` value.

Section 30.10: Read a file between a range of lines

So let's suppose you want to iterate only between some specific lines of a file

You can make use of `itertools` for that

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

This will read through the lines 13 to 20 as in python indexing starts from 0. So line number 1 is indexed as 0

As can also read some extra lines by making use of the `next()` keyword here.

And when you are using the file object as an iterable, please don't use the `readline()` statement here as the two techniques of traversing a file are not to be mixed together

Section 30.11: Copy a directory tree

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

The destination directory **must not exist** already.

Section 30.12: Copying contents of one file to a different file

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Using the `shutil` module:

```
import shutil
shutil.copyfile(src, dst)
```

Chapter 31: os.path

This module implements some useful functions on pathnames. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings.

Section 31.1: Join Paths

To join two or more path components together, firstly import os module of python and then use following:

```
import os
os.path.join('a', 'b', 'c')
```

The advantage of using os.path is that it allows code to remain compatible over all operating systems, as this uses the separator appropriate for the platform it's running on.

For example, the result of this command on Windows will be:

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

In an Unix OS:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

Section 31.2: Path Component Manipulation

To split one component off of the path:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')
>>> p
'/Users/csaftoiu/tmp/foo.txt'
>>> os.path.dirname(p)
'/Users/csaftoiu/tmp'
>>> os.path.basename(p)
'foo.txt'
>>> os.path.split(os.getcwd())
('/Users/csaftoiu/tmp', 'foo.txt')
>>> os.path.splitext(os.path.basename(p))
('foo', '.txt')
```

Section 31.3: Get the parent directory

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

Section 31.4: If the given path exists

to check if the given path exists

```
path = '/home/john/temp'
os.path.exists(path)
#this returns false if path doesn't exist or if the path is a broken symbolic link
```

Section 31.5: check if the given path is a directory, file, symbolic link, mount point etc

to check if the given path is a directory

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

to check if the given path is a file

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

to check if the given path is symbolic link

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

to check if the given path is a mount point

```
mount_path = '/home'
os.path.ismount(mount_path)
```

Section 31.6: Absolute Path from Relative Path

Use `os.path.abspath`:

```
>>> os.getcwd()
'/Users/csaftoiu/tmp'
>>> os.path.abspath('foo')
'/Users/csaftoiu/tmp/foo'
>>> os.path.abspath('../foo')
'/Users/csaftoiu/foo'
>>> os.path.abspath('/foo')
'/foo'
```

Chapter 32: Iterables and Iterators

Section 32.1: Iterator vs Iterable vs Generator

An **iterable** is an object that can return an **iterator**. Any object with state that has an `__iter__` method and returns an iterator is an iterable. It may also be an object *without* state that implements a `__getitem__` method. - The method can take indices (starting from zero) and raise an `IndexError` when the indices are no longer valid.

Python's `str` class is an example of a `__getitem__` iterable.

An **Iterator** is an object that produces the next value in a sequence when you call `next(*object*)` on some object. Moreover, any object with a `__next__` method is an iterator. An iterator raises `StopIteration` after exhausting the iterator and *cannot* be re-used at this point.

Iterable classes:

Iterable classes define an `__iter__` and a `__next__` method. Example of an iterable class:

```
class MyIterable:

    def __iter__(self):

         return self

    def __next__(self):
         #code

#Classic iterable object in older versions of python, __getitem__ is still supported...
class MySequence:

    def __getitem__(self, index):

         if (condition):
             raise IndexError
         return (item)

 #Can produce a plain `iterator` instance by using iter(MySequence())
```

Trying to instantiate the abstract class from the `collections` module to better see this.

Example:

Python 2.x `Version ≥ 2.3`

```
import collections
>>> collections.Iterator()
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Python 3.x `Version ≥ 3.0`

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Handle Python 3 compatibility for iterable classes in Python 2 by doing the following:

Python 2.x `Version ≥ 2.3`

```
class MyIterable(object): #or collections.Iterator, which I'd recommend....

     ....
```

```
    def __iter__(self):

        return self

    def next(self): #code

    __next__ = next
```

Both of these are now iterators and can be looped through:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Generators are simple ways to create iterators. A generator *is* an iterator and an iterator is an iterable.

Section 32.2: Extract values one by one

Start with `iter()` built-in to get **iterator** over iterable and use `next()` to get elements one by one until `StopIteration` is raised signifying the end:

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

Section 32.3: Iterating over entire iterable

```
s = {1, 2, 3}

# get every element in s
for a in s:
    print a  # prints 1, then 2, then 3

# copy into list
l1 = list(s)  # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2]  # l2 = [6]
```

Section 32.4: Verify only one element in iterable

Use unpacking to extract the first element and ensure it's the only one:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Section 32.5: What can be iterable

Iterable can be anything for which items are received *one by one, forward only*. Built-in Python collections are iterable:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Generators return iterables:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Section 32.6: Iterator isn't reentrant!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a

# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
gen()
```

Chapter 33: Functions

Parameter	Details
arg1, ..., *argN*	Regular arguments
**args*	Unnamed positional arguments
kw1, ..., *kwN*	Keyword-only arguments
***kwargs*	The rest of keyword arguments

Functions in Python provide organized, reusable and modular code to perform a set of specific actions. Functions simplify the coding process, prevent redundant logic, and make the code easier to follow. This topic describes the declaration and utilization of functions in Python.

Python has many *built-in functions* like `print()`, `input()`, `len()`. Besides built-ins you can also create your own functions to do more specific jobs—these are called *user-defined functions*.

Section 33.1: Defining and calling simple functions

Using the `def` statement is the most common way to define a function in python. This statement is a so called *single clause compound statement* with the following syntax:

```
def function_name(parameters):
    statement(s)
```

`function_name` is known as the *identifier* of the function. Since a function definition is an executable statement its execution *binds* the function name to the function object which can be called later on using the identifier.

`parameters` is an optional list of identifiers that get bound to the values supplied as arguments when the function is called. A function may have an arbitrary number of arguments which are separated by commas.

`statement(s)` – also known as the *function body* – are a nonempty sequence of statements executed each time the function is called. This means a function body cannot be empty, just like any *indented block*.

Here's an example of a simple function definition which purpose is to print `Hello` each time it's called:

```
def greet():
    print("Hello")
```

Now let's call the defined `greet()` function:

```
greet()
# Out: Hello
```

That's another example of a function definition which takes one single argument and displays the passed in value each time the function is called:

```
def greet_two(greeting):
    print(greeting)
```

After that the `greet_two()` function must be called with an argument:

```
greet_two("Howdy")
# Out: Howdy
```

Also you can give a default value to that function argument:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Now you can call the function without giving a value:

```
greet_two()
# Out: Howdy
```

You'll notice that unlike many other languages, you do not need to explicitly declare a return type of the function. Python functions can return values of any type via the `return` keyword. One function can return any number of different types!

```
def many_types(x):
    if x < 0:
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

As long as this is handled correctly by the caller, this is perfectly valid Python code.

A function that reaches the end of execution without a return statement will always return `None`:

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

As mentioned previously a function definition must have a function body, a nonempty sequence of statements. Therefore the `pass` statement is used as function body, which is a null operation – when it is executed, nothing happens. It does what it means, it skips. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

Section 33.2: Defining a function with an arbitrary number of arguments

Arbitrary number of positional arguments:

Defining a function capable of taking an arbitrary number of arguments can be done by prefixing one of the arguments with a `*`

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3)  # Calling it with 3 arguments
```

```
# Out: 1
#      2
#      3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values)  # Calling it with list of values, * expands the list
# Out: 1
#      2
#      3

func()  # Calling it without arguments
# No Output
```

You **can't** provide a default for `args`, for example `func(*args=[1, 2, 3])` will raise a syntax error (won't even compile).

You **can't** provide these by name when calling the function, for example `func(*args=[1, 2, 3])` will raise a `TypeError`.

But if you already have your arguments in an array (or any other `Iterable`), you **can** invoke your function like this: `func(*my_stuff)`.

These arguments (`*args`) can be accessed by index, for example `args[0]` will return the first argument

Arbitrary number of keyword arguments

You can take an arbitrary number of arguments with a name by defining an argument in the definition with **two** `*` in front of it:

```
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3)   # Calling it with 3 arguments
# Out: value1 1
#      value2 2
#      value3 3

func()                               # Calling it without arguments
# No Out put

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)                      # Calling it with a dictionary
# Out: foo 1
#      bar 2
```

You **can't** provide these **without** names, for example `func(1, 2, 3)` will raise a `TypeError`.

`kwargs` is a plain native python dictionary. For example, `args['value1']` will give the value for argument `value1`. Be sure to check beforehand that there is such an argument or a `KeyError` will be raised.

Warning

You can mix these with other optional and required arguments but the order inside the definition matters.

The **positional/keyword** arguments come first. (Required arguments).
Then comes the **arbitrary** `*arg` arguments. (Optional).

Then **keyword-only** arguments come next. (Required).
Finally the **arbitrary keyword** `**kwargs` come. (Optional).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- `arg1` must be given, otherwise a `TypeError` is raised. It can be given as positional (`func(10)`) or keyword argument (`func(arg1=10)`).
- `kwarg1` must also be given, but it can only be provided as keyword-argument: `func(kwarg1=10)`.
- `arg2` and `kwarg2` are optional. If the value is to be changed the same rules as for `arg1` (either positional or keyword) and `kwarg1` (only keyword) apply.
- `*args` catches additional positional parameters. But note, that `arg1` and `arg2` must be provided as positional arguments to pass arguments to `*args`: `func(1, 1, 1, 1)`.
- `**kwargs` catches all additional keyword parameters. In this case any parameter that is not `arg1`, `arg2`, `kwarg1` or `kwarg2`. For example: `func(kwarg3=10)`.
- In Python 3, you can use `*` alone to indicate that all subsequent arguments must be specified as keywords. For instance the `math.isclose` function in Python 3.5 and higher is defined using `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)`, which means the first two arguments can be supplied positionally but the optional third and fourth parameters can only be supplied as keyword arguments.

Python 2.x doesn't support keyword-only parameters. This behavior can be emulated with `kwargs`:

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

Note on Naming

The convention of naming optional positional arguments `args` and optional keyword arguments `kwargs` is just a convention you **can** use any names you like **but** it is useful to follow the convention so that others know what you are doing, *or even yourself later* so please do.

Note on Uniqueness

Any function can be defined with **none or one** `*args` and **none or one** `**kwargs` but not with more than one of each. Also `*args` **must** be the last positional argument and `**kwargs` must be the last parameter. Attempting to use more than one of either **will** result in a Syntax Error exception.

Note on Nesting Functions with Optional Arguments

It is possible to nest such functions and the usual convention is to remove the items that the code has already handled **but** if you are passing down the parameters you need to pass optional positional args with a `*` prefix and optional keyword args with a `**` prefix, otherwise args with be passed as a list or tuple and kwargs as a single dictionary. e.g.:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)
```

```
def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Section 33.3: Lambda (Inline/Anonymous) Functions

The `lambda` keyword creates an inline function that contains a single expression. The value of this expression is what the function returns when invoked.

Consider the function:

```
def greeting():
    return "Hello"
```

which, when called as:

```
print(greeting())
```

prints:

```
Hello
```

This can be written as a lambda function as follows:

```
greet_me = lambda: "Hello"
```

> See note at the bottom of this section regarding the assignment of lambdas to variables. Generally, don't do it.

This creates an inline function with the name `greet_me` that returns `Hello`. Note that you don't write `return` when creating a function with lambda. The value after `:` is automatically returned.

Once assigned to a variable, it can be used just like a regular function:

```
print(greet_me())
```

prints:

```
Hello
```

`lambda`s can take arguments, too:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

returns the string:

```
HELLO
```

They can also take arbitrary number of arguments / keyword arguments, like normal functions.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

prints:

```
hello ('world',) {'world': 'world'}
```

`lambda`s are commonly used for short functions that are convenient to define at the point where they are called (typically with `sorted`, `filter` and `map`).

For example, this line sorts a list of strings ignoring their case and ignoring whitespace at the beginning and at the end:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Sort list just ignoring whitespaces:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Examples with `map`:

```
sorted( map( lambda s: s.strip().upper(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BaZ', 'bAR', 'foo']
```

Examples with numerical lists:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# Out:
# [3, 5, 1, 7]

list( map( lambda x: abs(x), my_list))
# Out:
[3, 4, 2, 5, 1, 7]
```

One can call other functions (with/without arguments) from inside a lambda function.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

prints:

```
hello world
```

This is useful because `lambda` may contain only one expression and by using a subsidiary function one can run multiple statements.

NOTE

Bear in mind that PEP-8 (the official Python style guide) does not recommend assigning lambdas to variables (as we did in the first two examples):

> Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.
>
> Yes:
>
> ```
> def f(x): return 2*x
> ```
>
> No:
>
> ```
> f = lambda x: 2*x
> ```
>
> The first form means that the name of the resulting function object is specifically `f` instead of the generic `<lambda>`. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit `def` statement (i.e. that it can be embedded inside a larger expression).

Section 33.4: Defining a function with optional arguments

Optional arguments can be defined by assigning (using `=`) a default value to the argument-name:

```
def make(action='nothing'):
    return action
```

Calling this function is possible in 3 different ways:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
```

> **Warning**
>
> Mutable types (`list`, `dict`, `set`, etc.) should be treated with care when given as **default** attribute. Any mutation of the default argument will change it permanently. See Defining a function with optional mutable arguments.

Section 33.5: Defining a function with optional mutable arguments

There is a problem when using **optional arguments** with a **mutable default type** (described in Defining a function with optional arguments), which can potentially lead to unexpected behaviour.

Explanation

This problem arises because a function's default arguments are initialised **once**, at the point when the function is *defined*, and **not** (like many other languages) when the function is *called*. The default values are stored inside the function object's `__defaults__` member variable.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

For **immutable** types (see Argument passing and mutability) this is not a problem because there is no way to mutate the variable; it can only ever be reassigned, leaving the original value unchanged. Hence, subsequent are guaranteed to have the same default value. However, for a **mutable** type, the original value can mutate, by making calls to its various member functions. Therefore, successive calls to the function are not guaranteed to have the initial default value.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

> **Note:** Some IDEs like PyCharm will issue a warning when a mutable type is specified as a default attribute.

Solution

If you want to ensure that the default argument is always the one you specify in the function definition, then the solution is to **always** use an immutable type as your default argument.

A common idiom to achieve this when a mutable type is needed as the default, is to use `None` (immutable) as the default argument and then assign the actual default value to the argument variable if it is equal to `None`.

```
def append(elem, to=None):
    if to is None:
        to = []
```

```
    to.append(elem)
    return to
```

Section 33.6: Argument passing and mutability

First, some terminology:

- **argument (*actual* parameter):** the actual variable being passed to a function;
- **parameter (*formal* parameter):** the receiving variable that is used in a function.

In Python, arguments are passed by *assignment* (as opposed to other languages, where arguments can be passed by value/reference/pointer).

- Mutating a parameter will mutate the argument (if the argument's type is mutable).

```
def foo(x):         # here x is the parameter
    x[0] = 9        # This mutates the list labelled by both x and y
    print(x)

y = [4, 5, 6]
foo(y)              # call foo with y as argument
# Out: [9, 5, 6]   # list labelled by x has been mutated
print(y)
# Out: [9, 5, 6]   # list labelled by y has been mutated too
```

- Reassigning the parameter won't reassign the argument.

```
def foo(x):         # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9        # This mutates the list labelled by both x and y
    x = [1, 2, 3]   # x is now labeling a different list (y is unaffected)
    x[2] = 8        # This mutates x's list, not y's list

y = [4, 5, 6]       # y is the argument, x is the parameter
foo(y)              # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

In Python, we don't really assign values to variables, instead we *bind* (i.e. assign, attach) variables (considered as *names*) to objects.

- **Immutable:** Integers, strings, tuples, and so on. All operations make copies.
- **Mutable:** Lists, dictionaries, sets, and so on. Operations may or may not mutate.

```
x = [3, 1, 9]
y = x
x.append(5)     # Mutates the list labelled by x and y, both x and y are bound to [3, 1, 9]
x.sort()        # Mutates the list labelled by x and y (in-place sorting)
x = x + [4]     # Does not mutate the list (makes a copy for x only, not y)
z = x           # z is x ([1, 3, 9, 4])
x += [6]        # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x)   # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
```

```
# Out: [1, 3, 5, 9, 4, 6]
```

Section 33.7: Returning values from functions

Functions can `return` a value that you can use directly:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

or save the value for later use:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

or use the value for any operations:

```
print(give_me_five() + 10)
# Out: 15
```

If `return` is encountered in the function the function will be exited immediately and subsequent operations will not be evaluated:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

You can also `return` multiple values (in the form of a tuple):

```
def give_me_two_fives():
    return 5, 5  # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

A function with *no* `return` statement implicitly returns `None`. Similarly a function with a `return` statement, but no return value or variable returns `None`.

Section 33.8: Closure

Closures in Python are created by function calls. Here, the call to `makeInc` creates a binding for `x` that is referenced inside the function `inc`. Each call to `makeInc` creates a new instance of this function, but each instance has a link to a different binding of `x`.

```
def makeInc(x):
  def inc(y):
     # x is "attached" in the definition of inc
```

```
    return y + x

  return inc

incOne = makeInc(1)
incFive = makeInc(5)

incOne(5) # returns 6
incFive(5) # returns 10
```

Notice that while in a regular closure the enclosed function fully inherits all variables from its enclosing environment, in this construct the enclosed function has only read access to the inherited variables but cannot make assignments to them

```
def makeInc(x):
  def inc(y):
    # incrementing x is not allowed
    x += y
    return x

  return inc

incOne = makeInc(1)
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 offers the `nonlocal` statement (Nonlocal Variables) for realizing a full closure with nested functions.

Python 3.x `Version ≥ 3.0`

```
def makeInc(x):
  def inc(y):
    nonlocal x
    # now assigning a value to x is allowed
    x += y
    return x

  return inc

incOne = makeInc(1)
incOne(5) # returns 6
```

Section 33.9: Forcing the use of named parameters

All parameters specified after the first asterisk in the function signature are keyword-only.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

In Python 3 it's possible to put a single asterisk in the function signature to ensure that the remaining arguments may only be passed using keyword arguments.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() takes 2 positional arguments but 3 were given
```

```
f(1, 2, c=3)
# No error
```

Section 33.10: Nested functions

Functions in python are first-class objects. They can be defined in any scope

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Functions capture their enclosing scope can be passed around like any other sort of object

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Section 33.11: Recursion limit

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))

cursing(0)
# Out: I recursed 1083 times!
```

It is possible to change the recursion depth limit by using `sys.setrecursionlimit(limit)` and check this limit by `sys.getrecursionlimit()`.

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

From Python 3.5, the exception is a `RecursionError`, which is derived from `RuntimeError`.

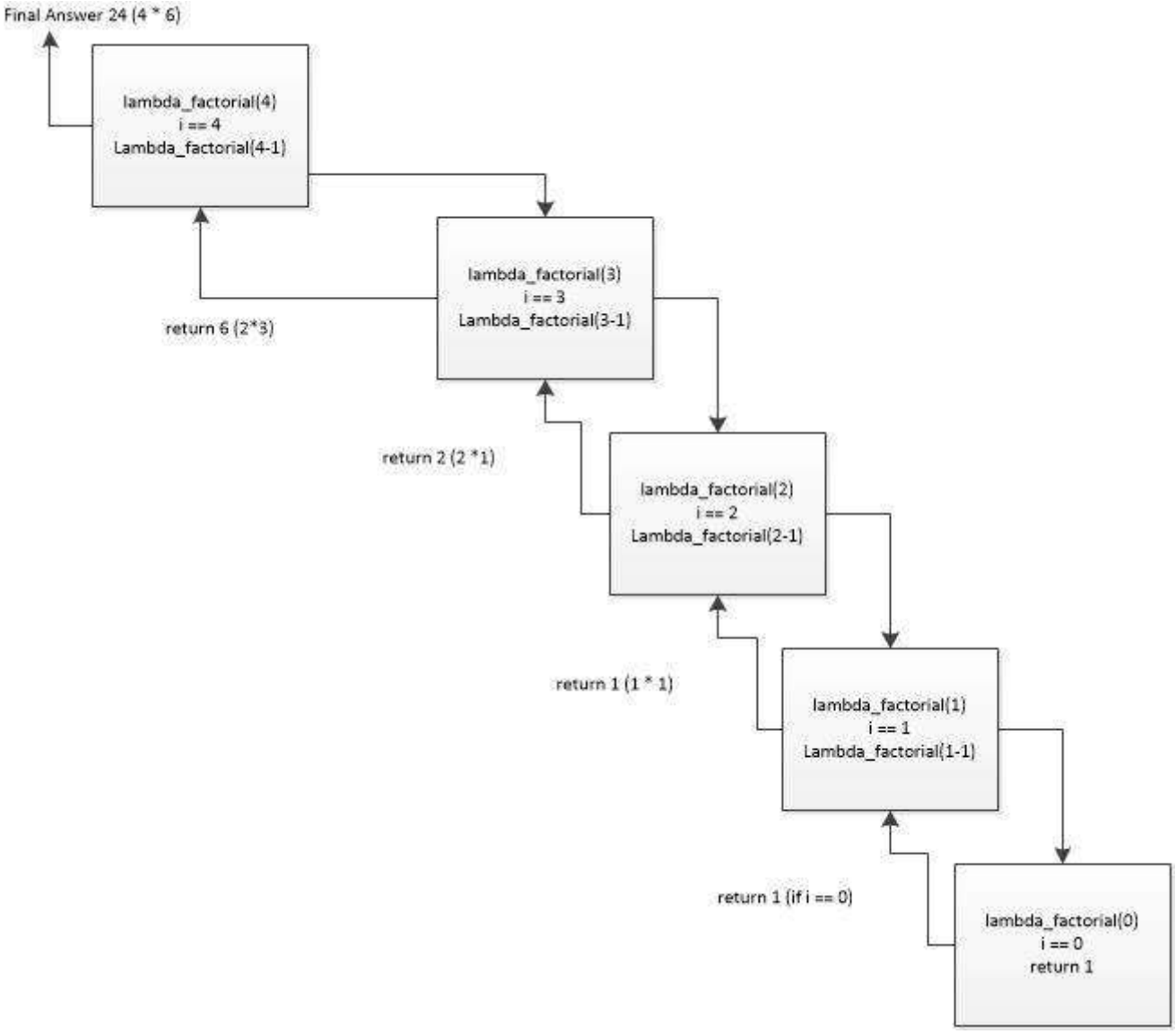
Section 33.12: Recursive Lambda using assigned variable

One method for creating recursive lambda functions involves assigning the function to a variable and then referencing that variable within the function itself. A common example of this is the recursive calculation of the factorial of a number - such as shown in the following code:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

Description of code

The lambda function, through its variable assignment, is passed a value (4) which it evaluates and returns 1 if it is 0 or else it returns the current value (`i`) * another calculation by the lambda function of the value - 1 (`i-1`). This continues until the passed value is decremented to 0 (`return 1`). A process which can be visualized as:



Section 33.13: Recursive functions

A recursive function is a function that calls itself in its definition. For example the mathematical function, factorial, defined by `factorial(n) = n*(n-1)*(n-2)*...*3*2*1`. can be programmed as

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

the outputs here are:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

as expected. Notice that this function is recursive because the second `return factorial(n-1)`, where the function calls itself in its definition.

Some recursive functions can be implemented using lambda, the factorial function using lambda would be something like this:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

The function outputs the same as above.

Section 33.14: Defining a function with arguments

Arguments are defined in parentheses after the function name:

```
def divide(dividend, divisor):  # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

The function name and its list of arguments are called the *signature* of the function. Each named argument is effectively a local variable of the function.

When calling the function, give values for the arguments by listing them in order

```
divide(10, 2)
# output: 5
```

or specify them in any order using the names from the function definition:

```
divide(divisor=2, dividend=10)
# output: 5
```

Section 33.15: Iterable and dictionary unpacking

Functions allow you to specify these types of parameters: positional, named, variable positional, Keyword args (kwargs). Here is a clear and concise use of each type.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
```

```
    print(a, b, c, d, args, kwargs)

>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}


>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'


>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'



>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}
```

```
>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'


>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}


>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

# Positional arguments take priority over any other form of argument passing
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

Section 33.16: Defining a function with multiple arguments

One can give a function as many arguments as one wants, the only fixed rules are that each argument name must be unique and that optional arguments must be after the not-optional ones:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

When calling the function you can either give each keyword without the name but then the order matters:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

Or combine giving the arguments with name and without. Then the ones with name must follow those without but the order of the ones with name doesn't matter:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Chapter 34: Defining functions with list arguments

Section 34.1: Function and Call

Lists as arguments are just another variable:

```
def func(myList):
    for item in myList:
        print(item)
```

and can be passed in the function call itself:

```
func([1,2,3,5,7])

1
2
3
5
7
```

Or as a variable:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Chapter 35: Functional Programming in Python

Functional programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input.below are functional techniques common to many languages: such as lambda, map, reduce.

Section 35.1: Lambda Function

An anonymous, inlined function defined with lambda. The parameters of the lambda are defined to the left of the colon. The function body is defined to the right of the colon. The result of running the function body is (implicitly) returned.

```
s=lambda x:x*x
s(2)    =>4
```

Section 35.2: Map Function

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection.

This is a simple map that takes a list of names and returns a list of the lengths of those names:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)    =>[4, 4, 3]
```

Section 35.3: Reduce Function

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

This is a simple reduce. It returns the sum of all the items in the collection.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)    =>10
```

Section 35.4: Filter Function

Filter takes a function and a collection. It returns a collection of every item for which the function returned True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Chapter 36: Partial functions

Param	details
x	the number to be raised
y	the exponent
raise	the function to be specialized

As you probably know if you came from OOP school, specializing an abstract class and use it is a practice you should keep in mind when writing your code.

What if you could define an abstract function and specialize it in order to create different versions of it? Thinks it as a sort of *function Inheritance* where you bind specific params to make them reliable for a specific scenario.

Section 36.1: Raise the power

Let's suppose we want raise *x* to a number *y*.

You'd write this as:

```
def raise_power(x, y):
    return x**y
```

What if your *y* value can assume a finite set of values?

Let's suppose *y* can be one of *[3,4,5]* and let's say you don't want offer end user the possibility to use such function since it is very computationally intensive. In fact you would check if provided *y* assumes a valid value and rewrite your function as:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Messy? Let's use the abstract form and specialize it to all three cases: let's implement them **partially**.

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

What happens here? We fixed the *y* params and we defined three different functions.

No need to use the abstract function defined above (you could make it *private*) but you could use **partial applied** functions to deal with raising a number to a fixed value.

Chapter 37: Decorators

Parameter	Details
f	The function to be decorated (wrapped)

Decorator functions are software design patterns. They dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the decorated function. When used correctly, decorators can become powerful tools in the development process. This topic covers implementation and applications of decorator functions in Python.

Section 37.1: Decorator function

Decorators augment the behavior of other functions or methods. Any function that takes a function as a parameter and returns an augmented function can be used as a **decorator**.

```
# This simplest decorator does nothing to the function being decorated. Such
# minimal decorators can occasionally be used as a kind of code markers.
def super_secret_function(f):
    return f

@super_secret_function
def my_function():
    print("This is my secret function.")
```

The `@`-notation is syntactic sugar that is equivalent to the following:

```
my_function = super_secret_function(my_function)
```

It is important to bear this in mind in order to understand how the decorators work. This "unsugared" syntax makes it clear why the decorator function takes a function as an argument, and why it should return another function. It also demonstrates what would happen if you *don't* return a function:

```
def disabled(f):
    """
    This function returns nothing, and hence removes the decorated function
    from the local scope.
    """
    pass

@disabled
def my_function():
    print("This function can no longer be called...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Thus, we usually define a *new function* inside the decorator and return it. This new function would first do something that it needs to do, then call the original function, and finally process the return value. Consider this simple decorator function that prints the arguments that the original function receives, then calls it.

```
#This is the decorator
def print_args(func):
    def inner_func(*args, **kwargs):
        print(args)
        print(kwargs)
```

```
        return func(*args, **kwargs) #Call the original function with its arguments.
    return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#Output:
# (3,5) - This is actually the 'args' that the function receives.
# {} - This is the 'kwargs', empty because we didn't specify keyword arguments.
# 15 - The result of the function.
```

Section 37.2: Decorator class

As mentioned in the introduction, a decorator is a function that can be applied to another function to augment its behavior. The syntactic sugar is equivalent to the following: `my_func = decorator(my_func)`. But what if the `decorator` was instead a class? The syntax would still work, except that now `my_func` gets replaced with an instance of the `decorator` class. If this class implements the `__call__()` magic method, then it would still be possible to use `my_func` as if it was a function:

```
class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.
```

Note that a function decorated with a class decorator will no longer be considered a "function" from type-checking perspective:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Decorating Methods

For decorating methods you need to define an additional `__get__`-method:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
```

```
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Inside the decorator.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Return a Method if it is called on an instance
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()
```

> Inside the decorator.

Warning!

Class Decorators only produce one instance for a specific function so decorating a method with a class decorator will share the same decorator between all instances of that class:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1   # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls   # 1
b = Test()
b.do_something()
b.do_something.ncalls   # 2
```

Section 37.3: Decorator with arguments (decorator factory)

A decorator takes just one argument: the function to be decorated. There is no way to pass other arguments.

But additional arguments are often desired. The trick is then to make a function which takes arbitrary arguments

and returns a decorator.

Decorator functions

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()
```

> The decorator wants to tell you: Hello World

Important Note:

With such decorator factories you **must** call the decorator with a pair of parentheses:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

> TypeError: decorator() missing 1 required positional argument: 'func'

Decorator classes

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

> Inside the decorator with arguments (10,)

Section 37.4: Making a decorator look like the decorated function

Decorators normally strip function metadata as they aren't the same. This can cause problems when using meta-programming to dynamically access function metadata. Metadata also includes function's docstrings and its name. `functools.wraps` makes the decorated function look like the original function by copying several attributes to the wrapper function.

```
from functools import wraps
```

The two methods of wrapping a decorator are achieving the same thing in hiding that the original function has been decorated. There is no reason to prefer the function version to the class version unless you're already using one over the other.

As a function

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

> 'test'

As a class

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

> 'Docstring of test.'

Section 37.5: Using a decorator to time a function

```
import time
def timer(func):
    def inner(*args, **kwargs):
```

```
        t1 = time.time()
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Runtime took {0} seconds'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    #do stuff


example_function()
```

Section 37.6: Create singleton class with a decorator

A singleton is a pattern that restricts the instantiation of a class to one instance/object. Using a decorator, we can define a class as a singleton by forcing the class to either return an existing instance of the class or create a new instance (if it doesn't exist).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

This decorator can be added to any class declaration and will make sure that at most one instance of the class is created. Any subsequent calls will return the already existing class instance.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass()  # prints: Created!
instance = SomeSingletonClass()  # doesn't print anything
print(instance.x)                # 2

instance.x = 3
print(SomeSingletonClass().x)    # 3
```

So it doesn't matter whether you refer to the class instance via your local variable or whether you create another "instance", you always get the same object.

Chapter 38: Classes

Python offers itself not only as a popular scripting language, but also supports the object-oriented programming paradigm. Classes describe data and provide methods to manipulate that data, all encompassed under a single object. Furthermore, classes allow for abstraction by separating concrete implementation details from abstract representations of data.

Code utilizing classes is generally easier to read, understand, and maintain.

Section 38.1: Introduction to classes

A class, functions as a template that defines the basic characteristics of a particular object. Here's an example:

```
class Person(object):
    """A simple class."""                          # docstring
    species = "Homo Sapiens"                       # class attribute

    def __init__(self, name):                      # special method
        """This is the initializer. It's a special
        method (see below).
        """
        self.name = name                           # instance attribute

    def __str__(self):                             # special method
        """This method is run when Python tries
        to cast the object to a string. Return
        this string when using print(), etc.
        """
        return self.name

    def rename(self, renamed):                     # regular method
        """Reassign and print the name attribute."""
        self.name = renamed
        print("Now my name is {}".format(self.name))
```

There are a few things to note when looking at the above example.

1. The class is made up of *attributes* (data) and *methods* (functions).
2. Attributes and methods are simply defined as normal variables and functions.
3. As noted in the corresponding docstring, the `__init__()` method is called the *initializer*. It's equivalent to the constructor in other object oriented languages, and is the method that is first run when you create a new object, or new instance of the class.
4. Attributes that apply to the whole class are defined first, and are called *class attributes*.
5. Attributes that apply to a specific instance of a class (an object) are called *instance attributes*. They are generally defined inside `__init__()`; this is not necessary, but it is recommended (since attributes defined outside of `__init__()` run the risk of being accessed before they are defined).
6. Every method, included in the class definition passes the object in question as its first parameter. The word `self` is used for this parameter (usage of `self` is actually by convention, as the word `self` has no inherent meaning in Python, but this is one of Python's most respected conventions, and you should always follow it).
7. Those used to object-oriented programming in other languages may be surprised by a few things. One is that Python has no real concept of `private` elements, so everything, by default, imitates the behavior of the C++/Java `public` keyword. For more information, see the "Private Class Members" example on this page.
8. Some of the class's methods have the following form: `__functionname__(self, other_stuff)`. All such methods are called "magic methods" and are an important part of classes in Python. For instance, operator overloading in Python is implemented with magic methods. For more information, see the relevant

documentation.

Now let's make a few instances of our `Person` class!

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

We currently have three `Person` objects, `kelly`, `joseph`, and `john_doe`.

We can access the attributes of the class from each instance using the dot operator `.` Note again the difference between class and instance attributes:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

We can execute the methods of the class using the same dot operator `.`:

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Section 38.2: Bound, unbound, and static methods

The idea of bound and unbound methods was removed in Python 3. In Python 3 when you declare a method within a class, you are using a `def` keyword, thus creating a function object. This is a regular function, and the surrounding class works as its namespace. In the following example we declare method `f` within class `A`, and it becomes a function `A.f`:

Python 3.x `Version ≥ 3.0`

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

In Python 2 the behavior was different: function objects within the class were implicitly replaced with objects of type `instancemethod`, which were called *unbound methods* because they were not bound to any particular class instance. It was possible to access the underlying function using `.__func__` property.

Python 2.x `Version ≥ 2.3`

```
A.f
```

```
# <unbound method A.f>   (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

The latter behaviors are confirmed by inspection - methods are recognized as functions in Python 3, while the distinction is upheld in Python 2.

Python 3.x `Version ≥ 3.0`

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x `Version ≥ 2.3`

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

In both versions of Python function/method `A.f` can be called directly, provided that you pass an instance of class `A` as the first argument.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#                      A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Now suppose `a` is an instance of class `A`, what is `a.f` then? Well, intuitively this should be the same method `f` of class `A`, only it should somehow "know" that it was applied to the object `a` – in Python this is called method *bound* to `a`.

The nitty-gritty details are as follows: writing `a.f` invokes the magic `__getattribute__` method of `a`, which first checks whether `a` has an attribute named `f` (it doesn't), then checks the class `A` whether it contains a method with such a name (it does), and creates a new object `m` of type `method` which has the reference to the original `A.f` in `m.__func__`, and a reference to the object `a` in `m.__self__`. When this object is called as a function, it simply does the following: `m(...) => m.__func__(m.__self__, ...)`. Thus this object is called a **bound method** because when invoked it knows to supply the object it was bound to as the first argument. (These things work same way in Python 2 and 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f  # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
```

```
a.f is a.f  # True
```

Finally, Python has **class methods** and **static methods** – special kinds of methods. Class methods work the same way as regular methods, except that when invoked on an object they bind to the *class* of the object instead of to the object. Thus `m.__self__ = type(a)`. When you call such bound method, it passes the class of `a` as the first argument. Static methods are even simpler: they don't bind anything at all, and simply return the underlying function without any transformations.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
        print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Note that class methods are bound to the class even when accessed on the instance:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

It is worth noting that at the lowest level, functions, methods, staticmethods, etc. are actually descriptors that invoke `__get__`, `__set__` and optionally `__del__` special methods. For more details on classmethods and staticmethods:

- What is the difference between @staticmethod and @classmethod in Python?
- Meaning of @classmethod and @staticmethod for beginner?

Section 38.3: Basic inheritance

Inheritance in Python is based on similar ideas used in other object oriented languages like Java, C++ etc. A new class can be derived from an existing class as follows.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

The `BaseClass` is the already existing (*parent*) class, and the `DerivedClass` is the new (*child*) class that inherits (or *subclasses*) attributes from `BaseClass`. **Note**: As of Python 2.2, all classes implicitly inherit from the `object` class, which is the base class for all built-in types.

We define a parent `Rectangle` class in the example below, which implicitly inherits from `object`:

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

The `Rectangle` class can be used as a base class for defining a `Square` class, as a square is a special case of rectangle.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super(Square, self).__init__(s, s)
        self.s = s
```

The `Square` class will automatically inherit all attributes of the `Rectangle` class as well as the object class. `super()` is used to call the `__init__()` method of `Rectangle` class, essentially calling any overridden method of the base class. **Note**: in Python 3, `super()` does not require arguments.

Derived class objects can access and modify the attributes of its base classes:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Built-in functions that work with inheritance

`issubclass(DerivedClass, BaseClass)`: returns `True` if `DerivedClass` is a subclass of the `BaseClass`

`isinstance(s, Class)`: returns `True` if s is an instance of `Class` or any of the derived classes of `Class`

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
```

```
# Output: False
# A rectangle is not a square

isinstance(s, Rectangle)
# Output: True
# A square is a rectangle
isinstance(s, Square)
# Output: True
```

Section 38.4: Monkey Patching

In this case, "monkey patching" means adding a new variable or method to a class after it's been defined. For instance, say we defined class `A` as

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

But now we want to add another function later in the code. Suppose this function is as follows.

```
def get_num(self):
    return self.num
```

But how do we add this as a method in `A`? That's simple we just essentially place that function into `A` with an assignment statement.

```
A.get_num = get_num
```

Why does this work? Because functions are objects just like any other object, and methods are functions that belong to the class.

The function `get_num` shall be available to all existing (already created) as well to the new instances of `A`

These additions are available on all instances of that class (or its subclasses) automatically. For example:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Note that, unlike some other languages, this technique does not work for certain built-in types, and it is not considered good style.

Section 38.5: New-style vs. old-style classes

Python 2.x `Version ≥ 2.2.0`

New-style classes were introduced in Python 2.2 to unify *classes* and *types*. They inherit from the top-level `object`

type. *A new-style class is a user-defined type*, and is very similar to built-in types.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Old-style classes do **not** inherit from `object`. Old-style instances are always implemented with a built-in `instance` type.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x `Version ≥ 3.0.0`

In Python 3, old-style classes were removed.

New-style classes in Python 3 implicitly inherit from `object`, so there is no need to specify `MyClass(object)` anymore.

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Section 38.6: Class methods: alternate initializers

Class methods present alternate ways to build instances of classes. To illustrate, let's look at an example.

Let's suppose we have a relatively simple `Person` class:

```
class Person(object):
```

```
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

It might be handy to have a way to build instances of this class specifying a full name instead of first and last name separately. One way to do this would be to have `last_name` be an optional parameter, and assuming that if it isn't given, we passed the full name in:

```
class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

However, there are two main problems with this bit of code:

1. The parameters `first_name` and `last_name` are now misleading, since you can enter a full name for `first_name`. Also, if there are more cases and/or more parameters that have this kind of flexibility, the if/elif/else branching can get annoying fast.
2. Not quite as important, but still worth pointing out: what if `last_name` is `None`, but `first_name` doesn't split into two or more things via spaces? We have yet another layer of input validation and/or exception handling...

Enter class methods. Rather than having a single initializer, we will create a separate initializer, called `from_full_name`, and decorate it with the (built-in) `classmethod` decorator.

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Notice `cls` instead of `self` as the first argument to `from_full_name`. Class methods are applied to the overall class, *not* an instance of a given class (which is what `self` usually denotes). So, if `cls` is our `Person` class, then the returned value from the `from_full_name` class method is `Person(first_name, last_name, age)`, which uses `Person`'s `__init__` to create an instance of the `Person` class. In particular, if we were to make a subclass `Employee` of `Person`, then `from_full_name` would work in the `Employee` class as well.

To show that this works as expected, let's create instances of `Person` in more than one way without the branching in `__init__`:

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Other references:

- Python @classmethod and @staticmethod for beginner?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Section 38.7: Multiple Inheritance

Python uses the C3 linearization algorithm to determine the order in which to resolve class attributes, including methods. This is known as the Method Resolution Order (MRO).

Here's a simple example:

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar' # we won't see this.
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Now if we instantiate FooBar, if we look up the foo attribute, we see that Foo's attribute is found first

```
fb = FooBar()
```

and

```
>>> fb.foo
'attr foo of Foo'
```

Here's the MRO of FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

It can be simply stated that Python's MRO algorithm is

1. Depth first (e.g. `FooBar` then `Foo`) unless
2. a shared parent (`object`) is blocked by a child (`Bar`) and
3. no circular relationships allowed.

That is, for example, Bar cannot inherit from FooBar while FooBar inherits from Bar.

For a comprehensive example in Python, see the wikipedia entry.

Another powerful feature in inheritance is `super`. super can fetch parent classes features.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Multiple inheritance with init method of class, when every class has own init method then we try for multiple inheritance then only init method get called of class which is inherit first.

for below example only Foo class **init** method getting called **Bar** class init not getting called

```
    class Foo(object):
        def __init__(self):
            print "foo init"

    class Bar(object):
        def __init__(self):
            print "bar init"

    class FooBar(Foo, Bar):
        def __init__(self):
            print "foobar init"
            super(FooBar, self).__init__()

    a = FooBar()
```

Output:

```
    foobar init
    foo init
```

But it doesn't mean that **Bar** class is not inherit. Instance of final **FooBar** class is also instance of **Bar** class and **Foo** class.

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
```

```
print isinstance(a,Bar)
```

Output:

```
True
True
True
```

Section 38.8: Properties

Python classes support **properties**, which look like regular object variables, but with the possibility of attaching custom behavior and documentation.

```
class MyClass(object):

    def __init__(self):
       self._my_string = ""

    @property
    def string(self):
        """A profoundly important string."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
               "Give me a string, not a %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

The object's of class `MyClass` will *appear* to have a property `.string`, however it's behavior is now tightly controlled:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

As well as the useful syntax as above, the property syntax allows for validation, or other augmentations to be added to those attributes. This could be especially useful with public APIs - where a level of help should be given to the user.

Another common use of properties is to enable the class to present 'virtual attributes' - attributes which aren't actually stored but are computed only when requested.

```
class Character(object):
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp

    # Make hp read only by not providing a set method
    @property
    def hp(self):
        return self._hp
```

```
    # Make name read only by not providing a set method
    @property
    def name(self):
        return self.name

    def take_damage(self, damage):
        self.hp -= damage
        self.hp = 0 if self.hp <0 else self.hp

    @property
    def is_alive(self):
        return self.hp != 0

    @property
    def is_wounded(self):
        return self.hp < self.max_hp if self.hp > 0 else False

    @property
    def is_dead(self):
        return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
# hp attribute is read only.

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )

bilbo.hp
# out : 50

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : True
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )
bilbo.hp
# out : 0

bilbo.is_alive
# out : False
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : True
```

Section 38.9: Default values for instance variables

If the variable contains a value of an immutable type (e.g. a string) then it is okay to assign a default value like this

```
class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width  * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

One needs to be careful when initializing mutable objects such as lists in the constructor. Consider the following example:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well
```

This behavior is caused by the fact that in Python default parameters are bound at function execution and not at function declaration. To get a default instance variable that's not shared among instances, one should use a construct like this:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed
```

See also Mutable Default Arguments and "Least Astonishment" and the Mutable Default Argument.

Section 38.10: Class and instance variables

Instance variables are unique for each instance, while class variables are shared by all instances.

```
class C:
    x = 2  # class variable
```

```
    def __init__(self, y):
        self.y = y  # instance variable

C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3

c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Class variables can be accessed on instances of this class, but assigning to the class attribute will create an instance variable which shadows the class variable

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Note that *mutating* class variables from instances can lead to some unexpected consequences.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item)  # note that this is not an assignment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

Section 38.11: Class composition

Class composition allows explicit relations between objects. In this example, people live in cities that belong to countries. Composition allows people to access the number of all people living in their country:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)
```

```
class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople


    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
                person(i).join_city(self)


class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

# 15
```

Section 38.12: Listing All Class Members

The `dir()` function can be used to get a list of the members of a class:

```
dir(Class)
```

For example:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

It is common to look only for "non-magic" members. This can be done using a simple comprehension that lists members with names not starting with `__`:

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Caveats:

Classes can define a `__dir__()` method. If that method exists calling `dir()` will call `__dir__()`, otherwise Python will try to create a list of members of the class. This means that the dir function can have unexpected results. Two quotes of importance from the official python documentation:

> If the object does not provide **dir**(), the function tries its best to gather information from the object's **dict** attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom **getattr**().

> **Note:** Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

Section 38.13: Singleton class

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see here.

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
        return it

    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__.upper())

    def __eq__(self, other):
         return other is self
```

Another method is to decorate your class. Following the example from this answer create a Singleton class:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
```

```
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance

    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
        return isinstance(inst, self._decorated)
```

To use you can use the `Instance` method

```
@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single
```

Section 38.14: Descriptors and Dotted Lookups

Descriptors are objects that are (usually) attributes of classes and that have any of `__get__`, `__set__`, or `__delete__` special methods.

Data Descriptors have any of `__set__`, or `__delete__`

These can control the dotted lookup on an instance, and are used to implement functions, `staticmethod`, `classmethod`, and `property`. A dotted lookup (e.g. instance `foo` of class `Foo` looking up attribute `bar` - i.e. `foo.bar`) uses the following algorithm:

1. `bar` is looked up in the class, `Foo`. If it is there and it is a **Data Descriptor**, then the data descriptor is used. That's how `property` is able to control access to data in an instance, and instances cannot override this. If a **Data Descriptor** is not there, then
2. `bar` is looked up in the instance `__dict__`. This is why we can override or block methods being called from an instance with a dotted lookup. If `bar` exists in the instance, it is used. If not, we then
3. look in the class `Foo` for `bar`. If it is a **Descriptor**, then the descriptor protocol is used. This is how functions (in this context, unbound methods), `classmethod`, and `staticmethod` are implemented. Else it simply returns the object there, or there is an `AttributeError`

Chapter 39: Metaclasses

Metaclasses allow you to deeply modify the behaviour of Python classes (in terms of how they're defined, instantiated, accessed, and more) by replacing the `type` metaclass that new classes use by default.

Section 39.1: Basic Metaclasses

When `type` is called with three arguments it behaves as the (meta)class it is, and creates a new instance, ie. it produces a new class/type.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__               # <type 'type'>
Dummy().__class__.__class__   # <type 'type'>
```

It is possible to subclass `type` to create an custom metaclass.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # call the base initializer
        type.__init__(cls, name, bases, dict)

        # perform custom initialization...
        cls.__custom_attribute__ = 2
```

Now, we have a new custom `mytype` metaclass which can be used to create classes in the same manner as `type`.

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__               # <class '__main__.mytype'>
MyDummy().__class__.__class__   # <class '__main__.mytype'>
MyDummy.__custom_attribute__    # 2
```

When we create a new class using the `class` keyword the metaclass is by default chosen based on upon the baseclasses.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

In the above example the only baseclass is `object` so our metaclass will be the type of `object`, which is `type`. It is possible override the default, however it depends on whether we use Python 2 or Python 3:

Python 2.x `Version ≤ 2.7`

A special class-level attribute `__metaclass__` can be used to specify the metaclass.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x `Version ≥ 3.0`

A special `metaclass` keyword argument specify the metaclass.

```
class MyDummy(metaclass=mytype):
```

```
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Any keyword arguments (except `metaclass`) in the class declaration will be passed to the metaclass. Thus `class MyDummy(metaclass=mytype, x=2)` will pass `x=2` as a keyword argument to the `mytype` constructor.

Read this in-depth description of python meta-classes for more details.

Section 39.2: Singletons using metaclasses

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see here.

```
class SingletonType(type):
    def __call__(cls, *args, **kwargs):
        try:
            return cls.__instance
        except AttributeError:
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)
            return cls.__instance
```

Python 2.x `Version ≤ 2.7`

```
class MySingleton(object):
    __metaclass__ = SingletonType
```

Python 3.x `Version ≥ 3.0`

```
class MySingleton(metaclass=SingletonType):
    pass
```

```
MySingleton() is MySingleton()  # True, only one instantiation occurs
```

Section 39.3: Using a metaclass

Metaclass syntax

Python 2.x `Version ≤ 2.7`

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x `Version ≥ 3.0`

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Python 2 and 3 compatibility with `six`

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Section 39.4: Introduction to Metaclasses

What is a metaclass?

In Python, everything is an object: integers, strings, lists, even functions and classes themselves are objects. And every object is an instance of a class.

To check the class of an object x, one can call `type(x)`, so:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Most classes in python are instances of `type`. `type` itself is also a class. Such classes whose instances are also classes are called metaclasses.

The Simplest Metaclass

OK, so there is already one metaclass in Python: `type`. Can we create another one?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

That does not add any functionality, but it is a new metaclass, see that MyClass is now an instance of SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

A Metaclass which does Something

A metaclass which does something usually overrides `type`'s `__new__`, to modify some properties of the class to be created, before calling the original `__new__` which creates the class:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls is this class
        # name is the name of the class to be created
        # parents is the list of the class's parent classes
        # dct is the list of class's attributes (methods, static variables)

        # here all of the attributes can be modified before creating the class, e.g.

        dct['x'] = 8  # now the class will have a static variable x = 8

        # return value is the new class. super will take care of that
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

Section 39.5: Custom functionality with metaclasses

Functionality in metaclasses can be changed so that whenever a class is built, a string is printed to standard output, or an exception is thrown. This metaclass will print the name of the class being built.

```
class VerboseMetaclass(type):
```

```
    def __new__(cls, class_name, class_parents, class_dict):
        print("Creating class ", class_name)
        new_class = super().__new__(cls, class_name, class_parents, class_dict)
        return new_class
```

You can use the metaclass like so:

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

The standard output will be:

```
Creating class Spam
[insert example string here]
```

Section 39.6: The default metaclass

You may have heard that everything in Python is an object. It is true, and all objects have a class:

```
>>> type(1)
int
```

The literal 1 is an instance of `int`. Let's declare a class:

```
>>> class Foo(object):
...     pass
...
```

Now let's instantiate it:

```
>>> bar = Foo()
```

What is the class of `bar`?

```
>>> type(bar)
Foo
```

Nice, `bar` is an instance of `Foo`. But what is the class of `Foo` itself?

```
>>> type(Foo)
type
```

Ok, `Foo` itself is an instance of `type`. How about `type` itself?

```
>>> type(type)
type
```

So what is a metaclass? For now let's pretend it is just a fancy name for the class of a class. Takeaways:

- Everything is an object in Python, so everything has a class
- The class of a class is called a metaclass
- The default metaclass is `type`, and by far it is the most common metaclass

But why should you know about metaclasses? Well, Python itself is quite "hackable", and the concept of metaclass is important if you are doing advanced stuff like meta-programming or if you want to control how your classes are initialized.

Chapter 40: String Formatting

When storing and transforming data for humans to see, string formatting can become very important. Python offers a wide variety of string formatting methods which are outlined in this topic.

Section 40.1: Basics of String Formatting

```
foo = 1
bar = 'bar'
baz = 3.14
```

You can use `str.format` to format output. Bracket pairs are replaced with arguments in the order in which the arguments are passed:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

Indexes can also be specified inside the brackets. The numbers correspond to indexes of the arguments passed to the `str.format` function (0-based).

```
print('{0}, {1}, {2}, and {1}'.format(foo, bar, baz))
# Out: "1, bar, 3.14, and bar"
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Out: index out of range error
```

Named arguments can be also used:

```
print("X value is: {x_val}. Y value is: {y_val}.".format(x_val=2, y_val=3))
# Out: "X value is: 2. Y value is: 3."
```

Object attributes can be referenced when passed into `str.format`:

```
class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('My value is: {0.value}'.format(my_value))  # "0" is optional
# Out: "My value is: 6"
```

Dictionary keys can be used as well:

```
my_dict = {'key': 6, 'other_key': 7}
print("My other key is: {0[other_key]}".format(my_dict))  # "0" is optional
# Out: "My other key is: 7"
```

Same applies to list and tuple indices:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

> Note: In addition to `str.format`, Python also provides the modulo operator `%`--also known as the *string formatting* or *interpolation operator* (see PEP 3101)--for formatting strings. `str.format` is a successor of `%`

> and it offers greater flexibility, for instance by making it easier to carry out multiple substitutions.

In addition to argument indexes, you can also include a *format specification* inside the curly brackets. This is an expression that follows special rules and must be preceded by a colon (`:`). See the docs for a full description of format specification. An example of format specification is the alignment directive `:~^20` (`^` stands for center alignment, total width 20, fill with ~ character):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

`format` allows behaviour not possible with `%`, for example repetition of arguments:

```
t = (12, 45, 22222, 103, 6)
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)
# Out: 12 22222 45 22222 103 22222 6 22222
```

As `format` is a function, it can be used as an argument in other functions:

```
number_list = [12,45,78]
print map('the number is {}'.format, number_list)
# Out: ['the number is 12', 'the number is 45', 'the number is 78']


from datetime import datetime,timedelta

once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)
delta = timedelta(days=13, hours=8,  minutes=20)

gen = (once_upon_a_time + x * delta for x in xrange(5))

print '\n'.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))
#Out: 2010-07-01 12:00:00
#     2010-07-14 20:20:00
#     2010-07-28 04:40:00
#     2010-08-10 13:00:00
#     2010-08-23 21:20:00
```

Section 40.2: Alignment and padding

Python 2.x `Version ≥ 2.6`

The `format()` method can be used to change the alignment of the string. You have to do it with a format expression of the form `:[fill_char][align_operator][width]` where `align_operator` is one of:

- `<` forces the field to be left-aligned within `width`.
- `>` forces the field to be right-aligned within `width`.
- `^` forces the field to be centered within `width`.
- `=` forces the padding to be placed after the sign (numeric types only).

`fill_char` (if omitted default is whitespace) is the character used for the padding.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'
```

```
'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Note: you could achieve the same results using the string functions `ljust()`, `rjust()`, `center()`, `zfill()`, however these functions are deprecated since version 2.5.

Section 40.3: Format literals (f-string)

Literal format strings were introduced in PEP 498 (Python3.6 and upwards), allowing you to prepend `f` to the beginning of a string literal to effectively apply `.format` to it with all variables in the current scope.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

This works with more advanced format strings too, including alignment and dot notation.

```
>>> f'{foo:^7s}'
'  bar  '
```

Note: The `f''` does not denote a particular type like `b''` for `bytes` or `u''` for `unicode` in python2. The formatting is immediately applied, resulting in a normal string.

The format strings can also be *nested*:

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'*************$478.23'
```

The expressions in an f-string are evaluated in left-to-right order. This is detectable only if the expressions have side effects:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

Section 40.4: Float formatting

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
```

```
'42.123'

>>> '{0:.5f}'.format(42.12345)
'42.12345'

>>> '{0:.7f}'.format(42.12345)
'42.1234500'
```

Same hold for other way of referencing:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

Floating point numbers can also be formatted in scientific notation or as percentages:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

You can also combine the `{0}` and `{name}` notations. This is especially useful when you want to round all variables to a pre-specified number of decimals *with 1 declaration*:

```
>>> s = 'Hello'
>>> a, b, c = 1.12345, 2.34567, 34.5678
>>> digits = 2

>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
'Hello! 1.12, 2.35, 34.57'
```

Section 40.5: Named placeholders

Format strings may contain named placeholders that are interpolated using keyword arguments to `format`.

Using a dictionary (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Using a dictionary (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map` allows to use dictionaries without having to unpack them first. Also the class of `data` (which might be a custom type) is used instead of a newly filled `dict`.

Without a dictionary:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

Section 40.6: String formatting with datetime

Any class can configure its own string formatting syntax through the `__format__` method. A type in the standard Python library that makes handy use of this is the `datetime` type, where one can use `strftime`-like formatting codes directly within `str.format`:

```
>>> from datetime import datetime
>>> 'North America: {dt:%m/%d/%Y}.  ISO: {dt:%Y-%m-%d}.'.format(dt=datetime.now())
'North America: 07/21/2016.  ISO: 2016-07-21.'
```

A full list of list of datetime formatters can be found in the official documentation.

Section 40.7: Formatting Numerical Values

The `.format()` method can interpret a number in different formats, such as:

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Format integers to different bases (hex, oct, binary)

```
>>> '{0:x}'.format(10) # base 16, lowercase - Hexadecimal
'a'

>>> '{0:X}'.format(10) # base 16, uppercase - Hexadecimal
'A'

>>> '{:o}'.format(10) # base 8 - Octal
'12'

>>> '{:b}'.format(10) # base 2 - Binary
'1010'

>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # With prefix
'0b101010, 0o52, 0x2a'

>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # Add zero padding
'8 bit: 00101010; Three bytes: 00002a'
```

Use formatting to convert an RGB float tuple to a color hex string:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Only integers can be converted:

```
>>> '{:x}'.format(42.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'x' for object of type 'float'
```

Section 40.8: Nested formatting

Some formats can take additional parameters, such as the width of the formatted string, or the alignment:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

Those can also be provided as parameters to `format` by nesting more `{}` inside the `{}`:

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

In the latter example, the format string `'{:{}{}{}}'` is modified to `'{:*^15}'` (i.e. "center and pad with * to total length of 15") before applying it to the actual string `'foo'` to be formatted that way.

This can be useful in cases when parameters are not known beforehand, for instances when aligning tabular data:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Section 40.9: Format using Getitem and Getattr

Any data structure that supports `__getitem__` can have their nested structure formatted:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Object attributes can be accessed using `getattr()`:

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Section 40.10: Padding and truncating strings, combined

Say you want to print variables in a 3 character column.

Note: doubling { and } escapes them.

```
s = """

pad
{{:3}}             :{a:3}:

truncate
```

```
{{:.3}}            :{e:.3}:

combined
{{:>3.3}}          :{a:>3.3}:
{{:3.3}}           :{a:3.3}:
{{:3.3}}           :{c:3.3}:
{{:3.3}}           :{e:3.3}:
"""

print (s.format(a="1"*1, c="3"*3, e="5"*5))
```

Output:

```
pad
{:3}               :1  :

truncate
{:.3}              :555:

combined
{:>3.3}            :  1:
{:3.3}             :1  :
{:3.3}             :333:
{:3.3}             :555:
```

Section 40.11: Custom formatting for a class

Note:

> Everything below applies to the `str.format` method, as well as the `format` function. In the text below, the two are interchangeable.

For every value which is passed to the `format` function, Python looks for a `__format__` method for that argument. Your own custom class can therefore have their own `__format__` method to determine how the `format` function will display and format your class and it's attributes.

This is different than the `__str__` method, as in the `__format__` method you can take into account the formatting language, including alignment, field width etc, and even (if you wish) implement your own format specifiers, and your own formatting language extensions.[1]

```
object.__format__(self, format_spec)
```

For example:

```
# Example in Python 2 - but can be easily applied to Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Implement special semantics for the 's' format specifier """
        # Reject anything that isn't an s
        if format_spec[-1] != 's':
            raise ValueError('{} format specifier not understood for this object',
format_spec[:-1])
```

```
        # Output in this example will be (<a>,<b>,<c>)
        raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
        # Honor the format language by using the inbuilt string format
        # Since we know the original format_spec ends in an 's'
        # we can take advantage of the str.format method with a
        # string argument we constructed above
        return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# out :              (1,2,3)
# Note how the right align and field width of 20 has been honored.
```

Note:

> If your custom class does not have a custom `__format__` method and an instance of the class is passed to the `format` function, **Python2** will always use the return value of the `__str__` method or `__repr__` method to determine what to print (and if neither exist then the default `repr` will be used), and you will need to use the `s` format specifier to format this. With **Python3**, to pass your custom class to the `format` function, you will need define `__format__` method on your custom class.

Chapter 41: String Methods

Section 41.1: Changing the capitalization of a string

Python's string type provides many functions that act on the capitalization of a string. These include:

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

With unicode strings (the default in Python 3), these operations are **not** 1:1 mappings or reversible. Most of these operations are intended for display purposes, rather than normalization.

Python 3.x `Version ≥ 3.3`

`str.casefold()`

`str.casefold` creates a lowercase string that is suitable for case insensitive comparisons. This is more aggressive than `str.lower` and may modify strings that are already in lowercase or cause strings to grow in length, and is not intended for display purposes.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

The transformations that take place under casefolding are defined by the Unicode Consortium in the CaseFolding.txt file on their website.

`str.upper()`

`str.upper` takes every character in a string and converts it to its uppercase equivalent, for example:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

`str.lower()`

`str.lower` does the opposite; it takes every character in a string and converts it to its lowercase equivalent:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

`str.capitalize()`

`str.capitalize` returns a capitalized version of the string, that is, it makes the first character have upper case and the rest lower:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
```

```
# "This is a 'string'."
```

`str.title()`

`str.title` returns the title cased version of the string, that is, every letter in the beginning of a word is made upper case and all others are made lower case:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

`str.swapcase()`

`str.swapcase` returns a new string object in which all lower case characters are swapped to upper case and all upper case characters to lower:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

Usage as `str` class methods

It is worth noting that these methods may be called either on string objects (as shown above) or as a class method of the `str` class (with an explicit call to `str.upper`, etc.)

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

This is most useful when applying one of these methods to many strings at once in say, a *`map`* function.

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

Section 41.2: str.translate: Translating characters in a string

Python supports a `translate` method on the `str` type which allows you to specify the translation table (used for replacements) as well as any characters which should be deleted in the process.

str.translate(table[, deletechars])

Parameter	Description
`table`	It is a lookup table that defines the mapping from one character to another.
`deletechars`	A list of characters which are to be removed from the string.

The `maketrans` method (`str.maketrans` in Python 3 and `string.maketrans` in Python 2) allows you to generate a translation table.

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

The `translate` method returns a string which is a translated copy of the original string.

You can set the `table` argument to `None` if you only need to delete characters.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
```

```
'ths syntx s vry sfl'
```

Section 41.3: str.format and f-strings: Format values into a string

Python provides string interpolation and formatting functionality through the `str.format` function, introduced in version 2.6 and f-strings introduced in version 3.6.

Given the following variables:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

The following statements are all equivalent

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
>>> "{} {} {} {} {}".format(i, f, s, l, d)
>>> str.format("{} {} {} {} {}", i, f, s, l, d)
>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)
>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)
>>> "{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
>>> f"{i} {f} {s} {l} {d}"
>>> f"{i:d} {f:0.1f} {s} {l!r} {d!r}"
```

For reference, Python also supports C-style qualifiers for string formatting. The examples below are equivalent to those above, but the `str.format` versions are preferred due to benefits in flexibility, consistency of notation, and extensibility:

```
"%d %0.1f %s %r %r" % (i, f, s, l, d)
"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

The braces uses for interpolation in `str.format` can also be numbered to reduce duplication when formatting strings. For example, the following are equivalent:

```
"I am from Australia. I love cupcakes from Australia!"
>>> "I am from {}. I love cupcakes from {}!".format("Australia", "Australia")
>>> "I am from {0}. I love cupcakes from {0}!".format("Australia")
```

While the official python documentation is, as usual, thorough enough, pyformat.info has a great set of examples with detailed explanations.

Additionally, the { and } characters can be escaped by using double brackets:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)

>>> f"{{'{'a'}': {5}, '{'b'}': {6}}}"
```

See String Formatting for additional information. `str.format()` was proposed in PEP 3101 and f-strings in PEP 498.

Section 41.4: String module's useful constants

Python's `string` module provides constants for string related operations. To use them, import the `string` module:

```
>>> import string
```

`string.ascii_letters`:

Concatenation of `ascii_lowercase` and `ascii_uppercase`:

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.ascii_lowercase`:

Contains all lower case ASCII characters:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`string.ascii_uppercase`:

Contains all upper case ASCII characters:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits`:

Contains all decimal digit characters:

```
>>> string.digits
'0123456789'
```

`string.hexdigits`:

Contains all hex digit characters:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits`:

Contains all octal digit characters:

```
>>> string.octaldigits
'01234567'
```

`string.punctuation`:

Contains all characters which are considered punctuation in the `C` locale:

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace`:

Contains all ASCII characters considered whitespace:

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

In script mode, `print(string.whitespace)` will print the actual characters, use `str` to get the string returned above.

`string.printable`:

Contains all characters which are considered printable; a combination of `string.digits`, `string.ascii_letters`, `string.punctuation`, and `string.whitespace`.

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~
\t\n\r\x0b\x0c'
```

Section 41.5: Stripping unwanted leading/trailing characters from a string

Three methods are provided that offer the ability to strip leading and trailing characters from a string: `str.strip`, `str.rstrip` and `str.lstrip`. All three methods have the same signature and all three return a new string object with unwanted characters removed.

`str.strip([chars])`

`str.strip` acts on a given string and removes (strips) any leading or trailing characters contained in the argument `chars`; if `chars` is not supplied or is `None`, all white space characters are removed by default. For example:

```
>>> "    a line with leading and trailing space     ".strip()
'a line with leading and trailing space'
```

If `chars` is supplied, all characters contained in it are removed from the string, which is returned. For example:

```
>>> ">>> a Python prompt".strip('> ')  # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` and `str.lstrip([chars])`

These methods have similar semantics and arguments with `str.strip()`, their difference lies in the direction from which they start. `str.rstrip()` starts from the end of the string while `str.lstrip()` splits from the start of the string.

For example, using `str.rstrip`:

```
>>> "     spacious string      ".rstrip()
'     spacious string'
```

While, using `str.lstrip`:

```
>>> "     spacious string      ".rstrip()
'spacious string      '
```

Section 41.6: Reversing a string

A string can reversed using the built-in `reversed()` function, which takes a string and returns an iterator in reverse order.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

`reversed()` can be wrapped in a call to `''.join()` to make a string from the iterator.

```
>>> ''.join(reversed('hello'))
'olleh'
```

While using `reversed()` might be more readable to uninitiated Python users, using extended slicing with a step of `-1` is faster and more concise. Here , try to implement it as function:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

Section 41.7: Split a string based on a delimiter into a list of strings

`str.split(sep=None, maxsplit=-1)`

`str.split` takes a string and returns a list of substrings of the original string. The behavior differs depending on whether the `sep` argument is provided or omitted.

If `sep` isn't provided, or is `None`, then the splitting takes place wherever there is whitespace. However, leading and trailing whitespace is ignored, and multiple consecutive whitespace characters are treated the same as a single whitespace character:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence.  ".split()
['This', 'is', 'a', 'sentence.']

>>> "            ".split()
[]
```

The `sep` parameter can be used to define a delimiter string. The original string is split where the delimiter string occurs, and the delimiter itself is discarded. Multiple consecutive delimiters are *not* treated the same as a single occurrence, but rather cause empty strings to be created.

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']

>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']

>>> " This is    a sentence.  ".split(' ')
['', 'This', 'is', '', '', '', 'a', 'sentence.', '', '']

>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '.']

>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']
```

The default is to split on *every* occurrence of the delimiter, however the `maxsplit` parameter limits the number of splittings that occur. The default value of `-1` means no limit:

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']

>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']

>>> "This is a sentence.".split('e', maxsplit=2)
['This is a s', 'nt', 'nce.']

>>> "This is a sentence.".split('e', maxsplit=-1)
['This is a s', 'nt', 'nc', '.']
```

```
str.rsplit(sep=None, maxsplit=-1)
```

`str.rsplit` ("right split") differs from `str.split` ("left split") when `maxsplit` is specified. The splitting starts at the end of the string rather than at the beginning:

```
>>> "This is a sentence.".rsplit('e', maxsplit=1)
['This is a sentenc', '.']

>>> "This is a sentence.".rsplit('e', maxsplit=2)
['This is a sent', 'nc', '.']
```

Note: Python specifies the maximum number of *splits* performed, while most other programming languages specify the maximum number of *substrings* created. This may create confusion when porting or comparing code.

Section 41.8: Replace all occurrences of one substring with another substring

Python's `str` type also has a method for replacing occurrences of one sub-string with another sub-string in a given string. For more demanding cases, one can use `re.sub`.

```
str.replace(old, new[, count]):
```

`str.replace` takes two arguments `old` and `new` containing the `old` sub-string which is to be replaced by the `new` sub-string. The optional argument `count` specifies the number of replacements to be made:

For example, in order to replace `'foo'` with `'spam'` in the following string, we can call `str.replace` with `old = 'foo'` and `new = 'spam'`:

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

If the given string contains multiple examples that match the `old` argument, **all** occurrences are replaced with the value supplied in `new`:

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

unless, of course, we supply a value for `count`. In this case `count` occurrences are going to get replaced:

```
>>> """It can foo multiple examples of foo if you want, \
... or you can limit the foo with the third argument.""".replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third
argument.'
```

Section 41.9: Testing what a string is composed of

Python's `str` type also features a number of methods that can be used to evaluate the contents of a string. These are `str.isalpha`, `str.isdigit`, `str.isalnum`, `str.isspace`. Capitalization can be tested with `str.isupper`, `str.islower` and `str.istitle`.

`str.isalpha`

`str.isalpha` takes no arguments and returns `True` if the all characters in a given string are alphabetic, for example:

```
>>> "Hello World".isalpha()  # contains a space
False
>>> "Hello2World".isalpha()  # contains a number
False
>>> "HelloWorld!".isalpha()  # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

As an edge case, the empty string evaluates to `False` when used with `"".isalpha()`.

`str.isupper`, `str.islower`, `str.istitle`

These methods test the capitalization in a given string.

`str.isupper` is a method that returns `True` if all characters in a given string are uppercase and `False` otherwise.

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
```

```
False
```

Conversely, `str.islower` is a method that returns `True` if all characters in a given string are lowercase and `False` otherwise.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` returns `True` if the given string is title cased; that is, every word begins with an uppercase character followed by lowercase characters.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal`, `str.isdigit`, `str.isnumeric`

`str.isdecimal` returns whether the string is a sequence of decimal digits, suitable for representing a decimal number.

`str.isdigit` includes digits not in a form suitable for representing a decimal number, such as superscript digits.

`str.isnumeric` includes any number values, even if not digits, such as values outside the range 0-9.

```
           isdecimal    isdigit   isnumeric

12345       True        True       True
?2??5       True        True       True
?²³?????     False       True       True
??         False       False      True
Five        False       False      False
```

Bytestrings (`bytes` in Python 3, `str` in Python 2), only support `isdigit`, which only checks for basic ASCII digits.

As with `str.isalpha`, the empty string evaluates to `False`.

`str.isalnum`

This is a combination of `str.isalpha` and `str.isnumeric`, specifically it evaluates to `True` if all characters in the given string are **alphanumeric**, that is, they consist of alphabetic *or* numeric characters:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
```

```
>>> "Hello World".isalnum()  # contains whitespace
False
```

str.isspace

Evaluates to `True` if the string contains only whitespace characters.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

Sometimes a string looks “empty” but we don't know whether it's because it contains just whitespace or no character at all

```
>>> "".isspace()
False
```

To cover this case we need an additional test

```
>>> my_str = ''
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

But the shortest way to test if a string is empty or just contains whitespace characters is to use `strip`(with no arguments it removes all leading and trailing whitespace characters)

```
>>> not my_str.strip()
True
```

Section 41.10: String Contains

Python makes it extremely intuitive to check if a string contains a given substring. Just use the `in` operator:

```
>>> "foo" in "foo.baz.bar"
True
```

Note: testing an empty string will always result in `True`:

```
>>> "" in "test"
True
```

Section 41.11: Join a list of strings into one string

A string can be used as a separator to join a list of strings together into a single string using the `join()` method. For example you can create a string where each element in a list is separated by a space.

```
>>> " ".join(["once","upon","a","time"])
"once upon a time"
```

The following example separates the string elements with three hyphens.

```
>>> "---".join(["once", "upon", "a", "time"])
```

```
"once---upon---a---time"
```

Section 41.12: Counting number of times a substring appears in a string

One method is available for counting the number of occurrences of a sub-string in another string, `str.count`.

```
str.count(sub[, start[, end]])
```

`str.count` returns an `int` indicating the number of non-overlapping occurrences of the sub-string `sub` in another string. The optional arguments `start` and `end` indicate the beginning and the end in which the search will take place. By default `start = 0` and `end = len(str)` meaning the whole string will be searched:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

By specifying a different value for `start`, `end` we can get a more localized search and count, for example, if `start` is equal to 13 the call to:

```
>>> s.count("sea", start)
1
```

is equivalent to:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

Section 41.13: Case insensitive string comparisons

Comparing string in a case insensitive way seems like something that's trivial, but it's not. This section only considers unicode strings (the default in Python 3). Note that Python 2 may have subtle weaknesses relative to Python 3 - the later's unicode handling is much more complete.

The first thing to note it that case-removing conversions in unicode aren't trivial. There is text for which `text.lower() != text.upper().lower()`, such as `"ß"`:

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

But let's say you wanted to caselessly compare `"BUSSE"` and `"Buße"`. You probably also want to compare `"BUSSE"` and `"BU□E"` equal - that's the newer capital form. The recommended way is to use `casefold`:

Python 3.x `Version ≥ 3.3`

```
>>> help(str.casefold)
"""
Help on method_descriptor:

casefold(...)
      S.casefold() -> str

     Return a version of S suitable for caseless comparisons.
"""
```

Do not just use `lower`. If `casefold` is not available, doing `.upper().lower()` helps (but only somewhat).

Then you should consider accents. If your font renderer is good, you probably think `"ê" == "ê"` - but it doesn't:

```
>>> "ê" == "ê"
False
```

This is because they are actually

```
>>> import unicodedata

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E WITH CIRCUMFLEX']

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

The simplest way to deal with this is `unicodedata.normalize`. You probably want to use **NFKD** normalization, but feel free to check the documentation. Then one does

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê")
True
```

To finish up, here this is expressed in functions:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

Section 41.14: Justify strings

Python provides functions for justifying strings, enabling text padding to make aligning various strings much easier.

Below is an example of `str.ljust` and `str.rjust`:

```
interstates_lengths = {
    5: (1381, 2222),
    19: (63, 102),
    40: (2555, 4112),
    93: (189,305),
}
for road, length in interstates_lengths.items():
    miles,kms = length
```

```
    print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4),
str(kms).ljust(4)))
```

```
  40 -> 2555 mi. (4112 km.)
  19 -> 63   mi. (102  km.)
   5 -> 1381 mi. (2222 km.)
  93 -> 189  mi. (305  km.)
```

`ljust` and `rjust` are very similar. Both have a `width` parameter and an optional `fillchar` parameter. Any string created by these functions is at least as long as the `width` parameter that was passed into the function. If the string is longer than `width` alread, it is not truncated. The `fillchar` argument, which defaults to the space character `' '` must be a single character, not a multicharacter string.

The `ljust` function pads the end of the string it is called on with the `fillchar` until it is `width` characters long. The `rjust` function pads the beginning of the string in a similar fashion. Therefore, the `l` and `r` in the names of these functions refer to the side that the original string, *not the `fillchar`*, is positioned in the output string.

Section 41.15: Test the starting and ending characters of a string

In order to test the beginning and ending of a given string in Python, one can use the methods `str.startswith()` and `str.endswith()`.

`str.startswith(prefix[, start[, end]])`

As its name implies, `str.startswith` is used to test whether a given string starts with the given characters in `prefix`.

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

The optional arguments `start` and `end` specify the start and end points from which the testing will start and finish. In the following example, by specifying a start value of `2` our string will be searched from position `2` and afterwards:

```
>>> s.startswith("is", 2)
True
```

This yields `True` since `s[2] == 'i'` and `s[3] == 's'`.

You can also use a `tuple` to check if it starts with any of a set of strings

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

`str.endswith(prefix[, start[, end]])`

`str.endswith` is exactly similar to `str.startswith` with the only difference being that it searches for ending characters and not starting characters. For example, to test if a string ends in a full stop, one could write:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

as with `startswith` more than one characters can used as the ending sequence:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

You can also use a `tuple` to check if it ends with any of a set of strings

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

Section 41.16: Conversion between str or bytes data and unicode characters

The contents of files and network messages may represent encoded characters. They often need to be converted to unicode for proper display.

In Python 2, you may need to convert str data to Unicode characters. The default (`''`, `""`, etc.) is an ASCII string, with any values outside of ASCII range displayed as escaped values. Unicode strings are `u''` (or `u""`, etc.).

Python 2.x `Version ≥ 2.3`

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\xa9 abc'  # s is a byte array, not a string of characters
                    # Doesn't know the original was UTF-8
                    # Default form of string literals in Python 2
s[0]                # '\xc2' - meaningless byte (without context such as an encoding)
type(s)             # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8')  # u'\xa9 abc'
                       # Now we have a Unicode string, which can be read as UTF-8 and printed
properly
                       # In Python 2, Unicode string literals need a leading u
                       # str.decode converts a string which may contain escaped bytes to a Unicode
string
u[0]                # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # unicode

u.encode('utf-8')   # '\xc2\xa9 abc'
                    # unicode.encode produces a string with escaped bytes for non-ASCII characters
```

In Python 3 you may need to convert arrays of bytes (referred to as a 'byte literal') to strings of Unicode characters. The default is now a Unicode string, and bytestring literals must now be entered as `b''`, `b""`, etc. A byte literal will return `True` to `isinstance(some_val, byte)`, assuming `some_val` to be a string that might be encoded as bytes.

Python 3.x `Version ≥ 3.0`

```
# You get from file or network "© abc" encoded in UTF-8
```

```
s = b'\xc2\xa9 abc' # s is a byte array, not characters
                    # In Python 3, the default string literal is Unicode; byte array literals need a
leading b
s[0]                # b'\xc2' - meaningless byte (without context such as an encoding)
type(s)             # bytes - now that byte arrays are explicit, Python can show that.

u = s.decode('utf-8')  # '© abc' on a Unicode terminal
                       # bytes.decode converts a byte array to a string (which will, in Python 3, be
Unicode)
u[0]                # '\u00a9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # str
                    # The default string literal in Python 3 is UTF-8 Unicode

u.encode('utf-8')   # b'\xc2\xa9 abc'
                    # str.encode produces a byte array, showing ASCII-range bytes as unescaped
characters.
```

Chapter 42: Using loops within functions

In Python function will be returned as soon as execution hits "return" statement.

Section 42.1: Return statement inside loop in a function

In this example, function will return as soon as value var has 1

```
def func(params):
    for value in params:
        print ('Got value {}'.format(value))

        if value == 1:
            # Returns from function as soon as value is 1
            print (">>>> Got 1")
            return

        print ("Still looping")

    return "Couldn't find 1"

func([5, 3, 1, 2, 8, 9])
```

output

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

Chapter 43: Importing modules

Section 43.1: Importing a module

Use the `import` statement:

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` will import a module and then allow you to reference its objects -- values, functions and classes, for example -- using the `module.name` syntax. In the above example, the `random` module is imported, which contains the `randint` function. So by importing `random` you can call `randint` with `random.randint`.

You can import a module and assign it to a different name:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

If your python file `main.py` is in the same folder as `custom.py`. You can import it like this:

```
import custom
```

It is also possible to import a function from a module:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

To import specific functions deeper down into a module, the dot operator may be used **only** on the left side of the `import` keyword:

```
from urllib.request import urlopen
```

In python, we have two ways to call function from top level. One is `import` and another is `from`. We should use `import` when we have a possibility of name collision. Suppose we have `hello.py` file and `world.py` files having same function named `function`. Then `import` statement will work good.

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

In general `import` will provide you a namespace.

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

But if you are sure enough, in your whole project there is no way having same function name you should use `from` statement

Multiple imports can be made on the same line:

```
>>> # Multiple modules
>>> import time, sockets, random
>>> # Multiple functions
>>> from math import sin, cos, tan
>>> # Multiple constants
>>> from math import pi, e

>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

The keywords and syntax shown above can also be used in combinations:

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys

>>> print(time.time())
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

Section 43.2: The __all__ special variable

Modules can have a special variable named `__all__` to restrict what variables are imported when using `from mymodule import *`.

Given the following module:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Only `imported_by_star` is imported when using `from mymodule import *`:

```
>>> from mymodule import *
>>> imported_by_star
42
>>> not_imported_by_star
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

However, `not_imported_by_star` can be imported explicitly:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
```

```
21
```

Section 43.3: Import modules from an arbitrary filesystem location

If you want to import a module that doesn't already exist as a built-in module in the Python Standard Library nor as a side-package, you can do this by adding the path to the directory where your module is found to `sys.path`. This may be useful where multiple python environments exist on a host.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

It is important that you append the path to the *directory* in which `mymodule` is found, not the path to the module itself.

Section 43.4: Importing all names from a module

```
from module_name import *
```

for example:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

This will import all names defined in the `math` module into the global namespace, other than names that begin with an underscore (which indicates that the writer feels that it is for internal use only).

Warning: If a function with the same name was already defined or imported, it will be **overwritten**. Almost always importing only specific names `from math import sqrt, ceil` is the **recommended way**:

```
def sqrt(num):
    print("I don't know what's the square root of {}.".format(num))

sqrt(4)
# Output: I don't know what's the square root of 4.

from math import *
sqrt(4)
# Output: 2.0
```

Starred imports are only allowed at the module level. Attempts to perform them in class or function definitions result in a `SyntaxError`.

```
def f():
    from math import *
```

and

```
class A:
    from math import *
```

both fail with:

```
SyntaxError: import * only allowed at module level
```

Section 43.5: Programmatic importing

Python 2.x `Version ≥ 2.7`

To import a module through a function call, use the `importlib` module (included in Python starting in version 2.7):

```
import importlib
random = importlib.import_module("random")
```

The `importlib.import_module()` function will also import the submodule of a package directly:

```
collections_abc = importlib.import_module("collections.abc")
```

For older versions of Python, use the `imp` module.

Python 2.x `Version ≤ 2.7`

Use the functions `imp.find_module` and `imp.load_module` to perform a programmatic import.

Taken from standard library documentation

```
import imp, sys
def import_module(name):
    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        if fp:
            fp.close()
```

Do **NOT** use `__import__()` to programmatically import modules! There are subtle details involving `sys.modules`, the `fromlist` argument, etc. that are easy to overlook which `importlib.import_module()` handles for you.

Section 43.6: PEP8 rules for Imports

Some recommended PEP8 style guidelines for imports:

1. Imports should be on separate lines:

```
from math import sqrt, ceil      # Not recommended
from math import sqrt            # Recommended
from math import ceil
```

2. Order imports as follows at the top of the module:

   - Standard library imports
   - Related third party imports
   - Local application/library specific imports

3. Wildcard imports should be avoided as it leads to confusion in names in the current namespace. If you do `from module import *`, it can be unclear if a specific name in your code comes from `module` or not. This is

doubly true if you have multiple `from module import *`-type statements.

4. Avoid using relative imports; use explicit imports instead.

Section 43.7: Importing specific names from a module

Instead of importing the complete module you can import only specified names:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

`from random` is needed, because the python interpreter has to know from which resource it should import a function or class and `import randint` specifies the function or class itself.

Another example below (similar to the one above):

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

The following example will raise an error, because we haven't imported a module:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Outputs:

```
NameError: name 'random' is not defined
```

The python interpreter does not understand what you mean with `random`. It needs to be declared by adding `import random` to the example:

```
import random
random.randrange(1, 10)
```

Section 43.8: Importing submodules

```
from module.submodule import function
```

This imports `function` from `module.submodule`.

Section 43.9: Re-importing a module

When using the interactive interpreter, you might want to reload a module. This can be useful if you're editing a module and want to import the newest version, or if you've monkey-patched an element of an existing module and want to revert your changes.

Note that you **can't** just `import` the module again to revert:

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

This is because the interpreter registers every module you import. And when you try to reimport a module, the interpreter sees it in the register and does nothing. So the hard way to reimport is to use `import` after removing the corresponding item from the register:

```
print(math.pi)    # 3
import sys
if 'math' in sys.modules:  # Is the ``math`` module in the register?
    del sys.modules['math']  # If so, remove it.
import math
print(math.pi)    # 3.141592653589793
```

But there is more a straightforward and simple way.

Python 2

Use the `reload` function:

Python 2.x `Version ≥ 2.3`

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

The `reload` function has moved to `importlib`:

Python 3.x `Version ≥ 3.0`

```
import math
math.pi = 3
print(math.pi)    # 3
from importlib import reload
reload(math)
print(math.pi)    # 3.141592653589793
```

Section 43.10: __import__() function

The `__import__()` function can be used to import modules where the name is only known at runtime

```
if user_input == "os":
    os = __import__("os")

# equivalent to import os
```

This function can also be used to specify the file path to a module

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Chapter 44: Difference between Module and Package

Section 44.1: Modules

A module is a single Python file that can be imported. Using a module looks like this:

`module.py`

```
def hi():
    print("Hello world!")
```

`my_script.py`

```
import module
module.hi()
```

in an interpreter

```
>>> from module import hi
>>> hi()
# Hello world!
```

Section 44.2: Packages

A package is made up of multiple Python files (or modules), and can even include libraries written in C or C++. Instead of being a single file, it is an entire folder structure which might look like this:

Folder `package`

- `__init__.py`
- `dog.py`
- `hi.py`

`__init__.py`

```
from package.dog import woof
from package.hi import hi
```

`dog.py`

```
def woof():
    print("WOOF!!!")
```

`hi.py`

```
def hi():
    print("Hello world!")
```

All Python packages must contain an `__init__.py` file. When you import a package in your script (`import package`), the `__init__.py` script will be run, giving you access to the all of the functions in the package. In this case, it allows you to use the `package.hi` and `package.woof` functions.

Chapter 45: Math Module

Section 45.1: Rounding: round, floor, ceil, trunc

In addition to the built-in `round` function, the `math` module provides the `floor`, `ceil`, and `trunc` functions.

```
x = 1.55
y = -1.55

# round to the nearest integer
round(x)       #  2
round(y)       # -2

# the second argument gives how many decimal places to round to (defaults to 0)
round(x, 1)    #  1.6
round(y, 1)    # -1.6

# math is a module so import it first, then use it.
import math

# get the largest integer less than x
math.floor(x)  #  1
math.floor(y)  # -2

# get the smallest integer greater than x
math.ceil(x)   #  2
math.ceil(y)   # -1

# drop fractional part of x
math.trunc(x)  #  1, equivalent to math.floor for positive numbers
math.trunc(y)  # -1, equivalent to math.ceil for negative numbers
```

Python 2.x `Version ≤ 2.7`

`floor`, `ceil`, `trunc`, and `round` always return a `float`.

```
round(1.3)  # 1.0
```

`round` always breaks ties away from zero.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x `Version ≥ 3.0`

`floor`, `ceil`, and `trunc` always return an `Integral` value, while `round` returns an `Integral` value if called with one argument.

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

`round` breaks ties towards the nearest even number. This corrects the bias towards larger numbers when performing a large number of calculations.

```
round(0.5)  # 0
round(1.5)  # 2
```

Warning!

As with any floating-point representation, some fractions *cannot be represented exactly*. This can lead to some unexpected rounding behavior.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Warning about the floor, trunc, and integer division of negative numbers

Python (and C++ and Java) round away from zero for negative numbers. Consider:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

Section 45.2: Trigonometry

Calculating the length of the hypotenuse

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Converting degrees to/from radians

All `math` functions expect **radians** so you need to convert degrees to radians:

```
math.radians(45)              # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

All results of the inverse trigonometric functions return the result in radians, so you may need to convert it back to degrees:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Sine, cosine, tangent and inverse functions

```
# Sine and arc sine
math.sin(math.pi / 2)
# Out: 1.0
math.sin(math.radians(90))   # Sine of 90 degrees
# Out: 1.0

math.asin(1)
# Out: 1.5707963267948966    # "= pi / 2"
math.asin(1) / math.pi
# Out: 0.5

# Cosine and arc cosine:
math.cos(math.pi / 2)
# Out: 6.123233995736766e-17
# Almost zero but not exactly because "pi" is a float with limited precision!

math.acos(1)
# Out: 0.0

# Tangent and arc tangent:
math.tan(math.pi/2)
# Out: 1.633123935319537e+16
# Very large but not exactly "Inf" because "pi" is a float with limited precision
```

Python 3.x `Version ≥ 3.5`

```
math.atan(math.inf)
```

```
# Out: 1.5707963267948966 # This is just "pi / 2"
```

```
math.atan(float('inf'))
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Apart from the `math.atan` there is also a two-argument `math.atan2` function, which computes the correct quadrant and avoids pitfalls of division by zero:

```
math.atan2(1, 2)   # Equivalent to "math.atan(1/2)"
# Out: 0.4636476090008061 # ≈ 26.57 degrees, 1st quadrant

math.atan2(-1, -2) # Not equal to "math.atan(-1/-2)" == "math.atan(1/2)"
# Out: -2.677945044588987 # ≈ -153.43 degrees (or 206.57 degrees), 3rd quadrant

math.atan2(1, 0)   # math.atan(1/0) would raise ZeroDivisionError
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Hyperbolic sine, cosine and tangent

```
# Hyperbolic sine function
math.sinh(math.pi) # = 11.548739357257746
math.asinh(1)      # = 0.8813735870195429

# Hyperbolic cosine function
math.cosh(math.pi) # = 11.591953275521519
math.acosh(1)      # = 0.0

# Hyperbolic tangent function
math.tanh(math.pi) # = 0.99627207622075
math.atanh(0.5)    # = 0.5493061443340549
```

Section 45.3: Pow for faster exponentiation

Using the timeit module from the command line:

```
> python -m timeit 'for x in xrange(50000): b = x**3'
10 loops, best of 3: 51.2 msec per loop
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'
100 loops, best of 3: 9.15 msec per loop
```

The built-in `**` operator often comes in handy, but if performance is of the essence, use math.pow. Be sure to note, however, that pow returns floats, even if the arguments are integers:

```
> from math import pow
> pow(5,5)
3125.0
```

Section 45.4: Infinity and NaN ("not a number")

In all versions of Python, we can represent infinity and NaN ("not a number") as follows:

```
pos_inf = float('inf')     # positive infinity
neg_inf = float('-inf')    # negative infinity
not_a_num = float('nan')   # NaN ("not a number")
```

In Python 3.5 and higher, we can also use the defined constants `math.inf` and `math.nan`:

Python 3.x `Version ≥ 3.5`

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

The string representations display as `inf` and `-inf` and `nan`:

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

We can test for either positive or negative infinity with the `isinf` method:

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

We can test specifically for positive infinity or for negative infinity by direct comparison:

```
pos_inf == float('inf')    # or  == math.inf in Python 3.5+
# Out: True

neg_inf == float('-inf')   # or  == -math.inf in Python 3.5+
# Out: True

neg_inf == pos_inf
# Out: False
```

Python 3.2 and higher also allows checking for finiteness:

Python 3.x `Version ≥ 3.2`

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

Comparison operators work as expected for positive and negative infinity:

```
import sys

sys.float_info.max
# Out: 1.7976931348623157e+308  (this is system-dependent)

pos_inf > sys.float_info.max
# Out: True

neg_inf < -sys.float_info.max
# Out: True
```

But if an arithmetic expression produces a value larger than the maximum that can be represented as a `float`, it will become infinity:

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
```

```
# Out: True
```

However division by zero does not give a result of infinity (or negative infinity where appropriate), rather it raises a `ZeroDivisionError` exception.

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Arithmetic operations on infinity just give infinite results, or sometimes NaN:

```
-5.0 * pos_inf == neg_inf
# Out: True

-5.0 * neg_inf == pos_inf
# Out: True

pos_inf * neg_inf == neg_inf
# Out: True

0.0 * pos_inf
# Out: nan

0.0 * neg_inf
# Out: nan

pos_inf / pos_inf
# Out: nan
```

NaN is never equal to anything, not even itself. We can test for it is with the `isnan` method:

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN always compares as "not equal", but never less than or greater than:

```
not_a_num != 5.0   # or any random value
# Out: True

not_a_num > 5.0   or   not_a_num < 5.0   or   not_a_num == 5.0
# Out: False
```

Arithmetic operations on NaN always give NaN. This includes multiplication by -1: there is no "negative NaN".

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x `Version ≥ 3.5`

```
-math.nan
```

```
# Out: nan
```

There is one subtle difference between the old `float` versions of NaN and infinity and the Python 3.5+ `math` library constants:

Python 3.x `Version ≥ 3.5`

```
math.inf is math.inf, math.nan is math.nan
# Out: (True, True)

float('inf') is float('inf'), float('nan') is float('nan')
# Out: (False, False)
```

Section 45.5: Logarithms

`math.log(x)` gives the natural (base `e`) logarithm of `x`.

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` can lose precision with numbers close to 1, due to the limitations of floating-point numbers. In order to accurately calculate logs close to 1, use `math.log1p`, which evaluates the natural logarithm of 1 plus the argument:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` can be used for logs base 10:

```
math.log10(10)  # 1.0
```

Python 2.x `Version ≥ 2.3.0`

When used with two arguments, `math.log(x, base)` gives the logarithm of `x` in the given `base` (i.e. `log(x) / log(base)`).

```
math.log(100, 10) # 2.0
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Section 45.6: Constants

`math` modules includes two commonly used mathematical constants.

- `math.pi` - The mathematical constant pi
- `math.e` - The mathematical constant *e* (base of natural logarithm)

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 and higher have constants for infinity and NaN ("not a number"). The older syntax of passing a string to `float()` still works.

Python 3.x Version ≥ 3.5

```
math.inf == float('inf')
# Out: True

-math.inf == float('-inf')
# Out: True

# NaN never compares equal to anything, even itself
math.nan == float('nan')
# Out: False
```

Section 45.7: Imaginary Numbers

Imaginary numbers in Python are represented by a "j" or "J" trailing the target number.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Section 45.8: Copying signs

In Python 2.6 and higher, `math.copysign(x, y)` returns `x` with the sign of `y`. The returned value is always a `float`.

Python 2.x Version ≥ 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, on a platform which supports signed zero
```

Section 45.9: Complex numbers and the cmath module

The `cmath` module is similar to the `math` module, but defines functions appropriately for the complex plane.

First of all, complex numbers are a numeric type that is part of the Python language itself rather than being provided by a library class. Thus we don't need to `import cmath` for ordinary arithmetic expressions.

Note that we use `j` (or `J`) and not `i`.

```
z = 1 + 3j
```

We must use `1j` since `j` would be the name of a variable rather than a numeric literal.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

We have the `real` part and the `imag` (imaginary) part, as well as the complex `conjugate`:

```
# real part and imaginary part are both float type
z.real, z.imag
# Out: (1.0, 3.0)

z.conjugate()
# Out: (1-3j)    # z.conjugate() == z.real - z.imag * 1j
```

The built-in functions `abs` and `complex` are also part of the language itself and don't require any import:

```
abs(1 + 1j)
# Out: 1.4142135623730951     # square root of 2

complex(1)
# Out: (1+0j)

complex(imag=1)
# Out: (1j)

complex(1, 1)
# Out: (1+1j)
```

The `complex` function can take a string, but it can't have spaces:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

But for most functions we do need the module, for instance `sqrt`:

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Naturally the behavior of `sqrt` is different for complex numbers and real numbers. In non-complex `math` the square root of a negative number raises an exception:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Functions are provided to convert to and from polar coordinates:

```
cmath.polar(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))

abs(1 + 1j), cmath.phase(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # same as previous calculation

cmath.rect(math.sqrt(2), math.atan(1))
# Out: (1.0000000000000002+1.0000000000000002j)
```

The mathematical field of complex analysis is beyond the scope of this example, but many functions in the complex plane have a "branch cut", usually along the real axis or the imaginary axis. Most modern platforms support "signed zero" as specified in IEEE 754, which provides continuity of those functions on both sides of the branch cut. The following example is from the Python documentation:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
```

```
# Out: -3.141592653589793
```

The `cmath` module also provides many functions with direct counterparts from the `math` module.

In addition to `sqrt`, there are complex versions of `exp`, `log`, `log10`, the trigonometric functions and their inverses (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`), and the hyperbolic functions and their inverses (`sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`). Note however there is no complex counterpart of `math.atan2`, the two-argument form of arctangent.

```
cmath.log(1+1j)
# Out: (0.34657359027997264+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

The constants `pi` and `e` are provided. Note these are `float` and not `complex`.

```
type(cmath.pi)
# Out: <class 'float'>
```

The `cmath` module also provides complex versions of `isinf`, and (for Python 3.2+) `isfinite`. See "Infinity and NaN". A complex number is considered infinite if either its real part or its imaginary part is infinite.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

Likewise, the `cmath` module provides a complex version of `isnan`. See "Infinity and NaN". A complex number is considered "not a number" if either its real part or its imaginary part is "not a number".

```
cmath.isnan(0.0, float('nan'))
# Out: True
```

Note there is no `cmath` counterpart of the `math.inf` and `math.nan` constants (from Python 3.5 and higher)

Python 3.x `Version ≥ 3.5`

```
cmath.isinf(complex(0.0, math.inf))
# Out: True

cmath.isnan(complex(math.nan, 0.0))
# Out: True

cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf'
```

In Python 3.5 and higher, there is an `isclose` method in both `cmath` and `math` modules.

Python 3.x `Version ≥ 3.5`

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Chapter 46: Complex math

Section 46.1: Advanced complex arithmetic

The module `cmath` includes additional functions to use complex numbers.

```
import cmath
```

This module can calculate the phase of a complex number, in radians:

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

It allows the conversion between the cartesian (rectangular) and polar representations of complex numbers:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

The module contains the complex version of

- Exponential and logarithmic functions (as usual, `log` is the natural logarithm and `log10` the decimal logarithm):

```
cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
cmath.log(z) # (1.2824746787307684+0.982793723247329j)
cmath.log10(-100) # (2+1.3643763538418412j)
```

- Square roots:

```
cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
```

- Trigonometric functions and their inverses:

```
cmath.sin(z)  # (9.15449914691143-4.168906959966565j)
cmath.cos(z)  # (-4.189625690968807-9.109227893755337j)
cmath.tan(z)  # (-0.003764025641504249+1.00323862735361j)
cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)
cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)
cmath.atan(z) # (1.409921049596575+0.22907268296853878j)
cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
```

- Hyperbolic functions and their inverses:

```
cmath.sinh(z)  # (-3.59056458998578+0.5309210862485197j)
cmath.cosh(z)  # (-3.7245455049153224+0.5118225699873846j)
cmath.tanh(z)  # (0.965385879022133-0.009884375038322495j)
cmath.asinh(z) # (0.5706527843210994+1.9833870299165355j)
cmath.acosh(z) # (1.9833870299165355+1.0001435424737972j)
cmath.atanh(z) # (0.1469466662255297+1.3389725222944935j)
cmath.cosh(z)**2 - cmath.sin(z)**2  # (1+0j)
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

Section 46.2: Basic complex arithmetic

Python has built-in support for complex arithmetic. The imaginary unit is denoted by `j`:

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

Complex numbers can be summed, subtracted, multiplied, divided and exponentiated:

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3  # (-46+9j)
```

Python can also extract the real and imaginary parts of complex numbers, and calculate their absolute value and conjugate:

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

Chapter 47: Collections module

The built-in `collections` package provides several specialized, flexible collection types that are both high-performance and provide alternatives to the general collection types of `dict`, `list`, `tuple` and `set`. The module also defines abstract base classes describing different types of collection functionality (such as `MutableSet` and `ItemsView`).

Section 47.1: collections.Counter

Counter is a dict sub class that allows you to easily count objects. It has utility methods for working with the frequencies of the objects that you are counting.

```
import collections
counts = collections.Counter([1,2,3])
```

the above code creates an object, counts, which has the frequencies of all the elements passed to the constructor. This example has the value `Counter({1: 1, 2: 1, 3: 1})`

Constructor examples

Letter Counter

```
>>> collections.Counter('Happy Birthday')
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't': 1})
```

Word Counter

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that Sam-I-am'.split())
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that': 1, 'not': 1, 'like': 1})
```

Recipes

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Get count of individual element

```
>>> c['a']
4
```

Set count of individual element

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Get total number of elements in counter (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative numbers are counted!
3
```

Get elements (only those with positive counter are kept)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Remove keys with 0 or negative value

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Remove everything

```
>>> c.clear()
>>> c
Counter()
```

Add remove individual elements

```
>>> c.update({'a': 3, 'b':3})
>>> c.update({'a': 2, 'c':2})  # adds to existing, sets if they don't exist
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3})  # subtracts (negative values are allowed)
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

Section 47.2: collections.OrderedDict

The order of keys in Python dictionaries is arbitrary: they are not governed by the order in which you add them.

For example:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
```

(The arbitrary ordering implied above means that you may get different results with the above code to that shown here.)

The order in which the keys appear is the order which they would be iterated over, e.g. using a `for` loop.

The `collections.OrderedDict` class provides dictionary objects that retain the order of keys. `OrderedDicts` can be created as shown below with a series of ordered items (here, a list of tuple key-value pairs):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
```

```
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Or we can create an empty `OrderedDict` and then add items:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

Iterating through an `OrderedDict` allows key access in the order they were added.

What happens if we assign a new value to an existing key?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

The key retains its original place in the `OrderedDict`.

Section 47.3: collections.defaultdict

collections.defaultdict(default_factory) returns a subclass of `dict` that has a default value for missing keys. The argument should be a function that returns the default value when called with no arguments. If there is nothing passed, it defaults to `None`.

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

returns a reference to a defaultdict that will create a string object with its default_factory method.

A typical usage of `defaultdict` is to use one of the builtin types such as `str`, `int`, `list` or `dict` as the default_factory, since these return empty types when called with no arguments:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Calling the defaultdict with a key that does not exist does not produce an error as it would in a normal dictionary.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Another example with `int`:

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2  # No errors should occur
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana']  # No errors should occur
```

```
0
>>> fruit_counts  # A new key is created
default_dict(int, {'apple': 2, 'banana': 0})
```

Normal dictionary methods work with the default dictionary

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Using `list` as the default_factory will create a list for each new key.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]
>>> dd = collections.defaultdict(list)
>>> for k, v in s:
...     dd[k].append(v)
>>> dd
defaultdict(<class 'list'>,
    {'VA': ['Richmond'],
     'NC': ['Raleigh', 'Asheville'],
     'WA': ['Seattle']})
```

Section 47.4: collections.namedtuple

Define a new type `Person` using `namedtuple` like this:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

The second argument is the list of attributes that the tuple will have. You can list these attributes also as either space or comma separated string:

```
Person = namedtuple('Person', 'age, height, name')
```

or

```
Person = namedtuple('Person', 'age height name')
```

Once defined, a named tuple can be instantiated by calling the object with the necessary parameters, e.g.:

```
dave = Person(30, 178, 'Dave')
```

Named arguments can also be used:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Now you can access the attributes of the namedtuple:

```
print(jack.age)  # 30
print(jack.name)  # 'Jack S.'
```

The first argument to the namedtuple constructor (in our example `'Person'`) is the `typename`. It is typical to use the same word for the constructor and the typename, but they can be different:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
```

```
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

Section 47.5: collections.deque

Returns a new `deque` object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new `deque` is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the `deque` with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

New in version 2.4.

If `maxlen` is not specified or is `None`, deques may grow to an arbitrary length. Otherwise, the `deque` is bounded to the specified maximum length. Once a bounded length `deque` is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Changed in version 2.6: Added maxlen parameter.

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
True
>>> d.extend('jkl')                  # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # right rotation
>>> d
```

```
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

Source: https://docs.python.org/2/library/collections.html

Section 47.6: collections.ChainMap

`ChainMap` is new in **version 3.3**

Returns a new `ChainMap` object given a number of `maps`. This object groups multiple dicts or other mappings together to create a single, updateable view.

`ChainMaps` are useful managing nested contexts and overlays. An example in the python world is found in the implementation of the `Context` class in Django's template engine. It is useful for quickly linking a number of mappings so that the result can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple `update()` calls.

Anytime one has a chain of lookup values there can be a case for `ChainMap`. An example includes having both user specified values and a dictionary of default values. Another example is the `POST` and `GET` parameter maps found in web use, e.g. Django or Flask. Through the use of `ChainMap` one returns a combined view of two distinct dictionaries.

The `maps` parameter list is ordered from first-searched to last-searched. Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

```
import collections

# define two dictionaries with at least some keys overlapping.
dict1 = {'apple': 1, 'banana': 2}
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}

# create two ChainMaps with different ordering of those dicts.
combined_dict = collections.ChainMap(dict1, dict2)
reverse_ordered_dict = collections.ChainMap(dict2, dict1)
```

Note the impact of order on which value is found first in the subsequent lookup

```
for k, v in combined_dict.items():
    print(k, v)

date 1
apple 1
banana 2
```

```
coconut 1

for k, v in reverse_ordered_dict.items():
    print(k, v)

date 1
apple 3
banana 2
coconut 1
```

Chapter 48: Operator module

Section 48.1: Itemgetter

Grouping the key-value pairs of a dictionary by the value with `itemgetter`:

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}

dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# Output: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

which is equivalent (but faster) to a `lambda` function like this:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

Or sorting a list of tuples by the second element first the first element as secondary:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

Section 48.2: Operators as alternative to an infix operator

For every infix operator, e.g. `+` there is an `operator`-function (`operator.add` for `+`):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

even though the main documentation states that for the arithmetic operators only numerical input is allowed it *is* possible:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

See also: mapping from operation to operator function in the official Python documentation.

Section 48.3: Methodcaller

Instead of this `lambda`-function that calls the method explicitly:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))     # Keep only elements that start with 'd'
# Output: ['duck']
```

one could use a operator-function that does the same:

```
from operator import methodcaller
```

```
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Chapter 49: JSON Module

Section 49.1: Storing data in a file

The following snippet encodes the data stored in `d` into JSON and stores it in a file (replace `filename` with the actual name of the file).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Section 49.2: Retrieving data from a file

The following snippet opens a JSON encoded file (replace `filename` with the actual name of the file) and returns the object that is stored in the file.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

Section 49.3: Formatting JSON output

Let's say we have the following data:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Just dumping this as JSON does not do anything special here:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Setting indentation to get prettier output

If we want pretty printing, we can set an `indent` size:

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Sorting keys alphabetically to get consistent output

By default the order of keys in the output is undefined. We can get them in alphabetical order to make sure we always get the same output:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

Getting rid of whitespace to get compact output

We might want to get rid of the unnecessary spaces, which is done by setting separator strings different from the default `', '` and `': '`:

```
>>>print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"},{"name":"Pepper","color":"black"}]}
```

Section 49.4: `load` vs `loads`, `dump` vs `dumps`

The `json` module contains functions for both reading and writing to and from unicode strings, and reading and writing to and from files. These are differentiated by a trailing `s` in the function name. In these examples we use a StringIO object, but the same functions would apply for any file-like object.

Here we use the string-based functions:

```
import json

data = {u"foo": u"bar", u"baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {u"foo": u"bar", u"baz": []}
```

And here we use the file-based functions:

```
import json

from io import StringIO

json_file = StringIO()
data = {u"foo": u"bar", u"baz": []}
json.dump(data, json_file)
json_file.seek(0)  # Seek back to the start of the file before reading
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0)  # Seek back to the start of the file before reading
json.load(json_file)
# {u"foo": u"bar", u"baz": []}
```

As you can see the main difference is that when dumping json data you must pass the file handle to the function, as opposed to capturing the return value. Also worth noting is that you must seek to the start of the file before reading or writing, in order to avoid data corruption. When opening a file the cursor is placed at position `0`, so the below would also work:

```
import json

json_file_path = './data.json'
data = {u"foo": u"bar", u"baz": []}
```

```
with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)

with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'

with open(json_file_path) as json_file:
    json.load(json_file)
    # {u"foo": u"bar", u"baz": []}
```

Having both ways of dealing with json data allows you to idiomatically and efficiently work with formats which build upon json, such as `pyspark`'s json-per-line:

```
# loading from a file
data = [json.loads(line) for line in open(file_path).splitlines()]

# dumping to a file
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write('\n')
```

Section 49.5: Calling `json.tool` from the command line to pretty-print JSON output

Given some JSON file "foo.json" like:

```
{"foo": {"bar": {"baz": 1}}}
```

we can call the module directly from the command line (passing the filename as an argument) to pretty-print it:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

The module will also take input from STDOUT, so (in Bash) we equally could do:

```
$ cat foo.json | python -m json.tool
```

Section 49.6: JSON encoding custom objects

If we just try the following:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

we get an error saying `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable`.

To be able to serialize the datetime object properly, we need to write custom code for how to convert it:

```
class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj has no isoformat method; let the builtin JSON encoder handle it
            return super(DatetimeJSONEncoder, self).default(obj)
```

and then use this encoder class instead of `json.dumps`:

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

Section 49.7: Creating JSON from Python dict

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

The above snippet will return the following:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

Section 49.8: Creating Python dict from JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

The above snippet will return the following:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Chapter 50: Sqlite3 Module

Section 50.1: Sqlite3 - Not require separate server process

The sqlite3 module was written by Gerhard Häring. To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name :memory: to create a database in RAM. Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
         (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

Section 50.2: Getting the values from the database and Error handling

Fetching the values from the SQLite3 database.

Print row values returned by select query

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute("SELECT * from table_name where id=cust_id")
for row in c:
    print row # will be a list
```

To fetch single matching fetchone() method

```
print c.fetchone()
```

For multiple rows use fetchall() method

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

Error handling can be done using sqlite3.Error built in function

```
try:
    #SQL Code
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

Chapter 51: The os Module

Parameter	Details
Path	A path to a file. The path separator may be determined by `os.path.sep`.
Mode	The desired permission, in octal (e.g. `0700`)

This module provides a portable way of using operating system dependent functionality.

Section 51.1: makedirs - recursive directory creation

Given a local directory with the following contents:

```
└── dir1
    ├── subdir1
    └── subdir2
```

We want to create the same subdir1, subdir2 under a new directory dir2, which does not exist yet.

```
import os

os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

Running this results in

```
├── dir1
│   ├── subdir1
│   └── subdir2
└── dir2
    ├── subdir1
    └── subdir2
```

dir2 is only created the first time it is needed, for subdir1's creation.

If we had used **os.mkdir** instead, we would have had an exception because dir2 would not have existed yet.

```
    os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

os.makedirs won't like it if the target directory exists already. If we re-run it again:

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

However, this could easily be fixed by catching the exception and checking that the directory has been created.

```
try:
    os.makedirs("./dir2/subdir1")
except OSError:
    if not os.path.isdir("./dir2/subdir1"):
        raise

try:
    os.makedirs("./dir2/subdir2")
except OSError:
```

```
    if not os.path.isdir("./dir2/subdir2"):
        raise
```

Section 51.2: Create a directory

```
os.mkdir('newdir')
```

If you need to specify permissions, you can use the optional `mode` argument:

```
os.mkdir('newdir', mode=0700)
```

Section 51.3: Get current directory

Use the `os.getcwd()` function:

```
print(os.getcwd())
```

Section 51.4: Determine the name of the operating system

The `os` module provides an interface to determine what type of operating system the code is currently running on.

```
os.name
```

This can return one of the following in Python 3:

- `posix`
- `nt`
- `ce`
- `java`

More detailed information can be retrieved from `sys.platform`

Section 51.5: Remove a directory

Remove the directory at `path`:

```
os.rmdir(path)
```

You should not use `os.remove()` to remove a directory. That function is for *files* and using it on directories will result in an `OSError`

Section 51.6: Follow a symlink (POSIX)

Sometimes you need to determine the target of a symlink. `os.readlink` will do this:

```
print(os.readlink(path_to_symlink))
```

Section 51.7: Change permissions on a file

```
os.chmod(path, mode)
```

where `mode` is the desired permission, in octal.

Chapter 52: The locale Module

Section 52.1: Currency Formatting US Dollars Using the locale Module

```
import locale

locale.setlocale(locale.LC_ALL, '')
Out[2]: 'English_United States.1252'

locale.currency(762559748.49)
Out[3]: '$762559748.49'

locale.currency(762559748.49, grouping=True)
Out[4]: '$762,559,748.49'
```

Chapter 53: Itertools Module

Section 53.1: Combinations method in Itertools Module

`itertools.combinations` will return a generator of the *k*-combination sequence of a list.

In other words: It will return a generator of tuples of all the possible k-wise combinations of the input list.

For Example:

If you have a list:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

Output:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

The above output is a generator converted to a list of tuples of all the possible *pair*-wise combinations of the input list `a`

You can also find all the 3-combinations:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
print b
```

Output:

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
 (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
 (2, 4, 5), (3, 4, 5)]
```

Section 53.2: itertools.dropwhile

itertools.dropwhile enables you to take items from a sequence after a condition first becomes `False`.

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

This outputs `[13, 14, 22, 23, 44]`.

(*This example is same as the example for `takewhile` but using `dropwhile`.*)

Note that, the first number that violates the predicate (i.e.: the function returning a Boolean value) `is_even` is, 13. All the elements before that, are discarded.

The **output produced** by `dropwhile` is similar to the output generated from the code below.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

The concatenation of results produced by `takewhile` and `dropwhile` produces the original iterable.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

Section 53.3: Zipping two iterators until they are both exhausted

Similar to the built-in function `zip()`, `itertools.zip_longest` will continue iterating beyond the end of the shorter of two iterables.

```
from itertools import zip_longest
a = [i for i in range(5)] # Length is 5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Length is 7
for i in zip_longest(a, b):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

An optional `fillvalue` argument can be passed (defaults to `''`) like so:

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

In Python 2.6 and 2.7, this function is called `itertools.izip_longest`.

Section 53.4: Take a slice of a generator

Itertools "islice" allows you to slice a generator:

```
results = fetch_paged_results()  # returns a generator
limit = 20  # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Normally you cannot slice a generator:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Will give

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[:3]:
TypeError: 'generator' object is not subscriptable
```

However, this works:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Note that like a regular slice, you can also use `start`, `stop` and `step` arguments:

```
itertools.islice(iterable, 1, 30, 3)
```

Section 53.5: Grouping items from an iterable object using a function

Start with an iterable which needs to be grouped

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Generate the grouped generator, grouping by the second element in each tuple:

```
def testGroupBy(lst):
    groups = itertools.groupby(lst, key=lambda x: x[1])
    for key, group in groups:
        print(key, list(group))

testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
```

Only groups of consecutive elements are grouped. You may need to sort by the same key before calling groupby For E.g, (Last element is changed)

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]
testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5)]
# 5 [('c', 5, 6)]
```

The group returned by groupby is an iterator that will be invalid before next iteration. E.g the following will not work if you want the groups to be sorted by key. Group 5 is empty below because when group 2 is fetched it invalidates 5

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted(groups):
```

```
    print(key, list(group))

# 2 [('c', 2, 6)]
# 5 []
```

To correctly do sorting, create a list from the iterator before sorting

```
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted((key, list(group)) for key, group in groups):
    print(key, list(group))

# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
# 5 [('a', 5, 6)]
```

Section 53.6: itertools.takewhile

itertools.takewhile enables you to take items from a sequence until a condition first becomes `False`.

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.takewhile(is_even, lst))

print(result)
```

This outputs `[0, 2, 4, 12, 18]`.

Note that, the first number that violates the predicate (i.e.: the function returning a Boolean value) `is_even` is, 13. Once `takewhile` encounters a value that produces `False` for the given predicate, it breaks out.

The **output produced** by `takewhile` is similar to the output generated from the code below.

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Note: The concatenation of results produced by `takewhile` and `dropwhile` produces the original iterable.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

Section 53.7: itertools.permutations

`itertools.permutations` returns a generator with successive r-length permutations of elements in the iterable.

```
a = [1,2,3]
list(itertools.permutations(a))
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

list(itertools.permutations(a, 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

if the list a has duplicate elements, the resulting permutations will have duplicate elements, you can use `set` to get unique permutations:

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]

set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

Section 53.8: itertools.repeat

Repeat something n times:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

Section 53.9: Get an accumulated sum of numbers in an iterable

Python 3.x `Version ≥ 3.2`

`accumulate` yields a cumulative sum (or product) of numbers.

```
>>> import itertools as it
>>> import operator

>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]

>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

Section 53.10: Cycle through elements in an iterator

`cycle` is an infinite iterator.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Therefore, take care to give boundaries when using this to avoid an infinite loop. Example:

```
>>> # Iterate over each element in cycle for a fixed range
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

Section 53.11: itertools.product

This function lets you iterate over the Cartesian product of a list of iterables.

For example,

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

is equivalent to

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Like all python functions that accept a variable number of arguments, we can pass a list to itertools.product for unpacking, with the * operator.

Thus,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

produces the same results as both of the previous examples.

```
>>> from itertools import product
>>> a=[1,2,3,4]
>>> b=['a','b','c']
>>> product(a,b)
<itertools.product object at 0x0000000002712F78>
>>> for i in product(a,b):
...     print i
...
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
(4, 'a')
(4, 'b')
(4, 'c')
```

Section 53.12: itertools.count

Introduction:

This simple function generates infinite series of numbers. For example...

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Note that we must break or it prints forever!

Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Arguments:

`count()` takes two arguments, `start` and `step`:

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Output:

```
10
14
18
22
```

Section 53.13: Chaining multiple iterators together

Use `itertools.chain` to create a single generator which will yield the values from several generators in sequence.

```
from itertools import chain
a = (x for x in ['1', '2', '3', '4'])
b = (x for x in ['x', 'y', 'z'])
' '.join(chain(a, b))
```

Results in:

```
'1 2 3 4 x y z'
```

As an alternate constructor, you can use the classmethod `chain.from_iterable` which takes as its single parameter an iterable of iterables. To get the same result as above:

```
' '.join(chain.from_iterable([a,b])
```

While `chain` can take an arbitrary number of arguments, `chain.from_iterable` is the only way to chain an *infinite* number of iterables.

Chapter 54: Asyncio Module

Section 54.1: Coroutine and Delegation Syntax

Before Python 3.5+ was released, the `asyncio` module used generators to mimic asynchronous calls and thus had a different syntax than the current Python 3.5 release.

Python 3.x `Version ≥ 3.5`

Python 3.5 introduced the `async` and `await` keywords. Note the lack of parentheses around the `await func()` call.

```
import asyncio

async def main():
    print(await func())

async def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x `Version ≥ 3.3` `Version < 3.5`

Before Python 3.5, the `@asyncio.coroutine` decorator was used to define a coroutine. The yield from expression was used for generator delegation. Note the parentheses around the `yield from func()`.

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))

@asyncio.coroutine
def func():
    # Do time intensive stuff..
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x `Version ≥ 3.5`

Here is an example that shows how two functions can be run asynchronously:

```
import asyncio

async def cor1():
    print("cor1 start")
    for i in range(10):
        await asyncio.sleep(1.5)
        print("cor1", i)

async def cor2():
    print("cor2 start")
    for i in range(15):
        await asyncio.sleep(1)
```

```
        print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

Section 54.2: Asynchronous Executors

Note: Uses the Python 3.5+ async/await syntax

`asyncio` supports the use of `Executor` objects found in `concurrent.futures` for scheduling tasks asynchronously. Event loops have the function `run_in_executor()` which takes an `Executor` object, a `Callable`, and the Callable's parameters.

Scheduling a task for an `Executor`

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Each event loop also has a "default" `Executor` slot that can be assigned to an `Executor`. To assign an `Executor` and schedule tasks from the loop you use the `set_default_executor()` method.

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    # NOTE: Using `None` as the first parameter designates the `default` Executor.
    result = await loop.run_in_executor(None, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

There are two main types of `Executor` in `concurrent.futures`, the `ThreadPoolExecutor` and the `ProcessPoolExecutor`. The `ThreadPoolExecutor` contains a pool of threads which can either be manually set to a specific number of threads through the constructor or defaults to the number of cores on the machine times 5. The `ThreadPoolExecutor` uses the pool of threads to execute tasks assigned to it and is generally better at CPU-bound operations rather than I/O bound operations. Contrast that to the `ProcessPoolExecutor` which spawns a new process for each task assigned to it. The `ProcessPoolExecutor` can only take tasks and parameters that are

picklable. The most common non-picklable tasks are the methods of objects. If you must schedule an object's method as a task in an `Executor` you must use a `ThreadPoolExecutor`.

Section 54.3: Using UVLoop

`uvloop` is an implementation for the `asyncio.AbstractEventLoop` based on libuv (Used by nodejs). It is compliant with 99% of `asyncio` features and is much faster than the traditional `asyncio.EventLoop`. `uvloop` is currently not available on Windows, install it with `pip install uvloop`.

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

One can also change the event loop factory by setting the `EventLoopPolicy` to the one in `uvloop`.

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

Section 54.4: Synchronization Primitive: Event

Concept

Use an `Event` to **synchronize the scheduling of multiple coroutines**.

Put simply, an event is like the gun shot at a running race: it lets the runners off the starting blocks.

Example

```
import asyncio

# event trigger function
def trigger(event):
    print('EVENT SET')
    event.set()  # wake up coroutines waiting

# event consumers
async def consumer_a(event):
    consumer_name = 'Consumer A'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

async def consumer_b(event):
    consumer_name = 'Consumer B'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

# event
event = asyncio.Event()

# wrap coroutines in one future
```

```
main_future = asyncio.wait([consumer_a(event),
                            consumer_b(event)])

# event loop
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event))  # trigger event in 0.1 sec

# complete main_future
done, pending = event_loop.run_until_complete(main_future)
```

Output:

> Consumer B waiting
> Consumer A waiting
> EVENT SET
> Consumer B triggered
> Consumer A triggered

Section 54.5: A Simple Websocket

Here we make a simple echo websocket using `asyncio`. We define coroutines for connecting to a server and sending/receiving messages. The communications of the websocket are run in a `main` coroutine, which is run by an event loop. This example is modified from a prior post.

```
import asyncio
import aiohttp

session = aiohttp.ClientSession()                          # handles the context manager
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

async def main():
    echo = EchoWebsocket()
    await echo.connect()
    await echo.send("Hello World!")
    print(await echo.receive())                            # "Hello World!"


if __name__ == '__main__':
    # The main loop
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Section 54.6: Common Misconception about asyncio

probably *the* most common misconception about `asnycio` is that it lets you run any task in parallel - sidestepping the GIL (global interpreter lock) and therefore execute blocking jobs in parallel (on separate threads). it does **not**!

`asyncio` (and libraries that are built to collaborate with `asyncio`) build on coroutines: functions that (collaboratively) yield the control flow back to the calling function. note `asyncio.sleep` in the examples above. this is an example of a non-blocking coroutine that waits 'in the background' and gives the control flow back to the calling function (when called with `await`). `time.sleep` is an example of a blocking function. the execution flow of the program will just stop there and only return after `time.sleep` has finished.

a real-live example is the `requests` library which consists (for the time being) on blocking functions only. there is no concurrency if you call any of its functions within `asyncio`. `aiohttp` on the other hand was built with `asyncio` in mind. its coroutines will run concurrently.

- if you have long-running CPU-bound tasks you would like to run in parallel `asyncio` is **not** for you. for that you need `threads` or `multiprocessing`.
- if you have IO-bound jobs running, you *may* run them concurrently using `asyncio`.

Chapter 55: Random module

Section 55.1: Creating a random user password

In order to create a random user password we can use the symbols provided in the `string` module. Specifically `punctuation` for punctuation symbols, `ascii_letters` for letters and `digits` for digits:

```
from string import punctuation, ascii_letters, digits
```

We can then combine all these symbols in a name named `symbols`:

```
symbols = ascii_letters + digits + punctuation
```

Remove either of these to create a pool of symbols with fewer elements.

After this, we can use `random.SystemRandom` to generate a password. For a 10 length password:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password)  # '^@g;J?]M6e'
```

Note that other routines made immediately available by the `random` module — such as `random.choice`, `random.randint`, etc. — are *unsuitable* for cryptographic purposes.

Behind the curtains, these routines use the Mersenne Twister PRNG, which does not satisfy the requirements of a CSPRNG. Thus, in particular, you should not use any of them to generate passwords you plan to use. Always use an instance of `SystemRandom` as shown above.

Python 3.x `Version ≥ 3.6`

Starting from Python 3.6, the `secrets` module is available, which exposes cryptographically safe functionality.

Quoting the official documentation, to generate *"a ten-character alphanumeric password with at least one lowercase character, at least one uppercase character, and at least three digits,"* you could:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Section 55.2: Create cryptographically secure random numbers

By default the Python random module use the Mersenne Twister PRNG to generate random numbers, which, although suitable in domains like simulations, fails to meet security requirements in more demanding environments.

In order to create a cryptographically secure pseudorandom number, one can use `SystemRandom` which, by using `os.urandom`, is able to act as a Cryptographically secure pseudorandom number generator, CPRNG.

The easiest way to use it simply involves initializing the `SystemRandom` class. The methods provided are similar to the ones exported by the random module.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

In order to create a random sequence of 10 `int`s in range `[0, 20]`, one can simply call `randrange()`:

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

To create a random integer in a given range, one can use `randint`:

```
print(secure_rand_gen.randint(0, 20))
# 5
```

and, accordingly for all other methods. The interface is exactly the same, the only change is the underlying number generator.

You can also use `os.urandom` directly to obtain cryptographically secure random bytes.

Section 55.3: Random and sequences: shuffle, choice and sample

```
import random
```

shuffle()

You can use `random.shuffle()` to mix up/randomize the items in a **mutable and indexable** sequence. For example a `list`:

```
laughs = ["Hi", "Ho", "He"]

random.shuffle(laughs)     # Shuffles in-place! Don't do: laughs = random.shuffle(laughs)

print(laughs)
# Out: ["He", "Hi", "Ho"]  # Output may vary!
```

choice()

Takes a random element from an arbitrary **sequence**:

```
print(random.choice(laughs))
# Out: He                  # Output may vary!
```

sample()

Like `choice` it takes random elements from an arbitrary **sequence** but you can specify how many:

```
#                   |--sequence--|--number--|
print(random.sample(    laughs    ,     1    ))  # Take one element
# Out: ['Ho']                    # Output may vary!
```

it will not take the same element twice:

```
print(random.sample(laughs, 3))  # Take 3 random element from the sequence.
# Out: ['Ho', 'He', 'Hi']        # Output may vary!
```

```
print(random.sample(laughs, 4))  # Take 4 random element from the 3-item sequence.
```

> ValueError: Sample larger than population

Section 55.4: Creating random integers and floats: randint, randrange, random, and uniform

```
import random
```

randint()

Returns a random integer between `x` and `y` (inclusive):

```
random.randint(x, y)
```

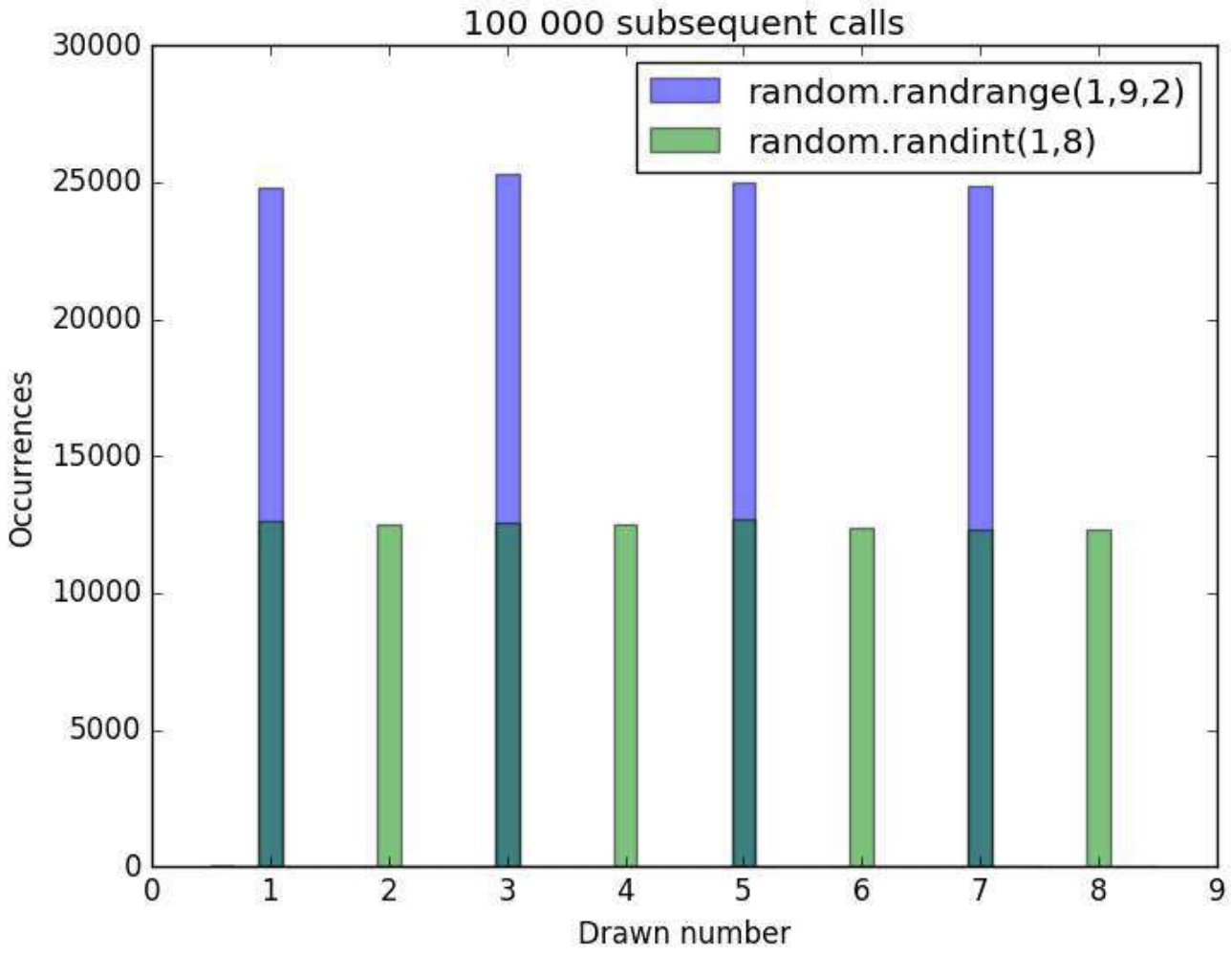
For example getting a random number between `1` and `8`:

```
random.randint(1, 8) # Out: 8
```

randrange()

`random.randrange` has the same syntax as `range` and unlike `random.randint`, the last value is **not** inclusive:

```
random.randrange(100)       # Random integer between 0 and 99
random.randrange(20, 50)    # Random integer between 20 and 49
random.rangrange(10, 20, 3) # Random integer between 10 and 19 with step 3 (10, 13, 16 and 19)
```



random

Returns a random floating point number between 0 and 1:

```
random.random() # Out: 0.66486093215306317
```

uniform

Returns a random floating point number between x and y (inclusive):

```
random.uniform(1, 8) # Out: 3.726062641730108
```

Section 55.5: Reproducible random numbers: Seed and State

Setting a specific Seed will create a fixed random-number series:

```
random.seed(5)                  # Create a fixed state
print(random.randrange(0, 10))  # Get a random integer between 0 and 9
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Resetting the seed will create the same "random" sequence again:

```
random.seed(5)                  # Reset the random module to the same fixed state.
print(random.randrange(0, 10))
```

```
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Since the seed is fixed these results are always `9` and `4`. If having specific numbers is not required only that the values will be the same one can also just use `getstate` and `setstate` to recover to a previous state:

```
save_state = random.getstate()  # Get the current state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8

random.setstate(save_state)     # Reset to saved state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8
```

To pseudo-randomize the sequence again you `seed` with `None`:

```
random.seed(None)
```

Or call the `seed` method with no arguments:

```
random.seed()
```

Section 55.6: Random Binary Decision

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

Chapter 56: Functools Module

Section 56.1: partial

The `partial` function creates partial function application from another function. It is used to *bind* values to some of the function's arguments (or keyword arguments) and produce a *callable* without the already defined arguments.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

`partial()`, as the name suggests, allows a partial evaluation of a function. Let's look at following example:

```
In [2]: from functools import partial

In [3]: def f(a, b, c, x):
   ...:     return 1000*a + 100*b + 10*c + x
   ...:

In [4]: g = partial(f, 1, 1, 1)

In [5]: print g(2)
1112
```

When `g` is created, `f`, which takes four arguments(`a, b, c, x`), is also partially evaluated for the first three arguments, `a, b, c,`. Evaluation of `f` is completed when `g` is called, `g(2)`, which passes the fourth argument to `f`.

One way to think of `partial` is a shift register; pushing in one argument at the time into some function. `partial` comes handy for cases where data is coming in as stream and we cannot pass more than one argument.

Section 56.2: cmp_to_key

Python changed its sorting methods to accept a key function. Those functions take a value and return a key which is used to sort the arrays.

Old comparison functions used to take two values and return -1, 0 or +1 if the first argument is small, equal or greater than the second argument respectively. This is incompatible to the new key-function.

That's where `functools.cmp_to_key` comes in:

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Example taken and adapted from the Python Standard Library Documentation.

Section 56.3: lru_cache

The `@lru_cache` decorator can be used wrap an expensive, computationally-intensive function with a Least Recently Used cache. This allows function calls to be memoized, so that future calls with the same parameters can return instantly instead of having to be recomputed.

```
@lru_cache(maxsize=None)  # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

In the example above, the value of `fibonacci(3)` is only calculated once, whereas if `fibonacci` didn't have an LRU cache, `fibonacci(3)` would have been computed upwards of 230 times. Hence, `@lru_cache` is especially great for recursive functions or dynamic programming, where an expensive function could be called multiple times with the same exact parameters.

`@lru_cache` has two arguments

- `maxsize`: Number of calls to save. When the number of unique calls exceeds `maxsize`, the LRU cache will remove the least recently used calls.
- `typed` (added in 3.3): Flag for determining if equivalent arguments of different types belong to different cache records (i.e. if `3.0` and `3` count as different arguments)

We can see cache stats too:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

NOTE: Since `@lru_cache` uses dictionaries to cache results, all parameters for the function must be hashable for the cache to work.

Official Python docs for `@lru_cache`. `@lru_cache` was added in 3.2.

Section 56.4: total_ordering

When we want to create an orderable class, normally we need to define the methods `__eq()__`, `__lt__()`, `__le__()`, `__gt__()` and `__ge__()`.

The `total_ordering` decorator, applied to a class, permits the definition of `__eq__()` and only one between `__lt__()`, `__le__()`, `__gt__()` and `__ge__()`, and still allow all the ordering operations on the class.

```
@total_ordering
class Employee:

    ...

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))
```

The decorator uses a composition of the provided methods and algebraic operations to derive the other comparison methods. For example if we defined `__lt__()` and `__eq()__` and we want to derive `__gt__()`, we can simply check `not __lt__() and not __eq()__`.

Note: The `total_ordering` function is only available since Python 2.7.

Section 56.5: reduce

In Python 3.x, the `reduce` function already explained here has been removed from the built-ins and must now be imported from `functools`.

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

Chapter 57: The dis module

Section 57.1: What is Python bytecode?

Python is a hybrid interpreter. When running a program, it first assembles it into *bytecode* which can then be run in the Python interpreter (also called a *Python virtual machine*). The `dis` module in the standard library can be used to make the Python bytecode human-readable by disassembling classes, methods, functions, and code objects.

```
>>> def hello():
...     print "Hello, World"
...
>>> dis.dis(hello)
  2           0 LOAD_CONST               1 ('Hello, World')
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

The Python interpreter is stack-based and uses a first-in last-out system.

Each operation code (opcode) in the Python assembly language (the bytecode) takes a fixed number of items from the stack and returns a fixed number of items to the stack. If there aren't enough items on the stack for an opcode, the Python interpreter will crash, possibly without an error message.

Section 57.2: Constants in the dis module

```
EXTENDED_ARG = 145 # All opcodes greater than this have 2 operands
HAVE_ARGUMENT = 90 # All opcodes greater than this have at least 1 operands

cmp_op = ('<', '<=', '==', '!=', '>', '>=', 'in', 'not in', 'is', 'is ...
       # A list of comparator id's. The indices are used as operands in some opcodes

# All opcodes in these lists have the respective types as there operands
hascompare = [107]
hasconst = [100]
hasfree = [135, 136, 137]
hasjabs = [111, 112, 113, 114, 115, 119]
hasjrel = [93, 110, 120, 121, 122, 143]
haslocal = [124, 125, 126]
hasname = [90, 91, 95, 96, 97, 98, 101, 106, 108, 109, 116]

# A map of opcodes to ids
opmap = {'BINARY_ADD': 23, 'BINARY_AND': 64, 'BINARY_DIVIDE': 21, 'BIN...
# A map of ids to opcodes
opname = ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', '...
```

Section 57.3: Disassembling modules

To disassemble a Python module, first this has to be turned into a `.pyc` file (Python compiled). To do this, run

```
python -m compileall <file>.py
```

Then in an interpreter, run

```
import dis
import marshal
```

```
with open("<file>.pyc", "rb") as code_f:
    code_f.read(8) # Magic number and modification time
    code = marshal.load(code_f) # Returns a code object which can be disassembled
    dis.dis(code) # Output the disassembly
```

This will compile a Python module and output the bytecode instructions with `dis`. The module is never imported so it is safe to use with untrusted code.

Chapter 58: The base64 Module

Parameter	Description
`base64.b64encode(s, altchars=None)`	
s	A bytes-like object
altchars	A bytes-like object of length 2+ of characters to replace the '+' and '=' characters when creating the Base64 alphabet. Extra characters are ignored.
`base64.b64decode(s, altchars=None, validate=False)`	
s	A bytes-like object
altchars	A bytes-like object of length 2+ of characters to replace the '+' and '=' characters when creating the Base64 alphabet. Extra characters are ignored.
validate	If validate is True, the characters not in the normal Base64 alphabet or the alternative alphabet are **not** discarded before the padding check
`base64.standard_b64encode(s)`	
s	A bytes-like object
`base64.standard_b64decode(s)`	
s	A bytes-like object
`base64.urlsafe_b64encode(s)`	
s	A bytes-like object
`base64.urlsafe_b64decode(s)`	
s	A bytes-like object
`b32encode(s)`	
s	A bytes-like object
`b32decode(s)`	
s	A bytes-like object
`base64.b16encode(s)`	
s	A bytes-like object
`base64.b16decode(s)`	
s	A bytes-like object
`base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)`	
b	A bytes-like object
foldspaces	If foldspaces is True, the character 'y' will be used instead of 4 consecutive spaces.
wrapcol	The number characters before a newline (0 implies no newlines)
pad	If pad is True, the bytes are padded to a multiple of 4 before encoding
adobe	If adobe is True, the encoded sequenced with be framed with '<~' and '~>' as used with Adobe products
`base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')`	
b	A bytes-like object
foldspaces	If foldspaces is True, the character 'y' will be used instead of 4 consecutive spaces.

adobe	If adobe is True, the encoded sequenced with be framed with '<~' and '~>' as used with Adobe products
ignorechars	A bytes-like object of characters to ignore in the encoding process

`base64.b85encode(b, pad=False)`

b	A bytes-like object
pad	If pad is True, the bytes are padded to a multiple of 4 before encoding

`base64.b85decode(b)`

b	A bytes-like object

Base 64 encoding represents a common scheme for encoding binary into ASCII string format using radix 64. The base64 module is part of the standard library, which means it installs along with Python. Understanding of bytes and strings is critical to this topic and can be reviewed here. This topic explains how to use the various features and number bases of the base64 module.

Section 58.1: Encoding and Decoding Base64

To include the base64 module in your script, you must import it first:

```
import base64
```

The base64 encode and decode functions both require a bytes-like object. To get our string into bytes, we must encode it using Python's built in encode function. Most commonly, the `UTF-8` encoding is used, however a full list of these standard encodings (including languages with different characters) can be found here in the official Python Documentation. Below is an example of encoding a string into bytes:

```
s = "Hello World!"
b = s.encode("UTF-8")
```

The output of the last line would be:

```
b'Hello World!'
```

The `b` prefix is used to denote the value is a bytes object.

To Base64 encode these bytes, we use the `base64.b64encode()` function:

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

That code would output the following:

```
b'SGVsbG8gV29ybGQh'
```

which is still in the bytes object. To get a string out of these bytes, we can use Python's `decode()` method with the `UTF-8` encoding:

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
```

```
s1 = e.decode("UTF-8")
print(s1)
```

The output would then be:

```
SGVsbG8gV29ybGQh
```

If we wanted to encode the string and then decode we could use the `base64.b64decode()` method:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base64 Encode the bytes
e = base64.b64encode(b)
# Decoding the Base64 bytes to string
s1 = e.decode("UTF-8")
# Printing Base64 encoded string
print("Base64 Encoded:", s1)
# Encoding the Base64 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base64 bytes
d = base64.b64decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

As you may have expected, the output would be the original string:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

Section 58.2: Encoding and Decoding Base32

The base64 module also includes encoding and decoding functions for Base32. These functions are very similar to the Base64 functions:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base32 Encode the bytes
e = base64.b32encode(b)
# Decoding the Base32 bytes to string
s1 = e.decode("UTF-8")
# Printing Base32 encoded string
print("Base32 Encoded:", s1)
# Encoding the Base32 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base32 bytes
d = base64.b32decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

This would produce the following output:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====
Hello World!
```

Section 58.3: Encoding and Decoding Base16

The base64 module also includes encoding and decoding functions for Base16. Base 16 is most commonly referred to as **hexadecimal**. These functions are very similar to the both the Base64 and Base32 functions:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base16 Encode the bytes
e = base64.b16encode(b)
# Decoding the Base16 bytes to string
s1 = e.decode("UTF-8")
# Printing Base16 encoded string
print("Base16 Encoded:", s1)
# Encoding the Base16 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base16 bytes
d = base64.b16decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

This would produce the following output:

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

Section 58.4: Encoding and Decoding ASCII85

Adobe created its own encoding called **ASCII85** which is similar to Base85, but has its differences. This encoding is used frequently in Adobe PDF files. These functions were released in Python version 3.4. Otherwise, the functions `base64.a85encode()` and `base64.a85encode()` are similar to the previous:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# ASCII85 Encode the bytes
e = base64.a85encode(b)
# Decoding the ASCII85 bytes to string
s1 = e.decode("UTF-8")
# Printing ASCII85 encoded string
print("ASCII85 Encoded:", s1)
# Encoding the ASCII85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the ASCII85 bytes
d = base64.a85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
```

```
print(s2)
```

This outputs the following:

```
ASCII85 Encoded: 87cURD]i,"Ebo80
Hello World!
```

Section 58.5: Encoding and Decoding Base85

Just like the Base64, Base32, and Base16 functions, the Base85 encoding and decoding functions are `base64.b85encode()` and `base64.b85decode()`:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base85 Encode the bytes
e = base64.b85encode(b)
# Decoding the Base85 bytes to string
s1 = e.decode("UTF-8")
# Printing Base85 encoded string
print("Base85 Encoded:", s1)
# Encoding the Base85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base85 bytes
d = base64.b85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

which outputs the following:

```
Base85 Encoded: NM&qnZy;B1a%^NF
Hello World!
```

Chapter 59: Queue Module

The Queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. There are three types of queues provides by queue module,Which are as following : 1. Queue 2. LifoQueue 3. PriorityQueue Exception which could be come: 1. Full (queue overflow) 2. Empty (queue underflow)

Section 59.1: Simple example

```
from Queue import Queue

question_queue = Queue()

for x in range(1,10):
    temp_dict = ('key', x)
    question_queue.put(temp_dict)

while(not question_queue.empty()):
    item = question_queue.get()
    print(str(item))
```

Output:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

Chapter 60: Deque Module

Parameter	Details
`iterable`	Creates the deque with initial elements copied from another iterable.
`maxlen`	Limits how large the deque can be, pushing out old elements as new are added.

Section 60.1: Basic deque using

The main methods that are useful with this class are `popleft` and `appendleft`

```
from collections import deque

d = deque([1, 2, 3])
p = d.popleft()         # p = 1, d = deque([2, 3])
d.appendleft(5)         # d = deque([5, 2, 3])
```

Section 60.2: Available methods in deque

Creating empty deque:

```
dl = deque()  # deque([]) creating empty deque
```

Creating deque with some elements:

```
dl = deque([1, 2, 3, 4])  # deque([1, 2, 3, 4])
```

Adding element to deque:

```
dl.append(5)  # deque([1, 2, 3, 4, 5])
```

Adding element left side of deque:

```
dl.appendleft(0)  # deque([0, 1, 2, 3, 4, 5])
```

Adding list of elements to deque:

```
dl.extend([6, 7])  # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Adding list of elements to from the left side:

```
dl.extendleft([-2, -1])  # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

Using `.pop()` element will naturally remove an item from the right side:

```
dl.pop()  # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Using `.popleft()` element to remove an item from the left side:

```
dl.popleft()  # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Remove element by its value:

```
dl.remove(1)  # deque([-2, 0, 2, 3, 4, 5, 6])
```

Reverse the order of the elements in deque:

```
dl.reverse()  # deque([6, 5, 4, 3, 2, 0, -2])
```

Section 60.3: limit deque size

Use the `maxlen` parameter while creating a deque to limit the size of the deque:

```
from collections import deque
d = deque(maxlen=3)  # only holds 3 items
d.append(1)  # deque([1])
d.append(2)  # deque([1, 2])
d.append(3)  # deque([1, 2, 3])
d.append(4)  # deque([2, 3, 4]) (1 is removed because its maxlen is 3)
```

Section 60.4: Breadth First Search

The Deque is the only Python data structure with fast Queue operations. (Note `queue.Queue` isn't normally suitable, since it's meant for communication between threads.) A basic use case of a Queue is the breadth first search.

```
from collections import deque

def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # The oldest seen (but not yet visited) node will be the left most one.
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # When we see a new node, we add it to the right side of the queue.
                q.append(neighbor)
    return distances
```

Say we have a simple directed graph:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

We can now find the distances from some starting position:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Chapter 61: Webbrowser Module

Parameter	Details
`webbrowser.open()`	
url	the URL to open in the web browser
new	0 opens the URL in the existing tab, 1 opens in a new window, 2 opens in new tab
autoraise	if set to True, the window will be moved on top of the other windows
`webbrowser.open_new()`	
url	the URL to open in the web browser
`webbrowser.open_new_tab()`	
url	the URL to open in the web browser
`webbrowser.get()`	
using	the browser to use
`webbrowser.register()`	
url	browser name
constructor	path to the executable browser (help)
instance	An instance of a web browser returned from the `webbrowser.get()` method

According to Python's standard documentation, the webbrowser module provides a high-level interface to allow displaying Web-based documents to users. This topic explains and demonstrates proper usage of the webbrowser module.

Section 61.1: Opening a URL with Default Browser

To simply open a URL, use the `webbrowser.open()` method:

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

If a browser window is currently open, the method will open a new tab at the specified URL. If no window is open, the method will open the operating system's default browser and navigate to the URL in the parameter. The open method supports the following parameters:

- `url` - the URL to open in the web browser (string) **[required]**
- `new` - 0 opens in existing tab, 1 opens new window, 2 opens new tab (integer) **[default 0]**
- `autoraise` - if set to True, the window will be moved on top of other applications' windows (Boolean) **[default False]**

Note, the `new` and `autoraise` arguments rarely work as the majority of modern browsers refuse these commands.

Webbrowser can also try to open URLs in new windows with the `open_new` method:

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

This method is commonly ignored by modern browsers and the URL is usually opened in a new tab. Opening a new tab can be tried by the module using the `open_new_tab` method:

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

Section 61.2: Opening a URL with Different Browsers

The webbrowser module also supports different browsers using the `register()` and `get()` methods. The get method is used to create a browser controller using a specific executable's path and the register method is used to attach these executables to preset browser types for future use, commonly when multiple browser types are used.

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Registering a browser type:

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
webbrowser.register('firefox', None, ff)
# Now to refer to use Firefox in the future you can use this
webbrowser.get('firefox').open("https://stackoverflow.com/")
```

Chapter 62: tkinter

Released in Tkinter is Python's most popular GUI (Graphical User Interface) library. This topic explains proper usage of this library and its features.

Section 62.1: Geometry Managers

Tkinter has three mechanisms for geometry management: `place`, `pack`, and `grid`.

The `place` manager uses absolute pixel coordinates.

The `pack` manager places widgets into one of 4 sides. New widgets are placed next to existing widgets.

The `grid` manager places widgets into a grid similar to a dynamically resizing spreadsheet.

Place

The most common keyword arguments for `widget.place` are as follows:

- `x`, the absolute x-coordinate of the widget
- `y`, the absolute y-coordinate of the widget
- `height`, the absolute height of the widget
- `width`, the absolute width of the widget

A code example using `place`:

```
class PlaceExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(master,text="This is on top at the origin")
        #top_text.pack()
        top_text.place(x=0,y=0,height=50,width=200)
        bottom_right_text=Label(master,text="This is at position 200,400")
        #top_text.pack()
        bottom_right_text.place(x=200,y=400,height=50,width=200)
# Spawn Window
if __name__=="__main__":
    root=Tk()
    place_frame=PlaceExample(root)
    place_frame.mainloop()
```

Pack

`widget.pack` can take the following keyword arguments:

- `expand`, whether or not to fill space left by parent
- `fill`, whether to expand to fill all space (NONE (default), X, Y, or BOTH)
- `side`, the side to pack against (TOP (default), BOTTOM, LEFT, or RIGHT)

Grid

The most commonly used keyword arguments of `widget.grid` are as follows:

- `row`, the row of the widget (default smallest unoccupied)
- `rowspan`, the number of columns a widget spans (default 1)
- `column`, the column of the widget (default 0)

- `columnspan`, the number of columns a widget spans (default 1)
- `sticky`, where to place widget if the grid cell is larger than it (combination of N,NE,E,SE,S,SW,W,NW)

The rows and columns are zero indexed. Rows increase going down, and columns increase going right.

A code example using `grid`:

```
from tkinter import *

class GridExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(self,text="This text appears on top left")
        top_text.grid() # Default position 0, 0
        bottom_text=Label(self,text="This text appears on bottom left")
        bottom_text.grid() # Default position 1, 0
        right_text=Label(self,text="This text appears on the right and spans both rows",
                         wraplength=100)
        # Position is 0,1
        # Rowspan means actual position is [0-1],1
        right_text.grid(row=0,column=1,rowspan=2)

# Spawn Window
if __name__=="__main__":
    root=Tk()
    grid_frame=GridExample(root)
    grid_frame.mainloop()
```

Never mix `pack` and `grid` within the same frame! Doing so will lead to application deadlock!

Section 62.2: A minimal tkinter Application

`tkinter` is a GUI toolkit that provides a wrapper around the Tk/Tcl GUI library and is included with Python. The following code creates a new window using `tkinter` and places some text in the window body.

> Note: In Python 2, the capitalization may be slightly different, see Remarks section below.

```
import tkinter as tk

# GUI window is a subclass of the basic tkinter Frame object
class HelloWorldFrame(tk.Frame):
    def __init__(self, master):
        # Call superclass constructor
        tk.Frame.__init__(self, master)
        # Place frame into main window
        self.grid()
        # Create text box with "Hello World" text
        hello = tk.Label(self, text="Hello World! This label can hold strings!")
        # Place text box into frame
        hello.grid(row=0, column=0)

# Spawn window
if __name__ == "__main__":
    # Create main window object
    root = tk.Tk()
    # Set title of window
    root.title("Hello World!")
```

```
    # Instantiate HelloWorldFrame object
    hello_frame = HelloWorldFrame(root)
    # Start GUI
    hello_frame.mainloop()
```

Chapter 63: pyautogui module

pyautogui is a module used to control mouse and keyboard. This module is basically used to automate mouse click and keyboard press tasks. For the mouse, the coordinates of the screen (0,0) start from the top-left corner. If you are out of control, then quickly move the mouse cursor to top-left, it will take the control of mouse and keyboard from the Python and give it back to you.

Section 63.1: Mouse Functions

These are some of useful mouse functions to control the mouse.

```
size()           #gave you the size of the screen
position()     #return current position of mouse
moveTo(200,0,duration=1.5)     #move the cursor  to (200,0) position  with 1.5 second delay
moveRel()          #move the cursor relative to your current position.
click(337,46)           #it will click on the position mention there
dragRel()              #it will drag the mouse relative to position
pyautogui.displayMousePosition()     #gave you the current mouse position but should be done on
terminal.
```

Section 63.2: Keyboard Functions

These are some of useful keyboard functions to automate the key pressing.

```
typewrite('')    #this will type the string on the screen where current window has focused.
typewrite(['a','b','left','left','X','Y'])
pyautogui.KEYBOARD_KEYS    #get the list of all the keyboard_keys.
pyautogui.hotkey('ctrl','o')    #for the combination of keys to enter.
```

Section 63.3: Screenshot And Image Recognition

These function will help you to take the screenshot and also match the image with the part of the screen.

```
.screenshot('c:\\path')        #get the screenshot.
.locateOnScreen('c:\\path')    #search that image on screen and get the coordinates for you.
locateCenterOnScreen('c:\\path')       #get the coordinate for the image on screen.
```

Chapter 64: Indexing and Slicing

Paramer	Description
`obj`	The object that you want to extract a "sub-object" from
`start`	The index of `obj` that you want the sub-object to start from (keep in mind that Python is zero-indexed, meaning that the first item of `obj` has an index of `0`). If omitted, defaults to `0`.
`stop`	The (non-inclusive) index of `obj` that you want the sub-object to end at. If omitted, defaults to `len(obj)`.
`step`	Allows you to select only every `step` item. If omitted, defaults to `1`.

Section 64.1: Basic Slicing

For any iterable (for eg. a string, list, etc), Python allows you to slice and return a substring or sublist of its data.

Format for slicing:

```
iterable_name[start:stop:step]
```

where,

- `start` is the first index of the slice. Defaults to 0 (the index of the first element)
- `stop` one past the last index of the slice. Defaults to len(iterable)
- `step` is the step size (better explained by the examples below)

Examples:

```
a = "abcdef"
a            # "abcdef"
             # Same as a[:] or a[::] since it uses the defaults for all three indices
a[-1]        # "f"
a[:]         # "abcdef"
a[::]        # "abcdef"
a[3:]        # "def" (from index 3, to end(defaults to size of iterable))
a[:4]        # "abcd" (from beginning(default 0) to position 4 (excluded))
a[2:4]       # "cd" (from position 2, to position 4 (excluded))
```

In addition, any of the above can be used with the step size defined:

```
a[::2]       # "ace" (every 2nd element)
a[1:4:2]     # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Indices can be negative, in which case they're computed from the end of the sequence

```
a[:-1]     # "abcde" (from index 0 (default), to the second last element (last element - 1))
a[:-2]     # "abcd" (from index 0 (default), to the third last element (last element -2))
a[-1:]     # "f" (from the last element to the end (default len())
```

Step sizes can also be negative, in which case slice will iterate through the list in reverse order:

```
a[3:1:-1]   # "dc" (from index 2 to None (default), in reverse order)
```

This construct is useful for reversing an iterable

```
a[::-1]     # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

Notice that for negative steps the default `end_index` is `None` (see http://stackoverflow.com/a/12521981)

```
a[5:None:-1] # "fedcba" (this is equivalent to a[::-1])
a[5:0:-1]    # "fedcb" (from the last element (index 5) to second element (index 1)
```

Section 64.2: Reversing an object

You can use slices to very easily reverse a `str`, `list`, or `tuple` (or basically any collection object that implements slicing with the step parameter). Here is an example of reversing a string, although this applies equally to the other types listed above:

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

Let's quickly look at the syntax. `[::-1]` means that the slice should be from the beginning until the end of the string (because `start` and `end` are omitted) and a step of `-1` means that it should move through the string in reverse.

Section 64.3: Slice assignment

Another neat feature using slices is slice assignment. Python allows you to assign new slices to replace old slices of a list in a single operation.

This means that if you have a list, you can replace multiple members in a single assignment:

```
lst = [1, 2, 3]
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]
```

The assignment shouldn't match in size as well, so if you wanted to replace an old slice with a new slice that is different in size, you could:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]
```

It's also possible to use the known slicing syntax to do things like replacing the entire list:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

Or just the last two members:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

Section 64.4: Making a shallow copy of an array

A quick way to make a copy of an array (as opposed to assigning a variable with another reference to the original array) is:

```
arr[:]
```

Let's examine the syntax. `[:]` means that `start`, `end`, and `slice` are all omitted. They default to `0`, `len(arr)`, and `1`, respectively, meaning that subarray that we are requesting will have all of the elements of `arr` from the beginning until the very end.

In practice, this looks something like:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

As you can see, `arr.append('d')` added `d` to `arr`, but `copy` remained unchanged!

Note that this makes a *shallow* copy, and is identical to `arr.copy()`.

Section 64.5: Indexing custom classes: __getitem__, __setitem__ and __delitem__

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
        if isinstance(item, int):
            self.value[item] = value
        elif isinstance(item, slice):
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value

    def __delitem__(self, item):
        if isinstance(item, int):
            del self.value[item]
        elif isinstance(item, slice):
            del self.value[item]
        else:
            if any(isinstance(elem, slice) for elem in item):
                raise ValueError('Cannot interpret slice with multiindexing')
            item = sorted(item, reverse=True)
            for elem in item:
                del self.value[elem]
```

This allows slicing and indexing for element access:

```
a = MultiIndexingList([1,2,3,4,5,6,7,8])
a
# Out: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
a[1,5,2,6,1]
# Out: [2, 6, 3, 7, 2]
a[4, 1, 5:, 2, ::2]
# Out: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
#       4|1-|----50:---|2-|-----::2-----   <-- indicated which element came from which index
```

While setting and deleting elements only allows for *comma separated* integer indexing (no slicing):

```
a[4] = 1000
a
# Out: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# Out: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5]
a
# Out: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5]
a
# Out: [1, 100, 4, 8]
```

Section 64.6: Basic Indexing

Python lists are 0-based *i.e.* the first element in the list can be accessed by the index `0`

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

You can access the second element in the list by index `1`, third element by index `2` and so on:

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

You can also use negative indices to access elements from the end of the list. eg. index `-1` will give you the last element of the list and index `-2` will give you the second-to-last element of the list:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

If you try to access an index which is not present in the list, an `IndexError` will be raised:

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

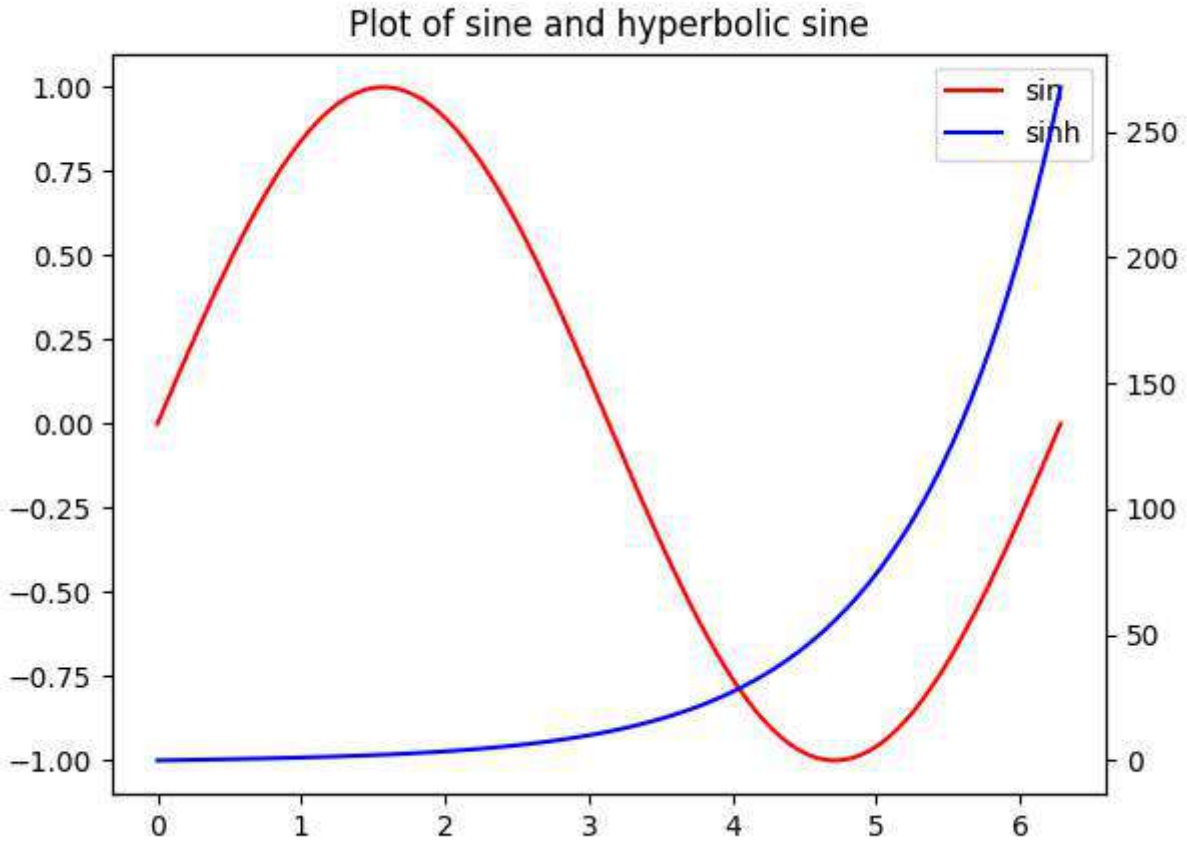
Chapter 65: Plotting with Matplotlib

Matplotlib (https://matplotlib.org/) is a library for 2D plotting based on NumPy. Here are some basic examples. More examples can be found in the official documentation (https://matplotlib.org/2.0.2/gallery.html and https://matplotlib.org/2.0.2/examples/index.html)

Section 65.1: Plots with Common X-axis but different Y-axis : Using twinx()

In this example, we will plot a sine curve and a hyperbolic sine curve in the same plot with a common x-axis having different y-axis. This is accomplished by the use of **twinx()** command.

```
# Plotting tutorials in Python
# Adding Multiple plots by twin x axis
# Good for plots having different y axis range
# Separate axes and figure objects
# replicate axes object and plot curves
# use axes to set attributes

# Note:
# Grid for second curve unsuccessful : let me know if you find it! :(

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.sinh(x)

# separate the figure object and axes object
# from the plotting object
fig, ax1 = plt.subplots()

# Duplicate the axes with a different y axis
# and the same x axis
ax2 = ax1.twinx() # ax2 and ax1 will have common x axis and different y axis

# plot the curves on axes 1, and 2, and get the curve handles
curve1, = ax1.plot(x, y, label="sin", color='r')
curve2, = ax2.plot(x, z, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
ax1.legend(curves, [curve.get_label() for curve in curves])
# ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```

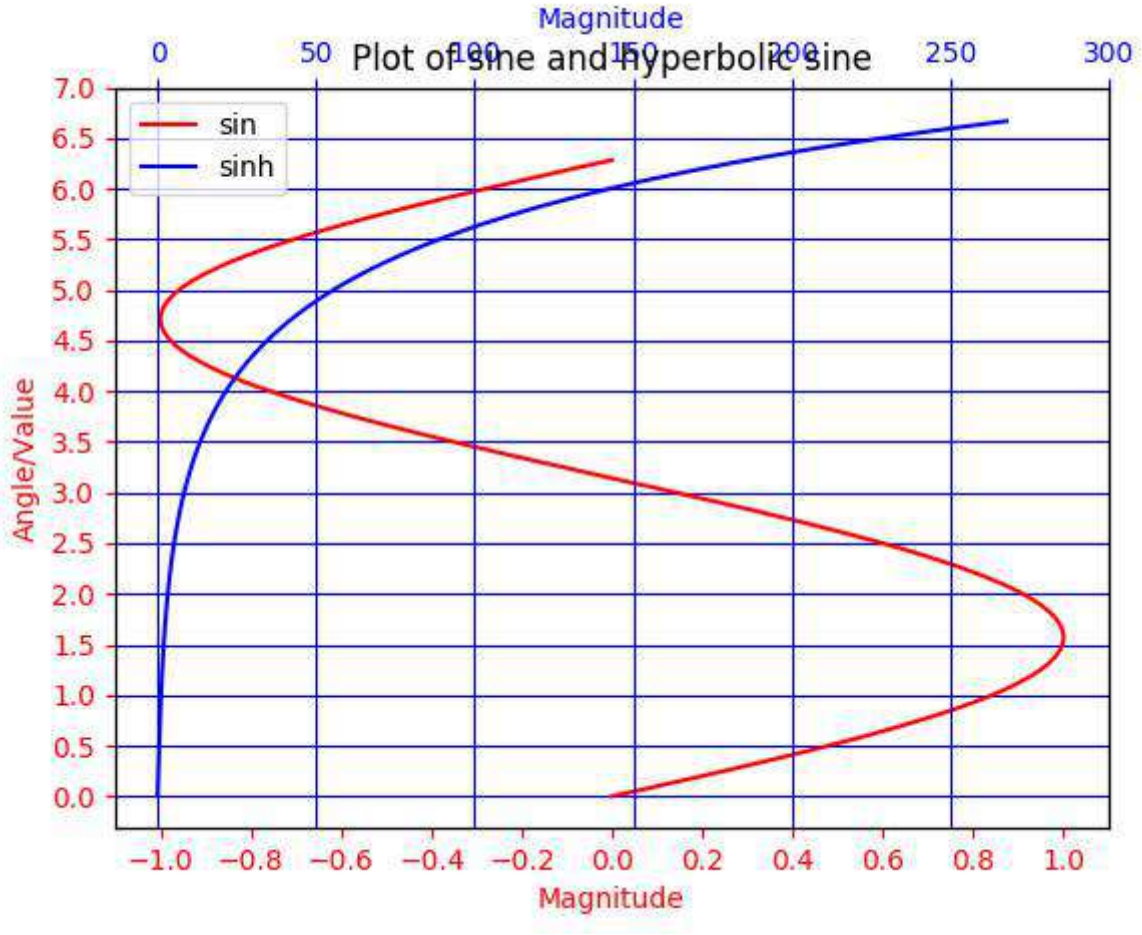


Section 65.2: Plots with common Y-axis and different X-axis using twiny()

In this example, a plot with curves having common y-axis but different x-axis is demonstrated using **twiny()** method. Also, some additional features such as the title, legend, labels, grids, axis ticks and colours are added to the plot.

```
# Plotting tutorials in Python
# Adding Multiple plots by twin y axis
# Good for plots having different x axis range
# Separate axes and figure objects
# replicate axes object and plot curves
# use axes to set attributes

import numpy as np
import matplotlib.pyplot as plt

y = np.linspace(0, 2.0*np.pi, 101)
x1 = np.sin(y)
x2 = np.sinh(y)

# values for making ticks in x and y axis
ynumbers = np.linspace(0, 7, 15)
xnumbers1 = np.linspace(-1, 1, 11)
xnumbers2 = np.linspace(0, 300, 7)

# separate the figure object and axes object
# from the plotting object
fig, ax1 = plt.subplots()
```

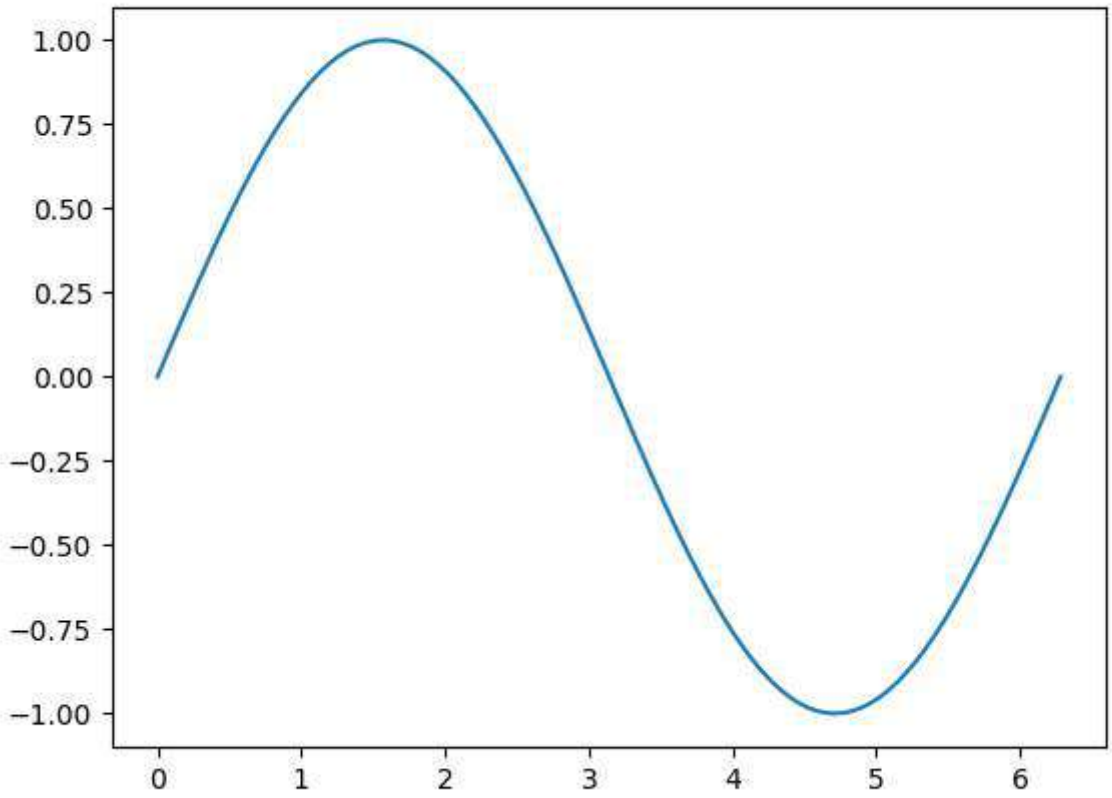
```
# Duplicate the axes with a different x axis
# and the same y axis
ax2 = ax1.twiny() # ax2 and ax1 will have common y axis and different x axis

# plot the curves on axes 1, and 2, and get the axes handles
curve1, = ax1.plot(x1, y, label="sin", color='r')
curve2, = ax2.plot(x2, y, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
# ax1.legend(curves, [curve.get_label() for curve in curves])
ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# x axis labels via the axes
ax1.set_xlabel("Magnitude", color=curve1.get_color())
ax2.set_xlabel("Magnitude", color=curve2.get_color())

# y axis label via the axes
ax1.set_ylabel("Angle/Value", color=curve1.get_color())
# ax2.set_ylabel("Magnitude", color=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# y ticks - make them coloured as well
ax1.tick_params(axis='y', colors=curve1.get_color())
# ax2.tick_params(axis='y', colors=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# x axis ticks via the axes
ax1.tick_params(axis='x', colors=curve1.get_color())
ax2.tick_params(axis='x', colors=curve2.get_color())

# set x ticks
ax1.set_xticks(xnumbers1)
ax2.set_xticks(xnumbers2)

# set y ticks
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # also works

# Grids via axes 1 # use this if axes 1 is used to
# define the properties of common x axis
# ax1.grid(color=curve1.get_color())

# To make grids using axes 2
ax1.grid(color=curve2.get_color())
ax2.grid(color=curve2.get_color())
ax1.xaxis.grid(False)

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```



Section 65.3: A Simple Plot in Matplotlib

This example illustrates how to create a simple sine curve using **Matplotlib**

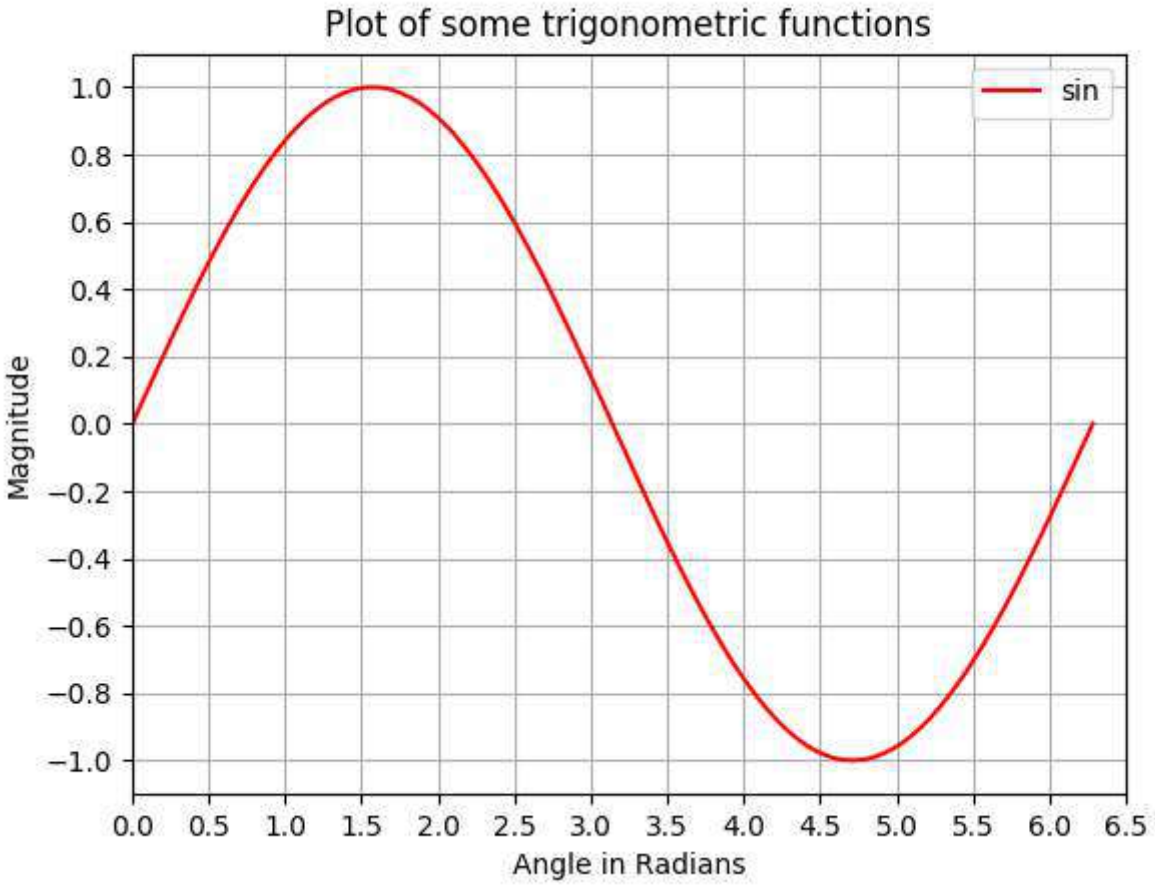
```
# Plotting tutorials in Python
# Launching a simple plot

import numpy as np
import matplotlib.pyplot as plt

# angle varying between 0 and 2*pi
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)                          # sine function

plt.plot(x, y)
plt.show()
```



Section 65.4: Adding more features to a simple plot : axis labels, title, axis ticks, grid, and legend

In this example, we take a sine curve plot and add more features to it; namely the title, axis labels, title, axis ticks, grid and legend.

```
# Plotting tutorials in Python
# Enhancing a plot

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

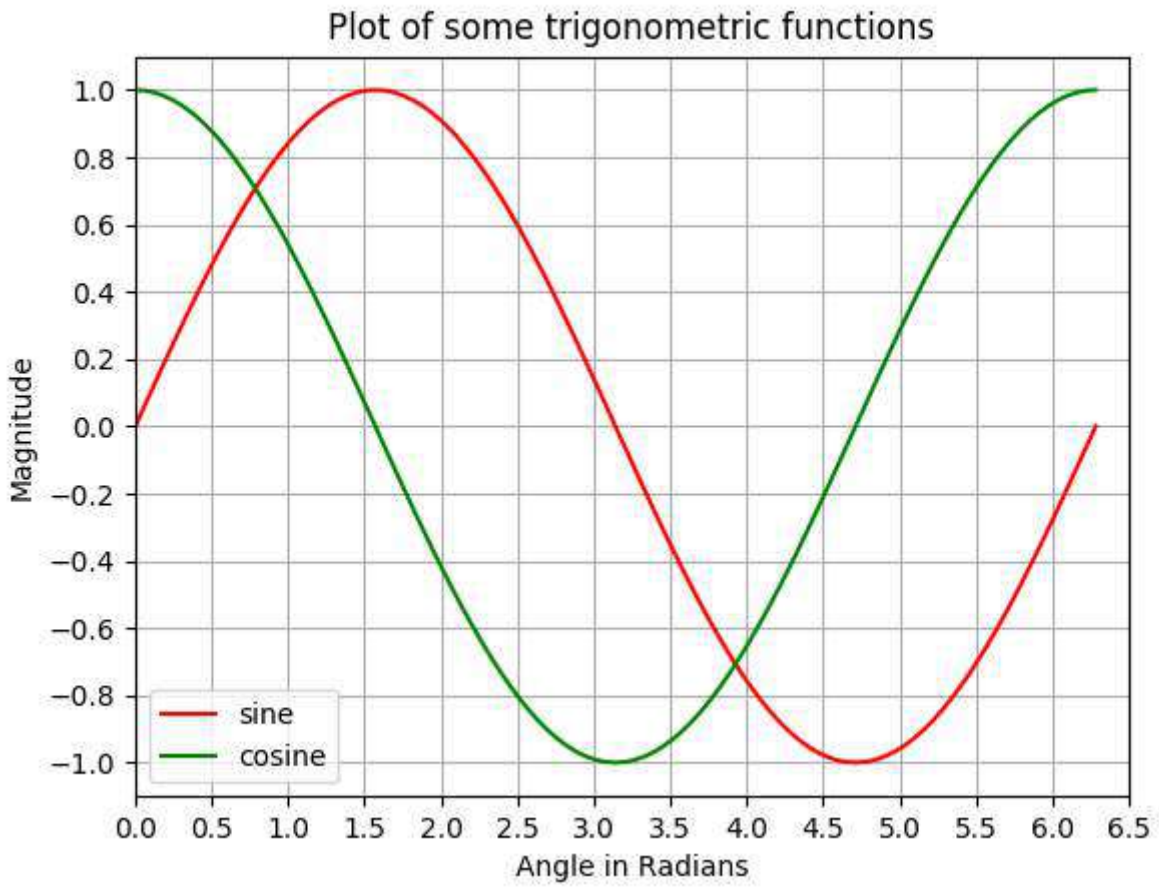


Section 65.5: Making multiple plots in the same figure by superimposition similar to MATLAB

In this example, a sine curve and a cosine curve are plotted in the same figure by superimposing the plots on top of each other.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using single plot command and legend

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, 'r', x, z, 'g') # r, g - red, green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend(['sine', 'cosine'])
plt.grid()
```

```
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

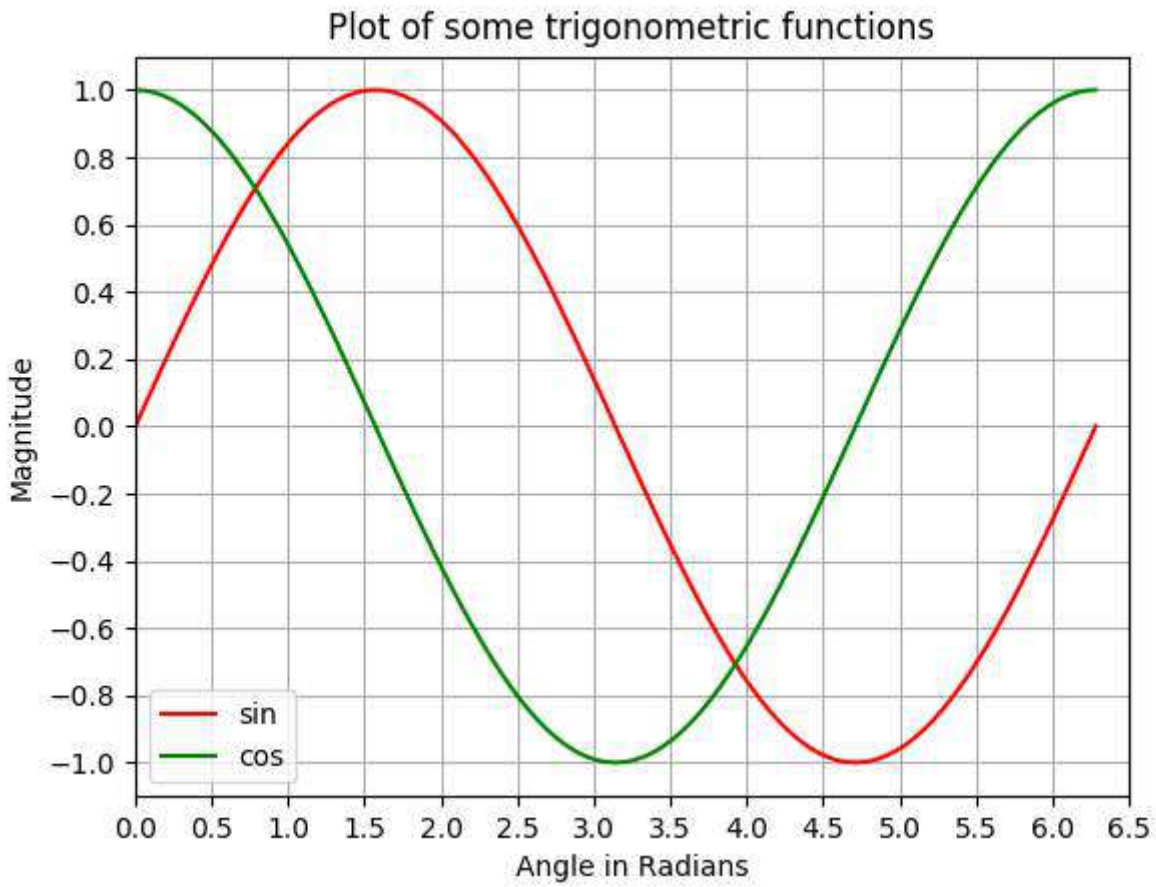


Section 65.6: Making multiple Plots in the same figure using plot superimposition with separate plot commands

Similar to the previous example, here, a sine and a cosine curve are plotted on the same figure using separate plot commands. This is more Pythonic and can be used to get separate handles for each plot.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using multiple plot commands
# Much better and preferred than previous

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.plot(x, z, color='g', label='cos') # g - green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
```

```
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```



Chapter 66: graph-tool

The python tools can be used to generate graph

Section 66.1: PyDotPlus

PyDotPlus is an improved version of the old pydot project that provides a Python Interface to Graphviz's Dot language.

Installation

For the latest stable version:

```
pip install pydotplus
```

For the development version:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

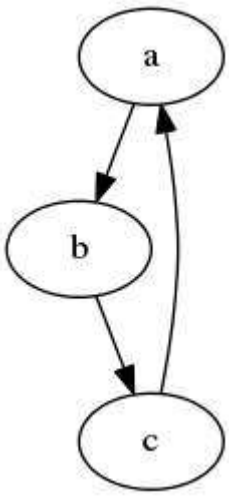
Load graph as defined by a DOT file

- The file is assumed to be in DOT format. It will be loaded, parsed and a Dot class will be returned, representing the graph. For example, a simple demo.dot:

> digraph demo1{ a -> b -> c; c ->a; }

```
import pydotplus
graph_a = pydotplus.graph_from_dot_file('demo.dot')
graph_a.write_svg('test.svg') # generate graph in svg.
```

You will get a svg(Scalable Vector Graphics) like this:



Section 66.2: PyGraphviz

Get PyGraphviz from the Python Package Index at http://pypi.python.org/pypi/pygraphviz

or install it with:

```
pip install pygraphviz
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

You can install the development version (at github.com) with:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Get PyGraphviz from the Python Package Index at http://pypi.python.org/pypi/pygraphviz

or install it with:

```
easy_install pygraphviz
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

Load graph as defined by a DOT file

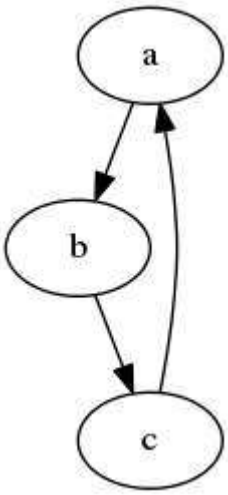
- The file is assumed to be in DOT format. It will be loaded, parsed and a Dot class will be returned, representing the graph. For example,a simple demo.dot:

> digraph demo1{ a -> b -> c; c ->a; }

- Load it and draw it.

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

You will get a svg(Scalable Vector Graphics) like this:



Chapter 67: Generators

Generators are lazy iterators created by generator functions (using `yield`) or generator expressions (using `(an_expression for x in an_iterator)`).

Section 67.1: Introduction

Generator expressions are similar to list, dictionary and set comprehensions, but are enclosed with parentheses. The parentheses do not have to be present when they are used as the sole argument for a function call.

```
expression = (x**2 for x in range(10))
```

This example generates the 10 first perfect squares, including 0 (in which x = 0).

Generator functions are similar to regular functions, except that they have one or more `yield` statements in their body. Such functions cannot `return` any values (however empty `return`s are allowed if you want to stop the generator early).

```
def function():
    for x in range(10):
        yield x**2
```

This generator function is equivalent to the previous generator expression, it outputs the same.

Note: all generator expressions have their own *equivalent* functions, but not vice versa.

A generator expression can be used without parentheses if both parentheses would be repeated otherwise:

```
sum(i for i in range(10) if i % 2 == 0)   #Output: 20
any(x = 0 for x in foo)                   #Output: True or False depending on foo
type(a > b for a in foo if a % 2 == 1)    #Output: <class 'generator'>
```

Instead of:

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

But not:

```
fooFunction(i for i in range(10) if i % 2 == 0,foo,bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

Calling a generator function produces a **generator object**, which can later be iterated over. Unlike other types of iterators, generator objects may only be traversed once.

```
g1 = function()
print(g1)  # Out: <generator object function at 0x1012e1888>
```

Notice that a generator's body is **not** immediately executed: when you call `function()` in the example above, it immediately returns a generator object, without executing even the first print statement. This allows generators to consume less memory than functions that return a list, and it allows creating generators that produce infinitely long sequences.

For this reason, generators are often used in data science, and other contexts involving large amounts of data. Another advantage is that other code can immediately use the values yielded by a generator, without waiting for the complete sequence to be produced.

However, if you need to use the values produced by a generator more than once, and if generating them costs more than storing, it may be better to store the yielded values as a `list` than to re-generate the sequence. See 'Resetting a generator' below for more details.

Typically a generator object is used in a loop, or in any function that requires an iterable:

```
for x in g1:
    print("Received", x)

# Output:
# Received 0
# Received 1
# Received 4
# Received 9
# Received 16
# Received 25
# Received 36
# Received 49
# Received 64
# Received 81

arr1 = list(g1)
# arr1 = [], because the loop above already consumed all the values.
g2 = function()
arr2 = list(g2)  # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Since generator objects are iterators, one can iterate over them manually using the `next()` function. Doing so will return the yielded values one by one on each subsequent invocation.

Under the hood, each time you call `next()` on a generator, Python executes statements in the body of the generator function until it hits the next `yield` statement. At this point it returns the argument of the `yield` command, and remembers the point where that happened. Calling `next()` once again will resume execution from that point and continue until the next `yield` statement.

If Python reaches the end of the generator function without encountering any more `yield`s, a `StopIteration` exception is raised (this is normal, all iterators behave in the same way).

```
g3 = function()
a = next(g3)  # a becomes 0
b = next(g3)  # b becomes 1
c = next(g3)  # c becomes 2
...
j = next(g3)  # Raises StopIteration, j remains undefined
```

Note that in Python 2 generator objects had `.next()` methods that could be used to iterate through the yielded values manually. In Python 3 this method was replaced with the `.__next__()` standard for all iterators.

Resetting a generator

Remember that you can only iterate through the objects generated by a generator *once*. If you have already iterated through the objects in a script, any further attempt do so will yield `None`.

If you need to use the objects generated by a generator more than once, you can either define the generator

function again and use it a second time, or, alternatively, you can store the output of the generator function in a list on first use. Re-defining the generator function will be a good option if you are dealing with large volumes of data, and storing a list of all data items would take up a lot of disc space. Conversely, if it is costly to generate the items initially, you may prefer to store the generated items in a list so that you can re-use them.

Section 67.2: Infinite sequences

Generators can be used to represent infinite sequences:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

Infinite sequence of numbers as above can also be generated with the help of `itertools.count`. The above code could be written as below

```
natural_numbers = itertools.count(1)
```

You can use generator comprehensions on infinite generators to produce new generators:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Be aware that an infinite generator does not have an end, so passing it to any function that will attempt to consume the generator entirely will have **dire consequences**:

```
list(multiples_of_two)  # will never terminate, or raise an OS-specific error
```

Instead, use list/set comprehensions with `range` (or `xrange` for python < 3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

or use `itertools.islice()` to slice the iterator to a subset:

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

Note that the original generator is updated too, just like all other generators coming from the same "root":

```
next(natural_numbers)    # yields 16
next(multiples_of_two)   # yields 34
next(multiples_of_four)  # yields 24
```

An infinite sequence can also be iterated with a `for`-loop. Make sure to include a conditional `break` statement so that the loop would terminate eventually:

```
for idx, number in enumerate(multiplies_of_two):
    print(number)
    if idx == 9:
```

```
        break  # stop after taking the first 10 multiplies of two
```

Classic example - Fibonacci numbers

```
import itertools

def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))

ninety_nineth_fib = nth_fib(99)  # 354224848179261915075
```

Section 67.3: Sending objects to a generator

In addition to receiving values from a generator, it is possible to *send* an object to a generator using the `send()` method.

```
def accumulator():
    total = 0
    value = None
    while True:
        # receive sent value
        value = yield total
        if value is None: break
        # aggregate values
        total += value

generator = accumulator()

# advance until the first "yield"
next(generator)      # 0

# from this point on, the generator aggregates values
generator.send(1)    # 1
generator.send(10)   # 11
generator.send(100)  # 111
# ...

# Calling next(generator) is equivalent to calling generator.send(None)
next(generator)      # StopIteration
```

What happens here is the following:

- When you first call `next(generator)`, the program advances to the first `yield` statement, and returns the value of `total` at that point, which is 0. The execution of the generator suspends at this point.
- When you then call `generator.send(x)`, the interpreter takes the argument `x` and makes it the return value of the last `yield` statement, which gets assigned to `value`. The generator then proceeds as usual, until it yields the next value.
- When you finally call `next(generator)`, the program treats this as if you're sending `None` to the generator. There is nothing special about `None`, however, this example uses `None` as a special value to ask the generator to stop.

Section 67.4: Yielding all values from another iterable

Python 3.x `Version ≥ 3.3`

Use `yield from` if you want to yield all values from another iterable:

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)

list(foob(5))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

This works with generators as well.

```
def fibto(n):
    a, b = 1, 1
    while True:
        if a >= n: break
        yield a
        a, b = b, a + b

def usefib():
    yield from fibto(10)
    yield from fibto(20)

list(usefib())  # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

Section 67.5: Iteration

A generator object supports the *iterator protocol*. That is, it provides a `next()` method (`__next__()` in Python 3.x), which is used to step through its execution, and its `__iter__` method returns itself. This means that a generator can be used in any language construct which supports generic iterable objects.

```
# naive partial implementation of the Python 2.x xrange()
def xrange(n):
    i = 0
    while i < n:
        yield i
        i += 1

# looping
for i in xrange(10):
    print(i)  # prints the values 0, 1, ..., 9

# unpacking
a, b, c = xrange(3)  # 0, 1, 2

# building a list
l = list(xrange(10))  # [0, 1, ..., 9]
```

Section 67.6: The next() function

The `next()` built-in is a convenient wrapper which can be used to receive a value from any iterator (including a generator iterator) and to provide a default value in case the iterator is exhausted.

```
def nums():
    yield 1
```

```
    yield 2
    yield 3
generator = nums()

next(generator, None)  # 1
next(generator, None)  # 2
next(generator, None)  # 3
next(generator, None)  # None
next(generator, None)  # None
# ...
```

The syntax is `next(iterator[, default])`. If iterator ends and a default value was passed, it is returned. If no default was provided, `StopIteration` is raised.

Section 67.7: Coroutines

Generators can be used to implement coroutines:

```
# create and advance generator to the first yield
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

# example coroutine
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# example use
s = adder()
s.send(1) # 1
s.send(2) # 3
```

Coroutines are commonly used to implement state machines, as they are primarily useful for creating single-method procedures that require a state to function properly. They operate on an existing state and return the value obtained on completion of the operation.

Section 67.8: Refactoring list-building code

Suppose you have complex code that creates and returns a list by starting with a blank list and repeatedly appending to it:

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

When it's not practical to replace the inner logic with a list comprehension, you can turn the entire function into a generator in-place, and then collect the results:

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

If the logic is recursive, use `yield from` to include all the values from the recursive call in a "flattened" result:

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

Section 67.9: Yield with recursion: recursively listing all files in a directory

First, import the libraries that work with files:

```
from os import listdir
from os.path import isfile, join, exists
```

A helper function to read only files from a directory:

```
def get_files(path):
    for file in listdir(path):
        full_path = join(path, file)
        if isfile(full_path):
            if exists(full_path):
                yield full_path
```

Another helper function to get only the subdirectories:

```
def get_directories(path):
    for directory in listdir(path):
        full_path = join(path, directory)
        if not isfile(full_path):
            if exists(full_path):
                yield full_path
```

Now use these functions to recursively get all files within a directory and all its subdirectories (using generators):

```
def get_files_recursive(directory):
    for file in get_files(directory):
        yield file
    for subdirectory in get_directories(directory):
        for file in get_files_recursive(subdirectory): # here the recursive call
            yield file
```

This function can be simplified using `yield from`:

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

Section 67.10: Generator expressions

It's possible to create generator iterators using a comprehension-like syntax.

```
generator = (i * 2 for i in range(3))

next(generator)  # 0
next(generator)  # 2
next(generator)  # 4
next(generator)  # raises StopIteration
```

If a function doesn't necessarily need to be passed a list, you can save on characters (and improve readability) by placing a generator expression inside a function call. The parenthesis from the function call implicitly make your expression a generator expression.

```
sum(i ** 2 for i in range(4))  # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Additionally, you will save on memory because instead of loading the entire list you are iterating over (`[0, 1, 2, 3]` in the above example), the generator allows Python to use values as needed.

Section 67.11: Using a generator to find Fibonacci Numbers

A practical use case of a generator is to iterate through values of an infinite series. Here's an example of finding the first ten terms of the Fibonacci Sequence.

```
def fib(a=0, b=1):
    """Generator that yields Fibonacci numbers. `a` and `b` are the seed values"""
    while True:
        yield a
        a, b = b, a + b

f = fib()
print(', '.join(str(next(f)) for _ in range(10)))
```

> 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Section 67.12: Searching

The `next` function is useful even without iterating. Passing a generator expression to `next` is a quick way to search for the first occurrence of an element matching some predicate. Procedural code like

```
def find_and_transform(sequence, predicate, func):
    for element in sequence:
        if predicate(element):
            return func(element)
    raise ValueError

item = find_and_transform(my_sequence, my_predicate, my_func)
```

can be replaced with:

```
item = next(my_func(x) for x in my_sequence if my_predicate(x))
# StopIteration will be raised if there are no matches; this exception can
# be caught and transformed, if desired.
```

For this purpose, it may be desirable to create an alias, such as `first = next`, or a wrapper function to convert the exception:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

Section 67.13: Iterating over generators in parallel

To iterate over several generators in parallel, use the `zip` builtin:

```
for x, y in zip(a,b):
    print(x,y)
```

Results in:

```
1 x
2 y
3 z
```

In python 2 you should use `itertools.izip` instead. Here we can also see that the all the `zip` functions yield tuples.

Note that zip will stop iterating as soon as one of the iterables runs out of items. If you'd like to iterate for as long as the longest iterable, use `itertools.zip_longest()`.

Chapter 68: Reduce

Parameter	Details
function	function that is used for reducing the iterable (must take two arguments). (*positional-only*)
iterable	iterable that's going to be reduced. (*positional-only*)
initializer	start-value of the reduction. (*optional*, *positional-only*)

Section 68.1: Overview

```
# No import needed


# No import required...
from functools import reduce # ... but it can be loaded from the functools module


from functools import reduce # mandatory
```

`reduce` reduces an iterable by applying a function repeatedly on the next element of an `iterable` and the cumulative result so far.

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence)  # equivalent to: add(add(1,2),3)
# Out: 6
```

In this example, we defined our own `add` function. However, Python comes with a standard equivalent function in the `operator` module:

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce` can also be passed a starting value:

```
reduce(add, asequence, 10)
# Out: 16
```

Section 68.2: Using reduce

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]
```

Given an `initializer` the function is started by applying it to the initializer and the first iterable element:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
```

```
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Without `initializer` parameter the `reduce` starts by applying the function to the first two list elements:

```
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

Section 68.3: Cumulative product

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

Section 68.4: Non short-circuit variant of any/all

`reduce` will not terminate the iteration before the `iterable` has been completely iterated over so it can be used to create a non short-circuit `any()` or `all()` function:

```
import operator
# non short-circuit "all"
reduce(operator.and_, [False, True, True, True]) # = False

# non short-circuit "any"
reduce(operator.or_, [True, False, False, False]) # = True
```

Chapter 69: Map Function

Parameter	Details
function	function for mapping (must take as many parameters as there are iterables) (*positional-only*)
iterable	the function is applied to each element of the iterable (*positional-only*)
*additional_iterables	see iterable, but as many as you like (*optional*, *positional-only*)

Section 69.1: Basic use of map, itertools.imap and future_builtins.map

The map function is the simplest one among Python built-ins used for functional programming. `map()` applies a specified function to each element in an iterable:

```
names = ['Fred', 'Wilma', 'Barney']
```

Python 3.x `Version ≥ 3.0`

```
map(len, names)  # map in Python 3.x is a class; its instances are iterable
# Out: <map object at 0x00000198B32E2CF8>
```

A Python 3-compatible `map` is included in the `future_builtins` module:

Python 2.x `Version ≥ 2.6`

```
from future_builtins import map  # contains a Python 3.x compatible map()
map(len, names)                  # see below
# Out: <itertools.imap instance at 0x3eb0a20>
```

Alternatively, in Python 2 one can use `imap` from `itertools` to get a generator

Python 2.x `Version ≥ 2.3`

```
map(len, names)   # map() returns a list
# Out: [4, 5, 6]

from itertools import imap
imap(len, names)  # itertools.imap() returns a generator
# Out: <itertools.imap at 0x405ea20>
```

The result can be explicitly converted to a `list` to remove the differences between Python 2 and 3:

```
list(map(len, names))
# Out: [4, 5, 6]
```

`map()` can be replaced by an equivalent *list comprehension* or *generator expression*:

```
[len(item) for item in names] # equivalent to Python 2.x map()
# Out: [4, 5, 6]

(len(item) for item in names) # equivalent to Python 3.x map()
# Out: <generator object <genexpr> at 0x00000195888D5FC0>
```

Section 69.2: Mapping each value in an iterable

For example, you can take the absolute value of each element:

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # the call to `list` is unnecessary in 2.x
```

```
# Out: [1, 1, 2, 2, 3, 3]
```

Anonymous function also support for mapping a list:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Out: [2, 4, 6, 8, 10]
```

or converting decimal values to percentages:

```
def to_percent(num):
    return num * 100

list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))
# Out: [95.0, 75.0, 101.0, 10.0]
```

or converting dollars to euros (given an exchange rate):

```
from functools import partial
from operator import mul

rate = 0.9  # fictitious exchange rate, 1 dollar = 0.9 euros
dollars = {'under_my_bed': 1000,
           'jeans': 45,
           'bank': 5000}

sum(map(partial(mul, rate), dollars.values()))
# Out: 5440.5
```

`functools.partial` is a convenient way to fix parameters of functions so that they can be used with `map` instead of using `lambda` or creating customized functions.

Section 69.3: Mapping values of different iterables

For example calculating the average of each `i`-th element of multiple iterables:

```
def average(*args):
    return float(sum(args)) / len(args)  # cast to float - only mandatory for python 2.x

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117, 91, 102]
measurement3 = [104, 102, 95, 101]

list(map(average, measurement1, measurement2, measurement3))
# Out: [102.0, 110.0, 95.0, 100.0]
```

There are different requirements if more than one iterable is passed to `map` depending on the version of python:

- The function must take as many parameters as there are iterables:

```
def median_of_three(a, b, c):
    return sorted((a, b, c))[1]

list(map(median_of_three, measurement1, measurement2))
```

> TypeError: median_of_three() missing 1 required positional argument: 'c'

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
```

> TypeError: median_of_three() takes 3 positional arguments but 4 were given

Python 2.x `Version ≥ 2.0.1`

- `map`: The mapping iterates as long as one iterable is still not fully consumed but assumes `None` from the fully consumed iterables:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

> TypeError: unsupported operand type(s) for -: 'int' and 'NoneType'

- `itertools.imap` and `future_builtins.map`: The mapping stops as soon as one iterable stops:

```
import operator
from itertools import imap

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(imap(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(imap(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Python 3.x `Version ≥ 3.0.0`

- The mapping stops as soon as one iterable stops:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(map(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Section 69.4: Transposing with Map: Using "None" as function argument (python 2.x only)

```
from itertools import imap
from future_builtins import map as fmap # Different name to highlight differences

image = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]

list(map(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]
list(map(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8), (3, None, 9)]  # Fill missing values with None
list(fmap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # ignore columns with missing values
list(imap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # dito
```

Python 3.x `Version ≥ 3.0.0`

```
list(map(None, *image))
```

> TypeError: 'NoneType' object is not callable

But there is a workaround to have similar results:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Section 69.5: Series and Parallel Mapping

map() is a built-in function, which means that it is available everywhere without the need to use an 'import' statement. It is available everywhere just like print() If you look at Example 5 you will see that I had to use an import statement before I could use pretty print (import pprint). Thus pprint is not a built-in function

Series mapping

In this case each argument of the iterable is supplied as argument to the mapping function in ascending order. This arises when we have just one iterable to map and the mapping function requires a single argument.

Example 1

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
print(list(map(f, insects))) # the function defined by f is executed on each item of the iterable
```

```
insects
```

results in

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Example 2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

results in

```
[3, 3, 6, 10]
```

Parallel mapping

In this case each argument of the mapping function is pulled from across all iterables (one from each iterable) in parallel. Thus the number of iterables supplied must match the number of arguments required by the function.

```
carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
omnivores = ['chicken', 'dove', 'mouse', 'pig']

def animals(w, x, y, z):
    return '{0}, {1}, {2}, and {3} ARE ALL ANIMALS'.format(w.title(), x, y, z)
```

Example 3

```
# Too many arguments
# observe here that map is trying to pass one item each from each of the four iterables to len. This
leads len to complain that
# it is being fed too many arguments
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

results in

```
TypeError: len() takes exactly one argument (4 given)
```

Example 4

```
# Too few arguments
# observe here that map is supposed to execute animal on individual elements of insects one-by-one.
But animals complain when
# it only gets one argument, whereas it was expecting four.
print(list(map(animals, insects)))
```

results in

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Example 5

```
# here map supplies w, x, y, z with one value from across the list
import pprint
pprint.pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

results in

```
['Fly, lion, african buffalo, and chicken ARE ALL ANIMALS',
'Ant, tiger, moose, and dove ARE ALL ANIMALS',
'Beetle, leopard, okapi, and mouse ARE ALL ANIMALS',
'Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS']
```

Chapter 70: Exponentiation

Section 70.1: Exponentiation using builtins: ** and pow()

Exponentiation can be used by using the builtin `pow`-function or the `**` operator:

```
2 ** 3    # 8
pow(2, 3) # 8
```

For most (all in Python 2.x) arithmetic operations the result's type will be that of the wider operand. This is not true for `**`; the following cases are exceptions from this rule:

- Base: `int`, exponent: `int < 0`:

  ```
  2 ** -3
  # Out: 0.125 (result is a float)
  ```

- This is also valid for Python 3.x.
- Before Python 2.2.0, this raised a `ValueError`.
- Base: `int < 0` or `float < 0`, exponent: `float != int`

  ```
  (-2) ** (0.5)  # also (-2.) ** (0.5)
  # Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
  ```

- Before python 3.0.0, this raised a `ValueError`.

The `operator` module contains two functions that are equivalent to the `**`-operator:

```
import operator
operator.pow(4, 2)      # 16
operator.__pow__(4, 3)  # 64
```

or one could directly call the `__pow__` method:

```
val1, val2 = 4, 2
val1.__pow__(val2)      # 16
val2.__rpow__(val1)     # 16
# in-place power operation isn't supported by immutable classes like int, float, complex:
# val1.__ipow__(val2)
```

Section 70.2: Square root: math.sqrt() and cmath.sqrt

The `math` module contains the `math.sqrt()`-function that can compute the square root of any number (that can be converted to a `float`) and the result will always be a `float`:

```
import math

math.sqrt(9)                # 3.0
math.sqrt(11.11)            # 3.3331666624997918
math.sqrt(Decimal('6.25'))  # 2.5
```

The `math.sqrt()` function raises a `ValueError` if the result would be `complex`:

```
math.sqrt(-10)
```

> ValueError: math domain error

`math.sqrt(x)` is *faster* than `math.pow(x, 0.5)` or `x ** 0.5` but the precision of the results is the same. The `cmath` module is extremely similar to the `math` module, except for the fact it can compute complex numbers and all of its results are in the form of a + bi. It can also use `.sqrt()`:

```
import cmath

cmath.sqrt(4)  # 2+0j
cmath.sqrt(-4) # 2j
```

What's with the `j`? `j` is the equivalent to the square root of -1. All numbers can be put into the form a + bi, or in this case, a + bj. `a` is the real part of the number like the 2 in `2+0j`. Since it has no imaginary part, `b` is 0. `b` represents part of the imaginary part of the number like the 2 in `2j`. Since there is no real part in this, `2j` can also be written as `0 + 2j`.

Section 70.3: Modular exponentiation: pow() with 3 arguments

Supplying `pow()` with 3 arguments `pow(a, b, c)` evaluates the modular exponentiation *ab mod c*:

```
pow(3, 4, 17)   # 13

# equivalent unoptimized expression:
3 ** 4 % 17     # 13

# steps:
3 ** 4          # 81
81 % 17         # 13
```

For built-in types using modular exponentiation is only possible if:

- First argument is an `int`
- Second argument is an `int >= 0`
- Third argument is an `int != 0`

These restrictions are also present in python 3.x

For example one can use the 3-argument form of `pow` to define a modular inverse function:

```
def modular_inverse(x, p):
    """Find a such as  a·x ≡ 1 (mod p), assuming p is prime."""
    return pow(x, p-2, p)

[modular_inverse(x, 13) for x in range(1,13)]
# Out: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

Section 70.4: Computing large integer roots

Even though Python natively supports big integers, taking the nth root of very large numbers can fail in Python.

```
x = 2 ** 100
cube = x ** 3
```

```
root = cube ** (1.0 / 3)
```

> OverflowError: long int too large to convert to float

When dealing with such large integers, you will need to use a custom function to compute the nth root of a number.

```
def nth_root(x, n):
    # Start with some reasonable bounds around the nth root.
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # Keep searching for a better result as long as the bounds make sense.
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # Found perfect nth root.
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# True
```

Section 70.5: Exponentiation using the math module: math.pow()

The `math`-module contains another `math.pow()` function. The difference to the builtin `pow()`-function or `**` operator is that the result is always a `float`:

```
import math
math.pow(2, 2)    # 4.0
math.pow(-2., 2)  # 4.0
```

Which excludes computations with complex inputs:

```
math.pow(2, 2+0j)
```

> TypeError: can't convert complex to float

and computations that would lead to complex results:

```
math.pow(-2, 0.5)
```

> ValueError: math domain error

Section 70.6: Exponential function: math.exp() and cmath.exp()

Both the `math` and `cmath`-module contain the Euler number: e and using it with the builtin `pow()`-function or `**`-operator works mostly like `math.exp()`:

```
import math

math.e ** 2  # 7.3890560989306495
math.exp(2)  # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

However the result is different and using the exponential function directly is more reliable than builtin exponentiation with base `math.e`:

```
print(math.e ** 10)       # 22026.465794806703
print(math.exp(10))       # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
#     difference starts here ---------------^
```

Section 70.7: Exponential function minus 1: math.expm1()

The `math` module contains the `expm1()`-function that can compute the expression `math.e ** x - 1` for very small `x` with higher precision than `math.exp(x)` or `cmath.exp(x)` would allow:

```
import math

print(math.e ** 1e-3 - 1)  # 0.0010005001667083846
print(math.exp(1e-3) - 1)  # 0.0010005001667083846
print(math.expm1(1e-3))    # 0.0010005001667083417
#                            ------------------^
```

For very small x the difference gets bigger:

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15))   # 1.0000000000000007e-15
#                              ^-------------------
```

The improvement is significant in scientific computing. For example the Planck's law contains an exponential function minus 1:

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)        # 4.139080074896474e-19
planks_law_naive(100, 5000)  # 4.139080073488451e-19
#                                        ^----------
```

```
planks_law(1000, 5000)       # 4.139080128493406e-23
planks_law_naive(1000, 5000) # 4.139080233183142e-23
#                                    ^------------
```

Section 70.8: Magic methods and exponentiation: builtin, math and cmath

Supposing you have a class that stores purely integer values:

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # Cast to an integer

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                     val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('Using __pow__')
            return self.__class__(self.value ** other)
        else:
            print('Using __pow__ with modulo')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('Using __float__')
        return float(self.value)

    def __complex__(self):
        print('Using __complex__')
        return complex(self.value, 0)
```

Using the builtin `pow` function or `**` operator always calls `__pow__`:

```
Integer(2) ** 2                # Integer(4)
# Prints: Using __pow__
Integer(2) ** 2.5              # Integer(5)
# Prints: Using __pow__
pow(Integer(2), 0.5)           # Integer(1)
# Prints: Using __pow__
operator.pow(Integer(2), 3)    # Integer(8)
# Prints: Using __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# Prints: Using __pow__
```

The second argument of the `__pow__()` method can only be supplied by using the builtin-`pow()` or by directly calling the method:

```
pow(Integer(2), 3, 4)          # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)       # Integer(0)
# Prints: Using __pow__ with modulo
```

While the `math`-functions always convert it to a `float` and use the float-computation:

```
import math
```

```
math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath`-functions try to convert it to `complex` but can also fallback to `float` if there is no explicit conversion to `complex`:

```
import cmath

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __complex__

del Integer.__complex__   # Deleting __complex__ method - instances cannot be cast to complex

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __float__
```

Neither `math` nor `cmath` will work if also the `__float__()`-method is missing:

```
del Integer.__float__  # Deleting __complex__ method

math.sqrt(Integer(2))  # also cmath.exp(Integer(2))
```

> TypeError: a float is required

Section 70.9: Roots: nth-root with fractional exponents

While the `math.sqrt` function is provided for the specific case of square roots, it's often convenient to use the exponentiation operator (`**`) with fractional exponents to perform nth-root operations, like cube roots.

The inverse of an exponentiation is exponentiation by the exponent's reciprocal. So, if you can cube a number by putting it to the exponent of 3, you can find the cube root of a number by putting it to the exponent of 1/3.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
>>> z == x
True
```

Chapter 71: Searching

Section 71.1: Searching for an element

All built-in collections in Python implement a way to check element membership using `in`.

List

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist   # True
10 in alist  # False
```

Tuple

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple    # False
'4' in atuple  # True
```

String

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

Set

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset  # True
10 in aset        # False
```

Dict

`dict` is a bit special: the normal `in` only checks the *keys*. If you want to search in *values* you need to specify it. The same if you want to search for *key-value* pairs.

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict                # True   - implicitly searches in keys
'a' in adict              # False
2 in adict.keys()         # True   - explicitly searches in keys
'a' in adict.values()     # True   - explicitly searches in values
(0, 'a') in adict.items()  # True   - explicitly searches key/value pairs
```

Section 71.2: Searching in custom classes: __contains__ and __iter__

To allow the use of `in` for custom classes the class must either provide the magic method `__contains__` or, failing that, an `__iter__`-method.

Suppose you have a class containing a `list` of `list`s:

```
class ListList:
    def __init__(self, value):
        self.value = value
        # Create a set of all values for fast access
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('Using __iter__.')
        # A generator over all sublist elements
        return (item for sublist in self.value for item in sublist)
```

```
    def __contains__(self, value):
        print('Using __contains__.')
        # Just lookup if the value is in the set
        return value in self.setofvalues

        # Even without the set you could use the iter method for the contains-check:
        # return any(item == value for item in iter(self))
```

Using membership testing is possible using `in`:

```
a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a    # False
# Prints: Using __contains__.
5 in a     # True
# Prints: Using __contains__.
```

even after deleting the `__contains__` method:

```
del ListList.__contains__
5 in a     # True
# Prints: Using __iter__.
```

Note: The looping `in` (as in `for i in a`) will always use `__iter__` even if the class implements a `__contains__` method.

Section 71.3: Getting the index for strings: str.index(), str.rindex() and str.find(), str.rfind()

`String` also have an `index` method but also more advanced options and the additional `str.find`. For both of these there is a complementary *reversed* method.

```
astring = 'Hello on StackOverflow'
astring.index('o')  # 4
astring.rindex('o') # 20

astring.find('o')   # 4
astring.rfind('o')  # 20
```

The difference between `index`/`rindex` and `find`/`rfind` is what happens if the substring is not found in the string:

```
astring.index('q') # ValueError: substring not found
astring.find('q')  # -1
```

All of these methods allow a start and end index:

```
astring.index('o', 5)    # 6
astring.index('o', 6)    # 6 - start is inclusive
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) #  - end is not inclusive
```

> ValueError: substring not found

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right
```

```
astring.rindex('o', 4, 7) # 6
```

Section 71.4: Getting the index list and tuples: list.index(), tuple.index()

`list` and `tuple` have an `index`-method to get the position of the element:

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# search for 16 in the list
alist.index(16) # 1
alist[1]        # 16

alist.index(15)
```

> ValueError: 15 is not in list

But only returns the position of the first found element:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26)   # 2
atuple[2]          # 26
atuple[7]          # 26 - is also 26!
```

Section 71.5: Searching key(s) for a value in dict

`dict` have no builtin method for searching a value or key because *dictionaries* are unordered. You can create a function that gets the key (or keys) for a specified value:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

This could also be written as an equivalent list comprehension:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

If you only care about one found key:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

The first two functions will return a `list` of all `keys` that have the specified value:

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10)     # ['c', 'a'] - order is random could as well be ['a', 'c']
getKeysForValueComp(adict, 10) # ['c', 'a'] - dito
getKeysForValueComp(adict, 20) # ['b']
getKeysForValueComp(adict, 25) # []
```

The other one will only return one key:

```
getOneKeyForValue(adict, 10)   # 'c'  - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)   # 'b'
```

and raise a `StopIteration-Exception` if the value is not in the `dict`:

```
getOneKeyForValue(adict, 25)
```

> StopIteration

Section 71.6: Getting the index for sorted sequences: bisect.bisect_left()

Sorted sequences allow the use of faster searching algorithms: `bisect.bisect_left()`[1]:

```
import bisect

def index_sorted(sorted_seq, value):
    """Locate the leftmost value exactly equal to x or raise a ValueError"""
    i = bisect.bisect_left(sorted_seq, value)
    if i != len(sorted_seq) and sorted_seq[i] == value:
        return i
    raise ValueError

alist = [i for i in range(1, 100000, 3)] # Sorted list from 1 to 100000 with step 3
index_sorted(alist, 97285) # 32428
index_sorted(alist, 4)     # 1
index_sorted(alist, 97286)
```

> ValueError

For very large **sorted sequences** the speed gain can be quite high. In case for the first search approximately 500 times as fast:

```
%timeit index_sorted(alist, 97285)
# 100000 loops, best of 3: 3 µs per loop
%timeit alist.index(97285)
# 1000 loops, best of 3: 1.58 ms per loop
```

While it's a bit slower if the element is one of the very first:

```
%timeit index_sorted(alist, 4)
# 100000 loops, best of 3: 2.98 µs per loop
%timeit alist.index(4)
# 1000000 loops, best of 3: 580 ns per loop
```

Section 71.7: Searching nested sequences

Searching in nested sequences like a `list` of `tuple` requires an approach like searching the keys for values in `dict` but needs customized functions.

The index of the outermost sequence if the value was found in the sequence:

```
def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
                      for item in inner
                      if item == value)

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a')  # 1
outer_index(alist_of_tuples, 4.3)  # 2
```

or the index of the outer and inner sequence:

```
def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
                                 for iindex, item in enumerate(inner)
                                 if item == value)

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2]  # 'a'

outer_inner_index(alist_of_tuples, 7)   # (2, 0)
alist_of_tuples[2][0]  # 7
```

In general (*not always*) using `next` and a **generator expression** with conditions to find the first occurrence of the searched value is the most efficient approach.

Chapter 72: Sorting, Minimum and Maximum

Section 72.1: Make custom classes orderable

`min`, `max`, and `sorted` all need the objects to be orderable. To be properly orderable, the class needs to define all of the 6 methods `__lt__`, `__gt__`, `__ge__`, `__le__`, `__ne__` and `__eq__`:

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - Test less than or equal to {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - Test greater than {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - Test greater than or equal to {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value
```

Though implementing all these methods would seem unnecessary, omitting some of them will make your code prone to bugs.

Examples:

```
alist = [IntegerContainer(5), IntegerContainer(3),
         IntegerContainer(10), IntegerContainer(7)
        ]

res = max(alist)
# Out: IntegerContainer(3) - Test greater than IntegerContainer(5)
#      IntegerContainer(10) - Test greater than IntegerContainer(5)
#      IntegerContainer(7) - Test greater than IntegerContainer(10)
print(res)
# Out: IntegerContainer(10)

res = min(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
```

```
#      IntegerContainer(7) - Test less than IntegerContainer(3)
print(res)
# Out: IntegerContainer(3)

res = sorted(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(10) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(10)
print(res)
# Out: [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]
```

`sorted` with `reverse=True` also uses `__lt__`:

```
res = sorted(alist, reverse=True)
# Out: IntegerContainer(10) - Test less than IntegerContainer(7)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(3)
print(res)
# Out: [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]
```

But `sorted` can use `__gt__` instead if the default is not implemented:

```
del IntegerContainer.__lt__   # The IntegerContainer no longer implements "less than"

res = min(alist)
# Out: IntegerContainer(5) - Test greater than IntegerContainer(3)
#      IntegerContainer(3) - Test greater than IntegerContainer(10)
#      IntegerContainer(3) - Test greater than IntegerContainer(7)
print(res)
# Out: IntegerContainer(3)
```

Sorting methods will raise a `TypeError` if neither `__lt__` nor `__gt__` are implemented:

```
del IntegerContainer.__gt__   # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

> TypeError: unorderable types: IntegerContainer() < IntegerContainer()

`functools.total_ordering` decorator can be used simplifying the effort of writing these rich comparison methods. If you decorate your class with `total_ordering`, you need to implement `__eq__`, `__ne__` and only one of the `__lt__`, `__le__`, `__ge__` or `__gt__`, and the decorator will fill in the rest:

```
import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)
```

```
    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value


IntegerContainer(5) > IntegerContainer(6)
# Output: IntegerContainer(5) - Test less than IntegerContainer(6)
# Returns: False

IntegerContainer(6) > IntegerContainer(5)
# Output: IntegerContainer(6) - Test less than IntegerContainer(5)
# Output: IntegerContainer(6) - Test equal to IntegerContainer(5)
# Returns True
```

Notice how the `>` (*greater than*) now ends up calling the *less than* method, and in some cases even the `__eq__` method. This also means that if speed is of great importance, you should implement each rich comparison method yourself.

Section 72.2: Special case: dictionaries

Getting the minimum or maximum or using `sorted` depends on iterations over the object. In the case of `dict`, the iteration is only over the keys:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

To keep the dictionary structure, you have to iterate over the `.items()`:

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

For `sorted`, you could create an `OrderedDict` to keep the sorting while having a `dict`-like structure:

```
from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Output: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Output: 3
```

By value

Again this is possible using the `key` argument:

```
min(adict.items(), key=lambda x: x[1])
# Output: ('c', 1)
max(adict.items(), key=operator.itemgetter(1))
# Output: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Output: [('b', 5), ('a', 3), ('c', 1)]
```

Section 72.3: Using the key argument

Finding the minimum/maximum of a sequence of sequences is possible:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

but if you want to sort by a specific element in each sequence use the `key`-argument:

```
min(list_of_tuples, key=lambda x: x[0])         # Sorting by first element
# Output: (0, 10)

min(list_of_tuples, key=lambda x: x[1])         # Sorting by second element
# Output: (2, 8)

sorted(list_of_tuples, key=lambda x: x[0])      # Sorting by first element (increasing)
# Output: [(0, 10), (1, 15), (2, 8)]

sorted(list_of_tuples, key=lambda x: x[1])      # Sorting by first element
# Output: [(2, 8), (0, 10), (1, 15)]

import operator
# The operator module contains efficient alternatives to the lambda function
max(list_of_tuples, key=operator.itemgetter(0)) # Sorting by first element
# Output: (2, 8)

max(list_of_tuples, key=operator.itemgetter(1)) # Sorting by second element
# Output: (1, 15)

sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # Reversed (decreasing)
# Output: [(2, 8), (1, 15), (0, 10)]

sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # Reversed(decreasing)
# Output: [(1, 15), (0, 10), (2, 8)]
```

Section 72.4: Default Argument to max, min

You can't pass an empty sequence into `max` or `min`:

```
min([])
```

> ValueError: min() arg is an empty sequence

However, with Python 3, you can pass in the keyword argument `default` with a value that will be returned if the sequence is empty, instead of raising an exception:

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

Section 72.5: Getting a sorted sequence

Using **one** sequence:

```
sorted((7, 2, 1, 5))                # tuple
# Output: [1, 2, 5, 7]

sorted(['c', 'A', 'b'])             # list
# Output: ['A', 'b', 'c']

sorted({11, 8, 1})                  # set
# Output: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15})  # dict
# Output: ['10', '11', '3']          # only iterates over the keys

sorted('bdca')                      # string
# Output: ['a','b','c','d']
```

The result is always a new `list`; the original data remains unchanged.

Section 72.6: Extracting N largest or N smallest items from an iterable

To find some number (more than one) of largest or smallest values of an iterable, you can use the `nlargest` and `nsmallest` of the `heapq` module:

```
import heapq

# get 5 largest items from the range

heapq.nlargest(5, range(10))
# Output: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# Output: [0, 1, 2, 3, 4]
```

This is much more efficient than sorting the whole iterable and then slicing from the end or beginning. Internally these functions use the binary heap priority queue data structure, which is very efficient for this use case.

Like `min`, `max` and `sorted`, these functions accept the optional `key` keyword argument, which must be a function that, given an element, returns its sort key.

Here is a program that extracts 1000 longest lines from a file:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Here we open the file, and pass the file handle `f` to `nlargest`. Iterating the file yields each line of the file as a separate string; `nlargest` then passes each element (or line) is passed to the function `len` to determine its sort key.

`len`, given a string, returns the length of the line in characters.

This only needs storage for a list of 1000 largest lines so far, which can be contrasted with

```
longest_lines = sorted(f, key=len)[1000:]
```

which will have to hold *the entire file in memory*.

Section 72.7: Getting the minimum or maximum of several values

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

Section 72.8: Minimum and Maximum of a sequence

Getting the minimum of a sequence (iterable) is equivalent of accessing the first element of a `sorted` sequence:

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

The maximum is a bit more complicated, because `sorted` keeps order and `max` returns the first encountered value. In case there are no duplicates the maximum is the same as the last element of the sorted return:

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

But not if there are multiple elements that are evaluated as having the maximum value:

```
class MyClass(object):
    def __init__(self, value, name):
        self.value = value
        self.name = name

    def __lt__(self, other):
        return self.value < other.value

    def __repr__(self):
        return str(self.name)

sorted([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: [second, first, third]
max([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: first
```

Any iterable containing elements that support `<` or `>` operations are allowed.

Chapter 73: Counting

Section 73.1: Counting all occurrence of all items in an iterable: collections.Counter

```
from collections import Counter

c = Counter(["a", "b", "c", "d", "a", "b", "a", "c", "d"])
c
# Out: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 2})
c["a"]
# Out: 3

c[7]     # not in the list (7 occurred 0 times!)
# Out: 0
```

The `collections.Counter` can be used for any iterable and counts every occurrence for every element.

Note: One exception is if a `dict` or another `collections.Mapping`-like class is given, then it will not count them, rather it creates a Counter with these values:

```
Counter({"e": 2})
# Out: Counter({"e": 2})

Counter({"e": "e"})        # warning Counter does not verify the values are int
# Out: Counter({"e": "e"})
```

Section 73.2: Getting the most common value(-s): collections.Counter.most_common()

Counting the *keys* of a `Mapping` isn't possible with `collections.Counter` but we can count the *values*:

```
from collections import Counter
adict = {'a': 5, 'b': 3, 'c': 5, 'd': 2, 'e':2, 'q': 5}
Counter(adict.values())
# Out: Counter({2: 2, 3: 1, 5: 3})
```

The most common elements are available by the `most_common`-method:

```
# Sorting them from most-common to least-common value:
Counter(adict.values()).most_common()
# Out: [(5, 3), (2, 2), (3, 1)]

# Getting the most common value
Counter(adict.values()).most_common(1)
# Out: [(5, 3)]

# Getting the two most common values
Counter(adict.values()).most_common(2)
# Out: [(5, 3), (2, 2)]
```

Section 73.3: Counting the occurrences of one item in a sequence: list.count() and tuple.count()

```
alist = [1, 2, 3, 4, 1, 2, 1, 3, 4]
```

```
alist.count(1)
# Out: 3

atuple = ('bear', 'weasel', 'bear', 'frog')
atuple.count('bear')
# Out: 2
atuple.count('fox')
# Out: 0
```

Section 73.4: Counting the occurrences of a substring in a string: str.count()

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

This works even for substrings longer than one character:

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

which would not be possible with `collections.Counter` which only counts single characters:

```
from collections import Counter
Counter(astring)
# Out: Counter({'a': 1, 'e': 1, 'h': 2, 'i': 2, 'o': 1, 'r': 1, 's': 3, 't': 4, 'x': 1})
```

Section 73.5: Counting occurrences in numpy array

To count the occurrences of a value in a numpy array. This will work:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

The logic is that the boolean statement produces a array where all occurrences of the requested values are 1 and all others are zero. So summing these gives the number of occurencies. This works for arrays of any shape or dtype.

There are two methods I use to count occurrences of all unique values in numpy. Unique and bincount. Unique automatically flattens multidimensional arrays, while bincount only works with 1d arrays only containing positive integers.

```
>>> unique,counts=np.unique(a,return_counts=True)
>>> print unique,counts # counts[i] is equal to occurrences of unique[i] in a
[0 3 4 5 7] [1 2 2 1 1]
>>> bin_count=np.bincount(a)
>>> print bin_count # bin_count[i] is equal to occurrences of i in a
[1 0 0 2 2 1 0 1]
```

If your data are numpy arrays it is generally much faster to use numpy methods then to convert your data to generic methods.

Chapter 74: The Print Function

Section 74.1: Print basics

In Python 3 and higher, `print` is a function rather than a keyword.

```
print('hello world!')
# out: hello world!

foo = 1
bar = 'bar'
baz = 3.14

print(foo)
# out: 1
print(bar)
# out: bar
print(baz)
# out: 3.14
```

You can also pass a number of parameters to `print`:

```
print(foo, bar, baz)
# out: 1 bar 3.14
```

Another way to `print` multiple parameters is by using a `+`

```
print(str(foo) + " " + bar + " " + str(baz))
# out: 1 bar 3.14
```

What you should be careful about when using `+` to print multiple parameters, though, is that the type of the parameters should be the same. Trying to print the above example without the cast to `string` first would result in an error, because it would try to add the number `1` to the string `"bar"` and add that to the number `3.14`.

```
# Wrong:
# type:int  str  float
print(foo + bar + baz)
# will result in an error
```

This is because the content of `print` will be evaluated first:

```
print(4 + 5)
# out: 9
print("4" + "5")
# out: 45
print([4] + [5])
# out: [4, 5]
```

Otherwise, using a `+` can be very helpful for a user to read output of variables In the example below the output is very easy to read!

The script below demonstrates this

```
import random
#telling python to include a function to create random numbers
randnum = random.randint(0, 12)
```

```
#make a random number between 0 and 12 and assign it to a variable
print("The randomly generated number was - " + str(randnum))
```

You can prevent the `print` function from automatically printing a newline by using the `end` parameter:

```
print("this has no newline at the end of it... ", end="")
print("see?")
# out: this has no newline at the end of it... see?
```

If you want to write to a file, you can pass it as the parameter `file`:

```
with open('my_file.txt', 'w+') as my_file:
    print("this goes to the file!", file=my_file)
```

> this goes to the file!

Section 74.2: Print parameters

You can do more than just print text. `print` also has several parameters to help you.

Argument `sep`: place a string between arguments.

Do you need to print a list of words separated by a comma or some other string?

```
>>> print('apples','bananas', 'cherries', sep=', ')
apple, bananas, cherries
>>> print('apple','banana', 'cherries', sep=', ')
apple, banana, cherries
>>>
```

Argument `end`: use something other than a newline at the end

Without the `end` argument, all `print()` functions write a line and then go to the beginning of the next line. You can change it to do nothing (use an empty string of ''), or double spacing between paragraphs by using two newlines.

```
>>> print("<a", end=''); print(" class='jidn'" if 1 else "", end=''); print("/>")
<a class='jidn'/>
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1

paragraph2
>>>
```

Argument `file`: send output to someplace other than sys.stdout.

Now you can send your text to either stdout, a file, or StringIO and not care which you are given. If it quacks like a file, it works like a file.

```
>>> def sendit(out, *values, sep=' ', end='\n'):
...     print(*values, sep=sep, end=end, file=out)
...
>>> sendit(sys.stdout, 'apples', 'bananas', 'cherries', sep='\t')
apples    bananas    cherries
>>> with open("delete-me.txt", "w+") as f:
...     sendit(f, 'apples', 'bananas', 'cherries', sep=' ', end='\n')
```

```
...
>>> with open("delete-me.txt", "rt") as f:
...     print(f.read())
...
apples bananas cherries

>>>
```

There is a fourth parameter `flush` which will forcibly flush the stream.

Chapter 75: Regular Expressions (Regex)

Python makes regular expressions available through the `re` module.

Regular expressions are combinations of characters that are interpreted as rules for matching substrings. For instance, the expression `'amount\D+\d+'` will match any string composed by the word `amount` plus an integral number, separated by one or more non-digits, such as:`amount=100`, `amount is 3`, `amount is equal to: 33`, etc.

Section 75.1: Matching the beginning of a string

The first argument of `re.match()` is the regular expression, the second is the string to match:

```
import re

pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Out: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)

match.group()
# Out: '123'
```

You may notice that the pattern variable is a string prefixed with `r`, which indicates that the string is a *raw string literal*.

A raw string literal has a slightly different syntax than a string literal, namely a backslash `\` in a raw string literal means "just a backslash" and there's no need for doubling up backlashes to escape "escape sequences" such as newlines (`\n`), tabs (`\t`), backspaces (`\`), form-feeds (`\r`), and so on. In normal string literals, each backslash must be doubled up to avoid being taken as the start of an escape sequence.

Hence, `r"\n"` is a string of 2 characters: `\` and `n`. Regex patterns also use backslashes, e.g. `\d` refers to any digit character. We can avoid having to double escape our strings (`"\\d"`) by using raw strings (`r"\d"`).

For instance:

```
string = "\\t123zzb" # here the backslash is escaped, so there's no tab, just '\' and 't'
pattern = "\\t123"   # this will match \t (escaping the backslash) followed by 123
re.match(pattern, string).group()   # no match
re.match(pattern, "\t123zzb").group()  # matches '\t123'

pattern = r"\\t123"
re.match(pattern, string).group()   # matches '\\t123'
```

Matching is done from the start of the string only. If you want to match anywhere use `re.search` instead:

```
match = re.match(r"(123)", "a123zzb")

match is None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
```

```
# Out: '123'
```

Section 75.2: Searching

```
pattern = r"(your base)"
sentence = "All your base are belong to us."

match = re.search(pattern, sentence)
match.group(1)
# Out: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# Out: 'belong to us.'
```

Searching is done anywhere in the string unlike `re.match`. You can also use `re.findall`.

You can also search at the beginning of the string (use `^`),

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

at the end of the string (use $),

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

or both (use both `^` and $):

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

Section 75.3: Precompiled patterns

```
import re

precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("The answer is 41!")
matches.group(1)
# Out: 41

matches = precompiled_pattern.search("Or was it 42?")
matches.group(1)
# Out: 42
```

Compiling a pattern allows it to be reused later on in a program. However, note that Python caches recently-used

expressions (docs, SO answer), so *"programs that use only a few regular expressions at a time needn't worry about compiling regular expressions"*.

```
import re

precompiled_pattern = re.compile(r"(.*\d+)")
matches = precompiled_pattern.match("The answer is 41!")
print(matches.group(1))
# Out: The answer is 41

matches = precompiled_pattern.match("Or was it 42?")
print(matches.group(1))
# Out: Or was it 42
```

It can be used with re.match().

Section 75.4: Flags

For some special cases we need to change the behavior of the Regular Expression, this is done using flags. Flags can be set in two ways, through the `flags` keyword or directly in the expression.

Flags keyword

Below an example for `re.search` but it works for most functions in the `re` module.

```
m = re.search("b", "ABC")
m is None
# Out: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# Out: 'B'

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE)
m is None
# Out: True

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# Out: 'A\nB'
```

Common Flags

Flag	Short Description
`re.IGNORECASE`, `re.I`	Makes the pattern ignore the case
`re.DOTALL`, `re.S`	Makes `.` match everything including newlines
`re.MULTILINE`, `re.M`	Makes `^` match the begin of a line and `$` the end of a line
`re.DEBUG`	Turns on debug information

For the complete list of all available flags check the docs

Inline flags

From the docs:

> `(?iLmsux)` (One or more letters from the set 'i', 'L', 'm', 's', 'u', 'x'.)

> The group matches the empty string; the letters set the corresponding flags: re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode dependent), and re.X (verbose), for the entire regular expression. This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function.
>
> Note that the (?x) flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

Section 75.5: Replacing

Replacements can be made on strings using `re.sub`.

Replacing strings

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# Out: 'my name foo is foo what foo ever foo'
```

Using group references

Replacements with a small number of groups can be made as follows:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

However, if you make a group ID like '10', this doesn't work: `\10` is read as 'ID number 1 followed by 0'. So you have to be more specific and use the `\g<i>` notation:

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Using a replacement function

```
items = ["zero", "one", "two"]
re.sub(r"a\[([0-3])\]", lambda match: items[int(match.group(1))], "Items: a[0], a[1], something,
a[2]")
# Out: 'Items: zero, one, something, two'
```

Section 75.6: Find All Non-Overlapping Matches

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Note that the `r` before `"[0-9]{2,3}"` tells python to interpret the string as-is; as a "raw" string.

You could also use `re.finditer()` which works in the same way as `re.findall()` but returns an iterator with `SRE_Match` objects instead of a list of strings:

```
results = re.finditer(r"([0-9]{2,3})", "some 1 text 12 is 945 here 4445588899")
print(results)
# Out: <callable-iterator object at 0x105245890>
for result in results:
    print(result.group(0))
''' Out:
12
945
444
558
```

```
889
'''
```

Section 75.7: Checking for allowed characters

If you want to check that a string contains only a certain set of characters, in this case a-z, A-Z and 0-9, you can do so like this,

```
import re

def is_allowed(string):
    characherRegex = re.compile(r'[^a-zA-Z0-9.]')
    string = characherRegex.search(string)
    return not bool(string)

print (is_allowed("abyzABYZ0099"))
# Out: 'True'

print (is_allowed("#*@#$%^"))
# Out: 'False'
```

You can also adapt the expression line from `[^a-zA-Z0-9.]` to `[^a-z0-9.]`, to disallow uppercase letters for example.

Partial credit: http://stackoverflow.com/a/1325265/2697955

Section 75.8: Splitting a string using regular expressions

You can also use regular expressions to split a string. For example,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

Section 75.9: Grouping

Grouping is done with parentheses. Calling `group()` returns a string formed of the matching parenthesized subgroups.

```
match.group() # Group without argument returns the entire match found
# Out: '123'
match.group(0) # Specifying 0 gives the same result as specifying no argument
# Out: '123'
```

Arguments can also be provided to `group()` to fetch a particular subgroup.

From the docs:

> If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument.

Calling `groups()` on the other hand, returns a list of tuples containing the subgroups.

```
sentence = "This is a phone number 672-123-456-9910"
pattern = r".*(phone).*?([\d-]+)"

match = re.match(pattern, sentence)

match.groups()   # The entire match as a list of tuples of the paranthesized subgroups
# Out: ('phone', '672-123-456-9910')

m.group()        # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(0)       # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(1)       # The first parenthesized subgroup.
# Out: 'phone'

m.group(2)       # The second parenthesized subgroup.
# Out: '672-123-456-9910'

m.group(1, 2)    # Multiple arguments give us a tuple.
# Out: ('phone', '672-123-456-9910')
```

Named groups

```
match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')
match.group('name')
# Out: 'John Smith'

match.group(1)
# Out: 'John Smith'
```

Creates a capture group that can be referenced by name as well as by index.

Non-capturing groups

Using `(?:)` creates a group, but the group isn't captured. This means you can use it as a group, but it won't pollute your "group space".

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')

re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

This example matches `11+22` or `11`, but not `11+`. This is since the `+` sign and the second term are grouped. On the other hand, the `+` sign isn't captured.

Section 75.10: Escaping Special Characters

Special characters (like the character class brackets `[` and `]` below) are not matched literally:

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

By escaping the special characters, they can be matched literally:

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
```

```
# Out: '[b]'
```

The `re.escape()` function can be used to do this for you:

```
re.escape('a[b]c')
# Out: 'a\\[b\\]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# Out: 'a[b]c'
```

The `re.escape()` function escapes all special characters, so it is useful if you are composing a regular expression based on user input:

```
username = 'A.C.'  # suppose this came from the user
re.findall(r'Hi {}!'.format(username), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!', 'Hi ABCD!']
re.findall(r'Hi {}!'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!']
```

Section 75.11: Match an expression only in specific locations

Often you want to match an expression only in *specific* places (leaving them untouched in others, that is). Consider the following sentence:

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

Here the "apple" occurs twice which can be solved with so called *backtracking control verbs* which are supported by the newer **regex** module. The idea is:

```
forget_this | or this | and this as well | (but keep this)
```

With our apple example, this would be:

```
import regex as re
string = "An apple a day keeps the doctor away (I eat an apple everyday)."
rx = re.compile(r'''
    \([^()]*\) (*SKIP)(*FAIL)  # match anything in parentheses and "throw it away"
    |                          # or
    apple                      # match an apple
    ''', re.VERBOSE)
apples = rx.findall(string)
print(apples)
# only one
```

This matches "apple" only when it can be found outside of the parentheses.

Here's how it works:

- While looking from **left to right**, the regex engine consumes everything to the left, the `(*SKIP)` acts as an "always-true-assertion". Afterwards, it correctly fails on `(*FAIL)` and backtracks.
- Now it gets to the point of `(*SKIP)` **from right to left** (aka while backtracking) where it is forbidden to go any further to the left. Instead, the engine is told to throw away anything to the left and jump to the point where the `(*SKIP)` was invoked.

Section 75.12: Iterating over matches using `re.finditer`

You can use `re.finditer` to iterate over all matches in a string. This gives you (in comparison to `re.findall` extra information, such as information about the match location in the string (indexes):

```
import re
text = 'You can try to find an ant in this string'
pattern = 'an?\w' # find 'an' either with or without a following word character

for match in re.finditer(pattern, text):
    # Start index of match (integer)
    sStart = match.start()

    # Final index of match (integer)
    sEnd = match.end()

    # Complete match (string)
    sGroup = match.group()

    # Print match
    print('Match "{}" found at: [{},{}]'.format(sGroup, sStart,sEnd))
```

Result:

```
Match "an" found at: [5,7]
Match "an" found at: [20,22]
Match "ant" found at: [23,26]
```

Chapter 76: Copying data

Section 76.1: Copy a dictionary

A dictionary object has the method `copy`. It performs a shallow copy of the dictionary.

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
True
```

Section 76.2: Performing a shallow copy

A shallow copy is a copy of a collection without performing a copy of its elements.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

Section 76.3: Performing a deep copy

If you have nested lists, it is desirable to clone the nested lists as well. This action is called deep copy.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

Section 76.4: Performing a shallow copy of a list

You can create shallow copies of lists using slices.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]     # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

Section 76.5: Copy a set

Sets also have a `copy`method. You can use this method to perform a shallow copy.

```
>>> s1 = {()}
>>> s2 = s1.copy()
>>> s1 is s2
```

```
False
>>> s2.add(3)
>>> s1
{[]}
>>> s2
{3,[]}
```

Chapter 77: Context Managers (“with” Statement)

While Python's context managers are widely used, few understand the purpose behind their use. These statements, commonly used with reading and writing files, assist the application in conserving system memory and improve resource management by ensuring specific resources are only in use for certain processes. This topic explains and demonstrates the use of Python's context managers.

Section 77.1: Introduction to context managers and the with statement

A context manager is an object that is notified when a context (a block of code) *starts* and *ends*. You commonly use one with the `with` statement. It takes care of the notifying.

For example, file objects are context managers. When a context ends, the file object is closed automatically:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

The above example is usually simplified by using the `as` keyword:

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Anything that ends execution of the block causes the context manager's exit method to be called. This includes exceptions, and can be useful when an error causes you to prematurely exit from an open file or connection. Exiting a script without properly closing files/connections is a bad idea, that may cause data loss or other problems. By using a context manager you can ensure that precautions are always taken to prevent damage or loss in this way. This feature was added in Python 2.5.

Section 77.2: Writing your own context manager

A context manager is any object that implements two magic methods `__enter__()` and `__exit__()` (although it can implement other methods as well):

```
class AContextManager():

    def __enter__(self):
        print("Entered")
        # optionally return an object
        return "A-instance"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (with an exception)" if exc_type else ""))
        # return True if you want to suppress the exception
```

If the context exits with an exception, the information about that exception will be passed as a triple `exc_type`, `exc_value`, `traceback` (these are the same variables as returned by the `sys.exc_info()` function). If the context

exits normally, all three of these arguments will be `None`.

If an exception occurs and is passed to the `__exit__` method, the method can return `True` in order to suppress the exception, or the exception will be re-raised at the end of the `__exit__` function.

```
with AContextManager() as a:
    print("a is %r" % a)
# Entered
# a is 'A-instance'
# Exited

with AContextManager() as a:
    print("a is %d" % a)
# Entered
# Exited (with an exception)
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# TypeError: %d format: a number is required, not str
```

Note that in the second example even though an exception occurs in the middle of the body of the with-statement, the `__exit__` handler still gets executed, before the exception propagates to the outer scope.

If you only need an `__exit__` method, you can return the instance of the context manager:

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

Section 77.3: Writing your own contextmanager using generator syntax

It is also possible to write a context manager using generator syntax thanks to the `contextlib.contextmanager` decorator:

```
import contextlib

@contextlib.contextmanager
def context_manager(num):
    print('Enter')
    yield num + 1
    print('Exit')

with context_manager(2) as cm:
    # the following instructions are run when the 'yield' point of the context
    # manager is reached.
    # 'cm' will have the value that was yielded
    print('Right in the middle with cm = {}'.format(cm))
```

produces:

```
Enter
Right in the middle with cm = 3
Exit
```

The decorator simplifies the task of writing a context manager by converting a generator into one. Everything before the yield expression becomes the `__enter__` method, the value yielded becomes the value returned by the generator (which can be bound to a variable in the with statement), and everything after the yield expression becomes the `__exit__` method.

If an exception needs to be handled by the context manager, a `try..except..finally`-block can be written in the generator and any exception raised in the `with`-block will be handled by this exception block.

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("Enter")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("Caught error")
    finally:
        print("Cleaning up")
    print("Exit")

with error_handling_context_manager(-1) as cm:
    print("Dividing by cm = {}".format(cm))
    print(2 / cm)
```

This produces:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

Section 77.4: Multiple context managers

You can open several content managers at the same time:

```
with open(input_path) as input_file, open(output_path, 'w') as output_file:

    # do something with both files.

    # e.g. copy the contents of input_file into output_file
    for line in input_file:
        output_file.write(line + '\n')
```

It has the same effect as nesting context managers:

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + '\n')
```

Section 77.5: Assigning to a target

Many context managers return an object when entered. You can assign that object to a new name in the `with` statement.

For example, using a database connection in a `with` statement could give you a cursor object:

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

File objects return themselves, this makes it possible to both open the file object and use it as a context manager in one expression:

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

Section 77.6: Manage Resources

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()
```

`__init__()` method sets up the object, in this case setting up the file name and mode to open file. `__enter__()` opens and returns the file and `__exit__()` just closes it.

Using these magic methods (`__enter__`, `__exit__`) allows you to implement objects which can be used easily `with` the with statement.

Use File class:

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

Chapter 78: The __name__ special variable

The `__name__` special variable is used to check whether a file has been imported as a module or not, and to identify a function, class, module object by their `__name__` attribute.

Section 78.1: __name__ == '__main__'

The special variable `__name__` is not set by the user. It is mostly used to check whether or not the module is being run by itself or run because an `import` was performed. To avoid your module to run certain parts of its code when it gets imported, check `if __name__ == '__main__'`.

Let **module_1.py** be just one line long:

```
import module2.py
```

And let's see what happens, depending on **module2.py**

Situation 1

module2.py

```
print('hello')
```

Running **module1.py** will print `hello`
Running **module2.py** will print `hello`

Situation 2

module2.py

```
if __name__ == '__main__':
    print('hello')
```

Running **module1.py** will print nothing
Running **module2.py** will print `hello`

Section 78.2: Use in logging

When configuring the built-in `logging` functionality, a common pattern is to create a logger with the `__name__` of the current module:

```
logger = logging.getLogger(__name__)
```

This means that the fully-qualified name of the module will appear in the logs, making it easier to see where messages have come from.

Section 78.3: function_class_or_module.__name__

The special attribute `__name__` of a function, class or module is a string containing its name.

```
import os
```

```
class C:
    pass

def f(x):
    x += 2
    return x


print(f)
# <function f at 0x029976B0>
print(f.__name__)
# f

print(C)
# <class '__main__.C'>
print(C.__name__)
# C

print(os)
# <module 'os' from '/spam/eggs/'>
print(os.__name__)
# os
```

The `__name__` attribute is not, however, the name of the variable which references the class, method or function, rather it is the name given to it when defined.

```
def f():
    pass

print(f.__name__)
# f - as expected

g = f
print(g.__name__)
# f - even though the variable is named g, the function is still named f
```

This can be used, among others, for debugging:

```
def enter_exit_info(func):
    def wrapper(*arg, **kw):
        print '-- entering', func.__name__
        res = func(*arg, **kw)
        print '-- exiting', func.__name__
        return res
    return wrapper

@enter_exit_info
def f(x):
    print 'In:', x
    res = x + 2
    print 'Out:', res
    return res

a = f(2)

# Outputs:
#     -- entering f
#     In: 2
#     Out: 4
#     -- exiting f
```

Chapter 79: Checking Path Existence and Permissions

Parameter	Details
os.F_OK	Value to pass as the mode parameter of access() to test the existence of path.
os.R_OK	Value to include in the mode parameter of access() to test the readability of path.
os.W_OK	Value to include in the mode parameter of access() to test the writability of path.
os.X_OK	Value to include in the mode parameter of access() to determine if path can be executed.

Section 79.1: Perform checks using os.access

`os.access` is much better solution to check whether directory exists and it's accessible for reading and writing.

```
import os
path = "/home/myFiles/directory1"

## Check if path exists
os.access(path, os.F_OK)

## Check if path is Readable
os.access(path, os.R_OK)

## Check if path is Writable
os.access(path, os.W_OK)

## Check if path is Executable
os.access(path, os.E_OK)
```

also it's possible to perform all checks together

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

All the above returns `True` if access is allowed and `False` if not allowed. These are available on unix and windows.

Chapter 80: Creating Python packages

Section 80.1: Introduction

Every package requires a `setup.py` file which describes the package.

Consider the following directory structure for a simple package:

```
+-- package_name
|       |
|       +-- __init__.py
|
+-- setup.py
```

The `__init__.py` contains only the line `def foo(): return 100`.

The following `setup.py` will define the package:

```
from setuptools import setup


setup(
    name='package_name',                    # package name
    version='0.1',                          # version
    description='Package Description',      # short description
    url='http://example.com',               # package URL
    install_requires=[],                    # list of packages this package depends
                                            # on.
    packages=['package_name'],              # List of module names that installing
                                            # this package will provide.
)
```

virtualenv is great to test package installs without modifying your other Python environments:

```
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
$ python setup.py install
running install
...
Installed .../package_name-0.1-....egg
...
$ python
>>> import package_name
>>> package_name.foo()
100
```

Section 80.2: Uploading to PyPI

Once your `setup.py` is fully functional (see Introduction), it is very easy to upload your package to PyPI.

Setup a .pypirc File

This file stores logins and passwords to authenticate your accounts. It is typically stored in your home directory.

```
# .pypirc file
```

```
[distutils]
index-servers =
  pypi
  pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[pypitest]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password
```

It is safer to use `twine` for uploading packages, so make sure that is installed.

```
$ pip install twine
```

Register and Upload to testpypi (optional)

Note: PyPI does not allow overwriting uploaded packages, so it is prudent to first test your deployment on a dedicated test server, e.g. testpypi. This option will be discussed. Consider a versioning scheme for your package prior to uploading such as calendar versioning or semantic versioning.

Either log in, or create a new account at testpypi. Registration is only required the first time, although registering more than once is not harmful.

```
$ python setup.py register -r pypitest
```

While in the root directory of your package:

```
$ twine upload dist/* -r pypitest
```

Your package should now be accessible through your account.

Testing

Make a test virtual environment. Try to `pip install` your package from either testpypi or PyPI.

```
# Using virtualenv
$ mkdir testenv
$ cd testenv
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
# Test from testpypi
(.virtualenv)  pip install --verbose --extra-index-url https://testpypi.python.org/pypi
package_name
...
# Or test from PyPI
(.virtualenv) $ pip install package_name
...

(.virtualenv) $ python
Python 3.5.1 (default, Jan 27 2016, 19:16:39)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import package_name
```

```
>>> package_name.foo()
100
```

If successful, your package is least importable. You might consider testing your API as well before your final upload to PyPI. If you package failed during testing, do not worry. You can still fix it, re-upload to testpypi and test again.

Register and Upload to PyPI

Make sure `twine` is installed:

```
$ pip install twine
```

Either log in, or create a new account at PyPI.

```
$ python setup.py register -r pypi
$ twine upload dist/*
```

That's it! Your package is now live.

If you discover a bug, simply upload a new version of your package.

Documentation

Don't forget to include at least some kind of documentation for your package. PyPi takes as the default formatting language reStructuredText.

Readme

If your package doesn't have a big documentation, include what can help other users in `README.rst` file. When the file is ready, another one is needed to tell PyPi to show it.

Create `setup.cfg` file and put these two lines in it:

```
[metadata]
description-file = README.rst
```

Note that if you try to put Markdown file into your package, PyPi will read it as a pure text file without any formatting.

Licensing

It's often more than welcome to put a `LICENSE.txt` file in your package with one of the OpenSource licenses to tell users if they can use your package for example in commercial projects or if your code is usable with their license.

In more readable way some licenses are explained at TL;DR.

Section 80.3: Making package executable

If your package isn't only a library, but has a piece of code that can be used either as a showcase or a standalone application when your package is installed, put that piece of code into `__main__.py` file.

Put the `__main__.py` in the `package_name` folder. This way you will be able to run it directly from console:

```
python -m package_name
```

If there's no `__main__.py` file available, the package won't run with this command and this error will be printed:

> python: No module named `package_name.__main__`; 'package_name' is a package and cannot be directly executed.

Chapter 81: Usage of "pip" module: PyPI Package Manager

Sometimes you may need to use pip package manager inside python eg. when some imports may raise `ImportError` and you want to handle the exception. If you unpack on Windows `Python_root/Scripts/pip.exe`inside is stored `__main__.py` file, where `main` class from `pip` package is imported. This means pip package is used whenever you use pip executable. For usage of pip as executable see: pip: PyPI Package Manager

Section 81.1: Example use of commands

```
import pip

command = 'install'
parameter = 'selenium'
second_param = 'numpy' # You can give as many package names as needed
switch = '--upgrade'

pip.main([command, parameter, second_param, switch])
```

Only needed parameters are obligatory, so both `pip.main(['freeze'])` and `pip.main(['freeze', '', ''])` are acceptable.

Batch install

It is possible to pass many package names in one call, but if one install/upgrade fails, whole installation process stops and ends with status '1'.

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

If you don't want to stop when some installs fail, call installation in loop.

```
for i in installed:
        pip.main(['install']+i.key+['--upgrade'])
```

Section 81.2: Handling ImportError Exception

When you use python file as module there is no need always check if package is installed but it is still useful for scripts.

```
if __name__ == '__main__':
    try:
        import requests
    except ImportError:
        print("To use this module you need 'requests' module")
        t = input('Install requests? y/n: ')
        if t == 'y':
```

```
            import pip
            pip.main(['install', 'requests'])
            import requests
            import os
            import sys
            pass
        else:
            import os
            import sys
            print('Some functionality can be unavailable.')
else:
    import requests
    import os
    import sys
```

Section 81.3: Force install

Many packages for example on version 3.4 would run on 3.6 just fine, but if there are no distributions for specific platform, they can't be installed, but there is workaround. In .whl files (known as wheels) naming convention decide whether you can install package on specified platform. Eg. `scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl`[package_name]-[version]-[python interpreter]-[python-interpreter]-[Operating System].whl. If name of wheel file is changed, so platform does match, pip tries to install package even if platform or python version does not match. Removing platform or interpreter from name will rise an error in newest version of pip module `kjhfkjdf.whl is not a valid wheel filename.`.

Alternatively .whl file can be unpacked using an archiver as 7-zip. - It usually contains distribution meta folder and folder with source files. These source files can be simply unpacked to `site-packages` directory unless this wheel contain installation script, if so, it has to be run first.

Chapter 82: pip: PyPI Package Manager

pip is the most widely-used package manager for the Python Package Index, installed by default with recent versions of Python.

Section 82.1: Install Packages

To install the latest version of a package named `SomePackage`:

```
$ pip install SomePackage
```

To install a specific version of a package:

```
$ pip install SomePackage==1.0.4
```

To specify a minimum version to install for a package:

```
$ pip install SomePackage>=1.0.4
```

If commands shows permission denied error on Linux/Unix then use `sudo` with the commands

Install from requirements files

```
$ pip install -r requirements.txt
```

Each line of the requirements file indicates something to be installed, and like arguments to pip install, Details on the format of the files are here: Requirements File Format.

After install the package you can check it using `freeze` command:

```
$ pip freeze
```

Section 82.2: To list all packages installed using `pip`

To list installed packages:

```
$ pip list
# example output
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

To list outdated packages, and show the latest version available:

```
$ pip list --outdated
# example output
docutils (Current: 0.9.1 Latest: 0.10)
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

Section 82.3: Upgrade Packages

Running

```
$ pip install --upgrade SomePackage
```

will upgrade package `SomePackage` and all its dependencies. Also, pip automatically removes older version of the package before upgrade.

To upgrade pip itself, do

```
$ pip install --upgrade pip
```

on Unix or

```
$ python -m pip install --upgrade pip
```

on Windows machines.

Section 82.4: Uninstall Packages

To uninstall a package:

```
$ pip uninstall SomePackage
```

Section 82.5: Updating all outdated packages on Linux

`pip` doesn't current contain a flag to allow a user to update all outdated packages in one shot. However, this can be accomplished by piping commands together in a Linux environment:

```
pip list --outdated --local | grep -v '^\-e' | cut -d = -f 1  | xargs -n1 pip install -U
```

This command takes all packages in the local virtualenv and checks if they are outdated. From that list, it gets the package name and then pipes that to a `pip install -U` command. At the end of this process, all local packages should be updated.

Section 82.6: Updating all outdated packages on Windows

`pip` doesn't current contain a flag to allow a user to update all outdated packages in one shot. However, this can be accomplished by piping commands together in a Windows environment:

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

This command takes all packages in the local virtualenv and checks if they are outdated. From that list, it gets the package name and then pipes that to a `pip install -U` command. At the end of this process, all local packages should be updated.

Section 82.7: Create a requirements.txt file of all packages on the system

`pip` assists in creating `requirements.txt` files by providing the `freeze` option.

```
pip freeze > requirements.txt
```

This will save a list of all packages and their version installed on the system to a file named `requirements.txt` in the current folder.

Section 82.8: Using a certain Python version with pip

If you have both Python 3 and Python 2 installed, you can specify which version of Python you would like pip to use. This is useful when packages only support Python 2 or 3 or when you wish to test with both.

If you want to install packages for Python 2, run either:

```
pip install [package]
```

or:

```
pip2 install [package]
```

If you would like to install packages for Python 3, do:

```
pip3 install [package]
```

You can also invoke installation of a package to a specific python installation with:

```
\path\to\that\python.exe -m pip install some_package # on Windows OR
/usr/bin/python25 -m pip install some_package # on OS-X/Linux
```

On OS-X/Linux/Unix platforms it is important to be aware of the distinction between the system version of python, (which upgrading make render your system inoperable), and the user version(s) of python. You **may**, *depending on which you are trying to upgrade*, need to prefix these commands with `sudo` and input a password.

Likewise on Windows some python installations, especially those that are a part of another package, can end up installed in system directories - those you will have to upgrade from a command window running in Admin mode - if you find that it looks like you need to do this it is a **very** good idea to check which python installation you are trying to upgrade with a command such as `python -c"import sys;print(sys.path);"` or `py -3.5 -c"import sys;print(sys.path);"` you can also check which pip you are trying to run with `pip --version`

On Windows, if you have both python 2 and python 3 installed, and on your path and your python 3 is greater than 3.4 then you will probably also have the python launcher `py` on your system path. You can then do tricks like:

```
py -3 -m pip install -U some_package # Install/Upgrade some_package to the latest python 3
py -3.3 -m pip install -U some_package # Install/Upgrade some_package to python 3.3 if present
py -2 -m pip install -U some_package # Install/Upgrade some_package to the latest python 2 - 64 bit
if present
py -2.7-32 -m pip install -U some_package # Install/Upgrade some_package to python 2.7 - 32 bit if
present
```

If you are running & maintaining multiple versions of python I would strongly recommend reading up about the python `virtualenv` or `venv` virtual environments which allow you to isolate both the version of python and which packages are present.

Section 82.9: Create a requirements.txt file of packages only in the current virtualenv

`pip` assists in creating `requirements.txt` files by providing the `freeze` option.

```
pip freeze --local > requirements.txt
```

The `--local` parameter will only output a list of packages and versions that are installed locally to a virtualenv.

Global packages will not be listed.

Section 82.10: Installing packages not yet on pip as wheels

Many, pure python, packages are not yet available on the Python Package Index as wheels but still install fine. However, some packages on Windows give the dreaded vcvarsall.bat not found error.

The problem is that the package that you are trying to install contains a C or C++ extension and is not *currently* available as a pre-built wheel from the python package index, *pypi*, and on windows you do not have the tool chain needed to build such items.

The simplest answer is to go to Christoph Gohlke's excellent site and locate the **appropriate** version of the libraries that you need. By appropriate in the package name a **-cp*NN*-** has to match your version of python, i.e. if you are using windows 32 bit python *even on win64* the name must include **-win32-** and if using the 64 bit python it must include **-win_amd64-** and then the python version must match, i.e. for Python 34 the filename **must** include **-cp*34*-**, etc. this is basically the magic that pip does for you on the pypi site.

Alternatively, you need to get the appropriate windows development kit for the version of python that you are using, the headers for any library that the package you are trying to build interfaces to, possibly the python headers for the version of python, etc.

Python 2.7 used Visual Studio 2008, Python 3.3 and 3.4 used Visual Studio 2010, and Python 3.5+ uses Visual Studio 2015.

- Install "Visual C++ Compiler Package for Python 2.7", which is available from Microsoft's website **or**
- Install "Windows SDK for Windows 7 and .NET Framework 4" (v7.1), which is available from Microsoft's website **or**
- Install Visual Studio 2015 Community Edition, *(or any later version, when these are released)*, **ensuring you select the options to install C & C++ support** *no longer the default -I am told that this can take up to **8 hours*** to download and install so make **sure** that those options are set on the first try.

Then you *may need* to locate the header files, *at the matching revision* for any libraries that your desired package links to and download those to an appropriate locations.

Finally you can let pip do your build - of course if the package has dependencies that you don't yet have you may also need to find the header files for them as well.

Alternatives: It is also worth looking out, *both on pypi or Christop's site*, for any slightly earlier version of the package that you are looking for that is either pure python or pre-built for your platform and python version and possibly using those, if found, until your package does become available. Likewise if you are using the very latest version of python you may find that it takes the package maintainers a little time to catch up so for projects that really **need** a specific package you may have to use a slightly older python for the moment. You can also check the packages source site to see if there is a forked version that is available pre-built or as pure python and searching for alternative packages that provide the functionality that you require but are available - one example that springs to mind is the Pillow, *actively maintained*, drop in replacement for PIL *currently not updated in 6 years and not available for python 3*.

Afterword, I would encourage anybody who is having this problem to go to the bug tracker for the package and add to, or raise if there isn't one already, a ticket **politely** requesting that the package maintainers provide a wheel on pypi for your specific combination of platform and python, if this is done then normally things will get better with time, some package maintainers don't realise that they have missed a given combination that people may be using.

Note on Installing Pre-Releases

Pip follows the rules of Semantic Versioning and by default prefers released packages over pre-releases. So if a given package has been released as `V0.98` and there is also a release candidate `V1.0-rc1` the default behaviour of `pip install` will be to install `V0.98` - if you wish to install the release candidate, *you are advised to test in a virtual environment first*, you can enable do so with `--pip install --pre` *package-name* or `--pip install --pre --upgrade` *package-name*. In many cases pre-releases or release candidates may not have wheels built for all platform & version combinations so you are more likely to encounter the issues above.

Note on Installing Development Versions

You can also use pip to install development versions of packages from github and other locations, since such code is in flux it is very unlikely to have wheels built for it, so any impure packages will require the presence of the build tools, and they may be broken at any time so the user is **strongly** encouraged to only install such packages in a virtual environment.

Three options exist for such installations:

1. Download compressed snapshot, most online version control systems have the option to download a compressed snapshot of the code. This can be downloaded manually and then installed with `pip install` *path/to/downloaded/file* note that for most compression formats pip will handle unpacking to a cache area, etc.
2. Let pip handle the download & install for you with: `pip install` *URL/of/package/repository* - you may also need to use the `--trusted-host`, `--client-cert` and/or `--proxy` flags for this to work correctly, especially in a corporate environment. e.g:

```
 > py -3.5-32 -m venv demo-pip
> demo-pip\Scripts\activate.bat
> python -m pip install -U pip
Collecting pip
Using cached pip-9.0.1-py2.py3-none-any.whl
Installing collected packages: pip
Found existing installation: pip 8.1.1
Uninstalling pip-8.1.1:
Successfully uninstalled pip-8.1.1
Successfully installed pip-9.0.1
> pip install git+https://github.com/sphinx-doc/sphinx/
Collecting git+https://github.com/sphinx-doc/sphinx/
Cloning https://github.com/sphinx-doc/sphinx/ to c:\users\steve-
~1\appdata\local\temp\pip-04yn9hpp-build
Collecting six>=1.5 (from Sphinx==1.7.dev20170506)
Using cached six-1.10.0-py2.py3-none-any.whl
Collecting Jinja2>=2.3 (from Sphinx==1.7.dev20170506)
Using cached Jinja2-2.9.6-py2.py3-none-any.whl
Collecting Pygments>=2.0 (from Sphinx==1.7.dev20170506)
Using cached Pygments-2.2.0-py2.py3-none-any.whl
Collecting docutils>=0.11 (from Sphinx==1.7.dev20170506)
Using cached docutils-0.13.1-py3-none-any.whl
Collecting snowballstemmer>=1.1 (from Sphinx==1.7.dev20170506)
Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
Collecting babel!=2.0,>=1.3 (from Sphinx==1.7.dev20170506)
Using cached Babel-2.4.0-py2.py3-none-any.whl
Collecting alabaster<0.8,>=0.7 (from Sphinx==1.7.dev20170506)
Using cached alabaster-0.7.10-py2.py3-none-any.whl
Collecting imagesize (from Sphinx==1.7.dev20170506)
Using cached imagesize-0.7.1-py2.py3-none-any.whl
Collecting requests>=2.0.0 (from Sphinx==1.7.dev20170506)
Using cached requests-2.13.0-py2.py3-none-any.whl
```

```
Collecting typing (from Sphinx==1.7.dev20170506)
Using cached typing-3.6.1.tar.gz
Requirement already satisfied: setuptools in f:\toolbuild\temp\demo-pip\lib\site-packages (from
Sphinx==1.7.dev20170506)
Collecting sphinxcontrib-websupport (from Sphinx==1.7.dev20170506)
Downloading sphinxcontrib_websupport-1.0.0-py2.py3-none-any.whl
Collecting colorama>=0.3.5 (from Sphinx==1.7.dev20170506)
Using cached colorama-0.3.9-py2.py3-none-any.whl
Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.7.dev20170506)
Using cached MarkupSafe-1.0.tar.gz
Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.7.dev20170506)
Using cached pytz-2017.2-py2.py3-none-any.whl
Collecting sqlalchemy>=0.9 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
Downloading SQLAlchemy-1.1.9.tar.gz (5.2MB)
100% |################################| 5.2MB 220kB/s
Collecting whoosh>=2.0 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
Downloading Whoosh-2.7.4-py2.py3-none-any.whl (468kB)
100% |################################| 471kB 1.1MB/s
Installing collected packages: six, MarkupSafe, Jinja2, Pygments, docutils, snowballstemmer, pytz,
babel, alabaster, imagesize, requests, typing, sqlalchemy, whoosh, sphinxcontrib-websupport,
colorama, Sphinx
Running setup.py install for MarkupSafe ... done
Running setup.py install for typing ... done
Running setup.py install for sqlalchemy ... done
Running setup.py install for Sphinx ... done
Successfully installed Jinja2-2.9.6 MarkupSafe-1.0 Pygments-2.2.0 Sphinx-1.7.dev20170506
alabaster-0.7.10 babel-2.4.0 colorama-0.3.9 docutils-0.13.1 imagesize-0.7.1 pytz-2017.2
requests-2.13.0 six-1.10.0 snowballstemmer-1.2.1 sphinxcontrib-websupport-1.0.0 sqlalchemy-1.1.9
typing-3.6.1 whoosh-2.7.4
```

Note the `git+` prefix to the URL.

3. Clone the repository using `git`, `mercurial` or other acceptable tool, *preferably a DVCS tool,* and use `pip install` *path/to/cloned/repo* - this will **both** process any requires.text file and perform the build and setup steps, *you can manually change directory to your cloned repository and run `pip install -r requires.txt` and then `python setup.py install` to get the same effect.* The big advantages of this approach is that while the initial clone operation may take longer than the snapshot download you can update to the latest with, in the case of git: `git pull origin master` and if the current version contains errors you can use `pip uninstall` *package-name* then use `git checkout` commands to move back through the repository history to earlier version(s) and re-try.

Chapter 83: Parsing Command Line arguments

Most command line tools rely on arguments passed to the program upon its execution. Instead of prompting for input, these programs expect data or specific flags (which become booleans) to be set. This allows both the user and other programs to run the Python file passing it data as it starts. This section explains and demonstrates the implementation and usage of command line arguments in Python.

Section 83.1: Hello world in argparse

The following program says hello to the user. It takes one positional argument, the name of the user, and can also be told the greeting.

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
    help='name of user'
)

parser.add_argument('-g', '--greeting',
    default='Hello',
    help='optional alternate greeting'
)

args = parser.parse_args()

print("{greeting}, {name}!".format(
       greeting=args.greeting,
       name=args.name)
)
```

```
$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name

positional arguments:
  name                  name of user

optional arguments:
  -h, --help            show this help message and exit
  -g GREETING, --greeting GREETING
                        optional alternate greeting
```

```
$ python hello.py world
Hello, world!
$ python hello.py John -g Howdy
Howdy, John!
```

For more details please read the argparse documentation.

Section 83.2: Using command line arguments with argv

Whenever a Python script is invoked from the command line, the user may supply additional **command line arguments** which will be passed on to the script. These arguments will be available to the programmer from the system variable `sys.argv` ("argv" is a traditional name used in most programming languages, and it means "**arg**ument **v**ector").

By convention, the first element in the `sys.argv` list is the name of the Python script itself, while the rest of the elements are the tokens passed by the user when invoking the script.

```
# cli.py
import sys
print(sys.argv)

$ python cli.py
=> ['cli.py']

$ python cli.py fizz
=> ['cli.py', 'fizz']

$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

Here's another example of how to use `argv`. We first strip off the initial element of sys.argv because it contains the script's name. Then we combine the rest of the arguments into a single sentence, and finally print that sentence prepending the name of the currently logged-in user (so that it emulates a chat program).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

The algorithm commonly used when "manually" parsing a number of non-positional arguments is to iterate over the `sys.argv` list. One way is to go over the list and pop each element of it:

```
# reverse and copy sys.argv
argv = reversed(sys.argv)
# extract the first element
arg = argv.pop()
# stop iterating when there's no more args to pop()
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('seen foo!')
    elif arg in ('-b', '--bar'):
        print('seen bar!')
    elif arg in ('-a', '--with-arg'):
        arg = arg.pop()
        print('seen value: {}'.format(arg))
    # get the next value
    arg = argv.pop()
```

Section 83.3: Setting mutually exclusive arguments with argparse

If you want two or more arguments to be mutually exclusive. You can use the function `argparse.ArgumentParser.add_mutually_exclusive_group()`. In the example below, either foo or bar can exist but not both at the same time.

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
```

```
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar
```

If you try to run the script specifying both `--foo` and `--bar` arguments, the script will complain with the below message.

```
error: argument -b/--bar: not allowed with argument -f/--foo
```

Section 83.4: Basic example with docopt

docopt turns command-line argument parsing on its head. Instead of parsing the arguments, you just **write the usage string** for your program, and docopt **parses the usage string** and uses it to extract the command line arguments.

```
"""
Usage:
    script_name.py [-a] [-b] <path>

Options:
    -a            Print all the things.
    -b            Get more bees into the path.
"""
from docopt import docopt


if __name__ == "__main__":
    args = docopt(__doc__)
    import pprint; pprint.pprint(args)
```

Sample runs:

```
$ python script_name.py
Usage:
    script_name.py [-a] [-b] <path>
$ python script_name.py something
{'-a': False,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py something -a
{'-a': True,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
 '-b': True,
 '<path>': 'something'}
```

Section 83.5: Custom parser error message with argparse

You can create parser error messages according to your script needs. This is through the `argparse.ArgumentParser.error` function. The below example shows the script printing a usage and an error message to `stderr` when `--foo` is given but not `--bar`.

```
import argparse
```

```
parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar
```

Assuming your script name is sample.py, and we run: `python sample.py --foo ds_in_fridge`

The script will complain with the following:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

Section 83.6: Conceptual grouping of arguments with argparse.add_argument_group()

When you create an argparse ArgumentParser() and run your program with '-h' you get an automated usage message explaining what arguments you can run your software with. By default, positional arguments and conditional arguments are separated into two categories, for example, here is a small script (example.py) and the output when you run `python example.py -h`.

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()
```

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                  [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                  name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit
  --bar_this BAR_THIS
  --bar_that BAR_THAT
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

There are some situations where you want to separate your arguments into further conceptual sections to assist your user. For example, you may wish to have all the input options in one group, and all the output formatting options in another. The above example can be adjusted to separate the `--foo_*` args from the `--bar_*` args like so.

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
```

```
parser.add_argument('name', help='Who to greet', default='World')
# Create two argument groups
foo_group = parser.add_argument_group(title='Foo options')
bar_group = parser.add_argument_group(title='Bar options')
# Add arguments to those groups
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()
```

Which produces this output when `python example.py -h` is run:

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                       [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                       name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit

Foo options:
  --bar_this BAR_THIS
  --bar_that BAR_THAT

Bar options:
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Section 83.7: Advanced example with docopt and docopt_dispatch

As with docopt, with [docopt_dispatch] you craft your `--help` in the `__doc__` variable of your entry-point module. There, you call `dispatch` with the doc string as argument, so it can run the parser over it.

That being done, instead of handling manually the arguments (which usually ends up in a high cyclomatic if/else structure), you leave it to dispatch giving only how you want to handle the set of arguments.

This is what the `dispatch.on` decorator is for: you give it the argument or sequence of arguments that should trigger the function, and that function will be executed with the matching values as parameters.

```
"""Run something in development or production mode.

Usage: run.py --development <host> <port>
       run.py --production <host> <port>
       run.py items add <item>
       run.py items delete <item>

"""
from docopt_dispatch import dispatch

@dispatch.on('--development')
def development(host, port, **kwargs):
    print('in *development* mode')
```

```
@dispatch.on('--production')
def development(host, port, **kwargs):
    print('in *production* mode')

@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('adding item...')

@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('deleting item...')

if __name__ == '__main__':
    dispatch(__doc__)
```

Chapter 84: Subprocess Library

Parameter	Details
`args`	A single executable, or sequence of executable and arguments - `'ls'`, `['ls', '-la']`
`shell`	Run under a shell? The default shell to `/bin/sh` on POSIX.
`cwd`	Working directory of the child process.

Section 84.1: More flexibility with Popen

Using `subprocess.Popen` give more fine-grained control over launched processes than `subprocess.call`.

Launching a subprocess

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

The signature for `Popen` is very similar to the `call` function; however, `Popen` will return immediately instead of waiting for the subprocess to complete like `call` does.

Waiting on a subprocess to complete

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

Reading output from a subprocess

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout=subprocess.PIPE, stderr=subprocess.PIPE)

# This will block until process completes
stdout, stderr = process.communicate()
print stdout
print stderr
```

Interactive access to running subprocesses

You can read and write on `stdin` and `stdout` even while the subprocess hasn't completed. This could be useful when automating functionality in another program.

Writing to a subprocess

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout = subprocess.PIPE, stdin =
subprocess.PIPE)


process.stdin.write('line of input\n') # Write input

line  = process.stdout.readline() # Read a line from stdout

# Do logic on line read.
```

However, if you only need one set of input and output, rather than dynamic interaction, you should use `communicate()` rather than directly accessing `stdin` and `stdout`.

Reading a stream from a subprocess

In case you want to see the output of a subprocess line by line, you can use the following snippet:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

in the case the subcommand output do not have EOL character, the above snippet does not work. You can then read the output character by character as follows:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

The `1` specified as argument to the `read` method tells read to read 1 character at time. You can specify to read as many characters you want using a different number. Negative number or 0 tells to `read` to read as a single string until the EOF is encountered (see here).

In both the above snippets, the `process.poll()` is `None` until the subprocess finishes. This is used to exit the loop once there is no more output to read.

The same procedure could be applied to the `stderr` of the subprocess.

Section 84.2: Calling External Commands

The simplest use case is using the `subprocess.call` function. It accepts a list as the first argument. The first item in the list should be the external application you want to call. The other items in the list are arguments that will be passed to that application.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

For shell commands, set `shell=True` and provide the command as a string instead of a list.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Note that the two command above return only the `exit status` of the subprocess. Moreover, pay attention when using `shell=True` since it provides security issues (see here).

If you want to be able to get the standard output of the subprocess, then substitute the `subprocess.call` with `subprocess.check_output`. For more advanced use, refer to this.

Section 84.3: How to create the command list argument

The subprocess method that allows running commands needs the command in form of a list (at least using `shell_mode=True`).

The rules to create the list are not always straightforward to follow, especially with complex commands. Fortunately, there is a very helpful tool that allows doing that: `shlex`. The easiest way of creating the list to be used as command is the following:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

A simple example:

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

Chapter 85: setup.py

Parameter	Usage
`name`	Name of your distribution.
`version`	Version string of your distribution.
`packages`	List of Python packages (that is, directories containing modules) to include. This can be specified manually, but a call to `setuptools.find_packages()` is typically used instead.
`py_modules`	List of top-level Python modules (that is, single `.py` files) to include.

Section 85.1: Purpose of setup.py

The setup script is the center of all activity in building, distributing, and installing modules using the Distutils. Its purpose is the correct installation of the software.

If all you want to do is distribute a module called foo, contained in a file foo.py, then your setup script can be as simple as this:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

To create a source distribution for this module, you would create a setup script, setup.py, containing the above code, and run this command from a terminal:

```
python setup.py sdist
```

sdist will create an archive file (e.g., tarball on Unix, ZIP file on Windows) containing your setup script setup.py, and your module foo.py. The archive file will be named foo-1.0.tar.gz (or .zip), and will unpack into a directory foo-1.0.

If an end-user wishes to install your foo module, all she has to do is download foo-1.0.tar.gz (or .zip), unpack it, and—from the foo-1.0 directory—run

```
python setup.py install
```

Section 85.2: Using source control metadata in setup.py

`setuptools_scm` is an officially-blessed package that can use Git or Mercurial metadata to determine the version number of your package, and find Python packages and package data to include in it.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

This example uses both features; to only use SCM metadata for the version, replace the call to `find_packages()` with your manual package list, or to only use the package finder, remove `use_scm_version=True`.

Section 85.3: Adding command line scripts to your python package

Command line scripts inside python packages are common. You can organise your package in such a way that when a user installs the package, the script will be available on their path.

If you had the `greetings` package which had the command line script `hello_world.py`.

```
greetings/
   greetings/
      __init__.py
      hello_world.py
```

You could run that script by running:

```
python greetings/greetings/hello_world.py
```

However if you would like to run it like so:

```
hello_world.py
```

You can achieve this by adding `scripts` to your `setup()` in `setup.py` like this:

```
from setuptools import setup
setup(
  name='greetings',
  scripts=['hello_world.py']
)
```

When you install the greetings package now, `hello_world.py` will be added to your path.

Another possibility would be to add an entry point:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

This way you just have to run it like:

```
greetings
```

Section 85.4: Adding installation options

As seen in previous examples, basic use of this script is:

```
python setup.py install
```

But there is even more options, like installing the package and have the possibility to change the code and test it without having to re-install it. This is done using:

```
python setup.py develop
```

If you want to perform specific actions like compiling a *Sphinx* documentation or building *fortran* code, you can create your own option like this:

```
cmdclasses = dict()
```

```
class BuildSphinx(Command):

    """Build Sphinx documentation."""

    description = 'Build Sphinx documentation'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])

cmdclasses['build_sphinx'] = BuildSphinx

setup(
...
cmdclass=cmdclasses,
)
```

`initialize_options` and `finalize_options` will be executed before and after the `run` function as their names suggests it.

After that, you will be able to call your option:

```
python setup.py build_sphinx
```

Chapter 86: Recursion

Section 86.1: The What, How, and When of Recursion

Recursion occurs when a function call causes that same function to be called again before the original function call terminates. For example, consider the well-known mathematical expression `x!` (i.e. the factorial operation). The factorial operation is defined for all nonnegative integers as follows:

- If the number is 0, then the answer is 1.
- Otherwise, the answer is that number times the factorial of one less than that number.

In Python, a naïve implementation of the factorial operation can be defined as a function as follows:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Recursion functions can be difficult to grasp sometimes, so let's walk through this step-by-step. Consider the expression `factorial(3)`. This and *all* function calls create a new **environment**. An environment is basically just a table that maps identifiers (e.g. `n`, `factorial`, `print`, etc.) to their corresponding values. At any point in time, you can access the current environment using `locals()`. In the first function call, the only local variable that gets defined is `n = 3`. Therefore, printing `locals()` would show `{'n': 3}`. Since `n == 3`, the return value becomes `n * factorial(n - 1)`.

At this next step is where things might get a little confusing. Looking at our new expression, we already know what `n` is. However, we don't yet know what `factorial(n - 1)` is. First, `n - 1` evaluates to 2. Then, 2 is passed to `factorial` as the value for `n`. Since this is a new function call, a second environment is created to store this new `n`. Let *A* be the first environment and *B* be the second environment. *A* still exists and equals `{'n': 3}`, however, *B* (which equals `{'n': 2}`) is the current environment. Looking at the function body, the return value is, again, `n * factorial(n - 1)`. Without evaluating this expression, let's substitute it into the original return expression. By doing this, we're mentally discarding *B*, so remember to substitute `n` accordingly (i.e. references to *B*'s `n` are replaced with `n - 1` which uses *A*'s `n`). Now, the original return expression becomes `n * ((n - 1) * factorial((n - 1) - 1))`. Take a second to ensure that you understand why this is so.

Now, let's evaluate the `factorial((n - 1) - 1))` portion of that. Since *A*'s `n == 3`, we're passing 1 into `factorial`. Therefore, we are creating a new environment *C* which equals `{'n': 1}`. Again, the return value is `n * factorial(n - 1)`. So let's replace `factorial((n - 1) - 1))` of the "original" return expression similarly to how we adjusted the original return expression earlier. The "original" expression is now `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))`.

Almost done. Now, we need to evaluate `factorial((n - 2) - 1)`. This time, we're passing in `0`. Therefore, this evaluates to `1`. Now, let's perform our last substitution. The "original" return expression is now `n * ((n - 1) * ((n - 2) * 1))`. Recalling that the original return expression is evaluated under *A*, the expression becomes `3 * ((3 - 1) * ((3 - 2) * 1))`. This, of course, evaluates to 6. To confirm that this is the correct answer, recall that `3! == 3 * 2 * 1 == 6`. Before reading any further, be sure that you fully understand the concept of environments and how they apply to recursion.

The statement `if n == 0: return 1` is called a base case. This is because, it exhibits no recursion. A base case is absolutely required. Without one, you'll run into infinite recursion. With that said, as long as you have at least one base case, you can have as many cases as you want. For example, we could have equivalently written `factorial` as

follows:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

You may also have multiple recursion cases, but we won't get into that since it's relatively uncommon and is often difficult to mentally process.

You can also have "parallel" recursive function calls. For example, consider the Fibonacci sequence which is defined as follows:

- If the number is 0, then the answer is 0.
- If the number is 1, then the answer is 1.
- Otherwise, the answer is the sum of the previous two Fibonacci numbers.

We can define this is as follows:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

I won't walk through this function as thoroughly as I did with `factorial(3)`, but the final return value of `fib(5)` is equivalent to the following (*syntactically* invalid) expression:

```
(
  fib((n - 2) - 2)
  +
  (
    fib(((n - 2) - 1) - 2)
    +
    fib(((n - 2) - 1) - 1)
  )
)
+
(
  (
    fib(((n - 1) - 2) - 2)
    +
    fib(((n - 1) - 2) - 1)
  )
  +
  (
    fib(((n - 1) - 1) - 2)
    +
    (
      fib((((n - 1) - 1) - 1) - 2)
      +
      fib((((n - 1) - 1) - 1) - 1)
    )
  )
)
```

This becomes `(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))` which of course evaluates to 5.

Now, let's cover a few more vocabulary terms:

- A **tail call** is simply a recursive function call which is the last operation to be performed before returning a value. To be clear, `return foo(n - 1)` is a tail call, but `return foo(n - 1) + 1` is not (since the addition is the last operation).
- **Tail call optimization** (TCO) is a way to automatically reduce recursion in recursive functions.
- **Tail call elimination** (TCE) is the reduction of a tail call to an expression that can be evaluated without recursion. TCE is a type of TCO.

Tail call optimization is helpful for a number of reasons:

- The interpreter can minimize the amount of memory occupied by environments. Since no computer has unlimited memory, excessive recursive function calls would lead to a **stack overflow**.
- The interpreter can reduce the number of **stack frame** switches.

Python has no form of TCO implemented for a number of a reasons. Therefore, other techniques are required to skirt this limitation. The method of choice depends on the use case. With some intuition, the definitions of `factorial` and `fib` can relatively easily be converted to iterative code as follows:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

This is usually the most efficient way to manually eliminate recursion, but it can become rather difficult for more complex functions.

Another useful tool is Python's lru_cache decorator which can be used to reduce the number of redundant calculations.

You now have an idea as to how to avoid recursion in Python, but when *should* you use recursion? The answer is “not often”. All recursive functions can be implemented iteratively. It's simply a matter of figuring out how to do so. However, there are rare cases in which recursion is okay. Recursion is common in Python when the expected inputs wouldn't cause a significant number of a recursive function calls.

If recursion is a topic that interests you, I implore you to study functional languages such as Scheme or Haskell. In such languages, recursion is much more useful.

Please note that the above example for the Fibonacci sequence, although good at showing how to apply the definition in python and later use of the lru cache, has an inefficient running time since it makes 2 recursive calls for each non base case. The number of calls to the function grows exponentially to `n`.
Rather non-intuitively a more efficient implementation would use linear recursion:

```
def fib(n):
    if n <= 1:
```

```
        return (n,0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

But that one has the issue of returning a *pair* of numbers. This emphasizes that some functions really do not gain much from recursion.

Section 86.2: Tree exploration with recursion

Say we have the following tree:

```
root
- A
  - AA
  - AB
- B
  - BA
  - BB
    - BBA
```

Now, if we wish to list all the names of the elements, we could do this with a simple for-loop. We assume there is a function `get_name()` to return a string of the name of a node, a function `get_children()` to return a list of all the sub-nodes of a given node in the tree, and a function `get_root()` to get the root node.

```
root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# prints: A, AA, AB, B, BA, BB, BBA
```

This works well and fast, but what if the sub-nodes, got sub-nodes of its own? And those sub-nodes might have more sub-nodes... What if you don't know beforehand how many there will be? A method to solve this is the use of recursion.

```
def list_tree_names(node):
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# prints: A, AA, AB, B, BA, BB, BBA
```

Perhaps you wish to not print, but return a flat list of all node names. This can be done by passing a rolling list as a parameter.

```
def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# returns ['A', 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']
```

Section 86.3: Sum of numbers from 1 to n

If I wanted to find out the sum of numbers from `1` to `n` where `n` is a natural number, I can do `1 + 2 + 3 + 4 + ... + (several hours later) + n`. Alternatively, I could write a `for` loop:

```
n = 0
for i in range (1, n+1):
    n += i
```

Or I could use a technique known as recursion:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

Recursion has advantages over the above two methods. Recursion takes less time than writing out `1 + 2 + 3` for a sum from 1 to 3. For `recursion(4)`, recursion can be used to work backwards:

Function calls: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

Whereas the `for` loop is working strictly forwards: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). Sometimes the recursive solution is simpler than the iterative solution. This is evident when implementing a reversal of a linked list.

Section 86.4: Increasing the Maximum Recursion Depth

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Here's a sample of a program that would cause this error:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))
cursing(0)
# Out: I recursed 1083 times!
```

It is possible to change the recursion depth limit by using

```
sys.setrecursionlimit(limit)
```

You can check what the current parameters of the limit are by running:

```
sys.getrecursionlimit()
```

Running the same method above with our new limit we get

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

From Python 3.5, the exception is a RecursionError, which is derived from RuntimeError.

Section 86.5: Tail Recursion - Bad Practice

When the only thing returned from a function is a recursive call, it is referred to as tail recursion.

Here's an example countdown written using tail recursion:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Any computation that can be made using iteration can also be made using recursion. Here is a version of find_max written using tail recursion:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

Tail recursion is considered a bad practice in Python, since the Python compiler does not handle optimization for tail recursive calls. The recursive solution in cases like this use more system resources than the equivalent iterative solution.

Section 86.6: Tail Recursion Optimization Through Stack Introspection

By default Python's recursion stack cannot exceed 1000 frames. This can be changed by setting the `sys.setrecursionlimit(15000)` which is faster however, this method consumes more memory. Instead, we can also solve the Tail Recursion problem using stack introspection.

```
#!/usr/bin/env python2.4
# This program shows off a python decorator which implements tail call optimization. It
# does this by throwing an exception if it is its own grandparent, and catching such
# exceptions to recall the stack.

import sys

class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

def tail_call_optimized(g):
"""
This function decorates a function with tail call
optimization. It does this by throwing an exception
if it is its own grandparent, and catching such
exceptions to fake the tail call optimization.

This function fails if the decorated
```

```
function recurses in a non-tail context.
"""

    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while 1:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException, e:
                    args = e.args
                    kwargs = e.kwargs
    func.__doc__ = g.__doc__
    return func
```

To optimize the recursive functions, we can use the `@tail_call_optimized` decorator to call our function. Here's a few of the common recursion examples using the decorator described above:

Factorial Example:

```
@tail_call_optimized
def factorial(n, acc=1):
  "calculate a factorial"
  if n == 0:
    return acc
  return factorial(n-1, n*acc)

print factorial(10000)
# prints a big, big number,
# but doesn't hit the recursion limit.
```

Fibonacci Example:

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
  if i == 0:
    return current
  else:
    return fib(i - 1, next, current + next)

print fib(10000)
# also prints a big number,
# but doesn't hit the recursion limit.
```

Chapter 87: Type Hints

Section 87.1: Adding types to a function

Let's take an example of a function which receives two arguments and returns a value indicating their sum:

```
def two_sum(a, b):
    return a + b
```

By looking at this code, one can not safely and without doubt indicate the type of the arguments for function `two_sum`. It works both when supplied with `int` values:

```
print(two_sum(2, 1))  # result: 3
```

and with strings:

```
print(two_sum("a", "b"))  # result: "ab"
```

and with other values, such as `list`s, `tuple`s et cetera.

Due to this dynamic nature of python types, where many are applicable for a given operation, any type checker would not be able to reasonably assert whether a call for this function should be allowed or not.

To assist our type checker we can now provide type hints for it in the Function definition indicating the type that we allow.

To indicate that we only want to allow `int` types we can change our function definition to look like:

```
def two_sum(a: int, b: int):
    return a + b
```

Annotations follow the argument name and are separated by a `:` character.

Similarly, to indicate only `str` types are allowed, we'd change our function to specify it:

```
def two_sum(a: str, b: str):
    return a + b
```

Apart from specifying the type of the arguments, one could also indicate the return value of a function call. This is done by adding the `->` character followed by the type after the closing parenthesis in the argument list *but* before the `:` at the end of the function declaration:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Now we've indicated that the return value when calling `two_sum` should be of type `int`. Similarly we can define appropriate values for `str`, `float`, `list`, `set` and others.

Although type hints are mostly used by type checkers and IDEs, sometimes you may need to retrieve them. This can be done using the `__annotations__` special attribute:

```
two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Section 87.2: NamedTuple

Creating a namedtuple with type hints is done using the function NamedTuple from the `typing` module:

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Note that the name of the resulting type is the first argument to the function, but it should be assigned to a variable with the same name to ease the work of type checkers.

Section 87.3: Generic Types

The `typing.TypeVar` is a generic type factory. It's primary goal is to serve as a parameter/placeholder for generic function/class/method annotations:

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """Gets the first element of a sequence."""
    return l[0]
```

Section 87.4: Variables and Attributes

Variables are annotated using comments:

```
x = 3  # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x `Version ≥ 3.6`

Starting from Python 3.6, there is also new syntax for variable annotations. The code above might use the form

```
x: int = 3
```

Unlike with comments, it is also possible to just add a type hint to a variable that was not previously declared, without setting a value to it:

```
y: int
```

Additionally if these are used in the module or the class level, the type hints can be retrieved using `typing.get_type_hints(class_or_module)`:

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

Alternatively, they can be accessed by using the `__annotations__` special variable or attribute:

```
x: int
print(__annotations__)
```

```
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

Section 87.5: Class Members and Methods

```
class A:
    x = None  # type: float
    def __init__(self, x: float) -> None:
        """
        self should not be annotated
        init should be annotated to return None
        """
        self.x = x

    @classmethod
    def from_int(cls, x: int) -> 'A':
        """
        cls should not be annotated
        Use forward reference to refer to current class with string literal 'A'
        """
        return cls(float(x))
```

Forward reference of the current class is needed since annotations are evaluated when the function is defined. Forward references can also be used when referring to a class that would cause a circular import if imported.

Section 87.6: Type hints for keyword arguments

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Note the spaces around the equal sign as opposed to how keyword arguments are usually styled.

Chapter 88: Exceptions

Errors detected during execution are called exceptions and are not unconditionally fatal. Most exceptions are not handled by programs; it is possible to write programs that handle selected exceptions. There are specific features in Python to deal with exceptions and exception logic. Furthermore, exceptions have a rich type hierarchy, all inheriting from the `BaseException` type.

Section 88.1: Catching Exceptions

Use `try...except`: to catch exceptions. You should specify as precise an exception as you can:

```
try:
    x = 5 / 0
except ZeroDivisionError as e:
    # `e` is the exception object
    print("Got a divide by zero! The exception was:", e)
    # handle exceptional case
    x = 0
finally:
    print "The END"
    # it runs no matter what execute.
```

The exception class that is specified - in this case, `ZeroDivisionError` - catches any exception that is of that class or of any subclass of that exception.

For example, `ZeroDivisionError` is a subclass of `ArithmeticError`:

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)
```

And so, the following will still catch the `ZeroDivisionError`:

```
try:
    5 / 0
except ArithmeticError:
    print("Got arithmetic error")
```

Section 88.2: Do not catch everything!

While it's often tempting to catch every `Exception`:

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gratiously
finally:
    print "The END"
    # it runs no matter what execute.
```

Or even everything (that includes `BaseException` and all its children including `Exception`):

```
try:
    even_more_difficult_function()
except:
    pass  # do whatever needed
```

In most cases it's bad practice. It might catch more than intended, such as `SystemExit`, `KeyboardInterrupt` and `MemoryError` - each of which should generally be handled differently than usual system or logic errors. It also means there's no clear understanding for what the internal code may do wrong and how to recover properly from that condition. If you're catching every error, you won't know what error occurred or how to fix it.

This is more commonly referred to as 'bug masking' and should be avoided. Let your program crash instead of silently failing or even worse, failing at deeper level of execution. (Imagine it's a transactional system)

Usually these constructs are used at the very outer level of the program, and will log the details of the error so that the bug can be fixed, or the error can be handled more specifically.

Section 88.3: Re-raising exceptions

Sometimes you want to catch an exception just to inspect it, e.g. for logging purposes. After the inspection, you want the exception to continue propagating as it did before.

In this case, simply use the `raise` statement with no parameters.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Got an error")
    raise
```

Keep in mind, though, that someone further up in the caller stack can still catch the exception and handle it somehow. The done output could be a nuisance in this case because it will happen in any case (caught or not caught). So it might be a better idea to raise a different exception, containing your comment about the situation as well as the original exception:

```
try:
    5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Got an error", e)
```

But this has the drawback of reducing the exception trace to exactly this `raise` while the `raise` without argument retains the original exception trace.

In Python 3 you can keep the original stack by using the `raise-from` syntax:

```
    raise ZeroDivisionError("Got an error") from e
```

Section 88.4: Catching multiple exceptions

There are a few ways to catch multiple exceptions.

The first is by creating a tuple of the exception types you wish to catch and handle in the same manner. This example will cause the code to ignore `KeyError` and `AttributeError` exceptions.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except (KeyError, AttributeError) as e:
    print("A KeyError or an AttributeError exception has been caught.")
```

If you wish to handle different exceptions in different ways, you can provide a separate exception block for each type. In this example, we still catch the `KeyError` and `AttributeError`, but handle the exceptions in different manners.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except KeyError as e:
    print("A KeyError has occurred. Exception message:", e)
except AttributeError as e:
    print("An AttributeError has occurred. Exception message:", e)
```

Section 88.5: Exception Hierarchy

Exception handling occurs based on an exception hierarchy, determined by the inheritance structure of the exception classes.

For example, `IOError` and `OSError` are both subclasses of `EnvironmentError`. Code that catches an `IOError` will not catch an `OSError`. However, code that catches an `EnvironmentError` will catch both `IOError`s and `OSError`s.

The hierarchy of built-in exceptions:

Python 2.x `Version ≥ 2.3`

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StandardError
| +-- BufferError
| +-- ArithmeticError
| | +-- FloatingPointError
| | +-- OverflowError
| | +-- ZeroDivisionError
| +-- AssertionError
| +-- AttributeError
| +-- EnvironmentError
| | +-- IOError
| | +-- OSError
| | +-- WindowsError (Windows)
| | +-- VMSError (VMS)
| +-- EOFError
| +-- ImportError
| +-- LookupError
| | +-- IndexError
| | +-- KeyError
| +-- MemoryError
| +-- NameError
| | +-- UnboundLocalError
| +-- ReferenceError
| +-- RuntimeError
| | +-- NotImplementedError
| +-- SyntaxError
| | +-- IndentationError
| | +-- TabError
| +-- SystemError
| +-- TypeError
| +-- ValueError
```

```
| +-- UnicodeError
| +-- UnicodeDecodeError
| +-- UnicodeEncodeError
| +-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
```

Python 3.x `Version ≥ 3.0`

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StopAsyncIteration
+-- ArithmeticError
| +-- FloatingPointError
| +-- OverflowError
| +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
+-- LookupError
| +-- IndexError
| +-- KeyError
+-- MemoryError
+-- NameError
| +-- UnboundLocalError
+-- OSError
| +-- BlockingIOError
| +-- ChildProcessError
| +-- ConnectionError
| | +-- BrokenPipeError
| | +-- ConnectionAbortedError
| | +-- ConnectionRefusedError
| | +-- ConnectionResetError
| +-- FileExistsError
| +-- FileNotFoundError
| +-- InterruptedError
| +-- IsADirectoryError
| +-- NotADirectoryError
| +-- PermissionError
| +-- ProcessLookupError
| +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
| +-- NotImplementedError
| +-- RecursionError
+-- SyntaxError
| +-- IndentationError
| +-- TabError
+-- SystemError
+-- TypeError
```

```
+-- ValueError
|    +-- UnicodeError
|    +-- UnicodeDecodeError
|    +-- UnicodeEncodeError
|    +-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
+-- ResourceWarning
```

Section 88.6: Else

Code in an else block will only be run if no exceptions were raised by the code in the `try` block. This is useful if you have some code you don't want to run if an exception is thrown, but you don't want exceptions thrown by that code to be caught.

For example:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Note that this kind of `else:` cannot be combined with an `if` starting the else-clause to an `elif`. If you have a following `if` it needs to stay indented below that `else:`:

```
try:
    ...
except ...:
    ...
else:
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
```

Section 88.7: Raising Exceptions

If your code encounters a condition it doesn't know how to handle, such as an incorrect parameter, it should raise the appropriate exception.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")
```

```
    return odds + 1
```

Section 88.8: Creating custom exception types

Create a class inheriting from `Exception`:

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
    print("A FooException was raised.")
```

or another exception type:

```
class NegativeError(ValueError):
      pass

def foo(x):
    # function that only accepts positive values of x
    if x < 0:
        raise NegativeError("Cannot process negative numbers")
    ...  # rest of function body
try:
    result = foo(int(input("Enter a positive integer: ")))  # raw_input in Python 2.x
except NegativeError:
    print("You entered a negative number!")
else:
    print("The result was " + str(result))
```

Section 88.9: Practical examples of exception handling

User input

Imagine you want a user to enter a number via `input`. You want to ensure that the input is a number. You can use `try`/`except` for this:

Python 3.x `Version ≥ 3.0`

```
while True:
    try:
        nb = int(input('Enter a number: '))
        break
    except ValueError:
        print('This is not a number, try again.')
```

Note: Python 2.x would use `raw_input` instead; the function `input` exists in Python 2.x but has different semantics. In the above example, `input` would also accept expressions such as `2 + 2` which evaluate to a number.

If the input could not be converted to an integer, a `ValueError` is raised. You can catch it with `except`. If no exception is raised, `break` jumps out of the loop. After the loop, `nb` contains an integer.

Dictionaries

Imagine you are iterating over a list of consecutive integers, like `range(n)`, and you have a list of dictionaries `d` that contains information about things to do when you encounter some particular integers, say *skip the `d[i]` next ones*.

```
d = [{7: 3}, {25: 9}, {38: 5}]

for i in range(len(d)):
    do_stuff(i)
    try:
       dic = d[i]
       i += dic[i]
    except KeyError:
       i += 1
```

A `KeyError` will be raised when you try to get a value from a dictionary for a key that doesn't exist.

Section 88.10: Exceptions are Objects too

Exceptions are just regular Python objects that inherit from the built-in `BaseException`. A Python script can use the `raise` statement to interrupt execution, causing Python to print a stack trace of the call stack at that point and a representation of the exception instance. For example:

```
>>> def failing_function():
...     raise ValueError('Example error!')
>>> failing_function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in failing_function
ValueError: Example error!
```

which says that a `ValueError` with the message `'Example error!'` was raised by our `failing_function()`, which was executed in the interpreter.

Calling code can choose to handle any and all types of exception that a call can raise:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

You can get hold of the exception objects by assigning them in the `except...` part of the exception handling code:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

A complete list of built-in Python exceptions along with their descriptions can be found in the Python Documentation: https://docs.python.org/3.5/library/exceptions.html. And here is the full list arranged hierarchically: Exception Hierarchy.

Section 88.11: Running clean-up code with finally

Sometimes, you may want something to occur regardless of whatever exception happened, for example, if you have to clean up some resources.

The `finally` block of a `try` clause will happen regardless of whether any exceptions were raised.

```
resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise  # re-raise the error
finally:
    free_expensive_resource(resource)
```

This pattern is often better handled with context managers (using the `with` statement).

Section 88.12: Chain exceptions with raise from

In the process of handling an exception, you may want to raise another exception. For example, if you get an `IOError` while reading from a file, you may want to raise an application-specific error to present to the users of your library, instead.

Python 3.x `Version ≥ 3.0`

You can chain exceptions to show how the handling of exceptions proceeded:

```
>>> try:
    5 / 0
except ZeroDivisionError as e:
    raise ValueError("Division failed") from e

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Division failed
```

Chapter 89: Raise Custom Errors / Exceptions

Python has many built-in exceptions which force your program to output an error when something in it goes wrong.

However, sometimes you may need to create custom exceptions that serve your purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived from this class.

Section 89.1: Custom Exception

Here, we have created a user-defined exception called CustomError which is derived from the Exception class. This new exception can be raised, like other exceptions, using the raise statement with an optional error message.

```
class CustomError(Exception):
       pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

Output:

```
Traceback (most recent call last):
File "error_custom.py", line 8, in
raise CustomError('This is custom error')
__main__.CustomError: This is custom error
```

Section 89.2: Catch custom Exception

This example shows how to catch custom Exception

```
class CustomError(Exception):
     pass

try:
    raise CustomError('Can you catch me ?')
except CustomError as e:
    print ('Catched CustomError :{}'.format(e))
except Exception as e:
    print ('Generic exception: {}'.format(e))
```

Output:

```
Catched CustomError :Can you catch me ?
```

Chapter 90: Commonwealth Exceptions

Here in Stack Overflow we often see duplicates talking about the same errors: `"ImportError: No module named '??????'`, `SyntaxError: invalid syntax` or `NameError: name '???' is not defined`. This is an effort to reduce them and to have some documentation to link to.

Section 90.1: Other Errors

AssertError

The `assert` statement exists in almost every programming language. When you do:

```
assert condition
```

or:

```
assert condition, message
```

It's equivalent to this:

```
if __debug__:
    if not condition: raise AssertionError(message)
```

Assertions can include an optional message, and you can disable them when you're done debugging.

Note: the built-in variable **debug** is True under normal circumstances, False when optimization is requested (command line option -O). Assignments to **debug** are illegal. The value for the built-in variable is determined when the interpreter starts.

KeyboardInterrupt

Error raised when the user presses the interrupt key, normally `Ctrl` + `C` or `del`.

ZeroDivisionError

You tried to calculate `1/0` which is undefined. See this example to find the divisors of a number:

Python 2.x `Version ≥ 2.0` `Version ≤ 2.7`

```
div = float(raw_input("Divisors of: "))
for x in xrange(div+1): #includes the number itself and zero
    if div/x == div//x:
        print x, "is a divisor of", div
```

Python 3.x `Version ≥ 3.0`

```
div = int(input("Divisors of: "))
for x in range(div+1): #includes the number itself and zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

It raises `ZeroDivisionError` because the `for` loop assigns that value to `x`. Instead it should be:

Python 2.x `Version ≥ 2.0` `Version ≤ 2.7`

```
div = float(raw_input("Divisors of: "))
for x in xrange(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
```

```
        print x, "is a divisor of", div
```

Python 3.x Version ≥ 3.0

```
div = int(input("Divisors of: "))
for x in range(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

Section 90.2: NameError: name '???' is not defined

Is raised when you tried to use a variable, method or function that is not initialized (at least not before). In other words, it is raised when a requested local or global name is not found. It's possible that you misspelt the name of the object or forgot to `import` something. Also maybe it's in another scope. We'll cover those with separate examples.

It's simply not defined nowhere in the code

It's possible that you forgot to initialize it, especially if it is a constant

```
foo   # This variable is not defined
bar() # This function is not defined
```

Maybe it's defined later:

```
baz()

def baz():
    pass
```

Or it wasn't `import`ed:

```
#needs import math

def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Python scopes and the LEGB Rule:

The so-called LEGB Rule talks about the Python scopes. Its name is based on the different scopes, ordered by the correspondent priorities:

```
Local → Enclosed → Global → Built-in.
```

- **L**ocal: Variables not declared global or assigned in a function.
- **E**nclosing: Variables defined in a function that is wrapped inside another function.
- **G**lobal: Variables declared global, or assigned at the top-level of a file.
- **B**uilt-in: Variables preassigned in the built-in names module.

As an example:

```
for i in range(4):
    d = i * 2
print(d)
```

d is accessible because the `for` loop does not mark a new scope, but if it did, we would have an error and its behavior would be similar to:

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python says `NameError: name 'd' is not defined`

Section 90.3: TypeErrors

These exceptions are caused when the type of some object should be different

TypeError: [definition/method] takes ? positional arguments but ? was given

A function or method was called with more (or less) arguments than the ones it can accept.

Example

If more arguments are given:

```
def foo(a): return a
foo(a,b,c,d) #And a,b,c,d are defined
```

If less arguments are given:

```
def foo(a,b,c,d): return a += b + c + d
foo(a) #And a is defined
```

Note: if you want use an unknown number of arguments, you can use `*args` or `**kwargs`. See *args and **kwargs

TypeError: unsupported operand type(s) for [operand]: '???' and '???'

Some types cannot be operated together, depending on the operand.

Example

For example: `+` is used to concatenate and add, but you can't use any of them for both types. For instance, trying to make a `set` by concatenating (+ing) `'set1'` and `'tuple1'` gives the error. Code:

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

Some types (eg: `int` and `string`) use both `+` but for different things:

```
b = 400 + 'foo'
```

Or they may not be even used for anything:

```
c = ["a","b"] - [1,2]
```

But you can for example add a `float` to an `int`:

```
d = 1 + 1.0
```

TypeError: '???' object is not iterable/subscriptable:

For an object to be iterable it can take sequential indexes starting from zero until the indexes are no longer valid and a `IndexError` is raised (More technically: it has to have an `__iter__` method which returns an `__iterator__`, or which defines a `__getitem__` method that does what was previously mentioned).

Example

Here we are saying that `bar` is the zeroth item of 1. Nonsense:

```
foo = 1
bar = foo[0]
```

This is a more discrete version: In this example `for` tries to set `x` to `amount[0]`, the first item in an iterable but it can't because amount is an int:

```
amount = 10
for x in amount: print(x)
```

TypeError: '???' object is not callable

You are defining a variable and calling it later (like what you do with a function or method)

Example

```
foo = "notAFunction"
foo()
```

Section 90.4: Syntax Error on good code

The gross majority of the time a SyntaxError which points to an uninteresting line means there is an issue on the line before it (in this example, it's a missing parenthesis):

```
def my_print():
    x = (1 + 1
    print(x)
```

Returns

```
  File "<input>", line 3
    print(x)
        ^
SyntaxError: invalid syntax
```

The most common reason for this issue is mismatched parentheses/brackets, as the example shows.

There is one major caveat for print statements in Python 3:

Python 3.x `Version ≥ 3.0`

```
>>> print "hello world"
 File "<stdin>", line 1
   print "hello world"
                     ^
```

```
SyntaxError: invalid syntax
```

Because the `print` statement was replaced with the `print()` function, so you want:

```
print("hello world")  # Note this is valid for both Py2 & Py3
```

Section 90.5: IndentationErrors (or indentation SyntaxErrors)

In most other languages indentation is not compulsory, but in Python (and other languages: early versions of FORTRAN, Makefiles, Whitespace (esoteric language), etc.) that is not the case, what can be confusing if you come from another language, if you were copying code from an example to your own, or simply if you are new.

IndentationError/SyntaxError: unexpected indent

This exception is raised when the indentation level increases with no reason.

Example

There is no reason to increase the level here:

Python 2.x `Version ≥ 2.0` `Version ≤ 2.7`

```
 print "This line is ok"
     print "This line isn't ok"
```

Python 3.x `Version ≥ 3.0`

```
 print("This line is ok")
     print("This line isn't ok")
```

Here there are two errors: the last one and that the indentation does not match any indentation level. However just one is shown:

Python 2.x `Version ≥ 2.0` `Version ≤ 2.7`

```
 print "This line is ok"
  print "This line isn't ok"
```

Python 3.x `Version ≥ 3.0`

```
 print("This line is ok")
  print("This line isn't ok")
```

IndentationError/SyntaxError: unindent does not match any outer indentation level

Appears you didn't unindent completely.

Example

Python 2.x `Version ≥ 2.0` `Version ≤ 2.7`

```
def foo():
    print "This should be part of foo()"
   print "ERROR!"
print "This is not a part of foo()"
```

Python 3.x `Version ≥ 3.0`

```
 print("This line is ok")
  print("This line isn't ok")
```

IndentationError: expected an indented block

After a colon (and then a new line) the indentation level has to increase. This error is raised when that didn't happen.

Example

```
if ok:
doStuff()
```

Note: Use the keyword `pass` (that makes absolutely nothing) to just put an `if`, `else`, `except`, `class`, `method` or `definition` but not say what will happen if called/condition is true (but do it later, or in the case of `except`: just do nothing):

```
def foo():
    pass
```

IndentationError: inconsistent use of tabs and spaces in indentation

Example

```
def foo():
    if ok:
      return "Two != Four != Tab"
        return "i don't care i do whatever i want"
```

How to avoid this error

Don't use tabs. It is discouraged by `PEP8`, the style guide for Python.

1. Set your editor to use 4 **spaces** for indentation.
2. Make a search and replace to replace all tabs with 4 spaces.
3. Make sure your editor is set to **display** tabs as 8 spaces, so that you can realize easily that error and fix it.

See this question if you want to learn more.

Chapter 91: urllib

Section 91.1: HTTP GET

Python 2.x `Version ≤ 2.7`

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

Using `urllib.urlopen()` will return a response object, which can be handled similar to a file.

```
print response.code
# Prints: 200
```

The `response.code` represents the http return value. 200 is OK, 404 is NotFound, etc.

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

`response.read()` and `response.readlines()` can be used to read the actual html file returned from the request. These methods operate similarly to `file.read*`

Python 3.x `Version ≥ 3.0`

Python 3

```
import urllib.request

print(urllib.request.urlopen("http://stackoverflow.com/documentation/"))
# Prints: <http.client.HTTPResponse at 0x7f37a97e3b00>

response = urllib.request.urlopen("http://stackoverflow.com/documentation/")

print(response.code)
# Prints: 200
print(response.read())
# Prints: b'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack Overflow</title>
```

The module has been updated for Python 3.x, but use cases remain basically the same. `urllib.request.urlopen` will return a similar file-like object.

Section 91.2: HTTP POST

To POST data pass the encoded query arguments as data to urlopen()

Python 2.x `Version ≤ 2.7`

Python 2

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.urlencode(query_parms)
response = urllib.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: '<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Log In - Stack Overflow'
```

Python 3.x Version ≥ 3.0

Python 3

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.parse.urlencode(query_parms).encode('utf-8')
response = urllib.request.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: b'<!DOCTYPE html>\r\n<html>....etc'
```

Section 91.3: Decode received bytes according to content type encoding

The received bytes have to be decoded with the correct character encoding to be interpreted as text:

Python 3.x Version ≥ 3.0

```
import urllib.request

response = urllib.request.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Python 2.x Version ≤ 2.7

```
import urllib2
response = urllib2.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().getencoding()
html = data.decode(encoding)
```

Chapter 92: Web scraping with Python

Web scraping is an automated, programmatic process through which data can be constantly 'scraped' off webpages. Also known as screen scraping or web harvesting, web scraping can provide instant data from any publicly accessible webpage. On some websites, web scraping may be illegal.

Section 92.1: Scraping using the Scrapy framework

First you have to set up a new Scrapy project. Enter a directory where you'd like to store your code and run:

```
scrapy startproject projectName
```

To scrape we need a spider. Spiders define how a certain site will be scraped. Here's the code for a spider that follows the links to the top voted questions on StackOverflow and scrapes some data from each page (source):

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow'  # each spider has a unique name
    start_urls = ['http://stackoverflow.com/questions?sort=votes']  # the parsing starts from a specific set of urls

    def parse(self, response):  # for each request this generator yields, its response is sent to parse_question
        for href in response.css('.question-summary h3 a::attr(href)'):  # do some scraping stuff using css selectors to find question urls
            full_url = response.urljoin(href.extract())
            yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract_first(),
            'votes': response.css('.question .vote-count-post::text').extract_first(),
            'body': response.css('.question .post-text').extract_first(),
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

Save your spider classes in the `projectName\spiders` directory. In this case - `projectName\spiders\stackoverflow_spider.py`.

Now you can use your spider. For example, try running (in the project's directory):

```
scrapy crawl stackoverflow
```

Section 92.2: Scraping using Selenium WebDriver

Some websites don't like to be scraped. In these cases you may need to simulate a real user working with a browser. Selenium launches and controls a web browser.

```
from selenium import webdriver

browser = webdriver.Firefox()  # launch Firefox browser

browser.get('http://stackoverflow.com/questions?sort=votes')  # load url
```

```
title = browser.find_element_by_css_selector('h1').text  # page title (first h1 element)

questions = browser.find_elements_by_css_selector('.question-summary')  # question list

for question in questions:  # iterate over questions
    question_title = question.find_element_by_css_selector('.summary h3 a').text
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-
post').text

    print "%s\n%s\n%s votes\n-----------\n" % (question_title, question_excerpt, question_vote)
```

Selenium can do much more. It can modify browser's cookies, fill in forms, simulate mouse clicks, take screenshots of web pages, and run custom JavaScript.

Section 92.3: Basic example of using requests and lxml to scrape some data

```
# For Python 2 compatibility.
from __future__ import print_function

import lxml.html
import requests


def main():
    r = requests.get("https://httpbin.org")
    html_source = r.text
    root_element = lxml.html.fromstring(html_source)
    # Note root_element.xpath() gives a *list* of results.
    # XPath specifies a path to the element we want.
    page_title = root_element.xpath('/html/head/title/text()')[0]
    print(page_title)

if __name__ == '__main__':
    main()
```

Section 92.4: Maintaining web-scraping session with requests

It is a good idea to maintain a web-scraping session to persist the cookies and other parameters. Additionally, it can result into a *performance improvement* because `requests.Session` reuses the underlying TCP connection to a host:

```
import requests

with requests.Session() as session:
    # all requests through session now have User-Agent header set
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36'}

    # set cookies
    session.get('http://httpbin.org/cookies/set?key=value')

    # get cookies
    response = session.get('http://httpbin.org/cookies')
    print(response.text)
```

Section 92.5: Scraping using BeautifulSoup4

```
from bs4 import BeautifulSoup
import requests

# Use the requests module to obtain a page
res = requests.get('https://www.codechef.com/problems/easy')

# Create a BeautifulSoup object
page = BeautifulSoup(res.text, 'lxml')   # the text field contains the source of the page

# Now use a CSS selector in order to get the table containing the list of problems
datatable_tags = page.select('table.dataTable')  # The problems are in the <table> tag,
                                                 # with class "dataTable"
# We extract the first tag from the list, since that's what we desire
datatable = datatable_tags[0]
# Now since we want problem names, they are contained in <b> tags, which are
# directly nested under <a> tags
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

Section 92.6: Simple web content download with urllib.request

The standard library module `urllib.request` can be used to download web content:

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()

# The received bytes should usually be decoded according the response's character set
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

A similar module is also available in Python 2.

Section 92.7: Modify Scrapy user agent

Sometimes the default Scrapy user agent (`"Scrapy/VERSION (+http://scrapy.org)"`) is blocked by the host. To change the default user agent open **settings.py**, uncomment and edit the following line to whatever you want.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

For example

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36'
```

Section 92.8: Scraping with curl

imports:

```
from subprocess import Popen, PIPE
from lxml import etree
```

```
from io import StringIO
```

Downloading:

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.95 Safari/537.36'
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

`-s`: silent download

`-A`: user agent flag

Parsing:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Chapter 93: HTML Parsing

Section 93.1: Using CSS selectors in BeautifulSoup

BeautifulSoup has a limited support for CSS selectors, but covers most commonly used ones. Use `SELECT()` method to find multiple elements and `select_one()` to find a single element.

Basic example:

```
from bs4 import BeautifulSoup

data = """
<ul>
    <li class="item">item1</li>
    <li class="item">item2</li>
    <li class="item">item3</li>
</ul>
"""

soup = BeautifulSoup(data, "html.parser")

for item in soup.select("li.item"):
    print(item.get_text())
```

Prints:

```
item1
item2
item3
```

Section 93.2: PyQuery

pyquery is a jquery-like library for python. It has very well support for css selectors.

```
from pyquery import PyQuery

html = """
<h1>Sales</h1>
<table id="table">
<tr>
    <td>Lorem</td>
    <td>46</td>
</tr>
<tr>
    <td>Ipsum</td>
    <td>12</td>
</tr>
<tr>
    <td>Dolor</td>
    <td>27</td>
</tr>
<tr>
    <td>Sit</td>
    <td>90</td>
</tr>
</table>
"""
```

```
doc = PyQuery(html)

title = doc('h1').text()

print title

table_data = []

rows = doc('#table > tr')
for row in rows:
    name = PyQuery(row).find('td').eq(0).text()
    value = PyQuery(row).find('td').eq(1).text()

    print "%s\t  %s" % (name, value)
```

Section 93.3: Locate a text after an element in BeautifulSoup

Imagine you have the following HTML:

```
<div>
    <label>Name:</label>
    John Smith
</div>
```

And you need to locate the text "John Smith" after the `label` element.

In this case, you can locate the `label` element by text and then use `.next_sibling` property:

```
from bs4 import BeautifulSoup

data = """
<div>
    <label>Name:</label>
    John Smith
</div>
"""

soup = BeautifulSoup(data, "html.parser")

label = soup.find("label", text="Name:")
print(label.next_sibling.strip())
```

Prints `John Smith`.

Chapter 94: Manipulating XML

Section 94.1: Opening and reading using an ElementTree

Import the ElementTree object, open the relevant .xml file and get the root tag:

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

There are a few ways to search through the tree. First is by iteration:

```
for child in root:
    print(child.tag, child.attrib)
```

Otherwise you can reference specific locations like a list:

```
print(root[0][1].text)
```

To search for specific tags by name, use the `.find` or `.findall`:

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

Section 94.2: Create and Build XML Documents

Import Element Tree module

```
import xml.etree.ElementTree as ET
```

Element() function is used to create XML elements

```
p=ET.Element('parent')
```

SubElement() function used to create sub-elements to a give element

```
c = ET.SubElement(p, 'child1')
```

dump() function is used to dump xml elements.

```
ET.dump(p)
# Output will be like this
#<parent><child1 /></parent>
```

If you want to save to a file create a xml tree with ElementTree() function and to save to a file use write() method

```
tree = ET.ElementTree(p)
tree.write("output.xml")
```

Comment() function is used to insert comments in xml file.

```
comment = ET.Comment('user comment')
p.append(comment) #this comment will be appended to parent element
```

Section 94.3: Modifying an XML File

Import Element Tree module and open xml file, get an xml element

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #get first child of root element
```

Element object can be manipulated by changing its fields, adding and modifying attributes, adding and removing children

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

If you want to remove an element use Element.remove() method

```
root.remove(element)
```

ElementTree.write() method used to output xml object to xml files.

```
tree.write('output.xml')
```

Section 94.4: Searching the XML with XPath

Starting with version 2.7 `ElementTree` has a better support for XPath queries. XPath is a syntax to enable you to navigate through an xml like SQL is used to search through a database. Both `find` and `findall` functions support XPath. The xml below will be used for this example

```
<Catalog>
   <Books>
       <Book id="1" price="7.95">
           <Title>Do Androids Dream of Electric Sheep?</Title>
           <Author>Philip K. Dick</Author>
       </Book>
       <Book id="5" price="5.95">
           <Title>The Colour of Magic</Title>
           <Author>Terry Pratchett</Author>
       </Book>
       <Book id="7" price="6.95">
           <Title>The Eye of The World</Title>
           <Author>Robert Jordan</Author>
       </Book>
   </Books>
</Catalog>
```

Searching for all books:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Searching for the book with title = 'The Colour of Magic':

```
tree.find("Books/Book[Title='The Colour of Magic']")
# always use '' in the right side of the comparison
```

Searching for the book with id = 5:

```
tree.find("Books/Book[@id='5']")
# searches with xml attributes must have '@' before the name
```

Search for the second book:

```
tree.find("Books/Book[2]")
# indexes starts at 1, not 0
```

Search for the last book:

```
tree.find("Books/Book[last()]")
# 'last' is the only xpath function allowed in ElementTree
```

Search for all authors:

```
tree.findall(".//Author")
#searches with // must use a relative path
```

Section 94.5: Opening and reading large XML files using iterparse (incremental parsing)

Sometimes we don't want to load the entire XML file in order to get the information we need. In these instances, being able to incrementally load the relevant sections and then delete them when we are finished is useful. With the iterparse function you can edit the element tree that is stored while parsing the XML.

Import the ElementTree object:

```
import xml.etree.ElementTree as ET
```

Open the .xml file and iterate over all the elements:

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

Alternatively, we can only look for specific events, such as start/end tags or namespaces. If this option is omitted (as above), only "end" events are returned:

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Here is the complete example showing how to clear elements from the in-memory tree when we are finished with them:

```
for event, elem in ET.iterparse("yourXMLfile.xml", events=("start","end")):
    if elem.tag == "record_tag" and event == "end":
        print elem.text
        elem.clear()
    ... do something else ...
```

Chapter 95: Python Requests Post

Documentation for the Python Requests module in the context of the HTTP POST method and its corresponding Requests function

Section 95.1: Simple Post

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

Will perform a simple HTTP POST operation. Posted data can be inmost formats, however key value pairs are most prevalent.

Headers

Headers can be viewed:

```
print(foo.headers)
```

An example response:

```
{'Content-Length': '439', 'X-Processed-Time': '0.000802993774414', 'X-Powered-By': 'Flask',
'Server': 'meinheld/0.6.1', 'Connection': 'keep-alive', 'Via': '1.1 vegur', 'Access-Control-Allow-
Credentials': 'true', 'Date': 'Sun, 21 May 2017 20:56:05 GMT', 'Access-Control-Allow-Origin': '*',
'Content-Type': 'application/json'}
```

Headers can also be prepared before post:

```
headers = {'Cache-Control':'max-age=0',
        'Upgrade-Insecure-Requests':'1',
        'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/54.0.2840.99 Safari/537.36',
        'Content-Type':'application/x-www-form-urlencoded',
        'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
        'Referer':'https://www.groupon.com/signup',
        'Accept-Encoding':'gzip, deflate, br',
        'Accept-Language':'es-ES,es;q=0.8'
        }

 foo = post('http://httpbin.org/post', headers=headers, data = {'key':'value'})
```

Encoding

Encoding can be set and viewed in much the same way:

```
 print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

SSL Verification

Requests by default validates SSL certificates of domains. This can be overridden:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

Redirection

Any redirection will be followed (e.g. http to https) this can also be changed:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

If the post operation has been redirected, this value can be accessed:

```
print(foo.url)
```

A full history of redirects can be viewed:

```
print(foo.history)
```

Section 95.2: Form Encoded Data

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
           }

foo = post('http://httpbin.org/post', data=payload)
```

To pass form encoded data with the post operation, data must be structured as dictionary and supplied as the data parameter.

If the data does not want to be form encoded, simply pass a string, or integer to the data parameter.

Supply the dictionary to the json parameter for Requests to format the data automatically:

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}

foo = post('http://httpbin.org/post', json=payload)
```

Section 95.3: File Upload

With the Requests module, it's only necessary to provide a file handle as opposed to the contents retrieved with `.read()`:

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

Filename, content_type and headers can also be set:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel', {'Expires':
'0'})}
```

```
foo = requests.post('http://httpbin.org/post', files=files)
```

Strings can also be sent as a file, as long they are supplied as the `files` parameter.

Multiple Files

Multiple files can be supplied in much the same way as one file:

```
multiple_files = [
    ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
    ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]

foo = post('http://httpbin.org/post', files=multiple_files)
```

Section 95.4: Responses

Response codes can be viewed from a post operation:

```
from requests import post

foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

Returned Data

Accessing data that is returned:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

Raw Responses

In the instances where you need to access the underlying urllib3 response.HTTPResponse object, this can be done by the following:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw

print(res.read())
```

Section 95.5: Authentication

Simple HTTP Authentication

Simple HTTP Authentication can be achieved with the following:

```
from requests import post

foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

This is technically short hand for the following:

```
from requests import post
from requests.auth import HTTPBasicAuth
```

```
foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

HTTP Digest Authentication

HTTP Digest Authentication is done in a very similar way, Requests provides a different object for this:

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0', 'natas0'))
```

Custom Authentication

In some cases the built in authentication mechanisms may not be enough, imagine this example:

A server is configured to accept authentication if the sender has the correct user-agent string, a certain header value and supplies the correct credentials through HTTP Basic Authentication. To achieve this a custom authentication class should be prepared, subclassing AuthBase, which is the base for Requests authentication implementations:

```
from requests.auth import AuthBase
from requests.auth import _basic_auth_str
from requests._internal_utils import to_native_string

class CustomAuth(AuthBase):

    def __init__(self, secret_header, user_agent , username, password):
        # setup any auth-related data here
        self.secret_header =  secret_header
        self.user_agent = user_agent
        self.username = username
        self.password = password

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Secret'] = self.secret_header
        r.headers['User-Agent'] = self.user_agent
        r.headers['Authorization'] = _basic_auth_str(self.username, self.password)

        return r
```

This can then be utilized with the following code:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

Section 95.6: Proxies

Each request POST operation can be configured to use network proxies

HTTP/S Proxies

```
from requests import post

proxies = {
  'http': 'http://192.168.0.128:3128',
  'https': 'http://192.168.0.127:1080',
   }
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

HTTP Basic Authentication can be provided in this manner:

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

SOCKS Proxies

The use of socks proxies requires 3rd party dependencies `requests[socks]`, once installed socks proxies are used in a very similar way to HTTPBasicAuth:

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Chapter 96: Distribution

Section 96.1: py2app

To use the py2app framework you must install it first. Do this by opening terminal and entering the following command:

```
sudo easy_install -U py2app
```

You can also `pip` install the packages as:

```
pip install py2app
```

Then create the setup file for your python script:

```
py2applet --make-setup MyApplication.py
```

Edit the settings of the setup file to your liking, this is the default:

```
"""
This is a setup.py script generated by py2applet

Usage:
    python setup.py py2app
"""

from setuptools import setup

APP = ['test.py']
DATA_FILES = []
OPTIONS = {'argv_emulation': True}

setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
)
```

To add an icon file (this file must have a .icns extension), or include images in your application as reference, change your options as shown:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Finally enter this into terminal:

```
python setup.py py2app
```

The script should run and you will find your finished application in the dist folder.

Use the following options for more customization:

```
optimize (-O)         optimization level: -O1 for "python -O", -O2 for
                      "python -OO", and -O0 to disable [default: -O0]
```

```
includes (-i)          comma-separated list of modules to include

packages (-p)          comma-separated list of packages to include

extension              Bundle extension [default:.app for app, .plugin for
                       plugin]

extra-scripts          comma-separated list of additional scripts to include
                       in an application or plugin.
```

Section 96.2: cx_Freeze

Install cx_Freeze from here

Unzip the folder and run these commands from that directory:

```
python setup.py build
sudo python setup.py install
```

Create a new directory for your python script and create a **"setup.py"** file in the same directory with the following content:

```
application_title = "My Application" # Use your own application name
main_python_file = "my_script.py" # Your python script

import sys

from cx_Freeze import setup, Executable

base = None
if sys.platform == "win32":
    base = "Win32GUI"

includes = ["atexit","re"]

setup(
        name = application_title,
        version = "0.1",
        description = "Your Description",
        options = {"build_exe" : {"includes" : includes }},
        executables = [Executable(main_python_file, base = base)])
```

Now run your setup.py from terminal:

```
python setup.py bdist_mac
```

NOTE: On El Capitan this will need to be run as root with SIP mode disabled.

Chapter 97: Property Objects

Section 97.1: Using the @property decorator for read-write properties

If you want to use `@property` to implement custom behavior for setting and getting, use this pattern:

```
class Cash(object):
    def __init__(self, value):
        self.value = value
    @property
    def formatted(self):
        return '${:.2f}'.format(self.value)
    @formatted.setter
    def formatted(self, new):
        self.value = float(new[1:])
```

To use this:

```
>>> wallet = Cash(2.50)
>>> print(wallet.formatted)
$2.50
>>> print(wallet.value)
2.5
>>> wallet.formatted = '$123.45'
>>> print(wallet.formatted)
$123.45
>>> print(wallet.value)
123.45
```

Section 97.2: Using the @property decorator

The `@property` decorator can be used to define methods in a class which act like attributes. One example where this can be useful is when exposing information which may require an initial (expensive) lookup and simple retrieval thereafter.

Given some module `foobar.py`:

```
class Foo(object):
    def __init__(self):
        self.__bar = None

    @property
    def bar(self):
        if self.__bar is None:
            self.__bar = some_expensive_lookup_operation()
        return self.__bar
```

Then

```
>>> from foobar import Foo
>>> foo = Foo()
>>> print(foo.bar)  # This will take some time since bar is None after initialization
42
>>> print(foo.bar)  # This is much faster since bar has a value now
42
```

Section 97.3: Overriding just a getter, setter or a deleter of a property object

When you inherit from a class with a property, you can provide a new implementation for one or more of the property `getter`, `setter` or `deleter` functions, by referencing the property object *on the parent class*:

```
class BaseClass(object):
    @property
    def foo(self):
        return some_calculated_value()

    @foo.setter
    def foo(self, value):
        do_something_with_value(value)


class DerivedClass(BaseClass):
    @BaseClass.foo.setter
    def foo(self, value):
        do_something_different_with_value(value)
```

You can also add a setter or deleter where there was not one on the base class before.

Section 97.4: Using properties without decorators

While using decorator syntax (with the @) is convenient, it also a bit concealing. You can use properties directly, without decorators. The following Python 3.x example shows this:

```
class A:
    p = 1234
    def getX (self):
        return self._x

    def setX (self, value):
        self._x = value

    def getY (self):
        return self._y

    def setY (self, value):
        self._y = 1000 + value    # Weird but possible

    def getY2 (self):
        return self._y

    def setY2 (self, value):
        self._y = value

    def getT    (self):
        return self._t

    def setT (self, value):
        self._t = value

    def getU (self):
        return self._u + 10000

    def setU (self, value):
        self._u = value - 5000
```

```
    x, y, y2 = property (getX, setX), property (getY, setY), property (getY2, setY2)
    t = property (getT, setT)
    u = property (getU, setU)

A.q = 5678

class B:
    def getZ (self):
        return self.z_

    def setZ (self, value):
        self.z_ = value

    z = property (getZ, setZ)

class C:
    def __init__ (self):
        self.offset = 1234

    def getW (self):
        return self.w_ + self.offset

    def setW (self, value):
        self.w_ = value - self.offset

    w = property (getW, setW)

a1 = A ()
a2 = A ()

a1.y2 = 1000
a2.y2 = 2000

a1.x = 5
a1.y = 6

a2.x = 7
a2.y = 8

a1.t = 77
a1.u = 88

print (a1.x, a1.y, a1.y2)
print (a2.x, a2.y, a2.y2)
print (a1.p, a2.p, a1.q, a2.q)

print (a1.t, a1.u)

b = B ()
c = C ()

b.z = 100100
c.z = 200200
c.w = 300300

print (a1.x, b.z, c.z, c.w)

c.w = 400400
c.z = 500500
b.z = 600600
```

```
print (a1.x, b.z, c.z, c.w)
```

Chapter 98: Overloading

Section 98.1: Operator overloading

Below are the operators that can be overloaded in classes, along with the method definitions that are required, and an example of the operator in use within an expression.

N.B. The use of `other` as a variable name is not mandatory, but is considered the norm.

Operator	Method	Expression
`+` Addition	`__add__(self, other)`	`a1 + a2`
`-` Subtraction	`__sub__(self, other)`	`a1 - a2`
`*` Multiplication	`__mul__(self, other)`	`a1 * a2`
`@` Matrix Multiplication	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
`/` Division	`__div__(self, other)`	`a1 / a2` (*Python 2 only*)
`/` Division	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
`//` Floor Division	`__floordiv__(self, other)`	`a1 // a2`
`%` Modulo/Remainder	`__mod__(self, other)`	`a1 % a2`
`**` Power	`__pow__(self, other[, modulo])`	`a1 ** a2`
`<<` Bitwise Left Shift	`__lshift__(self, other)`	`a1 << a2`
`>>` Bitwise Right Shift	`__rshift__(self, other)`	`a1 >> a2`
`&` Bitwise AND	`__and__(self, other)`	`a1 & a2`
`^` Bitwise XOR	`__xor__(self, other)`	`a1 ^ a2`
`|` (Bitwise OR)	`__or__(self, other)`	`a1 | a2`
`-` Negation (Arithmetic)	`__neg__(self)`	`-a1`
`+` Positive	`__pos__(self)`	`+a1`
`~` Bitwise NOT	`__invert__(self)`	`~a1`
`<` Less than	`__lt__(self, other)`	`a1 < a2`
`<=` Less than or Equal to	`__le__(self, other)`	`a1 <= a2`
`==` Equal to	`__eq__(self, other)`	`a1 == a2`
`!=` Not Equal to	`__ne__(self, other)`	`a1 != a2`
`>` Greater than	`__gt__(self, other)`	`a1 > a2`
`>=` Greater than or Equal to	`__ge__(self, other)`	`a1 >= a2`
`[index]` Index operator	`__getitem__(self, index)`	`a1[index]`
`in` In operator	`__contains__(self, other)`	`a2 in a1`
`(*args, ...)` Calling	`__call__(self, *args, **kwargs)`	`a1(*args, **kwargs)`

The optional parameter `modulo` for `__pow__` is only used by the `pow` built-in function.

Each of the methods corresponding to a *binary* operator has a corresponding "right" method which start with `__r`, for example `__radd__`:

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
```

```
        return other + self.a

A(1) + 2  # Out:  3
2 + A(1)  # prints radd. Out: 3
```

as well as a corresponding inplace version, starting with `__i`:

```
class B:
    def __init__(self, b):
        self.b = b
    def __iadd__(self, other):
        self.b += other
        print("iadd")
        return self

b = B(2)
b.b       # Out: 2
b += 1    # prints iadd
b.b       # Out: 3
```

Since there's nothing special about these methods, many other parts of the language, parts of the standard library, and even third-party modules add magic methods on their own, like methods to cast an object to a type or checking properties of the object. For example, the builtin `str()` function calls the object's `__str__` method, if it exists. Some of these uses are listed below.

Function	Method	Expression
Casting to `int`	`__int__(self)`	`int(a1)`
Absolute function	`__abs__(self)`	`abs(a1)`
Casting to `str`	`__str__(self)`	`str(a1)`
Casting to `unicode`	`__unicode__(self)`	`unicode(a1)` (Python 2 only)
String representation	`__repr__(self)`	`repr(a1)`
Casting to `bool`	`__nonzero__(self)`	`bool(a1)`
String formatting	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
Hashing	`__hash__(self)`	`hash(a1)`
Length	`__len__(self)`	`len(a1)`
Reversed	`__reversed__(self)`	`reversed(a1)`
Floor	`__floor__(self)`	`math.floor(a1)`
Ceiling	`__ceil__(self)`	`math.ceil(a1)`

There are also the special methods `__enter__` and `__exit__` for context managers, and many more.

Section 98.2: Magic/Dunder Methods

Magic (also called dunder as an abbreviation for double-underscore) methods in Python serve a similar purpose to operator overloading in other languages. They allow a class to define its behavior when it is used as an operand in unary or binary operator expressions. They also serve as implementations called by some built-in functions.

Consider this implementation of two-dimensional vectors.

```
import math

class Vector(object):
    # instantiation
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y

    # unary negation (-v)
    def __neg__(self):
        return Vector(-self.x, -self.y)

    # addition (v + u)
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # subtraction (v - u)
    def __sub__(self, other):
        return self + (-other)

    # equality (v == u)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # abs(v)
    def __abs__(self):
        return math.hypot(self.x, self.y)

    # str(v)
    def __str__(self):
        return '<{0.x}, {0.y}>'.format(self)

    # repr(v)
    def __repr__(self):
        return 'Vector({0.x}, {0.y})'.format(self)
```

Now it is possible to naturally use instances of the `Vector` class in various expressions.

```
v = Vector(1, 4)
u = Vector(2, 0)

u + v           # Vector(3, 4)
print(u + v)    # "<3, 4>" (implicit string conversion)
u - v           # Vector(1, -4)
u == v          # False
u + v == v + u  # True
abs(u + v)      # 5.0
```

Section 98.3: Container and sequence types

It is possible to emulate container types, which support accessing values by key or index.

Consider this naive implementation of a sparse list, which stores only its non-zero elements to conserve memory.

```
class sparselist(object):
    def __init__(self, size):
        self.size = size
        self.data = {}

    # l[index]
    def __getitem__(self, index):
        if index < 0:
            index += self.size
        if index >= self.size:
            raise IndexError(index)
```

```
        try:
            return self.data[index]
        except KeyError:
            return 0.0

    # l[index] = value
    def __setitem__(self, index, value):
        self.data[index] = value

    # del l[index]
    def __delitem__(self, index):
        if index in self.data:
            del self.data[index]

    # value in l
    def __contains__(self, value):
        return value == 0.0 or value in self.data.values()

    # len(l)
    def __len__(self):
        return self.size

    # for value in l: ...
    def __iter__(self):
        return (self[i] for i in range(self.size)) # use xrange for python2
```

Then, we can use a `sparselist` much like a regular `list`.

```
l = sparselist(10 ** 6)  # list with 1 million elements
0 in l                   # True
10 in l                  # False

l[12345] = 10
10 in l                  # True
l[12345]                 # 10

for v in l:
    pass  # 0, 0, 0, ... 10, 0, 0 ... 0
```

Section 98.4: Callable types

```
class adder(object):
    def __init__(self, first):
        self.first = first

    # a(...)
    def __call__(self, second):
        return self.first + second

add2 = adder(2)
add2(1)  # 3
add2(2)  # 4
```

Section 98.5: Handling unimplemented behaviour

If your class doesn't implement a specific overloaded operator for the argument types provided, it should `return NotImplemented` (**note** that this is a special constant, not the same as `NotImplementedError`). This will allow Python to fall back to trying other methods to make the operation work:

> When `NotImplemented` is returned, the interpreter will then try the reflected operation on the other type, or some other fallback, depending on the operator. If all attempted operations return `NotImplemented`, the interpreter will raise an appropriate exception.

For example, given `x + y`, if `x.__add__(y)` returns unimplemented, `y.__radd__(x)` is attempted instead.

```
class NotAddable(object):

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return NotImplemented


class Addable(NotAddable):

    def __add__(self, other):
        return Addable(self.value + other.value)

    __radd__ = __add__
```

As this is the *reflected* method we have to implement `__add__` **and** `__radd__` to get the expected behaviour in all cases; fortunately, as they are both doing the same thing in this simple example, we can take a shortcut.

In use:

```
>>> x = NotAddable(1)
>>> y = Addable(2)
>>> x + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NotAddable' and 'NotAddable'
>>> y + y
<so.Addable object at 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

Chapter 99: Polymorphism

Section 99.1: Duck Typing

Polymorphism without inheritance in the form of duck typing as available in Python due to its dynamic typing system. This means that as long as the classes contain the same methods the Python interpreter does not distinguish between them, as the only checking of the calls occurs at run-time.

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(obj):
    obj.quack()
    obj.feathers()

donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

The output is:

> Quaaaaaack!
> The duck has white and gray feathers.
> The person imitates a duck.
> The person takes a feather from the ground and shows it.

Section 99.2: Basic Polymorphism

Polymorphism is the ability to perform an action on an object regardless of its type. This is generally implemented by creating a base class and having two or more subclasses that all implement methods with the same signature. Any other function or method that manipulates these objects can call the same methods regardless of which type of object it is operating on, without needing to do a type check first. In object-oriented terminology when class X extend class Y , then Y is called super class or base class and X is called subclass or derived class.

```
class Shape:
    """
    This is a parent class that is intended to be inherited by other classes
    """

    def calculate_area(self):
        """
        This method is intended to be overridden in subclasses.
        If a subclass doesn't implement it but it is called, NotImplemented will be raised.
```

```
        """
        raise NotImplemented

class Square(Shape):
    """
    This is a subclass of the Shape class, and represents a square
    """
    side_length = 2     # in this example, the sides are 2 units long

    def calculate_area(self):
        """
        This method overrides Shape.calculate_area(). When an object of type
        Square has its calculate_area() method called, this is the method that
        will be called, rather than the parent class' version.

        It performs the calculation necessary for this shape, a square, and
        returns the result.
        """
        return self.side_length * 2

class Triangle(Shape):
    """
    This is also a subclass of the Shape class, and it represents a triangle
    """
    base_length = 4
    height = 3

    def calculate_area(self):
        """
        This method also overrides Shape.calculate_area() and performs the area
        calculation for a triangle, returning the result.
        """

        return 0.5 * self.base_length * self.height

def get_area(input_obj):
    """
    This function accepts an input object, and will call that object's
    calculate_area() method. Note that the object type is not specified. It
    could be a Square, Triangle, or Shape object.
    """

    print(input_obj.calculate_area())

# Create one object of each class
shape_obj = Shape()
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(shape_obj)
get_area(square_obj)
get_area(triangle_obj)
```

We should see this output:

```
None
4
```

```
6.0
```

What happens without polymorphism?
Without polymorphism, a type check may be required before performing an action on an object to determine the correct method to call. The following **counter example** performs the same task as the previous code, but without the use of polymorphism, the `get_area()` function has to do more work.

```
class Square:

    side_length = 2

    def calculate_square_area(self):
        return self.side_length ** 2

class Triangle:

    base_length = 4
    height = 3

    def calculate_triangle_area(self):
        return (0.5 * self.base_length) * self.height

def get_area(input_obj):

    # Notice the type checks that are now necessary here. These type checks
    # could get very complicated for a more complex example, resulting in
    # duplicate and difficult to maintain code.

    if type(input_obj).__name__ == "Square":
        area = input_obj.calculate_square_area()

    elif type(input_obj).__name__ == "Triangle":
        area = input_obj.calculate_triangle_area()

    print(area)

# Create one object of each class
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(square_obj)
get_area(triangle_obj)
```

We should see this output:

```
4
6.0
```

Important Note
Note that the classes used in the counter example are "new style" classes and implicitly inherit from the object class if Python 3 is being used. Polymorphism will work in both Python 2.x and 3.x, but the polymorphism counterexample code will raise an exception if run in a Python 2.x interpreter because type(input_obj).**name** will return "instance" instead of the class name if they do not explicitly inherit from object, resulting in area never being assigned to.

Chapter 100: Method Overriding

Section 100.1: Basic method overriding

Here is an example of basic overriding in Python (for the sake of clarity and compatibility with both Python 2 and 3, using new style class and `print` with `()`):

```
class Parent(object):
    def introduce(self):
        print("Hello!")

    def print_name(self):
        print("Parent")


class Child(Parent):
    def print_name(self):
        print("Child")


p = Parent()
c = Child()

p.introduce()
p.print_name()

c.introduce()
c.print_name()

$ python basic_override.py
Hello!
Parent
Hello!
Child
```

When the `Child` class is created, it inherits the methods of the `Parent` class. This means that any methods that the parent class has, the child class will also have. In the example, the `introduce` is defined for the `Child` class because it is defined for `Parent`, despite not being defined explicitly in the class definition of `Child`.

In this example, the overriding occurs when `Child` defines its own `print_name` method. If this method was not declared, then `c.print_name()` would have printed `"Parent"`. However, `Child` has overridden the `Parent`'s definition of `print_name`, and so now upon calling `c.print_name()`, the word `"Child"` is printed.

Chapter 101: User-Defined Methods

Section 101.1: Creating user-defined method objects

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object, an unbound user-defined method object, or a class method object.

```
class A(object):
    # func: A user-defined function object
    #
    # Note that func is a function object when it's defined,
    # and an unbound method object when it's retrieved.
    def func(self):
        pass

    # classMethod: A class method
    @classmethod
    def classMethod(self):
        pass

class B(object):
    # unboundMeth: A unbound user-defined method object
    #
    # Parent.func is an unbound user-defined method object here,
    # because it's retrieved.
    unboundMeth = A.func

a = A()
b = B()

print A.func
# output: <unbound method A.func>
print a.func
# output: <bound method A.func of <__main__.A object at 0x10e9ab910>>
print B.unboundMeth
# output: <unbound method A.func>
print b.unboundMeth
# output: <unbound method A.func>
print A.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
print a.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
```

When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object; otherwise, the original method object is used as it is.

```
# Parent: The class stored in the original method object
class Parent(object):
    # func: The underlying function of original method object
    def func(self):
        pass
    func2 = func

# Child: A derived class of Parent
class Child(Parent):
    func = Parent.func
```

```
# AnotherClass: A different class, neither subclasses nor subclassed
class AnotherClass(object):
    func = Parent.func

print Parent.func is Parent.func                # False, new object created
print Parent.func2 is Parent.func2              # False, new object created
print Child.func is Child.func                  # False, new object created
print AnotherClass.func is AnotherClass.func    # True, original object used
```

Section 101.2: Turtle example

The following is an example of using an user-defined function to be called multiple(∞) times in a script with ease.

```
import turtle, time, random #tell python we need 3 different modules
turtle.speed(0) #set draw speed to the fastest
turtle.colormode(255) #special colormode
turtle.pensize(4) #size of the lines that will be drawn
def triangle(size): #This is our own function, in the parenthesis is a variable we have defined that
will be used in THIS FUNCTION ONLY. This fucntion creates a right triangle
    turtle.forward(size) #to begin this function we go forward, the amount to go forward by is the
variable size
    turtle.right(90) #turn right by 90 degree
    turtle.forward(size) #go forward, again with variable
    turtle.right(135) #turn right again
    turtle.forward(size * 1.5) #close the triangle. thanks to the Pythagorean theorem we know that
this line must be 1.5 times longer than the other two(if they are equal)
while(1): #INFINITE LOOP
    turtle.setpos(random.randint(-200, 200), random.randint(-200, 200)) #set the draw point to a
random (x,y) position
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize the RGB color
    triangle(random.randint(5, 55)) #use our function, because it has only one variable we can
simply put a value in the parenthesis. The value that will be sent will be random between 5 - 55, end
the end it really just changes ow big the triangle is.
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize color again
```

Chapter 102: String representations of class instances: __str__ and __repr__ methods

Section 102.1: Motivation

So you've just created your first class in Python, a neat little class that encapsulates a playing card:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

Elsewhere in your code, you create a few instances of this class:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)
```

You've even created a list of cards, in order to represent a "hand":

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Now, during debugging, you want to see what your hand looks like, so you do what comes naturally and write:

```
print(my_hand)
```

But what you get back is a bunch of gibberish:

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

Confused, you try just printing a single card:

```
print(ace_of_spades)
```

And again, you get this weird output:

```
<__main__.Card instance at 0x0000000002533788>
```

Have no fear. We're about to fix this.

First, however, it's important to understand what's going on here. When you wrote `print(ace_of_spades)` you told Python you wanted it to print information about the `Card` instance your code is calling `ace_of_spades`. And to be fair, it did.

That output is comprised of two important bits: the `type` of the object and the object's `id`. The second part alone (the hexadecimal number) is enough to uniquely identify the object at the time of the `print` call.[1]

What really went on was that you asked Python to "put into words" the essence of that object and then display it to you. A more explicit version of the same machinery might be:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

In the first line, you try to turn your `Card` instance into a string, and in the second you display it.

The Problem

The issue you're encountering arises due to the fact that, while you told Python everything it needed to know about the `Card` class for you to *create* cards, you *didn't* tell it how you wanted `Card` instances to be converted to strings.

And since it didn't know, when you (implicitly) wrote `str(ace_of_spades)`, it gave you what you saw, a generic representation of the `Card` instance.

The Solution (Part 1)

But *we can* tell Python how we want instances of our custom classes to be converted to strings. And the way we do this is with the `__str__` "dunder" (for double-underscore) or "magic" method.

Whenever you tell Python to create a string from a class instance, it will look for a `__str__` method on the class, and call it.

Consider the following, updated version of our `Card` class:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

        card_name = special_names.get(self.pips, str(self.pips))

        return "%s of %s" % (card_name, self.suit)
```

Here, we've now defined the `__str__` method on our `Card` class which, after a simple dictionary lookup for face cards, **returns** a string formatted however we decide.

(Note that "returns" is in bold here, to stress the importance of returning a string, and not simply printing it. Printing it may seem to work, but then you'd have the card printed when you did something like `str(ace_of_spades)`, without even having a print function call in your main program. So to be clear, make sure that `__str__` returns a string.).

The `__str__` method is a method, so the first argument will be `self` and it should neither accept, nor be passed additional arguments.

Returning to our problem of displaying the card in a more user-friendly manner, if we again run:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

We'll see that our output is much better:

Ace of Spades

So great, we're done, right?

Well just to cover our bases, let's double check that we've solved the first issue we encountered, printing the list of `Card` instances, the `hand`.

So we re-check the following code:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

And, to our surprise, we get those funny hex codes again:

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

What's going on? We told Python how we wanted our `Card` instances to be displayed, why did it apparently seem to forget?

The Solution (Part 2)

Well, the behind-the-scenes machinery is a bit different when Python wants to get the string representation of items in a list. It turns out, Python doesn't care about `__str__` for this purpose.

Instead, it looks for a different method, `__repr__`, and if *that's* not found, it falls back on the "hexadecimal thing".[2]

So you're saying I have to make two methods to do the same thing? One for when I want to ***`print`*** *my card by itself and another when it's in some sort of container?*

No, but first let's look at what our class *would* be like if we were to implement both `__str__` and `__repr__` methods:

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (S)" % (card_name, self.suit)

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (R)" % (card_name, self.suit)
```

Here, the implementation of the two methods `__str__` and `__repr__` are exactly the same, except that, to differentiate between the two methods, `(S)` is added to strings returned by `__str__` and `(R)` is added to strings returned by `__repr__`.

Note that just like our `__str__` method, `__repr__` accepts no arguments and **returns** a string.

We can see now what method is responsible for each case:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)

my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

```
print(my_hand)          # [Ace of Spades (R), 4 of Clubs (R), 6 of Hearts (R)]

print(ace_of_spades)    # Ace of Spades (S)
```

As was covered, the `__str__` method was called when we passed our `Card` instance to `print` and the `__repr__` method was called when we passed *a list of our instances* to `print`.

At this point it's worth pointing out that just as we can explicitly create a string from a custom class instance using `str()` as we did earlier, we can also explicitly create a **string representation** of our class with a built-in function called `repr()`.

For example:

```
str_card = str(four_of_clubs)
print(str_card)                    # 4 of Clubs (S)

repr_card = repr(four_of_clubs)
print(repr_card)                   # 4 of Clubs (R)
```

And additionally, if defined, we *could* call the methods directly (although it seems a bit unclear and unnecessary):

```
print(four_of_clubs.__str__())     # 4 of Clubs (S)

print(four_of_clubs.__repr__())    # 4 of Clubs (R)
```

About those duplicated functions...

Python developers realized, in the case you wanted identical strings to be returned from `str()` and `repr()` you might have to functionally-duplicate methods -- something nobody likes.

So instead, there is a mechanism in place to eliminate the need for that. One I snuck you past up to this point. It turns out that if a class implements the `__repr__` method *but not* the `__str__` method, and you pass an instance of that class to `str()` (whether implicitly or explicitly), Python will fallback on your `__repr__` implementation and use that.

So, to be clear, consider the following version of the `Card` class:

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)
```

Note this version *only* implements the `__repr__` method. Nonetheless, calls to `str()` result in the user-friendly version:

```
print(six_of_hearts)            # 6 of Hearts  (implicit conversion)
print(str(six_of_hearts))       # 6 of Hearts  (explicit conversion)
```

as do calls to `repr()`:

```
print([six_of_hearts])          #[6 of Hearts] (implicit conversion)
```

```
print(repr(six_of_hearts))      # 6 of Hearts  (explicit conversion)
```

Summary

In order for you to empower your class instances to "show themselves" in user-friendly ways, you'll want to consider implementing at least your class's `__repr__` method. If memory serves, during a talk Raymond Hettinger said that ensuring classes implement `__repr__` is one of the first things he looks for while doing Python code reviews, and by now it should be clear why. The amount of information you *could* have added to debugging statements, crash reports, or log files with a simple method is overwhelming when compared to the paltry, and often less-than-helpful (type, id) information that is given by default.

If you want *different* representations for when, for example, inside a container, you'll want to implement both `__repr__` and `__str__` methods. (More on how you might use these two methods differently below).

Section 102.2: Both methods implemented, eval-round-trip style __repr__()

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    # Called when instance is converted to a string via str()
    # Examples:
    #   print(card1)
    #   print(str(card1)
    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)

    # Called when instance is converted to a string via repr()
    # Examples:
    #   print([card1, card2, card3])
    #   print(repr(card1))
    def __repr__(self):
        return "Card(%s, %d)" % (self.suit, self.pips)
```

Chapter 103: Debugging

Section 103.1: Via IPython and ipdb

If IPython (or Jupyter) are installed, the debugger can be invoked using:

```
import ipdb
ipdb.set_trace()
```

When reached, the code will exit and print:

```
 /home/usr/ook.py(3)<module>()
      1 import ipdb
      2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

Clearly, this means that one has to edit the code. There is a simpler way:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

This will cause the debugger to be called if there is an uncaught exception raised.

Section 103.2: The Python Debugger: Step-through Debugging with _pdb_

The Python Standard Library includes an interactive debugging library called *pdb*. *pdb* has extensive capabilities, the most commonly used being the ability to 'step-through' a program.

To immediately enter into step-through debugging use:

```
python -m pdb <my_file.py>
```

This will start the debugger at the first line of the program.

Usually you will want to target a specific section of the code for debugging. To do this we import the pdb library and use *set_trace()* to interrupt the flow of this troubled example code.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

Running this program will launch the interactive debugger.

```
python foo.py
> ~/scratch/foo.py(5)divide()
```

```
-> return a/b
(Pdb)
```

Often this command is used on one line so it can be commented out with a single # character

```
import pdf; pdb.set_trace()
```

At the *(Pdb)* prompt commands can be entered. These commands can be debugger commands or python. To print variables we can use *p* from the debugger, or python's *print*.

```
(Pdb) p a
1
(Pdb) print a
1
```

To see list of all local variables use

```
locals
```

build-in function

These are good debugger commands to know:

```
b  | : set breakpoint at line *n* or function named *f*.
# b 3
# b divide
b: show all breakpoints.
c: continue until the next breakpoint.
s: step through this line (will enter a function).
n: step over this line (jumps over a function).
r: continue until the current function returns.
l: list a window of code around this line.
p : print variable named *var*.
# p x
q: quit debugger.
bt: print the traceback of the current execution call stack
up: move your scope up the function call stack to the caller of the current function
down: Move your scope back down the function call stack one level
step: Run the program until the next line of execution in the program, then return control back to
the debugger
next: run the program until the next line of execution in the current function, then return
control back to the debugger
return: run the program until the current function returns, then return control back to the
debugger
continue: continue running the program until the next breakpoint (or set_trace si called again)
```

The debugger can also evaluate python interactively:

```
-> return a/b
(Pdb) p a+b
3
(Pdb) [ str(m) for m in [a,b]]
['1', '2']
(Pdb) [ d for d in xrange(5)]
[0, 1, 2, 3, 4]
```

Note:

If any of your variable names coincide with the debugger commands, use an exclamation mark '**!**' before the var to explicitly refer to the variable and not the debugger command. For example, often it might so happen that you use the variable name '**c**' for a counter, and you might want to print it while in the debugger. a simple '**c**' command would continue execution till the next breakpoint. Instead use '**!c**' to print the value of the variable as follows:

```
(Pdb) !c
4
```

Section 103.3: Remote debugger

Sometimes you need to debug python code which is executed by another process and in this cases rpdb comes in handy.

> rpdb is a wrapper around pdb that re-routes stdin and stdout to a socket handler. By default it opens the debugger on port 4444

Usage:

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

And then you need run this in terminal to connect to this process.

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

And you will get pdb promt

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
(Pdb)
```

Chapter 104: Reading and Writing CSV

Section 104.1: Using pandas

Write a CSV file from a `dict` or a `DataFrame`.

```
import pandas as pd

d = {'a': (1, 101), 'b': (2, 202), 'c': (3, 303)}
pd.DataFrame.from_dict(d, orient="index")
df.to_csv("data.csv")
```

Read a CSV file as a `DataFrame` and convert it to a `dict`:

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

Section 104.2: Writing a TSV file

Python

```
import csv

with open('/tmp/output.tsv', 'wt') as out_file:
    tsv_writer = csv.writer(out_file, delimiter='\t')
    tsv_writer.writerow(['name', 'field'])
    tsv_writer.writerow(['Dijkstra', 'Computer Science'])
    tsv_writer.writerow(['Shelah', 'Math'])
    tsv_writer.writerow(['Aumann', 'Economic Sciences'])
```

Output file

```
$ cat /tmp/output.tsv

name    field
Dijkstra    Computer Science
Shelah    Math
Aumann    Economic Sciences
```

Chapter 105: Writing to CSV from String or List

Parameter	Details
open (**"/path/"**, "mode")	Specify the path to your CSV file
open (path, **"mode"**)	Specify mode to open file in (read, write, etc.)
csv.writer(**file**, delimiter)	Pass opened CSV file here
csv.writer(file, **delimiter=' '**)	Specify delimiter character or pattern

Writing to a .csv file is not unlike writing to a regular file in most regards, and is fairly straightforward. I will, to the best of my ability, cover the easiest, and most efficient approach to the problem.

Section 105.1: Basic Write Example

```
import csv

#------ We will write to CSV in this function ------------

def csv_writer(data, path):

    #Open CSV file whose path we passed.
    with open(path, "wb") as csv_file:

        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)



#---- Define our list here, and call function ------------

if __name__ == "__main__":

    """
    data = our list that we want to write.
    Split it so we get a list of lists.
    """
    data = ["first_name,last_name,age".split(","),
            "John,Doe,22".split(","),
            "Jane,Doe,31".split(","),
            "Jack,Reacher,27".split(",")
            ]

    # Path to CSV file we want to write to.
    path = "output.csv"
    csv_writer(data, path)
```

Section 105.2: Appending a String as a newline in a CSV file

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

Chapter 106: Dynamic code execution with `exec` and `eval`

Argument	Details
`expression`	The expression code as a string, or a `code` object
`object`	The statement code as a string, or a `code` object
`globals`	The dictionary to use for global variables. If locals is not specified, this is also used for locals. If omitted, the `globals()` of calling scope are used.
`locals`	A *mapping* object that is used for local variables. If omitted, the one passed for `globals` is used instead. If both are omitted, then the `globals()` and `locals()` of the calling scope are used for `globals` and `locals` respectively.

Section 106.1: Executing code provided by untrusted user using exec, eval, or ast.literal_eval

It is not possible to use `eval` or `exec` to execute code from untrusted user securely. Even `ast.literal_eval` is prone to crashes in the parser. It is sometimes possible to guard against malicious code execution, but it doesn't exclude the possibility of outright crashes in the parser or the tokenizer.

To evaluate code by an untrusted user you need to turn to some third-party module, or perhaps write your own parser and your own virtual machine in Python.

Section 106.2: Evaluating a string containing a Python literal with ast.literal_eval

If you have a string that contains Python literals, such as strings, floats etc, you can use `ast.literal_eval` to evaluate its value instead of `eval`. This has the added feature of allowing only certain syntax.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

However, this is not secure for execution of code provided by untrusted user, and it is trivial to crash an interpreter with carefully crafted input

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped)  python3
```

Here, the input is a string of `()` repeated one million times, which causes a crash in CPython parser. CPython developers do not consider bugs in parser as security issues.

Section 106.3: Evaluating statements with exec

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
```

Section 106.4: Evaluating an expression with eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

Section 106.5: Precompiling an expression to evaluate it multiple times

`compile` built-in function can be used to precompile an expression to a code object; this code object can then be passed to eval. This will speed up the repeated executions of the evaluated code. The 3rd parameter to `compile` needs to be the string `'eval'`.

```
>>> code = compile('a * b + c', '<string>', 'eval')
>>> code
<code object <module> at 0x7f0e51a58830, file "<string>", line 1>
>>> a, b, c = 1, 2, 3
>>> eval(code)
5
```

Section 106.6: Evaluating an expression with eval using custom globals

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

As a plus, with this the code cannot accidentally refer to the names defined outside:

```
>>> eval('variables')
{'a': 6, 'b': 7}
>>> eval('variables', globals=variables)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'variables' is not defined
```

Using `defaultdict` allows for example having undefined variables set to zero:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables)  # note that 'c' is not explicitly defined
0
```

Chapter 107: PyInstaller - Distributing Python Code

Section 107.1: Installation and Setup

Pyinstaller is a normal python package. It can be installed using pip:

```
pip install pyinstaller
```

Installation in Windows
For Windows, pywin32 or pypiwin32 is a prerequisite. The latter is installed automatically when pyinstaller is installed using pip.

Installation in Mac OS X
PyInstaller works with the default Python 2.7 provided with current Mac OS X. If later versions of Python are to be used or if any major packages such as PyQT, Numpy, Matplotlib and the like are to be used, it is recommended to install them using either MacPorts or Homebrew.

Installing from the archive
If pip is not available, download the compressed archive from PyPI.
To test the development version, download the compressed archive from the *develop* branch of PyInstaller Downloads page.

Expand the archive and find the `setup.py` script. Execute `python setup.py install` with administrator privilege to install or upgrade PyInstaller.

Verifying the installation
The command `pyinstaller` should exist on the system path for all platforms after a successful installation.
Verify it by typing `pyinstaller --version` in the command line. This will print the current version of pyinstaller.

Section 107.2: Using Pyinstaller

In the simplest use-case, just navigate to the directory your file is in, and type:

```
pyinstaller myfile.py
```

Pyinstaller analyzes the file and creates:

- A **myfile.spec** file in the same directory as `myfile.py`
- A **build** folder in the same directory as `myfile.py`
- A **dist** folder in the same directory as `myfile.py`
- Log files in the **build** folder

The bundled app can be found in the **dist** folder

Options
There are several options that can be used with pyinstaller. A full list of the options can be found here.

Once bundled your app can be run by opening 'dist\myfile\myfile.exe'.

Section 107.3: Bundling to One Folder

When PyInstaller is used without any options to bundle `myscript.py` , the default output is a single folder (named `myscript`) containing an executable named `myscript` (`myscript.exe` in windows) along with all the necessary dependencies.
The app can be distributed by compressing the folder into a zip file.

One Folder mode can be explicitly set using the option `-D` or `--onedir`

```
pyinstaller myscript.py -D
```

Advantages:

One of the major advantages of bundling to a single folder is that it is easier to debug problems. If any modules fail to import, it can be verified by inspecting the folder.
Another advantage is felt during updates. If there are a few changes in the code but the dependencies used are *exactly* the same, distributors can just ship the executable file (which is typically smaller than the entire folder).

Disadvantages

The only disadvantage of this method is that the users have to search for the executable among a large number of files.
Also users can delete/modify other files which might lead to the app not being able to work correctly.

Section 107.4: Bundling to a Single File

```
pyinstaller myscript.py -F
```

The options to generate a single file are `-F` or `--onefile`. This bundles the program into a single `myscript.exe` file.

Single file executable are slower than the one-folder bundle. They are also harder to debug.

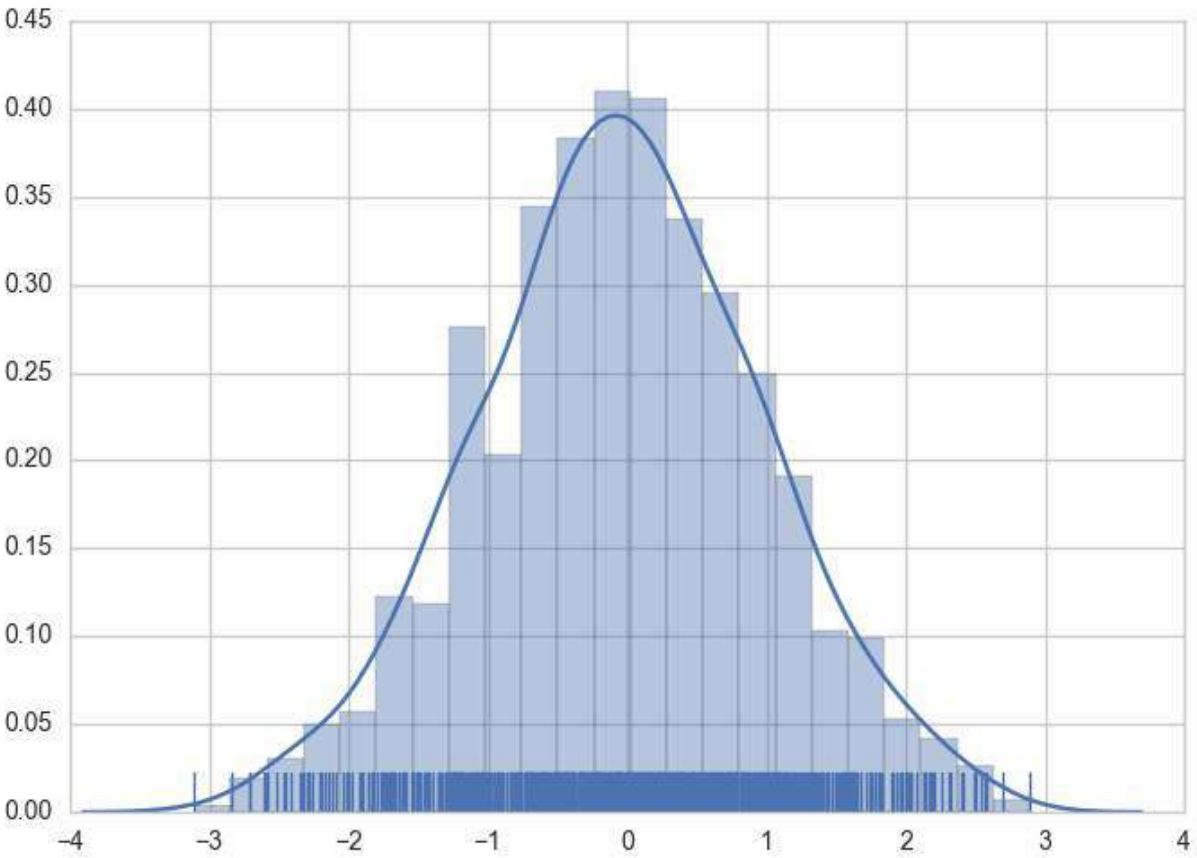
Chapter 108: Data Visualization with Python

Section 108.1: Seaborn

Seaborn is a wrapper around Matplotlib that makes creating common statistical plots easy. The list of supported plots includes univariate and bivariate distribution plots, regression plots, and a number of methods for plotting categorical variables. The full list of plots Seaborn provides is in their API reference.

Creating graphs in Seaborn is as simple as calling the appropriate graphing function. Here is an example of creating a histogram, kernel density estimation, and rug plot for randomly generated data.

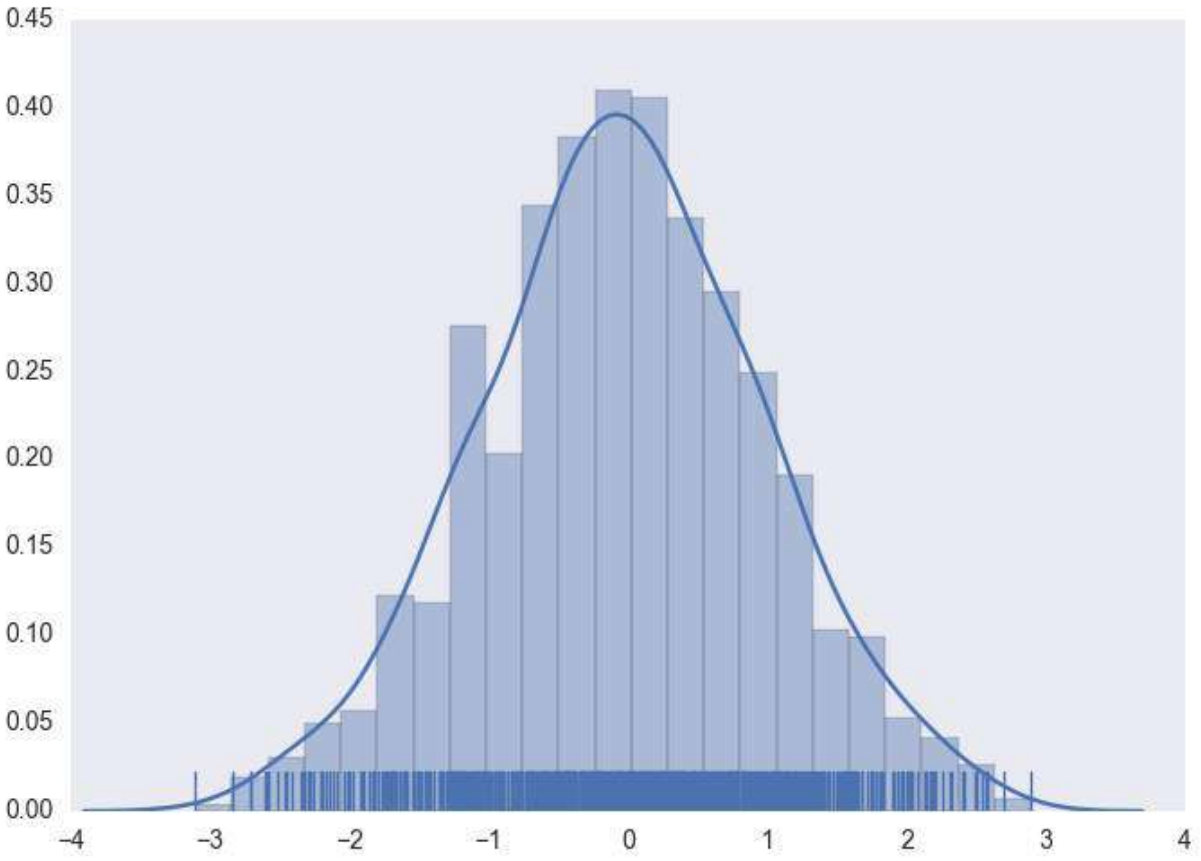
```
import numpy as np  # numpy used to create data from plotting
import seaborn as sns  # common form of importing seaborn

# Generate normally distributed data
data = np.random.randn(1000)

# Plot a histogram with both a rugplot and kde graph superimposed
sns.distplot(data, kde=True, rug=True)
```



The style of the plot can also be controlled using a declarative syntax.

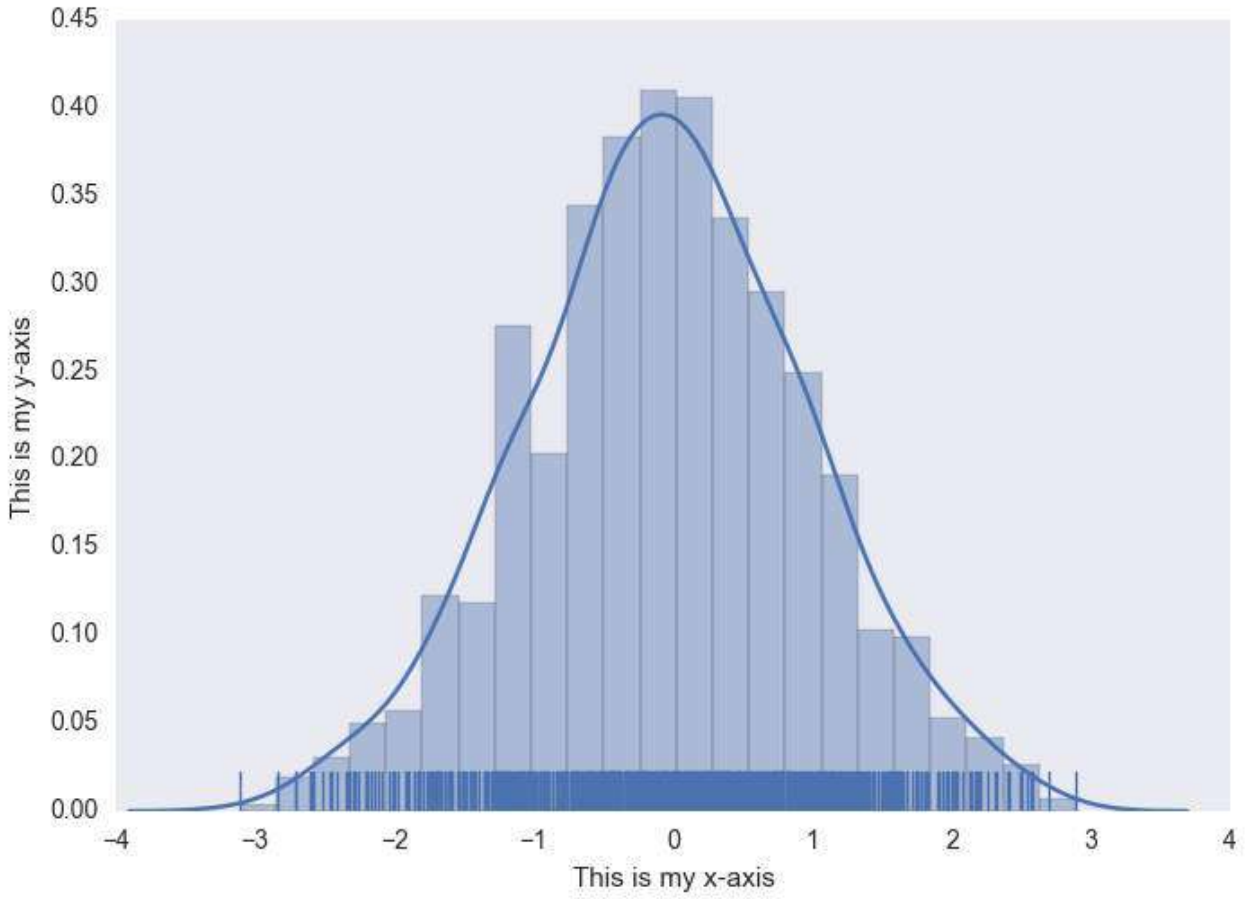
```
# Using previously created imports and data.

# Use a dark background with no grid.
sns.set_style('dark')
```

```
# Create the plot again
sns.distplot(data, kde=True, rug=True)
```



As an added bonus, normal matplotlib commands can still be applied to Seaborn plots. Here's an example of adding axis titles to our previously created histogram.

```
# Using previously created data and style

# Access to matplotlib commands
import matplotlib.pyplot as plt

# Previously created plot.
sns.distplot(data, kde=True, rug=True)
# Set the axis labels.
plt.xlabel('This is my x-axis')
plt.ylabel('This is my y-axis')
```



Section 108.2: Matplotlib

Matplotlib is a mathematical plotting library for Python that provides a variety of different plotting functionality.

The matplotlib documentation can be found here, with the SO Docs being available here.

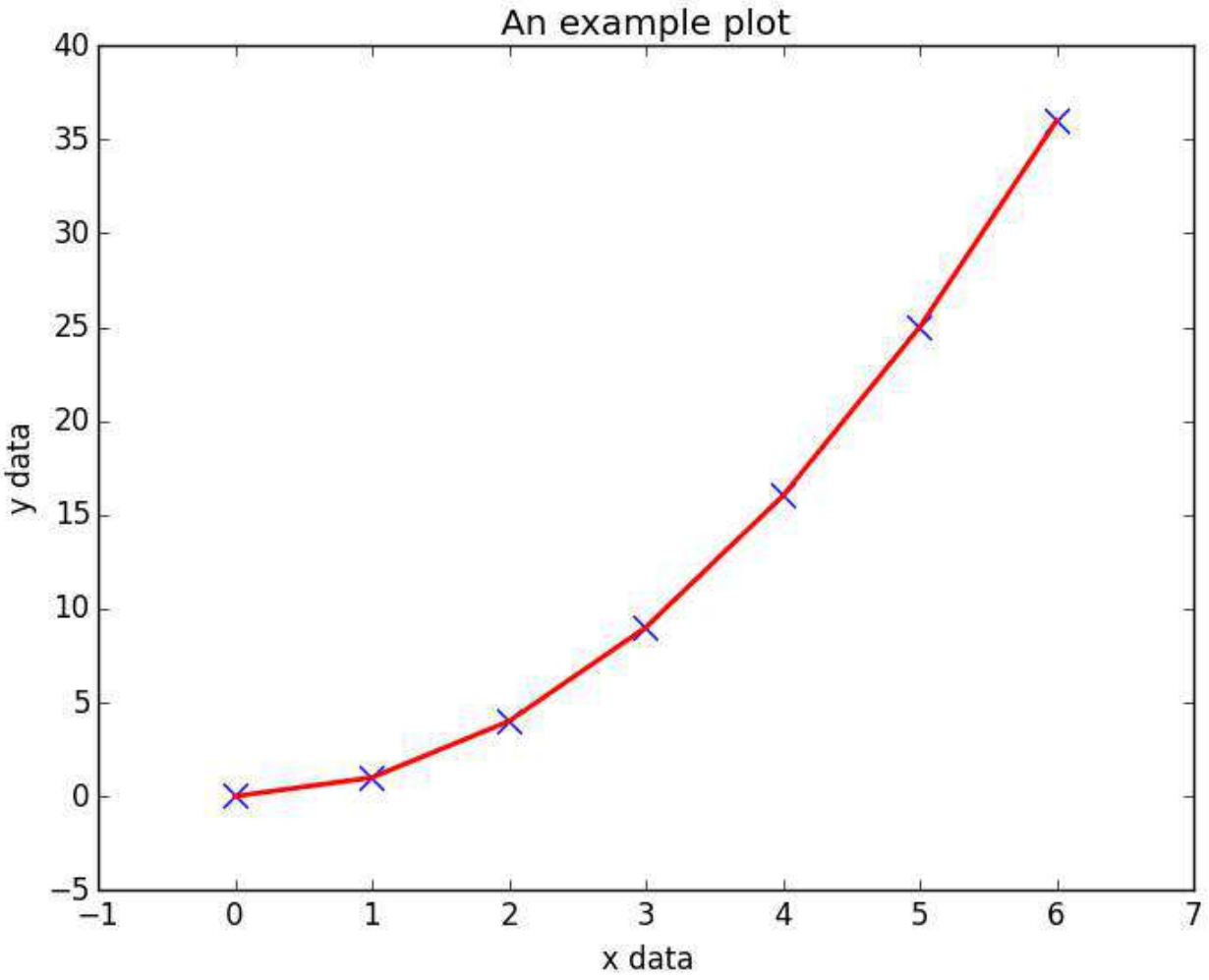
Matplotlib provides two distinct methods for plotting, though they are interchangeable for the most part:

- Firstly, matplotlib provides the `pyplot` interface, direct and simple-to-use interface that allows plotting of complex graphs in a MATLAB-like style.
- Secondly, matplotlib allows the user to control the different aspects (axes, lines, ticks, etc) directly using an object-based system. This is more difficult but allows complete control over the entire plot.

Below is an example of using the `pyplot` interface to plot some generated data:

```
import matplotlib.pyplot as plt

# Generate some data for plotting.
x = [0, 1, 2, 3, 4, 5, 6]
y = [i**2 for i in x]

# Plot the data x, y with some keyword arguments that control the plot style.
# Use two different plot commands to plot both points (scatter) and a line (plot).

plt.scatter(x, y, c='blue', marker='x', s=100) # Create blue markers of shape "x" and size 100
plt.plot(x, y, color='red', linewidth=2) # Create a red line with linewidth 2.

# Add some text to the axes and a title.
plt.xlabel('x data')
```

```
plt.ylabel('y data')
plt.title('An example plot')

# Generate the plot and show to the user.
plt.show()
```



Note that `plt.show()` is known to be problematic in some environments due to running `matplotlib.pyplot` in interactive mode, and if so, the blocking behaviour can be overridden explicitly by passing in an optional argument, `plt.show(block=True)`, to alleviate the issue.

Section 108.3: Plotly

Plotly is a modern platform for plotting and data visualization. Useful for producing a variety of plots, especially for data sciences, **Plotly** is available as a library for **Python**, **R**, **JavaScript**, **Julia** and, **MATLAB**. It can also be used as a web application with these languages.

Users can install plotly library and use it offline after user authentication. The installation of this library and offline authentication is given here. Also, the plots can be made in **Jupyter Notebooks** as well.

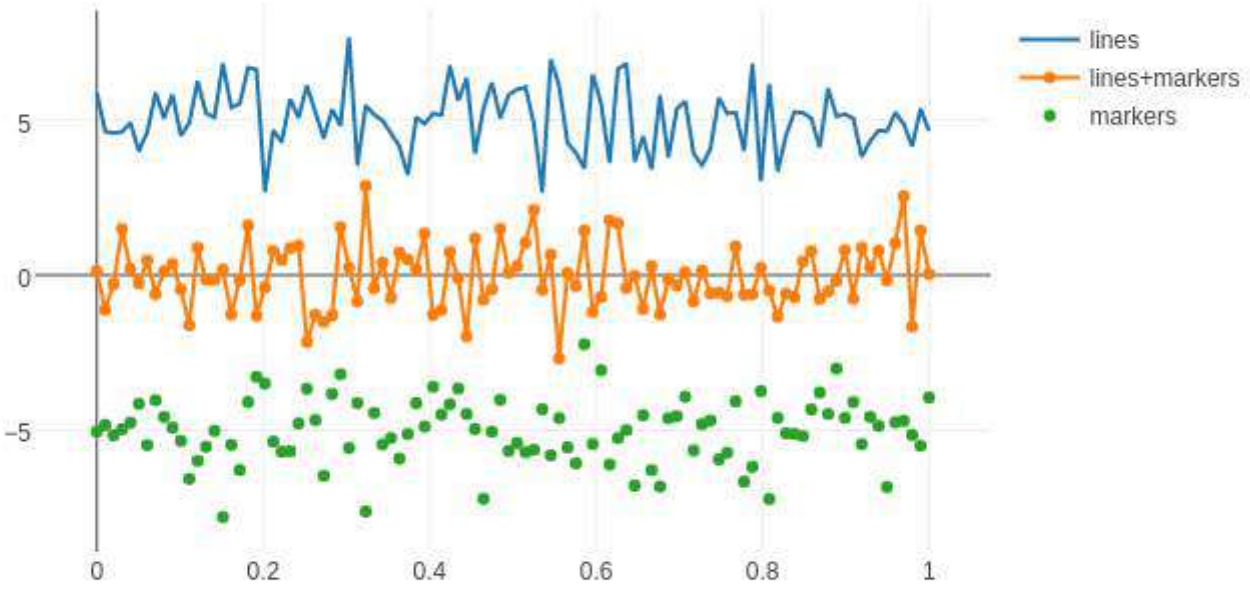
Usage of this library requires an account with username and password. This gives the workspace to save plots and data on the cloud.

The free version of the library has some slightly limited features and designed for making 250 plots per day. The paid version has all the features, unlimited plot downloads and more private data storage. For more details, one can visit the main page here.

For documentation and examples, one can go here

A sample plot from the documentation examples:

```
import plotly.graph_objs as go
import plotly as ply

# Create random data with numpy
import numpy as np

N = 100
random_x = np.linspace(0, 1, N)
random_y0 = np.random.randn(N)+5
random_y1 = np.random.randn(N)
random_y2 = np.random.randn(N)-5

# Create traces
trace0 = go.Scatter(
    x = random_x,
y = random_y0,
mode = 'lines',
name = 'lines'
)
trace1 = go.Scatter(
    x = random_x,
    y = random_y1,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = random_x,
    y = random_y2,
    mode = 'markers',
    name = 'markers'
)
data = [trace0, trace1, trace2]

ply.offline.plot(data, filename='line-mode')
```



Section 108.4: MayaVI

MayaVI is a 3D visualization tool for scientific data. It uses the Visualization Tool Kit or VTK under the hood. Using the power of VTK, **MayaVI** is capable of producing a variety of 3-Dimensional plots and figures. It is available as a separate software application and also as a library. Similar to Matplotlib, this library provides an object oriented programming language interface to create plots without having to know about **VTK**.

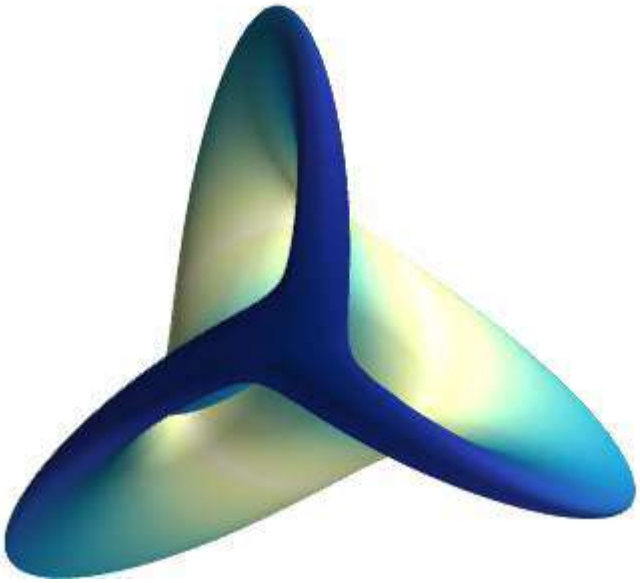
MayaVI is available only in Python 2.7x series! It is hoped to be available in Python 3-x series soon! (Although some success is noticed when using its dependencies in Python 3)

Documentation can be found here. Some gallery examples are found here

Here is a sample plot created using **MayaVI** from the documentation.

```
# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>
# Copyright (c) 2007, Enthought, Inc.
# License: BSD Style.


from numpy import sin, cos, mgrid, pi, sqrt
from mayavi import mlab

mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

X = 2 / 3. * (cos(u) * cos(2 * v)
        + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) -
                                                 sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) -
        sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2)
        - sin(2 * u) * sin(3 * v))
```

```
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)

mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )

# Nice view from the front
mlab.view(.0, - 5.0, 4)
mlab.show()
```



Chapter 109: The Interpreter (Command Line Console)

Section 109.1: Getting general help

If the `help` function is called in the console without any arguments, Python presents an interactive help console, where you can find out about Python modules, symbols, keywords and more.

```
>>> help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
```

Section 109.2: Referring to the last expression

To get the value of the last result from your last expression in the console, use an underscore `_`.

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

This magic underscore value is only updated when using a python expression that results in a value. Defining functions or for loops does not change the value. If the expression raises an exception there will be no changes to `_`.

```
>>> "Hello, {0}".format("World")
'Hello, World'
>>> _
'Hello, World'
>>> def wontchangethings():
...     pass
>>> _
'Hello, World'
>>> 27 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> _
'Hello, World'
```

Remember, this magic variable is only available in the interactive python interpreter. Running scripts will not do this.

Section 109.3: Opening the Python console

The console for the primary version of Python can usually be opened by typing `py` into your windows console or `python` on other platforms.

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you have multiple versions, then by default their executables will be mapped to `python2` or `python3` respectively.

This of course depends on the Python executables being in your PATH.

Section 109.4: The PYTHONSTARTUP variable

You can set an environment variable called PYTHONSTARTUP for Python's console. Whenever you enter the Python console, this file will be executed, allowing for you to add extra functionality to the console such as importing commonly-used modules automatically.

If the PYTHONSTARTUP variable was set to the location of a file containing this:

```
print("Welcome!")
```

Then opening the Python console would result in this extra output:

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Welcome!
>>>
```

Section 109.5: Command line arguments

Python has a variety of command-line switches which can be passed to `py`. These can be found by performing `py --help`, which gives this output on Python 3.4:

```
Python Launcher

usage: py [ launcher-arguments ] [ python-arguments ] script [ script-arguments ]

Launcher arguments:

-2 : Launch the latest Python 2.x version
-3 : Launch the latest Python 3.x version
-X.Y : Launch the specified Python version
-X.Y-32: Launch the specified 32bit Python version

The following help text is from Python:

usage: G:\\Python34\\python.exe [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b : issue warnings about str(bytes_instance), str(bytearray_instance)
and comparing bytes/bytearray with str. (-bb: issue errors)
-B : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
```

```
-d : debug output from parser; also PYTHONDEBUG=x
-E : ignore PYTHON* environment variables (such as PYTHONPATH)
-h : print this help message and exit (also --help)
-i : inspect interactively after running script; forces a prompt even
if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO : remove doc-strings in addition to the -O optimizations
-q : don't print version and copyright messages on interactive startup
-s : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S : don't imply 'import site' on initialization
-u : unbuffered binary stdout and stderr, stdin always buffered;
also PYTHONUNBUFFERED=x
see man page for details on internal buffering relating to '-u'
-v : verbose (trace import statements); also PYTHONVERBOSE=x
can be supplied multiple times to increase verbosity
-V : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
also PYTHONWARNINGS=arg
-x : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option
file : program read from script file
- : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH : ';'-separated list of directories prefixed to the
default module search path. The result is sys.path.
PYTHONHOME : alternate  directory (or ;).
The default module search path uses \\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
to seed the hashes of str, bytes and datetime objects. It can also be
set to an integer in the range [0,4294967295] to get hash values with a
predictable seed.
```

Section 109.6: Getting help about an object

The Python console adds a new function, `help`, which can be used to get information about a function or object.

For a function, `help` prints its signature (arguments) and its docstring, if the function has one.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

For an object, `help` lists the object's docstring and the different member functions which the object has.

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value...
```

Chapter 110: *args and **kwargs

Section 110.1: Using **kwargs when writing functions

You can define a function that takes an arbitrary number of keyword (named) arguments by using the double star `**` before a parameter name:

```
def print_kwargs(**kwargs):
    print(kwargs)
```

When calling the method, Python will construct a dictionary of all keyword arguments and make it available in the function body:

```
print_kwargs(a="two", b=3)
# prints: "{a: "two", b=3}"
```

Note that the **kwargs parameter in the function definition must always be the last parameter, and it will only match the arguments that were passed in after the previous ones.

```
def example(a, **kw):
    print kw

example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

Inside the function body, `kwargs` is manipulated in the same way as a dictionary; in order to access individual elements in `kwargs` you just loop through them as you would with a normal dictionary:

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

Now, calling `print_kwargs(a="two", b=1)` shows the following output:

```
print_kwargs(a = "two", b = 1)
key = a, value = "two"
key = b, value = 1
```

Section 110.2: Using *args when writing functions

You can use the star * when writing a function to collect all positional (ie. unnamed) arguments in a tuple:

```
def print_args(farg, *args):
   print("formal arg: %s" % farg)
   for arg in args:
       print("another positional arg: %s" % arg)
```

Calling method:

```
print_args(1, "two", 3)
```

In that call, farg will be assigned as always, and the two others will be fed into the args tuple, in the order they were received.

Section 110.3: Populating kwarg values with a dictionary

```
def foobar(foo=None, bar=None):
    return "{}{}".format(foo, bar)

values = {"foo": "foo", "bar": "bar"}

foobar(**values) # "foobar"
```

Section 110.4: Keyword-only and Keyword-required arguments

Python 3 allows you to define function arguments which can only be assigned by keyword, even without default values. This is done by using star * to consume additional positional parameters without setting the keyword parameters. All arguments after the * are keyword-only (i.e. non-positional) arguments. Note that if keyword-only arguments aren't given a default, they are still required when calling the function.

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("first positional arg: {}".format(arg1))
    for arg in args:
        print("another positional arg: {}".format(arg))
    print("keyword_required value: {}".format(keyword_required))
    print("keyword_only value: {}".format(keyword_only))

print(1, 2, 3, 4) # TypeError: print_args() missing 1 required keyword-only argument:
'keyword_required'
print(1, 2, 3, keyword_required=4)
# first positional arg: 1
# another positional arg: 2
# another positional arg: 3
# keyword_required value: 4
# keyword_only value: True
```

Section 110.5: Using **kwargs when calling functions

You can use a dictionary to assign values to the function's parameters; using parameters name as keys in the dictionary and the value of these arguments bound to each key:

```
def test_func(arg1, arg2, arg3): # Usual function with three arguments
   print("arg1: %s" % arg1)
   print("arg2: %s" % arg2)
   print("arg3: %s" % arg3)

# Note that dictionaries are unordered, so we can switch arg2 and arg3. Only the names matter.
kwargs = {"arg3": 3, "arg2": "two"}

# Bind the first argument (ie. arg1) to 1, and use the kwargs dictionary to bind the others
test_var_args_call(1, **kwargs)
```

Section 110.6: **kwargs and default values

To use default values with **kwargs

```
def fun(**kwargs):
    print kwargs.get('value', 0)
```

```
fun()
# print 0
fun(value=1)
# print 1
```

Section 110.7: Using *args when calling functions

The effect of using the `*` operator on an argument when calling a function is that of unpacking the list or a tuple argument

```
def print_args(arg1, arg2):
    print(str(arg1) + str(arg2))

a = [1,2]
b = tuple([3,4])

print_args(*a)
# 12
print_args(*b)
# 34
```

Note that the length of the starred argument need to be equal to the number of the function's arguments.

A common python idiom is to use the unpacking operator `*` with the `zip` function to reverse its effects:

```
a = [1,3,5,7,9]
b = [2,4,6,8,10]

zipped = zip(a,b)
# [(1,2), (3,4), (5,6), (7,8), (9,10)]

zip(*zipped)
# (1,3,5,7,9), (2,4,6,8,10)
```

Chapter 111: Garbage Collection

Section 111.1: Reuse of primitive objects

An interesting thing to note which may help optimize your applications is that primitives are actually also refcounted under the hood. Let's take a look at numbers; for all integers between -5 and 256, Python always reuses the same object:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

Note that the refcount increases, meaning that `a` and `b` reference the same underlying object when they refer to the `1` primitive. However, for larger numbers, Python actually doesn't reuse the underlying object:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Because the refcount for `999999999` does not change when assigning it to `a` and `b` we can infer that they refer to two different underlying objects, even though they both are assigned the same primitive.

Section 111.2: Effects of the del command

Removing a variable name from the scope using `del v`, or removing an object from a collection using `del v[item]` or `del[i:j]`, or removing an attribute using `del v.name`, or any other way of removing references to an object, *does not* trigger any destructor calls or any memory being freed in and of itself. Objects are only destructed when their reference count reaches zero.

```
>>> import gc
>>> gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def bar():
    return Track()
>>> t = bar()
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> del t          # not destructed yet - another_t still refers to it
>>> del another_t  # final reference gone, object is destructed
Destructed
```

Section 111.3: Reference Counting

The vast majority of Python memory management is handled with reference counting.

Every time an object is referenced (e.g. assigned to a variable), its reference count is automatically increased. When it is dereferenced (e.g. variable goes out of scope), its reference count is automatically decreased.

When the reference count reaches zero, the object is **immediately destroyed** and the memory is immediately freed. Thus for the majority of cases, the garbage collector is not even needed.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def foo():
        Track()
        # destructed immediately since no longer has any references
        print("---")
        t = Track()
        # variable is referenced, so it's not destructed yet
        print("---")
        # variable is destructed when function exits
>>> foo()
Initialized
Destructed
---
Initialized
---
Destructed
```

To demonstrate further the concept of references:

```
>>> def bar():
        return Track()
>>> t = bar()
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> t = None          # not destructed yet - another_t still refers to it
>>> another_t = None  # final reference gone, object is destructed
Destructed
```

Section 111.4: Garbage Collector for Reference Cycles

The only time the garbage collector is needed is if you have a *reference cycle*. The simples example of a reference cycle is one in which A refers to B and B refers to A, while nothing else refers to either A or B. Neither A or B are accessible from anywhere in the program, so they can safely be destructed, yet their reference counts are 1 and so they cannot be freed by the reference counting algorithm alone.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
```

```
>>> A = Track()
Initialized
>>> B = Track()
Initialized
>>> A.other = B
>>> B.other = A
>>> del A; del B  # objects are not destructed due to reference cycle
>>> gc.collect()  # trigger collection
Destructed
Destructed
4
```

A reference cycle can be arbitrary long. If A points to B points to C points to ... points to Z which points to A, then neither A through Z will be collected, until the garbage collection phase:

```
>>> objs = [Track() for _ in range(10)]
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
>>> for i in range(len(objs)-1):
...     objs[i].other = objs[i + 1]
...
>>> objs[-1].other = objs[0]  # complete the cycle
>>> del objs                  # no one can refer to objs now - still not destructed
>>> gc.collect()
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
20
```

Section 111.5: Forcefully deallocating objects

You can force deallocate objects even if their refcount isn't 0 in both Python 2 and 3.

Both versions use the `ctypes` module to do so.

WARNING: doing this *will* leave your Python environment unstable and prone to crashing without a traceback! Using this method could also introduce security problems (quite unlikely) Only deallocate objects you're sure you'll never reference again. Ever.

Python 3.x `Version ≥ 3.0`

```
import ctypes
deallocated = 12345
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))
```

Python 2.x `Version ≥ 2.3`

```
import ctypes, sys
deallocated = 12345
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[:4] = '\x00' * 4
```

After running, any reference to the now deallocated object will cause Python to either produce undefined behavior or crash - without a traceback. There was probably a reason why the garbage collector didn't remove that object...

If you deallocate `None`, you get a special message - `Fatal Python error: deallocating None` before crashing.

Section 111.6: Viewing the refcount of an object

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

Section 111.7: Do not wait for the garbage collection to clean up

The fact that the garbage collection will clean up does not mean that you should wait for the garbage collection cycle to clean up.

In particular you should not wait for garbage collection to close file handles, database connections and open network connections.

for example:

In the following code, you assume that the file will be closed on the next garbage collection cycle, if f was the last reference to the file.

```
>>> f = open("test.txt")
>>> del f
```

A more explicit way to clean up is to call `f.close()`. You can do it even more elegant, that is by using the `with` statement, also known as the context manager :

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exists, but it is closed
```

The `with` statement allows you to indent your code under the open file. This makes it explicit and easier to see how long a file is kept open. It also always closes a file, even if an exception is raised in the `while` block.

Section 111.8: Managing garbage collection

There are two approaches for influencing when a memory cleanup is performed. They are influencing how often the automatic process is performed and the other is manually triggering a cleanup.

The garbage collector can be manipulated by tuning the collection thresholds which affect the frequency at which the collector runs. Python uses a generation based memory management system. New objects are saved in the newest generation - **generation0** and with each survived collection, objects are promoted to older generations. After reaching the last generation - **generation2**, they are no longer promoted.

The thresholds can be changed using the following snippet:

```
import gc
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

The first argument represents the threshold for collecting **generation0**. Every time the number of **allocations** exceeds the number of **deallocations** by 1000 the garbage collector will be called.

The older generations are not cleaned at each run to optimize the process. The second and third arguments are **optional** and control how frequently the older generations are cleaned. If **generation0** was processed 100 times without cleaning **generation1**, then **generation1** will be processed. Similarly, objects in **generation2** will be processed only when the ones in **generation1** were cleaned 10 times without touching **generation2**.

One instance in which manually setting the thresholds is beneficial is when the program allocates a lot of small objects without deallocating them which leads to the garbage collector running too often (each **generation0_threshold** object allocations). Even though, the collector is pretty fast, when it runs on huge numbers of objects it poses a performance issue. Anyway, there's no one size fits all strategy for choosing the thresholds and it's use case dependable.

Manually triggering a collection can be done as in the following snippet:

```
import gc
gc.collect()
```

The garbage collection is automatically triggered based on the number of allocations and deallocations, not on the consumed or available memory. Consequently, when working with big objects, the memory might get depleted before the automated cleanup is triggered. This makes a good use case for manually calling the garbage collector.

Even though it's possible, it's not an encouraged practice. Avoiding memory leaks is the best option. Anyway, in big projects detecting the memory leak can be a tough task and manually triggering a garbage collection can be used as a quick solution until further debugging.

For long-running programs, the garbage collection can be triggered on a time basis or on an event basis. An example for the first one is a web server that triggers a collection after a fixed number of requests. For the later, a web server that triggers a garbage collection when a certain type of request is received.

Chapter 112: Pickle data serialisation

Parameter	Details
object	The object which is to be stored
file	The open file which will contain the object
protocol	The protocol used for pickling the object (optional parameter)
buffer	A bytes object that contains a serialized object

Section 112.1: Using Pickle to serialize and deserialize an object

The `pickle` module implements an algorithm for turning an arbitrary Python object into a series of bytes. This process is also called **serializing** the object. The byte stream representing the object can then be transmitted or stored, and later reconstructed to create a new object with the same characteristics.

For the simplest code, we use the `dump()` and `load()` functions.

To serialize the object

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

To deserialize the object

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Using pickle and byte objects

It is also possible to serialize into and deserialize out of byte objects, using the `dumps` and `loads` function, which are equivalent to `dump` and `load`.

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

Section 112.2: Customize Pickled Data

Some data cannot be pickled. Other data should not be pickled for other reasons.

What will be pickled can be defined in `__getstate__` method. This method must return something that is picklable.

On the opposite side is `__setstate__`: it will receive what `__getstate__` created and has to initialize the object.

```
class A(object):
    def __init__(self, important_data):
        self.important_data = important_data

        # Add data which cannot be pickled:
        self.func = lambda: 7

        # Add data which should never be pickled, because it expires quickly:
        self.is_up_to_date = False

    def __getstate__(self):
        return [self.important_data] # only this is needed

    def __setstate__(self, state):
        self.important_data = state[0]

        self.func = lambda: 7  # just some hard-coded unpicklable function

        self.is_up_to_date = False  # even if it was before pickling
```

Now, this can be done:

```
>>> a1 = A('very important')
>>>
>>> s = pickle.dumps(a1)  # calls a1.__getstate__()
>>>
>>> a2 = pickle.loads(s)  # calls a1.__setstate__(['very important'])
>>> a2
<__main__.A object at 0x0000000002742470>
>>> a2.important_data
'very important'
>>> a2.func()
7
```

The implementation here pikles a list with one value: `[self.important_data]`. That was just an example, `__getstate__` could have returned anything that is picklable, as long as `__setstate__` knows how to do the opposite. A good alternative is a dictionary of all values: `{'important_data': self.important_data}`.

Constructor is not called! Note that in the previous example instance `a2` was created in `pickle.loads` without ever calling `A.__init__`, so `A.__setstate__` had to initialize everything that `__init__` would have initialized if it were called.

Chapter 113: Binary Data

Section 113.1: Format a list of values into a byte object

```
from struct import pack

print(pack('I3c', 123, b'a', b'b', b'c'))  # b'{\x00\x00\x00abc'
```

Section 113.2: Unpack a byte object according to a format string

```
from struct import unpack

print(unpack('I3c', b'{\x00\x00\x00abc'))  # (123, b'a', b'b', b'c')
```

Section 113.3: Packing a structure

The module "**struct**" provides facility to pack python objects as contiguous chunk of bytes or dissemble a chunk of bytes to python structures.

The pack function takes a format string and one or more arguments, and returns a binary string. This looks very much like you are formatting a string except that the output is not a string but a chunk of bytes.

```
import struct
import sys
print "Native byteorder: ", sys.byteorder
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("ihb", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
print "Byte chunk unpacked: ", struct.unpack("ihb", buffer)
# Last element as unsigned short instead of unsigned char ( 2 Bytes)
buffer = struct.pack("ihh", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
```

Output:

> Native byteorder: little Byte chunk: '\x03\x00\x00\x00\x04\x00\x05' Byte chunk unpacked: (3, 4, 5) Byte chunk: '\x03\x00\x00\x00\x04\x00\x05\x00'

You could use network byte order with data received from network or pack data to send it to network.

```
import struct
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("hhh", 3, 4, 5)
print "Byte chunk native byte order: ", repr(buffer)
buffer = struct.pack("!hhh", 3, 4, 5)
print "Byte chunk network byte order: ", repr(buffer)
```

Output:

> Byte chunk native byte order: '\x03\x00\x04\x00\x05\x00'

> Byte chunk network byte order: '\x00\x03\x00\x04\x00\x05'

You can optimize by avoiding the overhead of allocating a new buffer by providing a buffer that was created earlier.

```
import struct
from ctypes import create_string_buffer
bufferVar = create_string_buffer(8)
bufferVar2 = create_string_buffer(8)
# We use a buffer that has already been created
# provide format, buffer, offset and data
struct.pack_into("hhh", bufferVar, 0, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar.raw)
struct.pack_into("hhh", bufferVar2, 2, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar2.raw)
```

Output:

> Byte chunk: '\x03\x00\x04\x00\x05\x00\x00\x00'
>
> Byte chunk: '\x00\x00\x03\x00\x04\x00\x05\x00'

Chapter 114: Idioms

Section 114.1: Dictionary key initializations

Prefer `dict.get` method if you are not sure if the key is present. It allows you to return a default value if key is not found. The traditional method `dict[key]` would raise a `KeyError` exception.

Rather than doing

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

Do

```
def add_student():
        students['count'] = students.get('count', 0) + 1
```

Section 114.2: Switching variables

To switch the value of two variables you can use tuple unpacking.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

Section 114.3: Use truth value testing

Python will implicitly convert any object to a Boolean value for testing, so use it wherever possible.

```
# Good examples, using implicit truth testing
if attr:
    # do something

if not attr:
    # do something

# Bad examples, using specific types
if attr == 1:
    # do something

if attr == True:
    # do something

if attr != '':
    # do something

# If you are looking to specifically check for None, use 'is' or 'is not'
if attr is None:
    # do something
```

This generally produces more readable code, and is usually much safer when dealing with unexpected types.

Click here for a list of what will be evaluated to `False`.

Section 114.4: Test for "__main__" to avoid unexpected code execution

It is good practice to test the calling program's `__name__` variable before executing your code.

```
import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Using this pattern ensures that your code is only executed when you expect it to be; for example, when you run your file explicitly:

```
python my_program.py
```

The benefit, however, comes if you decide to `import` your file in another program (for example if you are writing it as part of a library). You can then `import` your file, and the `__main__` trap will ensure that no code is executed unexpectedly:

```
# A new program file
import my_program        # main() is not run

# But you can run main() explicitly if you really want it to run:
my_program.main()
```

Chapter 115: Data Serialization

Parameter	Details
`protocol`	Using `pickle` or `cPickle`, it is the method that objects are being Serialized/Unserialized. You probably want to use `pickle.HIGHEST_PROTOCOL` here, which means the newest method.

Section 115.1: Serialization using JSON

JSON is a cross language, widely used method to serialize data

Supported data types : *int*, *float*, *boolean*, *string*, *list* and *dict*. See -> JSON Wiki for more

Here is an example demonstrating the **basic** usage of **JSON**:

```
import json

families = (['John'], ['Mark', 'David', {'name': 'Avraham'}])

# Dumping it into string
json_families = json.dumps(families)
# [["John"], ["Mark", "David", {"name": "Avraham"}]]

# Dumping it to file
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# Loading it from string
json_families = json.loads(json_families)

# Loading it from file
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```

See JSON-Module for detailed information about JSON.

Section 115.2: Serialization using Pickle

Here is an example demonstrating the **basic** usage of **pickle**:

```
# Importing pickle
try:
    import cPickle as pickle  # Python 2
except ImportError:
    import pickle  # Python 3

# Creating Pythonic object:
class Family(object):
    def __init__(self, names):
        self.sons = names

    def __str__(self):
        return ' '.join(self.sons)

my_family = Family(['John', 'David'])

# Dumping to string
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)
```

```
# Dumping to file
with open('family.p', 'w') as pickle_file:
    pickle.dump(families, pickle_file, pickle.HIGHEST_PROTOCOL)

# Loading from string
my_family = pickle.loads(pickle_data)

# Loading from file
with open('family.p', 'r') as pickle_file:
    my_family = pickle.load(pickle_file)
```

See Pickle for detailed information about Pickle.

WARNING: The official documentation for pickle makes it clear that there are no security guarantees. Don't load any data you don't trust its origin.

Chapter 116: Multiprocessing

Section 116.1: Running Two Simple Processes

A simple example of using multiple processes would be two processes (workers) that are executed separately. In the following example, two processes are started:

- `countUp()` counts 1 up, every second.
- `countDown()` counts 1 down, every second.

```
import multiprocessing
import time
from random import randint

def countUp():
    i = 0
    while i <= 3:
        print('Up:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i += 1

def countDown():
    i = 3
    while i >= 0:
        print('Down:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i -= 1

if __name__ == '__main__':
    # Initiate the workers.
    workerUp = multiprocessing.Process(target=countUp)
    workerDown = multiprocessing.Process(target=countDown)

    # Start the workers.
    workerUp.start()
    workerDown.start()

    # Join the workers. This will block in the main (parent) process
    # until the workers are complete.
    workerUp.join()
    workerDown.join()
```

The output is as follows:

```
Up: 0
Down: 3
Up: 1
Up: 2
Down: 2
Up: 3
Down: 1
Down: 0
```

Section 116.2: Using Pool and Map

```
from multiprocessing import Pool
```

```
def cube(x):
    return x ** 3

if __name__ == "__main__":
    pool = Pool(5)
    result = pool.map(cube, [0, 1, 2, 3])
```

`Pool` is a class which manages multiple `Workers` (processes) behind the scenes and lets you, the programmer, use.

`Pool(5)` creates a new Pool with 5 processes, and `pool.map` works just like map but it uses multiple processes (the amount defined when creating the pool).

Similar results can be achieved using `map_async`, `apply` and `apply_async` which can be found in the documentation.

Chapter 117: Multithreading

Threads allow Python programs to handle multiple functions at once as opposed to running a sequence of commands individually. This topic explains the principles behind threading and demonstrates its usage.

Section 117.1: Basics of multithreading

Using the `threading` module, a new thread of execution may be started by creating a new `threading.Thread` and assigning it a function to execute:

```
import threading

def foo():
  print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

The `target` parameter references the function (or callable object) to be run. The thread will not begin execution until `start` is called on the `Thread` object.

Starting a Thread

```
my_thread.start() # prints 'Hello threading!'
```

Now that `my_thread` has run and terminated, calling `start` again will produce a `RuntimeError`. If you'd like to run your thread as a daemon, passing the `daemon=True` kwarg, or setting `my_thread.daemon` to `True` before calling `start()`, causes your `Thread` to run silently in the background as a daemon.

Joining a Thread

In cases where you split up one big job into several small ones and want to run them concurrently, but need to wait for all of them to finish before continuing, `Thread.join()` is the method you're looking for.

For example, let's say you want to download several pages of a website and compile them into a single page. You'd do this:

```
import requests
from threading import Thread
from queue import Queue

q = Queue(maxsize=20)
def put_page_to_q(page_num):
    q.put(requests.get('http://some-website.com/page_%s.html' % page_num)

def compile(q):
    # magic function that needs all pages before being able to be executed
    if not q.full():
        raise ValueError
    else:
        print("Done compiling!")

threads = []
for page_num in range(20):
     t = Thread(target=requests.get, args=(page_num,))
     t.start()
     threads.append(t)
```

```
# Next, join all threads to make sure all threads are done running before
# we continue. join() is a blocking call (unless specified otherwise using
# the kwarg blocking=False when calling join)
for t in threads:
    t.join()

# Call compile() now, since all threads have completed
compile(q)
```

A closer look at how `join()` works can be found here.

Create a Custom Thread Class

Using `threading.Thread` class we can subclass new custom Thread class. we must override `run` method in a subclass.

```
from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello form Thread")

if __name__ == "__main__":
    t = Sleepy()
    t.start()      # start method automatic call Thread class run method.
    # print 'The main program continues to run in foreground.'
    t.join()
    print("The main program continues to run in the foreground.")
```

Section 117.2: Communicating between threads

There are multiple threads in your code and you need to safely communicate between them.

You can use a `Queue` from the `queue` library.

```
from queue import Queue
from threading import Thread

# create a data producer
def producer(output_queue):
    while True:
        data = data_computation()

        output_queue.put(data)

# create a consumer
def consumer(input_queue):
    while True:
        # retrieve data (blocking)
        data = input_queue.get()

        # do something with the data

        # indicate data has been consumed
        input_queue.task_done()
```

Creating producer and consumer threads with a shared queue

```
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Section 117.3: Creating a worker pool

Using `threading` & `queue`:

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

Using `concurrent.futures.Threadpoolexecutor`:

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

Python Cookbook, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.

Section 117.4: Advanced use of multithreads

This section will contain some of the most advanced examples realized using Multithreading.

Advanced printer (logger)

A thread that prints everything is received and modifies the output according to the terminal width. The nice part is that also the "already written" output is modified when the width of the terminal changes.

```
#!/usr/bin/env python2

import threading
import Queue
import time
import sys
import subprocess
from backports.shutil_get_terminal_size import get_terminal_size

printq = Queue.Queue()
interrupt = False
lines = []

def main():

    ptt = threading.Thread(target=printer) # Turn the printer on
    ptt.daemon = True
    ptt.start()

    # Stupid example of stuff to print
    for i in xrange(1,100):
        printq.put(' '.join([str(x) for x in range(1,i)]))           # The actual way to send stuff
to the printer
        time.sleep(.5)

def split_line(line, cols):
    if len(line) > cols:
        new_line = ''
        ww = line.split()
        i = 0
        while len(new_line) <= (cols - len(ww[i]) - 1):
            new_line += ww[i] + ' '
            i += 1
            print len(new_line)
        if new_line == '':
            return (line, '')

        return (new_line, ' '.join(ww[i:]))
    else:
        return (line, '')


def printer():

    while True:
        cols, rows = get_terminal_size() # Get the terminal dimensions
        msg = '#' + '-' * (cols - 2) + '#\n' # Create the
        try:
            new_line = str(printq.get_nowait())
            if new_line != '!@#EXIT#@!': # A nice way to turn the printer
                                         # thread out gracefully
                lines.append(new_line)
                printq.task_done()
            else:
                printq.task_done()
                sys.exit()
```

```
        except Queue.Empty:
            pass

        # Build the new message to show and split too long lines
        for line in lines:
            res = line          # The following is to split lines which are
                                # longer than cols.
            while len(res) !=0:
                toprint, res = split_line(res, cols)
                msg += '\n' + toprint

        # Clear the shell and print the new output
        subprocess.check_call('clear') # Keep the shell clean
        sys.stdout.write(msg)
        sys.stdout.flush()
        time.sleep(.5)
```

Section 117.5: Stoppable Thread with a while Loop

```
import threading
import time

class StoppableThread(threading.Thread):
    """Thread class with a stop() method. The thread itself has to check
    regularly for the stopped() condition."""

    def __init__(self):
        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()

    def stop(self):
        self._stop_event.set()

    def join(self, *args, **kwargs):
        self.stop()
        super(StoppableThread,self).join(*args, **kwargs)

    def run()
        while not self._stop_event.is_set():
            print("Still running!")
            time.sleep(2)
        print("stopped!"
```

Based on this Question.

Chapter 118: Processes and Threads

Most programs are executed line by line, only running a single process at a time. Threads allow multiple processes to flow independent of each other. Threading with multiple processors permits programs to run multiple processes simultaneously. This topic documents the implementation and usage of threads in Python.

Section 118.1: Global Interpreter Lock

Python multithreading performance can often suffer due to the **Global Interpreter Lock**. In short, even though you can have multiple threads in a Python program, only one bytecode instruction can execute in parallel at any one time, regardless of the number of CPUs.

As such, multithreading in cases where operations are blocked by external events - like network access - can be quite effective:

```
import threading
import time


def process():
    time.sleep(2)


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.00s
# Out: Four runs took 2.00s
```

Note that even though each `process` took 2 seconds to execute, the four processes together were able to effectively run in parallel, taking 2 seconds total.

However, multithreading in cases where intensive computations are being done in Python code - such as a lot of computation - does not result in much improvement, and can even be slower than running in parallel:

```
import threading
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
```

```
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.05s
# Out: Four runs took 14.42s
```

In the latter case, multiprocessing can be effective as multiple processes can, of course, execute multiple instructions simultaneously:

```
import multiprocessing
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.07s
# Out: Four runs took 2.30s
```

Section 118.2: Running in Multiple Threads

Use `threading.Thread` to run a function in another thread.

```
import threading
import os

def process():
    print("Pid is %s, thread id is %s" % (os.getpid(), threading.current_thread().name))

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

# Out: Pid is 11240, thread id is Thread-1
# Out: Pid is 11240, thread id is Thread-2
# Out: Pid is 11240, thread id is Thread-3
# Out: Pid is 11240, thread id is Thread-4
```

Section 118.3: Running in Multiple Processes

Use `multiprocessing.Process` to run a function in another process. The interface is similar to `threading.Thread`:

```
import multiprocessing
import os

def process():
    print("Pid is %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# Out: Pid is 11206
# Out: Pid is 11207
# Out: Pid is 11208
# Out: Pid is 11209
```

Section 118.4: Sharing State Between Threads

As all threads are running in the same process, all threads have access to the same data.

However, concurrent access to shared data should be protected with a lock to avoid synchronization issues.

```
import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj has %d values" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj now has %d values" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
for t in ts:
    t.start()
for t in ts:
```

```
    t.join()
print("Obj final result:")
import pprint; pprint.pprint(obj)

# Out: Obj has 0 values
# Out:  Obj has 0 values
# Out: Obj now has 1 values
# Out: Obj now has 2 valuesObj has 2 values
# Out: Obj now has 3 values
# Out:
# Out:  Obj has 3 values
# Out: Obj now has 4 values
# Out: Obj final result:
# Out: {'0': 0, '1': 1, '2': 2, '3': 3}
```

Section 118.5: Sharing State Between Processes

Code running in different processes do not, by default, share the same data. However, the `multiprocessing` module contains primitives to help share values across multiple processes.

```
import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # ordinary variable modifications are not visible across processes
        plain_num += 1
        # multiprocessing.Value modifications are
        shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# Out: plain_num is 0, shared_num is 4
```

Chapter 119: Python concurrency

Section 119.1: The multiprocessing module

```
from __future__ import print_function
import multiprocessing


def countdown(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=countdown, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=countdown, args=(20,))
    p2.start()

    p1.join()
    p2.join()
```

Here, each function is executed in a new process. Since a new instance of Python VM is running the code, there is no `GIL` and you get parallelism running on multiple cores.

The `Process.start` method launches this new process and run the function passed in the `target` argument with the arguments `args`. The `Process.join` method waits for the end of the execution of processes `p1` and `p2`.

The new processes are launched differently depending on the version of python and the platform on which the code is running *e.g.*:

- Windows uses `spawn` to create the new process.
- With unix systems and version earlier than 3.3, the processes are created using a `fork`.
  Note that this method does not respect the POSIX usage of fork and thus leads to unexpected behaviors, especially when interacting with other multiprocessing libraries.
- With unix system and version 3.4+, you can choose to start the new processes with either `fork`, `forkserver` or `spawn` using `multiprocessing.set_start_method` at the beginning of your program. `forkserver` and `spawn` methods are slower than forking but avoid some unexpected behaviors.

POSIX fork usage:

> After a fork in a multithreaded program, the child can safely call only async-signal-safe functions until such time as it calls execve.
> (see)

Using fork, a new process will be launched with the exact same state for all the current mutex but only the `MainThread` will be launched. This is unsafe as it could lead to race conditions *e.g.*:

- If you use a `Lock` in `MainThread` and pass it to another thread which is supposed to lock it at some point. If the `fork` occurs simultaneously, the new process will start with a locked lock which will never be released as the second thread does not exist in this new process.

Actually, this kind of behavior should not occurred in pure python as `multiprocessing` handles it properly but if you are interacting with other library, this kind of behavior can occurs, leading to crash of your system (for instance with numpy/accelerated on macOS).

Section 119.2: The threading module

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
t2 = threading.Thread(target=countdown,args=(20,))
t2.start()
```

In certain implementations of Python such as CPython, true parallelism is not achieved using threads because of using what is known as the GIL, or **G**lobal **I**nterpreter **L**ock.

Here is an excellent overview of Python concurrency:

Python concurrency by David Beazley (YouTube)

Section 119.3: Passing data between multiprocessing processes

Because data is sensitive when dealt with between two threads (think concurrent read and concurrent write can conflict with one another, causing race conditions), a set of unique objects were made in order to facilitate the passing of data back and forth between threads. Any truly atomic operation can be used between threads, but it is always safe to stick with Queue.

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
#Creates a queue with an undefined maximum size
#this can be dangerous as the queue becomes increasingly large
#it will take a long time to copy data to/from each read/write thread
```

Most people will suggest that when using queue, to always place the queue data in a try: except: block instead of using empty. However, for applications where it does not matter if you skip a scan cycle (data can be placed in the queue while it is flipping states from `queue.Empty==True` to `queue.Empty==False`) it is usually better to place read and write access in what I call an Iftry block, because an 'if' statement is technically more performant than catching the exception.

```
import multiprocessing
import queue
'''Import necessary Python standard libraries, multiprocessing for classes and queue for the queue
exceptions it provides'''
def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    '''This global method for the Iftry block is provided for its reuse and
standard functionality, the if also saves on performance as opposed to catching
 the exception, which is expensive.
       It also allows the user to specify a function for the outgoing data to use,
```

```
 and a default value to return if the function cannot return the value from the queue'''
        if get_queue.empty():
            if use_default:
                return default
        else:
            try:
                value = get_queue.get_nowait()
            except queue.Empty:
                if use_default:
                    return default
            else:
                if use_func:
                    return func(value)
                else:
                    return value
    def Queue_Iftry_Put(put_queue, value):
        '''This global method for the Iftry block is provided because of its reuse
and
standard functionality, the If also saves on performance as opposed to catching
 the exception, which is expensive.
        Return True if placing value in the queue was successful. Otherwise, false'''
        if put_queue.full():
            return False
        else:
            try:
                put_queue.put_nowait(value)
            except queue.Full:
                return False
            else:
                return True
```

Chapter 120: Parallel computation

Section 120.1: Using the multiprocessing module to parallelise tasks

```
import multiprocessing

def fib(n):
    """computing the Fibonacci in an inefficient way
    was chosen to slow down the CPU."""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))

# Out: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]
```

As the execution of each call to `fib` happens in parallel, the time of execution of the full example is **1.8× faster** than if done in a sequential way on a dual processor.

Python 2.2+

Section 120.2: Using a C-extension to parallelize tasks

The idea here is to move the computationally intensive jobs to C (using special macros), independent of Python, and have the C code release the GIL while it's working.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Section 120.3: Using Parent and Children scripts to execute code in parallel

child.py

```
import time

def main():
    print "starting work"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"

if __name__ == '__main__':
    main()
```

parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

if __name__ == '__main__':
    main()
```

This is useful for parallel, independent HTTP request/response tasks or Database select/inserts. Command line arguments can be given to the **child.py** script as well. Synchronization between scripts can be achieved by all scripts regularly checking a separate server (like a Redis instance).

Section 120.4: Using PyPar module to parallelize

PyPar is a library that uses the message passing interface (MPI) to provide parallelism in Python. A simple example in PyPar (as seen at https://github.com/daleroberts/pypar) looks like this:

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print 'I am rank %d of %d on node %s' % (rank, ncpus, node)

if rank == 0:
  msh = 'P0'
  pp.send(msg, destination=1)
  msg = pp.receive(source=rank-1)
  print 'Processor 0 received message "%s" from rank %d' % (msg, rank-1)
else:
  source = rank-1
  destination = (rank+1) % ncpus
  msg = pp.receive(source)
  msg = msg + 'P' + str(rank)
  pypar.send(msg, destination)
pp.finalize()
```

Chapter 121: Sockets

Parameter	Description
socket.AF_UNIX	UNIX Socket
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Many programming languages use sockets to communicate across processes or between devices. This topic explains proper usage the sockets module in Python to facilitate sending and receiving data over common networking protocols.

Section 121.1: Raw Sockets on Linux

First you disable your network card's automatic checksumming:

```
sudo ethtool -K eth1 tx off
```

Then send your packet, using a SOCK_RAW socket:

```
#!/usr/bin/env python
from socket import socket, AF_PACKET, SOCK_RAW
s = socket(AF_PACKET, SOCK_RAW)
s.bind(("eth1", 0))

# We're putting together an ethernet frame here,
# but you could have anything you want instead
# Have a look at the 'struct' module for more
# flexible packing/unpacking of binary data
# and 'binascii' for 32 bit CRC
src_addr = "\x01\x02\x03\x04\x05\x06"
dst_addr = "\x01\x02\x03\x04\x05\x06"
payload = ("["*30)+"PAYLOAD"+("]"*30)
checksum = "\x1a\x2b\x3c\x4d"
ethertype = "\x08\x01"

s.send(dst_addr+src_addr+ethertype+payload+checksum)
```

Section 121.2: Sending data via UDP

UDP is a connectionless protocol. Messages to other processes or computers are sent without establishing any sort of connection. There is no automatic confirmation if your message has been received. UDP is usually used in latency sensitive applications or in applications sending network wide broadcasts.

The following code sends a message to a process listening on localhost port 6667 using UDP

Note that there is no need to "close" the socket after the send, because UDP is connectionless.

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
msg = ("Hello you there!").encode('utf-8')  # socket.sendto() takes bytes as input, hence we must
encode the string first.
s.sendto(msg, ('localhost', 6667))
```

Section 121.3: Receiving data via UDP

UDP is a connectionless protocol. This means that peers sending messages do not require establishing a connection before sending messages. `socket.recvfrom`thus returns a tuple (`msg` [the message the socket received], `addr` [the address of the sender])

A UDP server using solely the `socket` module:

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('localhost', 6667))

while True:
    msg, addr = sock.recvfrom(8192)  # This is the amount of bytes to read at maximum
    print("Got message from %s: %s" % (addr, msg))
```

Below is an alternative implementation using `socketserver.UDPServer`:

```
from socketserver import BaseRequestHandler, UDPServer

class MyHandler(BaseRequestHandler):
    def handle(self):
        print("Got connection from: %s" % self.client_address)
        msg, sock = self.request
        print("It said: %s" % msg)
        sock.sendto("Got your message!".encode(), self.client_address) # Send reply

serv = UDPServer(('localhost', 6667), MyHandler)
serv.serve_forever()
```

By default, `sockets` block. This means that execution of the script will wait until the socket receives data.

Section 121.4: Sending data via TCP

Sending data over the internet is made possible using multiple modules. The sockets module provides low-level access to the underlying Operating System operations responsible for sending or receiving data from other computers or processes.

The following code sends the byte string `b'Hello'` to a TCP server listening on port 6667 on the host localhost and closes the connection when finished:

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 6667))  # The address of the TCP server listening
s.send(b'Hello')
s.close()
```

Socket output is blocking by default, that means that the program will wait in the connect and send calls until the action is 'completed'. For connect that means the server actually accepting the connection. For send it only means that the operating system has enough buffer space to queue the data to be send later.

Sockets should always be closed after use.

Section 121.5: Multi-threaded TCP Socket Server

When run with no arguments, this program starts a TCP socket server that listens for connections to `127.0.0.1` on

port `5000`. The server handles each connection in a separate thread.

When run with the `-c` argument, this program connects to the server, reads the client list, and prints it out. The client list is transferred as a JSON string. The client name may be specified by passing the `-n` argument. By passing different names, the effect on the client list may be observed.

client_list.py

```
import argparse
import json
import socket
import threading

def handle_client(client_list, conn, address):
    name = conn.recv(1024)
    entry = dict(zip(['name', 'address', 'port'], [name, address[0], address[1]]))
    client_list[name] = entry
    conn.sendall(json.dumps(client_list))
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()

def server(client_list):
    print "Starting server..."
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('127.0.0.1', 5000))
    s.listen(5)
    while True:
        (conn, address) = s.accept()
        t = threading.Thread(target=handle_client, args=(client_list, conn, address))
        t.daemon = True
        t.start()

def client(name):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('127.0.0.1', 5000))
    s.send(name)
    data = s.recv(1024)
    result = json.loads(data)
    print json.dumps(result, indent=4)

def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', dest='client', action='store_true')
    parser.add_argument('-n', dest='name', type=str, default='name')
    result = parser.parse_args()
    return result

def main():
    client_list = dict()
    args = parse_arguments()
    if args.client:
        client(args.name)
    else:
        try:
            server(client_list)
        except KeyboardInterrupt:
            print "Keyboard interrupt"

if __name__ == '__main__':
```

```
    main()
```

Server Output

```
$ python client_list.py
Starting server...
```

Client Output

```
$ python client_list.py -c -n name1
{
    "name1": {
        "address": "127.0.0.1",
        "port": 62210,
        "name": "name1"
    }
}
```

The receive buffers are limited to 1024 bytes. If the JSON string representation of the client list exceeds this size, it will be truncated. This will cause the following exception to be raised:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)
```

Chapter 122: Websockets

Section 122.1: Simple Echo with aiohttp

`aiohttp` provides asynchronous websockets.

Python 3.x `Version ≥ 3.5`

```
import asyncio
from aiohttp import ClientSession

with ClientSession() as session:
    async def hello_world():

        websocket = await session.ws_connect("wss://echo.websocket.org")

        websocket.send_str("Hello, world!")

        print("Received:", (await websocket.receive()).data)

        await websocket.close()

    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello_world())
```

Section 122.2: Wrapper Class with aiohttp

`aiohttp.ClientSession` may be used as a parent for a custom WebSocket class.

Python 3.x `Version ≥ 3.5`

```
import asyncio
from aiohttp import ClientSession

class EchoWebSocket(ClientSession):

    URL = "wss://echo.websocket.org"

    def __init__(self):
        super().__init__()
        self.websocket = None

    async def connect(self):
        """Connect to the WebSocket."""
        self.websocket = await self.ws_connect(self.URL)

    async def send(self, message):
        """Send a message to the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        self.websocket.send_str(message)
        print("Sent:", message)

    async def receive(self):
        """Receive one message from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        return (await self.websocket.receive()).data

    async def read(self):
        """Read messages from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
```

```
        while self.websocket.receive():
            message = await self.receive()
            print("Received:", message)
            if message == "Echo 9!":
                break

async def send(websocket):
    for n in range(10):
        await websocket.send("Echo {}!".format(n))
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()

with EchoWebSocket() as websocket:

    loop.run_until_complete(websocket.connect())

    tasks = (
        send(websocket),
        websocket.read()
    )

    loop.run_until_complete(asyncio.wait(tasks))

    loop.close()
```

Section 122.3: Using Autobahn as a Websocket Factory

The Autobahn package can be used for Python web socket server factories.

Python Autobahn package documentation

To install, typically one would simply use the terminal command

(For Linux):

```
sudo pip install autobahn
```

(For Windows):

```
python -m pip install autobahn
```

Then, a simple echo server can be created in a Python script:

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
class MyServerProtocol(WebSocketServerProtocol):
    '''When creating server protocol, the
    user defined class inheriting the
    WebSocketServerProtocol needs to override
    the onMessage, onConnect, et-c events for
    user specified functionality, these events
    define your server's protocol, in essence'''
    def onMessage(self,payload,isBinary):
        '''The onMessage routine is called
        when the server receives a message.
        It has the required arguments payload
        and the bool isBinary. The payload is the
        actual contents of the "message" and isBinary
        is simply a flag to let the user know that
```

```
        the payload contains binary data. I typically
        elsewise assume that the payload is a string.
        In this example, the payload is returned to sender verbatim.'''
        self.sendMessage(payload,isBinary)
if__name__=='__main__':
    try:
        importasyncio
    except ImportError:
        '''Trollius = 0.3 was renamed'''
        import trollius as asyncio
    from autobahn.asyncio.websocketimportWebSocketServerFactory
    factory=WebSocketServerFactory()
    '''Initialize the websocket factory, and set the protocol to the
    above defined protocol(the class that inherits from
    autobahn.asyncio.websocket.WebSocketServerProtocol)'''
    factory.protocol=MyServerProtocol
    '''This above line can be thought of as "binding" the methods
    onConnect, onMessage, et-c that were described in the MyServerProtocol class
    to the server, setting the servers functionality, ie, protocol'''
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    '''Run the server in an infinite loop'''
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()
```

In this example, a server is being created on the localhost (127.0.0.1) on port 9000. This is the listening IP and port. This is important information, as using this, you could identify your computer's LAN address and port forward from your modem, though whatever routers you have to the computer. Then, using google to investigate your WAN IP, you could design your website to send WebSocket messages to your WAN IP, on port 9000 (in this example).

It is important that you port forward from your modem back, meaning that if you have routers daisy chained to the modem, enter into the modem's configuration settings, port forward from the modem to the connected router, and so forth until the final router your computer is connected to is having the information being received on modem port 9000 (in this example) forwarded to it.

Chapter 123: Sockets And Message Encryption/Decryption Between Client and Server

Cryptography is used for security purposes. There are not so many examples of Encryption/Decryption in Python using IDEA encryption MODE CTR. **Aim of this documentation :**

Extend and implement of the RSA Digital Signature scheme in station-to-station communication. Using Hashing for integrity of message, that is SHA-1. Produce simple Key Transport protocol. Encrypt Key with IDEA encryption. Mode of Block Cipher is Counter Mode

Section 123.1: Server side Implementation

```
import socket
import hashlib
import os
import time
import itertools
import threading
import sys
import Crypto.Cipher.AES as AES
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#server address and port number input from admin
host= raw_input("Server Address - > ")
port = int(input("Port - > "))
#boolean for checking server and port
check = False
done = False

def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCHECKING IP ADDRESS AND NOT USED PORT '+c)
        sys.stdout.flush()
        time.sleep(0.1)
    sys.stdout.write('\r -----SERVER STARTED. WAITING FOR CLIENT-----\n')
try:
    #setting up socket
    server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    server.bind((host,port))
    server.listen(5)
    check = True
except BaseException:
    print "-----Check Server Address or Port-----"
    check = False

if check is True:
    # server Quit
    shutdown = False
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
```

```
done = True
#binding client and address
client,address = server.accept()
print ("CLIENT IS CONNECTED. CLIENT'S ADDRESS ->",address)
print ("\n-----WAITING FOR PUBLIC KEY & PUBLIC KEY HASH-----\n")

#client's message(Public Key)
getpbk = client.recv(2048)

#conversion of string to KEY
server_public_key = RSA.importKey(getpbk)

#hashing the public key in server side for validating the hash from client
hash_object = hashlib.sha1(getpbk)
hex_digest = hash_object.hexdigest()

if getpbk != "":
    print (getpbk)
    client.send("YES")
    gethash = client.recv(1024)
    print ("\n-----HASH OF PUBLIC KEY----- \n"+gethash)
if hex_digest == gethash:
    # creating session key
    key_128 = os.urandom(16)
    #encrypt CTR MODE session key
    en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128)
    encrypto = en.encrypt(key_128)
    #hashing sha1
    en_object = hashlib.sha1(encrypto)
    en_digest = en_object.hexdigest()

    print ("\n-----SESSION KEY-----\n"+en_digest)

    #encrypting session key and public key
    E = server_public_key.encrypt(encrypto,16)
    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY-----\n"+str(E))
    print ("\n-----HANDSHAKE COMPLETE-----")
    client.send(str(E))
    while True:
        #message from client
        newmess = client.recv(1024)
        #decoding the message from HEXADECIMAL to decrypt the encrypted version of the message only
        decoded = newmess.decode("hex")
        #making en_digest(session_key) as the key
        key = en_digest[:16]
        print ("\nENCRYPTED MESSAGE FROM CLIENT -> "+newmess)
        #decrypting message from the client
        ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
        dMsg = ideaDecrypt.decrypt(decoded)
        print ("\n**New Message**  "+time.ctime(time.time()) +" > "+dMsg+"\n")
        mess = raw_input("\nMessage To Client -> ")
        if mess != "":
            ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
            eMsg = ideaEncrypt.encrypt(mess)
            eMsg = eMsg.encode("hex").upper()
            if eMsg != "":
                print ("ENCRYPTED MESSAGE TO CLIENT-> " + eMsg)
            client.send(eMsg)
    client.close()
else:
    print ("\n-----PUBLIC KEY HASH DOESNOT MATCH-----\n")
```

Section 123.2: Client side Implementation

```
import time
import socket
import threading
import hashlib
import itertools
import sys
from Crypto import Random
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#animating loading
done = False
def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCONFIRMING CONNECTION TO SERVER '+c)
        sys.stdout.flush()
        time.sleep(0.1)

#public key and private key
random_generator = Random.new().read
key = RSA.generate(1024,random_generator)
public = key.publickey().exportKey()
private = key.exportKey()

#hashing the public key
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()

#Setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

#host and port input user
host = raw_input("Server Address To Be Connected -> ")
port = int(input("Port of The Server -> "))
#binding the address and port
server.connect((host, port))
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True

def send(t,name,key):
    mess = raw_input(name + " : ")
    key = key[:16]
    #merging the message and the name
    whole = name+" : "+mess
    ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
    eMsg = ideaEncrypt.encrypt(whole)
    #converting the encrypted message to HEXADECIMAL to readable
    eMsg = eMsg.encode("hex").upper()
    if eMsg != "":
        print ("ENCRYPTED MESSAGE TO SERVER-> "+eMsg)
    server.send(eMsg)
def recv(t,key):
    newmess = server.recv(1024)
```

```
    print ("\nENCRYPTED MESSAGE FROM SERVER-> " + newmess)
    key = key[:16]
    decoded = newmess.decode("hex")
    ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
    dMsg = ideaDecrypt.decrypt(decoded)
    print ("\n**New Message From Server**  " + time.ctime(time.time()) + " : " + dMsg + "\n")

while True:
    server.send(public)
    confirm = server.recv(1024)
    if confirm == "YES":
        server.send(hex_digest)

    #connected msg
    msg = server.recv(1024)
    en = eval(msg)
    decrypt = key.decrypt(en)
    # hashing sha1
    en_object = hashlib.sha1(decrypt)
    en_digest = en_object.hexdigest()

    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY FROM SERVER-----")
    print (msg)
    print ("\n-----DECRYPTED SESSION KEY-----")
    print (en_digest)
    print ("\n-----HANDSHAKE COMPLETE-----\n")
    alais = raw_input("\nYour Name -> ")

    while True:
        thread_send = threading.Thread(target=send,args=("------Sending Message------
",alais,en_digest))
        thread_recv = threading.Thread(target=recv,args=("------Receiving Message------
",en_digest))
        thread_send.start()
        thread_recv.start()

        thread_send.join()
        thread_recv.join()
        time.sleep(0.5)
    time.sleep(60)
    server.close()
```

Chapter 124: Python Networking

Section 124.1: Creating a Simple Http Server

To share files or to host simple websites(http and javascript) in your local network, you can use Python's builtin SimpleHTTPServer module. Python should be in your Path variable. Go to the folder where your files are and type:

For `python 2`:

```
$ python -m SimpleHTTPServer <portnumber>
```

For `python 3`:

```
$ python3 -m http.server <portnumber>
```

If port number is not given `8000` is the default port. So the output will be:

> Serving HTTP on 0.0.0.0 port 8000 ...

You can access to your files through any device connected to the local network by typing `http://hostipaddress:8000/`.

`hostipaddress` is your local IP address which probably starts with `192.168.x.x`.

To finish the module simply press `ctrl+c`.

Section 124.2: Creating a TCP server

You can create a TCP server using the `socketserver` library. Here's a simple echo server.

Server side

```
from sockerserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('connection from:', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    server = TCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Client side

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 5000))
sock.send(b'Monty Python')
sock.recv(8192)  # returns b'Monty Python'
```

`socketserver` makes it relatively easy to create simple TCP servers. However, you should be aware that, by default, the servers are single threaded and can only serve one client at a time. If you want to handle multiple clients, either instantiate a `ThreadingTCPServer` instead.

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Section 124.3: Creating a UDP Server

A UDP server is easily created using the `socketserver` library.

a simple time server:

```
import time
from socketserver import BaseRequestHandler, UDPServer

class CtimeHandler(BaseRequestHandler):
    def handle(self):
    print('connection from: ', self.client_address)
    # Get message and client socket
    msg, sock = self.request
    resp = time.ctime()
    sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    server = UDPServer(('', 5000), CtimeHandler)
    server.serve_forever()
```

Testing:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
>>> sick.sendto(b'', ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

Section 124.4: Start Simple HttpServer in a thread and open the browser

Useful if your program is outputting web pages along the way.

```
from http.server import HTTPServer, CGIHTTPRequestHandler
import webbrowser
import threading

def start_server(path, port=8000):
    '''Start a simple webserver serving path on port'''
    os.chdir(path)
    httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
    httpd.serve_forever()

# Start the server in a new thread
port = 8000
daemon = threading.Thread(name='daemon_server',
```

```
                          target=start_server,
                          args=('.', port)
daemon.setDaemon(True) # Set as a daemon so it will be killed once the main thread is dead.
daemon.start()

# Open the web browser
webbrowser.open('http://localhost:{}'.format(port))
```

Section 124.5: The simplest Python socket client-server example

Server side:

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5) # become a server socket, maximum 5 connections

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf) > 0:
        print(buf)
    break
```

Client Side:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

First run the SocketServer.py, and make sure the server is ready to listen/receive sth Then the client send info to the server; After the server received sth, it terminates

Chapter 125: Python HTTP Server

Section 125.1: Running a simple HTTP server

Python 2.x `Version ≥ 2.3`

```
python -m SimpleHTTPServer 9000
```

Python 3.x `Version ≥ 3.0`

```
python -m http.server 9000
```

Running this command serves the files of the current directory at port `9000`.

If no argument is provided as port number then server will run on default port `8000`.

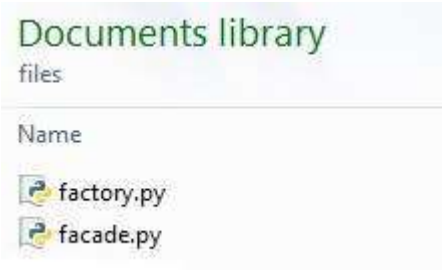
The `-m` flag will search `sys.path` for the corresponding `.py` file to run as a module.

If you want to only serve on localhost you'll need to write a custom Python program such as:

```
import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

HandlerClass = SimpleHTTPRequestHandler
ServerClass  = BaseHTTPServer.HTTPServer
Protocol     = "HTTP/1.0"

if sys.argv[1:]:
   port = int(sys.argv[1])
else:
   port = 8000
server_address = ('127.0.0.1', port)

HandlerClass.protocol_version = Protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

Section 125.2: Serving files

Assuming you have the following directory of files:



You can setup a web server to serve these files as follows:

Python 2.x `Version ≥ 2.3`

```
import SimpleHTTPServer
import SocketServer
```

```
PORT = 8000

handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("localhost", PORT), handler)
print "Serving files at port {}".format(PORT)
httpd.serve_forever()
```

Python 3.x `Version ≥ 3.0`

```
import http.server
import socketserver

PORT = 8000

handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), handler)
print("serving at port", PORT)
httpd.serve_forever()
```

The `SocketServer` module provides the classes and functionalities to setup a network server.

`SocketServer`'s `TCPServer` class sets up a server using the TCP protocol. The constructor accepts a tuple representing the address of the server (i.e. the IP address and port) and the class that handles the server requests.

The `SimpleHTTPRequestHandler` class of the `SimpleHTTPServer` module allows the files at the current directory to be served.

Save the script at the same directory and run it.

Run the HTTP Server :

Python 2.x `Version ≥ 2.3`

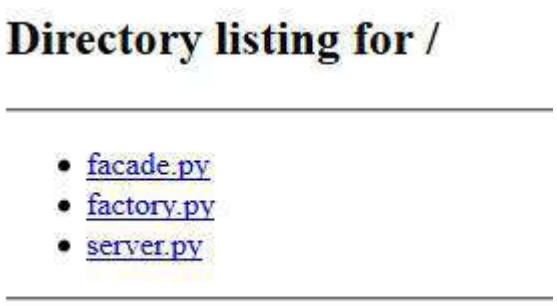
python -m SimpleHTTPServer 8000

Python 3.x `Version ≥ 3.0`

python -m http.server 8000

The '-m' flag will search 'sys.path' for the corresponding '.py' file to run as a module.

Open localhost:8000 in the browser, it will give you the following:



Section 125.3: Basic handling of GET, POST, PUT using BaseHTTPRequestHandler

```
# from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # python2
```

```
from http.server import BaseHTTPRequestHandler, HTTPServer # python3
class HandleRequests(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("received get request")

    def do_POST(self):
        '''Reads post request body'''
        self._set_headers()
        content_len = int(self.headers.getheader('content-length', 0))
        post_body = self.rfile.read(content_len)
        self.wfile.write("received post request:<br>{}".format(post_body))

    def do_PUT(self):
        self.do_POST()

host = ''
port = 80
HTTPServer((host, port), HandleRequests).serve_forever()
```

Example output using `curl`:

```
$ curl http://localhost/
received get request%



$ curl -X POST http://localhost/
received post request:<br>%



$ curl -X PUT http://localhost/
received post request:<br>%



$ echo 'hello world' | curl --data-binary @- http://localhost/
received post request:<br>hello world
```

Section 125.4: Programmatic API of SimpleHTTPServer

What happens when we execute `python -m SimpleHTTPServer 9000`?

To answer this question we should understand the construct of SimpleHTTPServer (https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py) and BaseHTTPServer(https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py).

Firstly, Python invokes the `SimpleHTTPServer` module with `9000` as an argument. Now observing the SimpleHTTPServer code,

```
def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)
```

```
if __name__ == '__main__':
    test()
```

The test function is invoked following request handlers and ServerClass. Now BaseHTTPServer.test is invoked

```
def test(HandlerClass = BaseHTTPRequestHandler,
         ServerClass = HTTPServer, protocol="HTTP/1.0"):
"""Test the HTTP request handler class.

This runs an HTTP server on port 8000 (or the first command line
argument).

"""

if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ('', port)

HandlerClass.protocol_version = protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

Hence here the port number, which the user passed as argument is parsed and is bound to the host address. Further basic steps of socket programming with given port and protocol is carried out. Finally socket server is initiated.

This is a basic overview of inheritance from SocketServer class to other classes:

```
 +------------+
| BaseServer |
+------------+
|
v
+-----------+ +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+ +------------------+
|
v
+-----------+ +--------------------+
| UDPServer |------->| UnixDatagramServer |
+-----------+ +--------------------+
```

The links https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py and https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py are useful for finding further information.

Chapter 126: Flask

Flask is a Python micro web framework used to run major websites including Pinterest, Twilio, and LinkedIn. This topic explains and demonstrates the variety of features Flask offers for both front and back end web development.

Section 126.1: Files and Templates

Instead of typing our HTML markup into the return statements, we can use the `render_template()` function:

```
from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

This will use our template file `about-us.html`. To ensure our application can find this file we must organize our directory in the following format:

```
- application.py
/templates
    - about-us.html
    - login-form.html
/static
    /styles
        - about-style.css
        - login-style.css
    /scripts
        - about-script.js
        - login-script.js
```

Most importantly, references to these files in the HTML must look like this:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-style.css')}}">
```

which will direct the application to look for `about-style.css` in the styles folder under the static folder. The same format of path applies to all references to images, styles, scripts, or files.

Section 126.2: The basics

The following example is an example of a basic server:

```
# Imports the Flask class
from flask import Flask
# Creates an app and checks if it's the main or imported
app = Flask(__name__)

# Specifies what URL triggers hello_world()
@app.route('/')
# The function run on the index route
def hello_world():
    # Returns the text to be displayed
```

```
    return "Hello World!"

# If this script isn't an import
if __name__ == "__main__":
    # Run the app until stopped
    app.run()
```

Running this script (with all the right dependencies installed) should start up a local server. The host is `127.0.0.1` commonly known as **localhost**. This server by default runs on port **5000**. To access your webserver, open a web browser and enter the URL `localhost:5000` or `127.0.0.1:5000` (no difference). Currently, only your computer can access the webserver.

`app.run()` has three parameters, **host**, **port**, and **debug**. The host is by default `127.0.0.1`, but setting this to `0.0.0.0` will make your web server accessible from any device on your network using your private IP address in the URL. the port is by default 5000 but if the parameter is set to port `80`, users will not need to specify a port number as browsers use port 80 by default. As for the debug option, during the development process (never in production) it helps to set this parameter to True, as your server will restart when changes made to your Flask project.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

Section 126.3: Routing URLs

With Flask, URL routing is traditionally done using decorators. These decorators can be used for static routing, as well as routing URLs with parameters. For the following example, imagine this Flask script is running the website `www.example.com`.

```
@app.route("/")
def index():
    return "You went to www.example.com"

@app.route("/about")
def about():
    return "You went to www.example.com/about"

@app.route("/users/guido-van-rossum")
    return "You went to www.example.com/guido-van-rossum"
```

With that last route, you can see that given a URL with /users/ and the profile name, we could return a profile. Since it would be horribly inefficient and messy to include a `@app.route()` for every user, Flask offers to take parameters from the URL:

```
@app.route("/users/<username>")
def profile(username):
    return "Welcome to the profile of " + username

cities = ["OMAHA", "MELBOURNE", "NEPAL", "STUTTGART", "LIMA", "CAIRO", "SHANGHAI"]

@app.route("/stores/locations/<city>")
def storefronts(city):
    if city in cities:
        return "Yes! We are located in " + city
    else:
        return "No. We are not located in " + city
```

Section 126.4: HTTP Methods

The two most common HTTP methods are **GET** and **POST**. Flask can run different code from the same URL dependent on the HTTP method used. For example, in a web service with accounts, it is most convenient to route the sign in page and the sign in process through the same URL. A GET request, the same that is made when you open a URL in your browser should show the login form, while a POST request (carrying login data) should be processed separately. A route is also created to handle the DELETE and PUT HTTP method.

```
@app.route("/login", methods=["GET"])
def login_form():
    return "This is the login form"
@app.route("/login", methods=["POST"])
def login_auth():
    return "Processing your data"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "This method is not allowed"
```

To simplify the code a bit, we can import the `request` package from flask.

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Processing your data"
```

To retrieve data from the POST request, we must use the `request` package:

```
from flask import request
@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Username was " + request.form["username"] + " and password was " +
request.form["password"]
```

Section 126.5: Jinja Templating

Similar to Meteor.js, Flask integrates well with front end templating services. Flask uses by default Jinja Templating. Templates allow small snippets of code to be used in the HTML file such as conditionals or loops.

When we render a template, any parameters beyond the template file name are passed into the HTML templating service. The following route will pass the username and joined date (from a function somewhere else) into the HTML.

```
@app.route("/users/<username>)
def profile(username):
    joinedDate = get_joined_date(username) # This function's code is irrelevant
    awards = get_awards(username) # This function's code is irrelevant
```

```
    # The joinDate is a string and awards is an array of strings
    return render_template("profile.html", username=username, joinDate=joinDate, awards=awards)
```

When this template is rendered, it can use the variables passed to it from the `render_template()` function. Here are the contents of `profile.html`:

```
<!DOCTYPE html>
<html>
    <head>
        # if username
            <title>Profile of {{ username }}</title>
        # else
            <title>No User Found</title>
        # endif
    <head>
    <body>
        {% if username %}
            <h1>{{ username }} joined on the date {{ date }}</h1>
            {% if len(awards) > 0 %}
                <h3>{{ username }} has the following awards:</h3>
                <ul>
                {% for award in awards %}
                    <li>{{award}}</li>
                {% endfor %}
                </ul>
            {% else %}
                <h3>{{ username }} has no awards</h3>
            {% endif %}
        {% else %}
            <h1>No user was found under that username</h1>
        {% endif %}
        {# This is a comment and doesn't affect the output #}
    </body>
</html>
```

The following delimiters are used for different interpretations:

- `{% ... %}` denotes a statement
- `{{ ... }}` denotes an expression where a template is outputted
- `{# ... #}` denotes a comment (not included in template output)
- `{# ... ##` implies the rest of the line should be interpreted as a statement

Section 126.6: The Request Object

The `request` object provides information on the request that was made to the route. To utilize this object, it must be imported from the flask module:

```
from flask import request
```

URL Parameters

In previous examples `request.method` and `request.form` were used, however we can also use the `request.args` property to retrieve a dictionary of the keys/values in the URL parameters.

```
@app.route("/api/users/<username>")
def user_api(username):
    try:
        token = request.args.get("key")
        if key == "pA55w0Rd":
```

```
            if isUser(username): # The code of this method is irrelevant
                joined = joinDate(username) # The code of this method is irrelevant
                return "User " + username + " joined on " + joined
            else:
                return "User not found"
        else:
            return "Incorrect key"
    # If there is no key parameter
    except KeyError:
        return "No key provided"
```

To correctly authenticate in this context, the following URL would be needed (replacing the username with any username:

```
www.example.com/api/users/guido-van-rossum?key=pa55w0Rd
```

File Uploads

If a file upload was part of the submitted form in a POST request, the files can be handled using the `request` object:

```
@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # Store with the original filename
```

Cookies

The request may also include cookies in a dictionary similar to the URL parameters.

```
@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "Your stored username is " + username
    except KeyError:
        return "No username cookies was found")
```

Chapter 127: Introduction to RabbitMQ using AMQPStorm

Section 127.1: How to consume messages from RabbitMQ

Start with importing the library.

```
from amqpstorm import Connection
```

When consuming messages, we first need to define a function to handle the incoming messages. This can be any callable function, and has to take a message object, or a message tuple (depending on the `to_tuple` parameter defined in `start_consuming`).

Besides processing the data from the incoming message, we will also have to Acknowledge or Reject the message. This is important, as we need to let RabbitMQ know that we properly received and processed the message.

```
def on_message(message):
    """This function is called on message received.

    :param message: Delivered message.
    :return:
    """
    print("Message:", message.body)

    # Acknowledge that we handled the message without any issues.
    message.ack()

    # Reject the message.
    # message.reject()

    # Reject the message, and put it back in the queue.
    # message.reject(requeue=True)
```

Next we need to set up the connection to the RabbitMQ server.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

After that we need to set up a channel. Each connection can have multiple channels, and in general when performing multi-threaded tasks, it's recommended (but not required) to have one per thread.

```
channel = connection.channel()
```

Once we have our channel set up, we need to let RabbitMQ know that we want to start consuming messages. In this case we will use our previously defined `on_message` function to handle all our consumed messages.

The queue we will be listening to on the RabbitMQ server is going to be `simple_queue`, and we are also telling RabbitMQ that we will be acknowledging all incoming messages once we are done with them.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Finally we need to start the IO loop to start processing messages delivered by the RabbitMQ server.

```
channel.start_consuming(to_tuple=False)
```

Section 127.2: How to publish messages to RabbitMQ

Start with importing the library.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Next we need to open a connection to the RabbitMQ server.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

After that we need to set up a channel. Each connection can have multiple channels, and in general when performing multi-threaded tasks, it's recommended (but not required) to have one per thread.

```
channel = connection.channel()
```

Once we have our channel set up, we can start to prepare our message.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# Create the message.
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

Now we can publish the message by simply calling `publish` and providing a `routing_key`. In this case we are going to send the message to a queue called `simple_queue`.

```
message.publish(routing_key='simple_queue')
```

Section 127.3: How to create a delayed queue in RabbitMQ

First we need to set up two basic channels, one for the main queue, and one for the delay queue. In my example at the end, I include a couple of additional flags that are not required, but makes the code more reliable; such as `confirm delivery`, `delivery_mode` and `durable`. You can find more information on these in the RabbitMQ manual.

After we have set up the channels we add a binding to the main channel that we can use to send messages from the delay channel to our main queue.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

Next we need to configure our delay channel to forward messages to the main queue once they have expired.

```
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000,
    'x-dead-letter-exchange': 'amq.direct',
    'x-dead-letter-routing-key': 'hello'
})
```

- x-message-ttl *(Message - Time To Live)*

  This is normally used to automatically remove old messages in the queue after a specific duration, but by

adding two optional arguments we can change this behaviour, and instead have this parameter determine in milliseconds how long messages will stay in the delay queue.

- x-dead-letter-routing-key

  This variable allows us to transfer the message to a different queue once they have expired, instead of the default behaviour of removing it completely.

- x-dead-letter-exchange

  This variable determines which Exchange used to transfer the message from hello_delay to hello queue.

Publishing to the delay queue

When we are done setting up all the basic Pika parameters you simply send a message to the delay queue using basic publish.

```
delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mod': 2})
```

Once you have executed the script you should see the following queues created in your RabbitMQ management module.

Overview					Messages			Message rates		
Name	**Exclusive**	Parameters	**Policy**	**Status**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
hello		D		Idle	1	0	1			
hello_delay		TTL DLX DLK D		Idle	0	0	0	0.00/s		

Example.

```
from amqpstorm import Connection

connection = Connection('127.0.0.1', 'guest', 'guest')

# Create normal 'Hello World' type channel.
channel = connection.channel()
channel.confirm_deliveries()
channel.queue.declare(queue='hello', durable=True)

# We need to bind this channel to an exchange, that will be used to transfer
# messages from our delay queue.
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')

# Create our delay channel.
delay_channel = connection.channel()
delay_channel.confirm_deliveries()

# This is where we declare the delay, and routing for our delay channel.
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000, # Delay until the message is transferred in milliseconds.
    'x-dead-letter-exchange': 'amq.direct', # Exchange used to transfer the message from A to B.
    'x-dead-letter-routing-key': 'hello' # Name of the queue we want the message transferred to.
```

```
})

delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mode': 2})

print("[x] Sent")
```

Chapter 128: Descriptor

Section 128.1: Simple descriptor

There are two different types of descriptors. Data descriptors are defined as objects that define both a `__get__()` and a `__set__()` method, whereas non-data descriptors only define a `__get__()` method. This distinction is important when considering overrides and the namespace of an instance's dictionary. If a data descriptor and an entry in an instance's dictionary share the same name, the data descriptor will take precedence. However, if instead a non-data descriptor and an entry in an instance's dictionary share the same name, the instance dictionary's entry will take precedence.

To make a read-only data descriptor, define both **get**() and **set**() with the **set**() raising an AttributeError when called. Defining the **set**() method with an exception raising placeholder is enough to make it a data descriptor.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

An implemented example:

```
class DescPrinter(object):
    """A data descriptor that logs activity."""
    _val = 7

    def __get__(self, obj, objtype=None):
        print('Getting ...')
        return self._val

    def __set__(self, obj, val):
        print('Setting', val)
        self._val = val

    def __delete__(self, obj):
        print('Deleting ...')
        del self._val


class Foo():
    x = DescPrinter()

i = Foo()
i.x
# Getting ...
# 7

i.x = 100
# Setting 100
i.x
# Getting ...
# 100

del i.x
# Deleting ...
i.x
# Getting ...
# 7
```

Section 128.2: Two-way conversions

Descriptor objects can allow related object attributes to react to changes automatically.

Suppose we want to model an oscillator with a given frequency (in Hertz) and period (in seconds). When we update the frequency we want the period to update, and when we update the period we want the frequency to update:

```
 >>> oscillator = Oscillator(freq=100.0)  # Set frequency to 100.0 Hz
>>> oscillator.period  # Period is 1 / frequency, i.e. 0.01 seconds
0.01
>>> oscillator.period = 0.02  # Set period to 0.02 seconds
>>> oscillator.freq # The frequency is automatically adjusted
50.0
>>> oscillator.freq = 200.0  # Set the frequency to 200.0 Hz
>>> oscillator.period  # The period is automatically adjusted
0.005
```

We pick one of the values (frequency, in Hertz) as the "anchor," i.e. the one that can be set with no conversion, and write a descriptor class for it:

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

The "other" value (period, in seconds) is defined in terms of the anchor. We write a descriptor class that does our conversions:

```
class Second(object):
    def __get__(self, instance, owner):
        # When reading period, convert from frequency
        return 1 / instance.freq

    def __set__(self, instance, value):
        # When setting period, update the frequency
        instance.freq = 1 / float(value)
```

Now we can write the Oscillator class:

```
class Oscillator(object):
    period = Second()  # Set the other value as a class attribute

    def __init__(self, freq):
        self.freq = Hertz()  # Set the anchor value as an instance attribute
        self.freq = freq  # Assign the passed value - self.period will be adjusted
```

Chapter 129: tempfile NamedTemporaryFile

param	description
mode	mode to open file, default=w+b
delete	To delete file on closure, default=True
suffix	filename suffix, default=''
prefix	filename prefix, default='tmp'
dir	dirname to place tempfile, default=None
buffsize	default=-1, (operating system default used)

Section 129.1: Create (and write to a) known, persistent temporary file

You can create temporary files which has a visible name on the file system which can be accessed via the `name` property. The file can, on unix systems, be configured to delete on closure (set by `delete` param, default is True) or can be reopened later.

The following will create and open a named temporary file and write 'Hello World!' to that file. The filepath of the temporary file can be accessed via `name`, in this example it is saved to the variable `path` and printed for the user. The file is then re-opened after closing the file and the contents of the tempfile are read and printed for the user.

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as t:
    t.write('Hello World!')
    path = t.name
    print path

with open(path) as t:
    print t.read()
```

Output:

```
/tmp/tmp6pireJ
Hello World!
```

Chapter 130: Input, Subset and Output External Data Files using Pandas

This section shows basic code for reading, sub-setting and writing external data files using pandas.

Section 130.1: Basic Code to Import, Subset and Write External Data Files Using Pandas

```
# Print the working directory
import os
print os.getcwd()
# C:\Python27\Scripts

# Set the working directory
os.chdir('C:/Users/general1/Documents/simple Python files')
print os.getcwd()
# C:\Users\general1\Documents\simple Python files

# load pandas
import pandas as pd

# read a csv data file named 'small_dataset.csv' containing 4 lines and 3 variables
my_data = pd.read_csv("small_dataset.csv")
my_data
#     x   y   z
# 0   1   2   3
# 1   4   5   6
# 2   7   8   9
# 3  10  11  12

my_data.shape       # number of rows and columns in data set
# (4, 3)

my_data.shape[0]    # number of rows in data set
# 4

my_data.shape[1]    # number of columns in data set
# 3

# Python uses 0-based indexing.  The first row or column in a data set is located
# at position 0.  In R the first row or column in a data set is located
# at position 1.

# Select the first two rows
my_data[0:2]
#    x   y   z
#0   1   2   3
#1   4   5   6

# Select the second and third rows
my_data[1:3]
#    x  y  z
# 1  4  5  6
# 2  7  8  9

# Select the third row
my_data[2:3]
#    x   y   z
#2   7   8   9
```

```
# Select the first two elements of the first column
my_data.iloc[0:2, 0:1]
#    x
# 0  1
# 1  4

# Select the first element of the variables y and z
my_data.loc[0, ['y', 'z']]
# y    2
# z    3

# Select the first three elements of the variables y and z
my_data.loc[0:2, ['y', 'z']]
#    y  z
# 0  2  3
# 1  5  6
# 2  8  9

# Write the first three elements of the variables y and z
# to an external file.  Here index = 0 means do not write row names.

my_data2 = my_data.loc[0:2, ['y', 'z']]

my_data2.to_csv('my.output.csv', index = 0)
```

Chapter 131: Unzipping Files

To extract or uncompress a tarball, ZIP, or gzip file, Python's tarfile, zipfile, and gzip modules are provided respectively. Python's tarfile module provides the `TarFile.extractall(path=".", members=None)` function for extracting from a tarball file. Python's zipfile module provides the `ZipFile.extractall([path[, members[, pwd]]])` function for extracting or unzipping ZIP compressed files. Finally, Python's gzip module provides the GzipFile class for decompressing.

Section 131.1: Using Python ZipFile.extractall() to decompress a ZIP file

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

Section 131.2: Using Python TarFile.extractall() to decompress a tarball

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

Chapter 132: Working with ZIP archives

Section 132.1: Examining Zipfile Contents

There are a few ways to inspect the contents of a zipfile. You can use the `printdir` to just get a variety of information sent to `stdout`

```
with zipfile.ZipFile(filename) as zip:
    zip.printdir()

    # Out:
    # File Name                                             Modified             Size
    # pyexpat.pyd                                    2016-06-25 22:13:34       157336
    # python.exe                                     2016-06-25 22:13:34        39576
    # python3.dll                                    2016-06-25 22:13:34        51864
    # python35.dll                                   2016-06-25 22:13:34      3127960
    # etc.
```

We can also get a list of filenames with the `namelist` method. Here, we simply print the list:

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())

# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

Instead of `namelist`, we can call the `infolist` method, which returns a list of `ZipInfo` objects, which contain additional information about each file, for instance a timestamp and file size:

```
with zipfile.ZipFile(filename) as zip:
    info = zip.infolist()
    print(zip[0].filename)
    print(zip[0].date_time)
    print(info[0].file_size)

# Out: pyexpat.pyd
# Out: (2016, 6, 25, 22, 13, 34)
# Out: 157336
```

Section 132.2: Opening Zip Files

To start, import the `zipfile` module, and set the filename.

```
import zipfile
filename = 'zipfile.zip'
```

Working with zip archives is very similar to working with files, you create the object by opening the zipfile, which lets you work on it before closing the file up again.

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
zip.close()
```

In Python 2.7 and in Python 3 versions higher than 3.2, we can use the `with` context manager. We open the file in "read" mode, and then print a list of filenames:

```
with zipfile.ZipFile(filename, 'r') as z:
    print(zip)
    # <zipfile.ZipFile object at 0x0000000002E51A90>
```

Section 132.3: Extracting zip file contents to a directory

Extract all file contents of a zip file

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

If you want extract single files use extract method, it takes name list and path as input parameter

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

Section 132.4: Creating new archives

To create new archive open zipfile with write mode.

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

To add files to this archive use write() method.

```
new_arch.write('filename.txt','filename_in_archive.txt') #first parameter is filename and second
parameter is filename in archive by default filename will be taken if not provided
new_arch.close()
```

If you want to write string of bytes into the archive you can use writestr() method.

```
str_bytes="string buffer"
new_arch.writestr('filename_string_in_archive.txt',str_bytes)
new_arch.close()
```

Chapter 133: Getting start with GZip

This module provides a simple interface to compress and decompress files just like the GNU programs gzip and gunzip would.

The data compression is provided by the zlib module.

The gzip module provides the GzipFile class which is modeled after Python's File Object. The GzipFile class reads and writes gzip-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

Section 133.1: Read and write GNU zip files

```
import gzip
import os

outfilename = 'example.txt.gz'
output = gzip.open(outfilename, 'wb')
try:
    output.write('Contents of the example file go here.\n')
finally:
    output.close()

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes of compressed data'
os.system('file -b --mime %s' % outfilename)
```

Save it as 1gzip_write.py1.Run it through terminal.

```
$ python gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

Chapter 134: Stack

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push the item into the stack, and pop the item out of the stack**. A stack is a limited access data structure - **elements can be added and removed from the stack only at the top**. Here is a structural definition of a Stack: a stack is either empty or it consists of a top and the rest which is a Stack.

Section 134.1: Creating a Stack class with a List Object

Using a `list` object you can create a fully functional generic Stack with helper methods such as peeking and checking if the stack is Empty. Check out the official python docs for using `list` as `Stack` here.

```
#define a stack class
class Stack:
    def __init__(self):
        self.items = []

    #method to check the stack is empty or not
    def isEmpty(self):
        return self.items == []

    #method for pushing an item
    def push(self, item):
        self.items.append(item)

    #method for popping an item
    def pop(self):
        return self.items.pop()

    #check what item is on top of the stack without removing it
    def peek(self):
        return self.items[-1]

    #method to get the size
    def size(self):
        return len(self.items)

    #to view the entire stack
    def fullStack(self):
        return self.items
```

An example run:

```
stack = Stack()
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
print('Pushing integer 1')
stack.push(1)
print('Pushing string "Told you, I am generic stack!"')
stack.push('Told you, I am generic stack!')
print('Pushing integer 3')
stack.push(3)
print('Current stack:', stack.fullStack())
print('Popped item:', stack.pop())
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
```

Output:

```
Current stack: []
Stack empty?: True
Pushing integer 1
Pushing string "Told you, I am generic stack!"
Pushing integer 3
Current stack: [1, 'Told you, I am generic stack!', 3]
Popped item: 3
Current stack: [1, 'Told you, I am generic stack!']
Stack empty?: False
```

Section 134.2: Parsing Parentheses

Stacks are often used for parsing. A simple parsing task is to check whether a string of parentheses are matching.

For example, the string `([])` is matching, because the outer and inner brackets form pairs. `()<>)` is not matching, because the last `)` has no partner. `([)]` is also not matching, because pairs must be either entirely inside or outside other pairs.

```
def checkParenth(str):
    stack = Stack()
    pushChars, popChars = "<({[", ">)}]"
    for c in str:
        if c in pushChars:
            stack.push(c)
        elif c in popChars:
            if stack.isEmpty():
                return False
            else:
                stackTop = stack.pop()
                # Checks to see whether the opening bracket matches the closing one
                balancingBracket = pushChars[popChars.index(c)]
                if stackTop != balancingBracket:
                    return False
        else:
            return False

    return not stack.isEmpty()
```

Chapter 135: Working around the Global Interpreter Lock (GIL)

Section 135.1: Multiprocessing.Pool

The simple answer, when asking how to use threads in Python is: "Don't. Use processes, instead." The multiprocessing module lets you create processes with similar syntax to creating threads, but I prefer using their convenient Pool object.

Using the code that David Beazley first used to show the dangers of threads against the GIL, we'll rewrite it using multiprocessing.Pool:

David Beazley's code that showed GIL threading problems

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Re-written using multiprocessing.Pool:

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
    pool.join()

end = time.time()
print(end-start)
```

Instead of creating threads, this creates new processes. Since each process is its own interpreter, there are no GIL collisions. multiprocessing.Pool will open as many processes as there are cores on the machine, though in the example above, it would only need two. In a real-world scenario, you want to design your list to have at least as much length as there are processors on your machine. The Pool will run the function you tell it to run with each argument, up to the number of processes it creates. When the function finishes, any remaining functions in the list will be run on that process.

I've found that, even using the `with` statement, if you don't close and join the pool, the processes continue to exist.

To clean up resources, I always close and join my pools.

Section 135.2: Cython nogil:

Cython is an alternative python interpreter. It uses the GIL, but lets you disable it. See their documentation

As an example, using the code that David Beazley first used to show the dangers of threads against the GIL, we'll rewrite it using nogil:

David Beazley's code that showed GIL threading problems

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Re-written using nogil (ONLY WORKS IN CYTHON):

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

with nogil:
    t1 = Thread(target=countdown,args=(COUNT/2,))
    t2 = Thread(target=countdown,args=(COUNT/2,))
    start = time.time()
    t1.start();t2.start()
    t1.join();t2.join()

end = time.time()
print end-start
```

It's that simple, as long as you're using cython. Note that the documentation says you must make sure not to change any python objects:

> Code in the body of the statement must not manipulate Python objects in any way, and must not call anything that manipulates Python objects without first re-acquiring the GIL. Cython currently does not check this.

Chapter 136: Deployment

Section 136.1: Uploading a Conda Package

Before starting you must have:

Anaconda installed on your system Account on Binstar If you are not using Anaconda 1.6+ install the binstar command line client:

```
$ conda install binstar
$ conda update binstar
```

If you are not using Anaconda the Binstar is also available on pypi:

```
$ pip install binstar
```

Now we can login:

```
$ binstar login
```

Test your login with the whoami command:

```
$ binstar whoami
```

We are going to be uploading a package with a simple 'hello world' function. To follow along start by getting my demonstration package repo from Github:

```
$ git clone https://github.com/<NAME>/<Package>
```

This a small directory that looks like this:

```
package/
        setup.py
        test_package/
           __init__.py
           hello.py
           bld.bat
           build.sh
           meta.yaml
```

`Setup.py` is the standard python build file and `hello.py` has our single hello_world() function.

The `bld.bat`, `build.sh`, and `meta.yaml` are scripts and metadata for the `Conda` package. You can read the Conda build page for more info on those three files and their purpose.

Now we create the package by running:

```
$ conda build test_package/
```

That is all it takes to create a Conda package.

The final step is uploading to binstar by copying and pasting the last line of the print out after running the conda build test_package/ command. On my system the command is:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Since it is your first time creating a package and release you will be prompted to fill out some text fields which could alternatively be done through the web app.

You will see a *done* printed out to confirm you have successfully uploaded your Conda package to Binstar.

Chapter 137: Logging

Section 137.1: Introduction to Python Logging

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

So, let's start:

Example Configuration Directly in Code

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('this is a %s test', 'debug')
```

Output example:

```
2016-07-26 18:53:55,332 root DEBUG this is a debug test
```

Example Configuration via an INI File

Assuming the file is named logging_config.ini. More details for the file format are in the logging configuration section of the logging tutorial.

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
```

```
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Then use `logging.config.fileConfig()` in the code:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Example Configuration via a Dictionary

As of Python 2.7, you can use a dictionary with configuration details. PEP 391 contains a list of the mandatory and optional elements in the configuration dictionary.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
        },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
        },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Section 137.2: Logging exceptions

If you want to log exceptions you can and should make use of the `logging.exception(msg)` method:

```
>>> import logging
>>> logging.basicConfig()
>>> try:
...     raise Exception('foo')
... except:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Do not pass the exception as argument:

As `logging.exception(msg)` expects a `msg` arg, it is a common pitfall to pass the exception into the logging call like this:

```
>>> try:
...     raise Exception('foo')
... except Exception as e:
...     logging.exception(e)
...
ERROR:root:foo
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

While it might look as if this is the right thing to do at first, it is actually problematic due to the reason how exceptions and various encoding work together in the logging module:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception(e)
...
Traceback (most recent call last):
  File "/.../python2.7/logging/__init__.py", line 861, in emit
    msg = self.format(record)
  File "/.../python2.7/logging/__init__.py", line 734, in format
    return fmt.format(record)
  File "/.../python2.7/logging/__init__.py", line 469, in format
    s = self._fmt % record.__dict__
UnicodeEncodeError: 'ascii' codec can't encode characters in position 1-2: ordinal not in
range(128)
Logged from file <stdin>, line 4
```

Trying to log an exception that contains unicode chars, this way will fail miserably. It will hide the stacktrace of the original exception by overriding it with a new one that is raised during formatting of your `logging.exception(e)` call.

Obviously, in your own code, you might be aware of the encoding in exceptions. However, 3rd party libs might handle this in a different way.

Correct Usage:

If instead of the exception you just pass a message and let python do its magic, it will work:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

As you can see we don't actually use `e` in that case, the call to `logging.exception(...)` magically formats the most recent exception.

Logging exceptions with non ERROR log levels

If you want to log an exception with another log level than ERROR, you can use the `exc_info` argument of the default loggers:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Accessing the exception's message

Be aware that libraries out there might throw exceptions with messages as any of unicode or (utf-8 if you're lucky) byte-strings. If you really need to access an exception's text, the only reliable way, that will always work, is to use `repr(e)` or the `%r` string formatting:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
...
ERROR:root:received this exception: Exception(u'f\xf6\xf6',)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Chapter 138: Web Server Gateway Interface (WSGI)

Parameter	Details
start_response	A function used to process the start

Section 138.1: Server Object (Method)

Our server object is given an 'application' parameter which can be any callable application object (see other examples). It writes first the headers, then the body of data returned by our application to the system standard output.

```
import os, sys

def run(application):
    environ['wsgi.input']        = sys.stdin
    environ['wsgi.errors']       = sys.stderr

    headers_set = []
    headers_sent = []

    def write (data):
        """
        Writes header data from 'start_response()' as well as body data from 'response'
        to system standard output.
        """
        if not headers_set:
            raise AssertionError("write() before start_response()")

        elif not headers_sent:
            status, response_headers = headers_sent[:] = headers_set
            sys.stdout.write('Status: %s\r\n' % status)
            for header in response_headers:
                sys.stdout.write('%s: %s\r\n' % header)
            sys.stdout.write('\r\n')

        sys.stdout.write(data)
        sys.stdout.flush()

    def start_response(status, response_headers):
        """ Sets headers for the response returned by this server."""
        if headers_set:
            raise AssertionError("Headers already set!")

        headers_set[:] = [status, response_headers]
        return write

    # This is the most important piece of the 'server object'
    # Our result will be generated by the 'application' given to this method as a parameter
    result = application(environ, start_response)
    try:
        for data in result:
            if data:
                write(data)          # Body isn't empty send its data to 'write()'
        if not headers_sent:
            write('')                # Body is empty, send empty string to 'write()'
```

Chapter 139: Python Server Sent Events

Server Sent Events (SSE) is a unidirectional connection between a server and a client (usually a web browser) that allows the server to "push" information to the client. It is much like websockets and long polling. The main difference between SSE and websockets is that SSE is unidirectional, only the server can send info to the client, where as with websockets, both can send info to each other. SSE is typically considered to be much simpler to use/implement than websockets.

Section 139.1: Flask SSE

```
@route("/stream")
def stream():
    def event_stream():
        while True:
            if message_to_send:
                yield "data:
                    {}\n\n".format(message_to_send)"

    return Response(event_stream(), mimetype="text/event-stream")
```

Section 139.2: Asyncio SSE

This example uses the asyncio SSE library: https://github.com/brutasse/asyncio-sse

```
import asyncio
import sse

class Handler(sse.Handler):
    @asyncio.coroutine
    def handle_request(self):
        yield from asyncio.sleep(2)
        self.send('foo')
        yield from asyncio.sleep(2)
        self.send('bar', event='wakeup')

start_server = sse.serve(Handler, 'localhost', 8888)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Chapter 140: Alternatives to switch statement from other languages

Section 140.1: Use what the language offers: the if/else construct

Well, if you want a `switch/case` construct, the most straightforward way to go is to use the good old `if/else` construct:

```
def switch(value):
    if value == 1:
        return "one"
    if value == 2:
        return "two"
    if value == 42:
        return "the answer to the question about life, the universe and everything"
    raise Exception("No case found!")
```

it might look redundant, and not always pretty, but that's by far the most efficient way to go, and it does the job:

```
>>> switch(1)
one
>>> switch(2)
two
>>> switch(3)
...
Exception: No case found!
>>> switch(42)
the answer to the question about life the universe and everything
```

Section 140.2: Use a dict of functions

Another straightforward way to go is to create a dictionary of functions:

```
switch = {
    1: lambda: 'one',
    2: lambda: 'two',
    42: lambda: 'the answer of life the universe and everything',
}
```

then you add a default function:

```
def default_case():
    raise Exception('No case found!')
```

and you use the dictionary's get method to get the function given the value to check and run it. If value does not exists in dictionary, then `default_case` is run.

```
>>> switch.get(1, default_case)()
one
>>> switch.get(2, default_case)()
two
>>> switch.get(3, default_case)()
...
Exception: No case found!
```

```
>>> switch.get(42, default_case)()
the answer of life the universe and everything
```

you can also make some syntactic sugar so the switch looks nicer:

```
def run_switch(value):
    return switch.get(value, default_case)()

>>> run_switch(1)
one
```

Section 140.3: Use class introspection

You can use a class to mimic the switch/case structure. The following is using introspection of a class (using the `getattr()` function that resolves a string into a bound method on an instance) to resolve the "case" part.

Then that introspecting method is aliased to the `__call__` method to overload the `()` operator.

```
class SwitchBase:
    def switch(self, case):
        m = getattr(self, 'case_{}'.format(case), None)
        if not m:
            return self.default
        return m

    __call__ = switch
```

Then to make it look nicer, we subclass the `SwitchBase` class (but it could be done in one class), and there we define all the `case` as methods:

```
class CustomSwitcher:
    def case_1(self):
        return 'one'

    def case_2(self):
        return 'two'

    def case_42(self):
        return 'the answer of life, the universe and everything!'

    def default(self):
        raise Exception('Not a case!')
```

so then we can finally use it:

```
>>> switch = CustomSwitcher()
>>> print(switch(1))
one
>>> print(switch(2))
two
>>> print(switch(3))
...
Exception: Not a case!
>>> print(switch(42))
the answer of life, the universe and everything!
```

Section 140.4: Using a context manager

Another way, which is very readable and elegant, but far less efficient than an if/else structure, is to build a class such as follows, that will read and store the value to compare with, expose itself within the context as a callable that will return true if it matches the stored value:

```
class Switch:
    def __init__(self, value):
        self._val = value
    def __enter__(self):
        return self
    def __exit__(self, type, value, traceback):
        return False # Allows traceback to occur
    def __call__(self, cond, *mconds):
        return self._val in (cond,)+mconds
```

then defining the cases is almost a match to the real `switch`/`case` construct (exposed within a function below, to make it easier to show off):

```
def run_switch(value):
    with Switch(value) as case:
        if case(1):
            return 'one'
        if case(2):
            return 'two'
        if case(3):
            return 'the answer to the question about life, the universe and everything'
        # default
        raise Exception('Not a case!')
```

So the execution would be:

```
>>> run_switch(1)
one
>>> run_switch(2)
two
>>> run_switch(3)
...
Exception: Not a case!
>>> run_switch(42)
the answer to the question about life, the universe and everything
```

Nota Bene:

- This solution is being offered as the *switch* module available on pypi.

Chapter 141: List destructuring (aka packing and unpacking)

Section 141.1: Destructuring assignment

In assignments, you can split an Iterable into values using the "unpacking" syntax:

Destructuring as values

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

If you try to unpack more than the length of the iterable, you'll get an error:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x `Version > 3.0`

Destructuring as a list

You can unpack a list of unknown length using the following syntax:

```
head, *tail = [1, 2, 3, 4, 5]
```

Here, we extract the first value as a scalar, and the other values as a list:

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

Which is equivalent to:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

It also works with multiple elements or elements form the end of the list:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

Ignoring values in destructuring assignments

If you're only interested in a given value, you can use `_` to indicate you aren't interested. Note: this will still set `_`, just most people don't use it as a variable.

```
a, _ = [1, 2]
print(a)
# Prints: 1
a, _, c = (1, 2, 3)
print(a)
# Prints: 1
```

```
print(c)
# Prints: 3
```

Python 3.x `Version > 3.0`

Ignoring lists in destructuring assignments

Finally, you can ignore many values using the `*_` syntax in the assignment:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

which is not really interesting, as you could using indexing on the list instead. Where it gets nice is to keep first and last values in one assignment:

```
a, *_, b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

or extract several values at once:

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# Prints: 1 3 5
```

Section 141.2: Packing function arguments

In functions, you can define a number of mandatory arguments:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

which will make the function callable only when the three arguments are given:

```
fun1(1, 2, 3)
```

and you can define the arguments as optional, by using default values:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

so you can call the function in many different ways, like:

```
fun2(1)              → (1,b,c)
fun2(1, 2)           → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
...
```

But you can also use the destructuring syntax to *pack* arguments up, so you can assign variables using a `list` or a `dict`.

Packing a list of arguments

Consider you have a list of values

```
l = [1,2,3]
```

You can call the function with the list of values as an argument using the `*` syntax:

```
fun1(*l)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

But if you do not provide a list which length matches the number of arguments:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

Packing keyword arguments

Now, you can also pack arguments using a dictionary. You can use the `**` operator to tell Python to unpack the `dict` as parameter values:

```
d = {
  'arg1': 1,
  'arg2': 2,
  'arg3': 3
}
fun1(**d)
# Returns: (1, 2, 3)
```

when the function only has positional arguments (the ones without default values) you need the dictionary to be contain of all the expected parameters, and have no extra parameter, or you'll get an error:

```
fun1(**{'arg1':1, 'arg2':2})
# Raises: TypeError: fun1() missing 1 required positional argument: 'arg3'
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun1() got an unexpected keyword argument 'arg4'
```

For functions that have optional arguments, you can pack the arguments as a dictionary the same way:

```
fun2(**d)
# Returns: (1, 2, 3)
```

But there you can omit values, as they will be replaced with the defaults:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

And the same as before, you cannot give extra values that are not existing parameters:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

In real world usage, functions can have both positional and optional arguments, and it works the same:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

you can call the function with just an iterable:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

or with just a dictionary:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

or you can use both in the same call:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Beware though that you cannot provide multiple values for the same argument:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

Section 141.3: Unpacking function arguments

When you want to create a function that can accept any number of arguments, and not enforce the position or the name of the argument at "compile" time, it's possible and here's how:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

The `*args` and `**kwargs` parameters are special parameters that are set to a `tuple` and a `dict`, respectively:

```
fun1(1,2,3)
# Prints: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# Prints: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# Prints: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

If you look at enough Python code, you'll quickly discover that it is widely being used when passing arguments over to another function. For example if you want to extend the string class:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

Chapter 142: Accessing Python source code and bytecode

Section 142.1: Display the bytecode of a function

The Python interpreter compiles code to bytecode before executing it on the Python's virtual machine (see also What is python bytecode?.

Here's how to view the bytecode of a Python function

```
import dis

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)

# Display the disassembled bytecode of the function.
dis.dis(fib)
```

The function `dis.dis` in the dis module will return a decompiled bytecode of the function passed to it.

Section 142.2: Display the source code of an object

Objects that are not built-in

To print the source code of a Python object use `inspect`. Note that this won't work for built-in objects nor for objects defined interactively. For these you will need other methods explained later.

Here's how to print the source code of the method `randint` from the `random` module:

```
import random
import inspect

print(inspect.getsource(random.randint))
# Output:
#    def randint(self, a, b):
#        """Return random integer in range [a, b], including both end points.
#        """
#
#        return self.randrange(a, b+1)
```

To just print the documentation string

```
print(inspect.getdoc(random.randint))
# Output:
# Return random integer in range [a, b], including both end points.
```

Print full path of the file where the method `random.randint` is defined:

```
print(inspect.getfile(random.randint))
# c:\Python35\lib\random.py
print(random.randint.__code__.co_filename) # equivalent to the above
# c:\Python35\lib\random.py
```

Objects defined interactively

If an object is defined interactively `inspect` cannot provide the source code but you can use `dill.source.getsource` instead

```
# define a new function in the interactive shell
def add(a, b):
   return a + b
print(add.__code__.co_filename) # Output: <stdin>

import dill
print dill.source.getsource(add)
# def add(a, b):
      return a + b
```

Built-in objects

The source code for Python's built-in functions is written in **c** and can only be accessed by looking at the Python's source code (hosted on Mercurial or downloadable from https://www.python.org/downloads/source/).

```
print(inspect.getsource(sorted)) # raises a TypeError
type(sorted) # <class 'builtin_function_or_method'>
```

Section 142.3: Exploring the code object of a function

CPython allows access to the code object for a function object.

The `__code__`object contains the raw bytecode (`co_code`) of the function as well as other information such as constants and variable names.

```
def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)
```

Chapter 143: Mixins

Section 143.1: Mixin

A **Mixin** is a set of properties and methods that can be used in different classes, which *don't* come from a base class. In Object Oriented Programming languages, you typically use *inheritance* to give objects of different classes the same functionality; if a set of objects have some ability, you put that ability in a base class that both objects *inherit* from.

> For instance, say you have the classes `Car`, `Boat`, and `Plane`. Objects from all of these classes have the ability to travel, so they get the function `travel`. In this scenario, they all travel the same basic way, too; by getting a route, and moving along it. To implement this function, you could derive all of the classes from `Vehicle`, and put the function in that shared class:
>
> ```
> class Vehicle(object):
>     """A generic vehicle class."""
>
>     def __init__(self, position):
>         self.position = position
>
>     def travel(self, destination):
>         route = calculate_route(from=self.position, to=destination)
>         self.move_along(route)
>
> class Car(Vehicle):
>     ...
>
> class Boat(Vehicle):
>     ...
>
> class Plane(Vehicle):
>     ...
> ```
>
> With this code, you can call `travel` on a car (`car.travel("Montana")`), boat (`boat.travel("Hawaii")`), and plane (`plane.travel("France")`)

However, what if you have functionality that's not available to a base class? Say, for instance, you want to give `Car` a radio and the ability to use it to play a song on a radio station, with `play_song_on_station`, but you also have a `Clock` that can use a radio too. `Car` and `Clock` could share a base class (`Machine`). However, not all machines can play songs; `Boat` and `Plane` can't (at least in this example). So how do you accomplish without duplicating code? You can use a mixin. In Python, giving a class a mixin is as simple as adding it to the list of subclasses, like this

```
class Foo(main_super, mixin): ...
```

`Foo` will inherit all of the properties and methods of `main_super`, but also those of `mixin` as well.

> So, to give the classes `Car` and clock the ability to use a radio, you could override `Car` from the last example and write this:
>
> ```
> class RadioUserMixin(object):
>     def __init__(self):
>         self.radio = Radio()
> ```

```
    def play_song_on_station(self, station):
        self.radio.set_station(station)
        self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
    ...

class Clock(Vehicle, RadioUserMixin):
    ...
```

Now you can call `car.play_song_on_station(98.7)` and `clock.play_song_on_station(101.3)`, but not something like `boat.play_song_on_station(100.5)`

The important thing with mixins is that they allow you to add functionality to much different objects, that don't share a "main" subclass with this functionality but still share the code for it nonetheless. Without mixins, doing something like the above example would be much harder, and/or might require some repetition.

Section 143.2: Overriding Methods in Mixins

Mixins are a sort of class that is used to "mix in" extra properties and methods into a class. This is usually fine because many times the mixin classes don't override each other's, or the base class' methods. But if you do override methods or properties in your mixins this can lead to unexpected results because in Python the class hierarchy is defined right to left.

For instance, take the following classes

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class BaseClass(object):
    def test(self):
        print "Base"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

In this case the Mixin2 class is the base class, extended by Mixin1 and finally by BaseClass. Thus, if we execute the following code snippet:

```
>>> x = MyClass()
>>> x.test()
Base
```

We see the result returned is from the Base class. This can lead to unexpected errors in the logic of your code and needs to be accounted for and kept in mind

Chapter 144: Attribute Access

Section 144.1: Basic Attribute Access using the Dot Notation

Let's take a sample class.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

In Python you can access the attribute *title* of the class using the dot notation.

```
>>> book1.title
'P.G. Wodehouse'
```

If an attribute doesn't exist, Python throws an error:

```
>>> book1.series
Traceback  (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Book' object has no attribute 'series'
```

Section 144.2: Setters, Getters & Properties

For the sake of data encapsulation, sometimes you want to have an attribute which value comes from other attributes or, in general, which value shall be computed at the moment. The standard way to deal with this situation is to create a method, called getter or a setter.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

In the example above, it's easy to see what happens if we create a new Book that contains a title and a author. If all books we're to add to our Library have authors and titles, then we can skip the getters and setters and use the dot notation. However, suppose we have some books that do not have an author and we want to set the author to "Unknown". Or if they have multiple authors and we plan to return a list of authors.

In this case we can create a getter and a setter for the *author* attribute.

```
class P:
    def __init__(self,title,author):
        self.title = title
        self.setAuthor(author)

    def get_author(self):
        return self.author

    def set_author(self, author):
        if not author:
            self.author = "Unknown"
        else:
```

```
            self.author = author
```

This scheme is not recommended.

One reason is that there is a catch: Let's assume we have designed our class with the public attribute and no methods. People have already used it a lot and they have written code like this:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
```

Now we have a problem. Because *author* is not an attribute! Python offers a solution to this problem called properties. A method to get properties is decorated with the @property before it's header. The method that we want to function as a setter is decorated with @attributeName.setter before it.

Keeping this in mind, we now have our new updated class.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Note, normally Python doesn't allow you to have multiple methods with the same name and different number of parameters. However, in this case Python allows this because of the decorators used.

If we test the code:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
>>> book.author
Unknown
```

Chapter 145: ArcPy

Section 145.1: createDissolvedGDB to create a file gdb on the workspace

```
def createDissolvedGDB(workspace, gdbName):
    gdb_name = workspace + "/" + gdbName + ".gdb"

    if(arcpy.Exists(gdb_name):
        arcpy.Delete_management(gdb_name)
        arcpy.CreateFileGDB_management(workspace, gdbName, "")
    else:
        arcpy.CreateFileGDB_management(workspace, gdbName, "")

    return gdb_name
```

Section 145.2: Printing one field's value for all rows of feature class in file geodatabase using Search Cursor

To print a test field (TestField) from a test feature class (TestFC) in a test file geodatabase (Test.gdb) located in a temporary folder (C:\Temp):

```
with arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC",["TestField"]) as cursor:
    for row in cursor:
        print row[0]
```

Chapter 146: Abstract Base Classes (abc)

Section 146.1: Setting the ABCMeta metaclass

Abstract classes are classes that are meant to be inherited but avoid implementing specific methods, leaving behind only method signatures that subclasses must implement.

Abstract classes are useful for defining and enforcing class abstractions at a high level, similar to the concept of interfaces in typed languages, without the need for method implementation.

One conceptual approach to defining an abstract class is to stub out the class methods, and then raise a NotImplementedError if accessed. This prevents children classes from accessing parent methods without overriding them first. Like so:

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("check_ripeness method not implemented!")


class Apple(Fruit):
    pass


a = Apple()
a.check_ripeness() # raises NotImplementedError
```

Creating an abstract class in this way prevents improper usage of methods that are not overridden, and certainly encourages methods to be defined in child classes, but it does not enforce their definition. With the `abc` module we can prevent child classes from being instantiated when they fail to override abstract class methods of their parents and ancestors:

```
from abc import ABCMeta

class AbstractClass(object):
    # the metaclass attribute must always be set as a class variable
    __metaclass__ = ABCMeta

   # the abstractmethod decorator registers this method as undefined
   @abstractmethod
   def virtual_method_subclasses_must_define(self):
       # Can be left completely blank, or a base implementation can be provided
       # Note that ordinarily a blank interpretation implicitly returns `None`,
       # but by registering, this behaviour is no longer enforced.
```

It is now possible to simply subclass and override:

```
class Subclass(AbstractClass):
    def virtual_method_subclasses_must_define(self):
        return
```

Section 146.2: Why/How to use ABCMeta and @abstractmethod

Abstract base classes (ABCs) enforce what derived classes implement particular methods from the base class.

To understand how this works and why we should use it, let's take a look at an example that Van Rossum would enjoy. Let's say we have a Base class "MontyPython" with two methods (joke & punchline) that must be implemented by all derived classes.

```
class MontyPython:
    def joke(self):
        raise NotImplementedError()

    def punchline(self):
        raise NotImplementedError()

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

When we instantiate an object and call it's two methods, we'll get an error (as expected) with the `punchline()` method.

```
 >>> sketch = ArgumentClinic()
 >>> sketch.punchline()
NotImplementedError
```

However, this still allows us to instantiate an object of the ArgumentClinic class without getting an error. In fact we don't get an error until we look for the punchline().

This is avoided by using the Abstract Base Class (ABC) module. Let's see how this works with the same example:

```
from abc import ABCMeta, abstractmethod

class MontyPython(metaclass=ABCMeta):
    @abstractmethod
    def joke(self):
        pass

@abstractmethod
def punchline(self):
    pass

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

This time when we try to instantiate an object from the incomplete class, we immediately get a TypeError!

```
>>> c = ArgumentClinic()
TypeError:
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

In this case, it's easy to complete the class to avoid any TypeErrors:

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"

    def punchline(self):
        return "Send in the constable!"
```

This time when you instantiate an object it works!

Chapter 147: Plugin and Extension Classes

Section 147.1: Mixins

In Object oriented programming language, a mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language.

It provides a mechanism for multiple inheritance by allowing multiple classes to use the common functionality, but without the complex semantics of multiple inheritance. Mixins are useful when a programmer wants to share functionality between different classes. Instead of repeating the same code over and over again, the common functionality can simply be grouped into a mixin and then inherited into each class that requires it.

When we use more than one mixins, Order of mixins are important. here is a simple example:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

In this example we call `MyClass` and `test` method,

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Result must be Mixin1 because Order is left to right. This could be show unexpected results when super classes add with it. So reverse order is more good just like this:

```
class MyClass(Mixin2, Mixin1):
    pass
```

Result will be:

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

Mixins can be used to define custom plugins.

Python 3.x `Version ≥ 3.0`

```
class Base(object):
    def test(self):
        print("Base.")

class PluginA(object):
    def test(self):
        super().test()
        print("Plugin A.")
```

```
class PluginB(object):
    def test(self):
        super().test()
        print("Plugin B.")

plugins = PluginA, PluginB

class PluginSystemA(PluginA, Base):
    pass

class PluginSystemB(PluginB, Base):
    pass

PluginSystemA().test()
# Base.
# Plugin A.

 PluginSystemB().test()
# Base.
# Plugin B.
```

Section 147.2: Plugins with Customized Classes

In Python 3.6, PEP 487 added the `__init_subclass__` special method, which simplifies and extends class customization without using metaclasses. Consequently, this feature allows for creating simple plugins. Here we demonstrate this feature by modifying a prior example:

Python 3.x `Version ≥ 3.6`

```
class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    def test(self):
        print("Base.")

class PluginA(Base):
    def test(self):
        super().test()
        print("Plugin A.")


class PluginB(Base):
    def test(self):
        super().test()
        print("Plugin B.")
```

Results:

```
PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
```

```
# [__main__.PluginA, __main__.PluginB]
```

Chapter 148: Immutable datatypes(int, float, str, tuple and frozensets)

Section 148.1: Individual characters of strings are not assignable

```
foo = "bar"
foo[0] = "c" # Error
```

Immutable variable value can not be changed once they are created.

Section 148.2: Tuple's individual members aren't assignable

```
foo = ("bar", 1, "Hello!",)
foo[1] = 2 # ERROR!!
```

Second line would return an error since tuple members once created aren't assignable. Because of tuple's immutability.

Section 148.3: Frozenset's are immutable and not assignable

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

Second line would return an error since frozenset members once created aren't assignable. Third line would return error as frozensets do not support functions that can manipulate members.

Chapter 149: Incompatibilities moving from Python 2 to Python 3

Unlike most languages, Python supports two major versions. Since 2008 when Python 3 was released, many have made the transition, while many have not. In order to understand both, this section covers the important differences between Python 2 and Python 3.

Section 149.1: Integer Division

The standard **division symbol** (`/`) operates differently in Python 3 and Python 2 when applied to integers.

When dividing an integer by another integer in Python 3, the division operation `x / y` represents a **true division** (uses `__truediv__` method) and produces a floating point result. Meanwhile, the same operation in Python 2 represents a **classic division** that rounds the result down toward negative infinity (also known as taking the *floor*).

For example:

Code	Python 2 output	Python 3 output
`3 / 2`	1	1.5
`2 / 3`	0	0.6666666666666666
`-3 / 2`	-2	-1.5

The rounding-towards-zero behavior was deprecated in Python 2.2, but remains in Python 2.7 for the sake of backward compatibility and was removed in Python 3.

Note: To get a *float* result in Python 2 (without floor rounding) we can specify one of the operands with the decimal point. The above example of `2/3` which gives `0` in Python 2 shall be used as `2 / 3.0` or `2.0 / 3` or `2.0/3.0` to get `0.6666666666666666`

Code	Python 2 output	Python 3 output
`3.0 / 2.0`	1.5	1.5
`2 / 3.0`	0.6666666666666666	0.6666666666666666
`-3.0 / 2`	-1.5	-1.5

There is also the **floor division operator** (`//`), which works the same way in both versions: it rounds down to the nearest integer. (although a float is returned when used with floats) In both versions the `//` operator maps to `__floordiv__`.

Code	Python 2 output	Python 3 output
`3 // 2`	1	1
`2 // 3`	0	0
`-3 // 2`	-2	-2
`3.0 // 2.0`	1.0	1.0
`2.0 // 3`	0.0	0.0
`-3 // 2.0`	-2.0	-2.0

One can explicitly enforce true division or floor division using native functions in the `operator` module:

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25           # equivalent to `/` in Python 3
assert floordiv(10, 8) == 1             # equivalent to `//`
```

While clear and explicit, using operator functions for every division can be tedious. Changing the behavior of the `/` operator will often be preferred. A common practice is to eliminate typical division behavior by adding `from __future__ import division` as the first statement in each module:

```
# needs to be the first statement in a module
from __future__ import division
```

Code	Python 2 output	Python 3 output
`3 / 2`	1.5	1.5
`2 / 3`	0.6666666666666666	0.6666666666666666
`-3 / 2`	-1.5	-1.5

`from __future__ import division` guarantees that the `/` operator represents true division and only within the modules that contain the `__future__` import, so there are no compelling reasons for not enabling it in all new modules.

Note: Some other programming languages use *rounding toward zero* (truncation) rather than *rounding down toward negative infinity* as Python does (i.e. in those languages `-3 / 2 == -1`). This behavior may create confusion when porting or comparing code.

Note on float operands: As an alternative to `from __future__ import division`, one could use the usual division symbol `/` and ensure that at least one of the operands is a float: `3 / 2.0 == 1.5`. However, this can be considered bad practice. It is just too easy to write `average = sum(items) / len(items)` and forget to cast one of the arguments to float. Moreover, such cases may frequently evade notice during testing, e.g., if you test on an array containing `float`s but receive an array of `int`s in production. Additionally, if the same code is used in Python 3, programs that expect `3 / 2 == 1` to be True will not work correctly.

See PEP 238 for more detailed rationale why the division operator was changed in Python 3 and why old-style division should be avoided.

See the *Simple Math* topic for more about division.

Section 149.2: Unpacking Iterables

Python 3.x `Version ≥ 3.0`

In Python 3, you can unpack an iterable without knowing the exact number of items in it, and even have a variable hold the end of the iterable. For that, you provide a variable that may collect a list of values. This is done by placing an asterisk before the name. For example, unpacking a `list`:

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)
# Out: 1
print(second)
# Out: 2
print(tail)
# Out: [3, 4]
print(last)
# Out: 5
```

Note: When using the `*variable` syntax, the `variable` will always be a list, even if the original type wasn't a list. It may contain zero or more elements depending on the number of elements in the original list.

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)
```

```
# Out: [3]

first, second, *tail, last = [1, 2, 3]
print(tail)
# Out: []
print(last)
# Out: 3
```

Similarly, unpacking a `str`:

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
# Out: ['e', 'l', 'l', 'o']
```

Example of unpacking a `date`; `_` is used in this example as a throwaway variable (we are interested only in `year` value):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

It is worth mentioning that, since `*` eats up a variable number of items, you cannot have two `*`s *for the same iterable* in an assignment - it wouldn't know how many elements go into the first unpacking, and how many in the second:

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x `Version ≥ 3.5`

So far we have discussed unpacking in assignments. `*` and `**` were extended in Python 3.5. It's now possible to have several unpacking operations in one expression:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x `Version ≥ 2.0`

It is also possible to unpack an iterable into function arguments:

```
iterable = [1, 2, 3, 4, 5]
print(iterable)
# Out: [1, 2, 3, 4, 5]
print(*iterable)
# Out: 1 2 3 4 5
```

Python 3.x `Version ≥ 3.5`

Unpacking a dictionary uses two adjacent stars `**` (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
 # Out: {'x': 1, 'y': 2, 'z': 3}
```

This allows for both overriding old values and merging dictionaries.

```
dict1 = {'x': 1, 'y': 1}
```

```
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

Python 3.x Version ≥ 3.0

Python 3 removed tuple unpacking in functions. Hence the following doesn't work in Python 3

```
# Works in Python 2, but syntax error in Python 3:
map(lambda (x, y): x + y, zip(range(5), range(5)))
# Same is true for non-lambdas:
def example((x, y)):
    pass

# Works in both Python 2 and Python 3:
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# And non-lambdas, too:
def working_example(x_y):
    x, y = x_y
    pass
```

See PEP 3113 for detailed rationale.

Section 149.3: Strings: Bytes versus Unicode

Python 2.x Version ≤ 2.7

In Python 2 there are two variants of string: those made of bytes with type (`str`) and those made of text with type (`unicode`).

In Python 2, an object of type `str` is always a byte sequence, but is commonly used for both text and binary data.

A string literal is interpreted as a byte string.

```
s = 'Cafe'    # type(s) == str
```

There are two exceptions: You can define a *Unicode (text) literal* explicitly by prefixing the literal with `u`:

```
s = u'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == str
```

Alternatively, you can specify that a whole module's string literals should create Unicode (text) literals:

```
from __future__ import unicode_literals

s = 'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == unicode
```

In order to check whether your variable is a string (either Unicode or a byte string), you can use:

```
isinstance(s, basestring)
```

Python 3.x Version ≥ 3.0

In Python 3, the `str` type is a Unicode text type.

```
s = 'Cafe'           # type(s) == str
```

```
s = 'Café'           # type(s) == str (note the accented trailing e)
```

Additionally, Python 3 added a bytes object, suitable for binary "blobs" or writing to encoding-independent files. To create a bytes object, you can prefix `b` to a string literal or call the string's `encode` method:

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode()  # type(s) == bytes
```

To test whether a value is a string, use:

```
isinstance(s, str)
```

Python 3.x `Version ≥ 3.3`

It is also possible to prefix string literals with a `u` prefix to ease compatibility between Python 2 and Python 3 code bases. Since, in Python 3, all strings are Unicode by default, prepending a string literal with `u` has no effect:

```
u'Cafe' == 'Cafe'
```

Python 2's raw Unicode string prefix `ur` is not supported, however:

```
>>> ur'Café'
  File "<stdin>", line 1
    ur'Café'
            ^
SyntaxError: invalid syntax
```

Note that you must `encode` a Python 3 text (`str`) object to convert it into a `bytes` representation of that text. The default encoding of this method is UTF-8.

You can use `decode` to ask a `bytes` object for what Unicode text it represents:

```
>>> b.decode()
'Café'
```

Python 2.x `Version ≥ 2.6`

While the `bytes` type exists in both Python 2 and 3, the `unicode` type only exists in Python 2. To use Python 3's implicit Unicode strings in Python 2, add the following to the top of your code file:

```
from __future__ import unicode_literals
print(repr("hi"))
# u'hi'
```

Python 3.x `Version ≥ 3.0`

Another important difference is that indexing bytes in Python 3 results in an `int` output like so:

```
b"abc"[0] == 97
```

Whilst slicing in a size of one results in a length 1 bytes object:

```
b"abc"[0:1] == b"a"
```

In addition, Python 3 fixes some unusual behavior with unicode, i.e. reversing byte strings in Python 2. For example, the following issue is resolved:

```
# -*- coding: utf8 -*-
print("Hi, my name is Łukasz Langa.")
print(u"Hi, my name is Łukasz Langa."[::-1])
print("Hi, my name is Łukasz Langa."[::-1])

# Output in Python 2
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsaku<U+FFFD><U+FFFD> si eman ym ,iH

# Output in Python 3
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsakuŁ si eman ym ,iH
```

Section 149.4: Print statement vs. Print function

In Python 2, `print` is a statement:

Python 2.x `Version ≤ 2.7`

```
print "Hello World"
print                           # print a newline
print "No newline",             # add trailing comma to remove newline
print >>sys.stderr, "Error"     # print to stderr
print("hello")                  # print "hello", since ("hello") == "hello"
print()                         # print an empty tuple "()"
print 1, 2, 3                   # print space-separated arguments: "1 2 3"
print(1, 2, 3)                  # print tuple "(1, 2, 3)"
```

In Python 3, `print()` is a function, with keyword arguments for common uses:

Python 3.x `Version ≥ 3.0`

```
print "Hello World"              # SyntaxError
print("Hello World")
print()                         # print a newline (must use parentheses)
print("No newline", end="")      # end specifies what to append (defaults to newline)
print("Error", file=sys.stderr)  # file specifies the output buffer
print("Comma", "separated", "output", sep=",")  # sep specifies the separator
print("A", "B", "C", sep="")     # null string for sep: prints as ABC
print("Flush this", flush=True)  # flush the output buffer, added in Python 3.3
print(1, 2, 3)                  # print space-separated arguments: "1 2 3"
print((1, 2, 3))                # print tuple "(1, 2, 3)"
```

The print function has the following parameters:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` is what separates the objects you pass to print. For example:

```
print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar
```

`end` is what the end of the print statement is followed by. For example:

```
print('foo', 'bar', end='!') # out: foo bar!
```

Printing again following a non-newline ending print statement *will* print to the same line:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

Note: For future compatibility, `print` *function* is also available in Python 2.6 onwards; however it cannot be used unless parsing of the `print` *statement* is disabled with

```
from __future__ import print_function
```

This function has exactly same format as Python 3's, except that it lacks the `flush` parameter.

See PEP 3105 for rationale.

Section 149.5: Differences between range and xrange functions

In Python 2, `range` function returns a list while `xrange` creates a special `xrange` object, which is an immutable sequence, which unlike other built-in sequence types, doesn't support slicing and has neither `index` nor `count` methods:

Python 2.x `Version ≥ 2.3`

```
print(range(1, 10))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(isinstance(range(1, 10), list))
# Out: True

print(xrange(1, 10))
# Out: xrange(1, 10)

print(isinstance(xrange(1, 10), xrange))
# Out: True
```

In Python 3, `xrange` was expanded to the `range` sequence, which thus now creates a `range` object. There is no `xrange` type:

Python 3.x `Version ≥ 3.0`

```
print(range(1, 10))
# Out: range(1, 10)

print(isinstance(range(1, 10), range))
# Out: True

# print(xrange(1, 10))
# The output will be:
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#NameError: name 'xrange' is not defined
```

Additionally, since Python 3.2, `range` also supports slicing, `index` and `count`:

```
print(range(1, 10)[3:7])
# Out: range(3, 7)
print(range(1, 10).count(5))
# Out: 1
```

```
print(range(1, 10).index(7))
# Out: 6
```

The advantage of using a special sequence type instead of a list is that the interpreter does not have to allocate memory for a list and populate it:

Python 2.x Version ≥ 2.3

```
# range(1000000000000000)
# The output would be:
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(100000000000000000))
# Out: xrange(100000000000000000)
```

Since the latter behaviour is generally desired, the former was removed in Python 3. If you still want to have a list in Python 3, you can simply use the `list()` constructor on a `range` object:

Python 3.x Version ≥ 3.0

```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Compatibility

In order to maintain compatibility between both Python 2.x and Python 3.x versions, you can use the `builtins` module from the external package `future` to achieve both *forward-compatibility* and *backward-compatibility*:

Python 2.x Version ≥ 2.0

```
#forward-compatible
from builtins import range

for i in range(10**8):
    pass
```

Python 3.x Version ≥ 3.0

```
#backward-compatible
from past.builtins import xrange

for i in xrange(10**8):
    pass
```

The `range` in `future` library supports slicing, `index` and `count` in all Python versions, just like the built-in method on Python 3.2+.

Section 149.6: Raising and handling Exceptions

This is the Python 2 syntax, note the commas `,` on the `raise` and `except` lines:

Python 2.x Version ≥ 2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

In Python 3, the `,` syntax is dropped and replaced by parenthesis and the `as` keyword:

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

For backwards compatibility, the Python 3 syntax is also available in Python 2.6 onwards, so it should be used for all new code that does not need to be compatible with previous versions.

Python 3.x `Version ≥ 3.0`

Python 3 also adds exception chaining, wherein you can signal that some other exception was the *cause* for this exception. For example

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

The exception raised in the `except` statement is of type `DatabaseError`, but the original exception is marked as the `__cause__` attribute of that exception. When the traceback is displayed, the original exception will also be displayed in the traceback:

```
Traceback (most recent call last):
File "", line 2, in
FileNotFoundError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
File "", line 4, in
DatabaseError('Cannot open database.db')
```

If you throw in an `except` block *without* explicit chaining:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

The traceback is

```
Traceback (most recent call last):
File "", line 2, in
FileNotFoundError

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
File "", line 4, in
DatabaseError('Cannot open database.db')
```

Python 2.x `Version ≥ 2.0`

Neither one is supported in Python 2.x; the original exception and its traceback will be lost if another exception is raised in the except block. The following code can be used for compatibility:

```
import sys
import traceback

try:
    funcWithError()
except:
    sys_vers = getattr(sys, 'version_info', (0,))
    if sys_vers < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")
```

Python 3.x Version ≥ 3.3

To "forget" the previously thrown exception, use `raise from None`

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from None
```

Now the traceback would simply be

```
Traceback (most recent call last):
File "", line 4, in
DatabaseError('Cannot open database.db')
```

Or in order to make it compatible with both Python 2 and 3 you may use the six package like so:

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

Section 149.7: Leaked variables in list comprehension

Python 2.x Version ≥ 2.3

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'U'
```

Python 3.x Version ≥ 3.0

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'hello world!'
```

As can be seen from the example, in Python 2 the value of `x` was leaked: it masked `hello world!` and printed out `U`, since this was the last value of `x` when the loop ended.

However, in Python 3 `x` prints the originally defined `hello world!`, since the local variable from the list

comprehension does not mask variables from the surrounding scope.

Additionally, neither generator expressions (available in Python since 2.5) nor dictionary or set comprehensions (which were backported to Python 2.7 from Python 3) leak variables in Python 2.

Note that in both Python 2 and Python 3, variables will leak into the surrounding scope when using a for loop:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Out: 'U'
```

Section 149.8: True, False and None

In Python 2, `True`, `False` and `None` are built-in constants. Which means it's possible to reassign them.

Python 2.x `Version ≥ 2.0`

```
True, False = False, True
True   # False
False  # True
```

You can't do this with `None` since Python 2.4.

Python 2.x `Version ≥ 2.4`

```
None = None  # SyntaxError: cannot assign to None
```

In Python 3, `True`, `False`, and `None` are now keywords.

Python 3.x `Version ≥ 3.0`

```
True, False = False, True  # SyntaxError: can't assign to keyword

None = None  # SyntaxError: can't assign to keyword
```

Section 149.9: User Input

In Python 2, user input is accepted using the `raw_input` function,

Python 2.x `Version ≥ 2.3`

```
user_input = raw_input()
```

While in Python 3 user input is accepted using the `input` function.

Python 3.x `Version ≥ 3.0`

```
user_input = input()
```

In Python 2, the `input` function will accept input and *interpret* it. While this can be useful, it has several security considerations and was removed in Python 3. To access the same functionality, `eval(input())` can be used.

To keep a script portable across the two versions, you can put the code below near the top of your Python script:

```
try:
    input = raw_input
except NameError:
```

```
    pass
```

Section 149.10: Comparison of different types

Python 2.x `Version ≥ 2.3`

Objects of different types can be compared. The results are arbitrary, but consistent. They are ordered such that `None` is less than anything else, numeric types are smaller than non-numeric types, and everything else is ordered lexicographically by type. Thus, an `int` is less than a `str` and a `tuple` is greater than a `list`:

```
[1, 2] > 'foo'
# Out: False
(1, 2) > 'foo'
# Out: True
[1, 2] > (1, 2)
# Out: False
100 < [1, 'x'] < 'xyz' < (1, 'x')
# Out: True
```

This was originally done so a list of mixed types could be sorted and objects would be grouped together by type:

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']
sorted(l)
# Out: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Python 3.x `Version ≥ 3.0`

An exception is raised when comparing different (non-numeric) types:

```
1 < 1.5
# Out: True

[1, 2] > 'foo'
# TypeError: unorderable types: list() > str()
(1, 2) > 'foo'
# TypeError: unorderable types: tuple() > str()
[1, 2] > (1, 2)
# TypeError: unorderable types: list() > tuple()
```

To sort mixed lists in Python 3 by types and to achieve compatibility between versions, you have to provide a key to the sorted function:

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]
>>> sorted(list, key=str)
# Out: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

Using `str` as the `key` function temporarily converts each item to a string only for the purposes of comparison. It then sees the string representation starting with either `[`, `'`, `{` or `0-9` and it's able to sort those (and all the following characters).

Section 149.11: .next() method on iterators renamed

In Python 2, an iterator can be traversed by using a method called `next` on the iterator itself:

Python 2.x `Version ≥ 2.3`

```
g = (i for i in range(0, 3))
g.next()  # Yields 0
```

```
g.next()  # Yields 1
g.next()  # Yields 2
```

In Python 3 the `.next` method has been renamed to `.__next__`, acknowledging its "magic" role, so calling `.next` will raise an `AttributeError`. The correct way to access this functionality in both Python 2 and Python 3 is to call the `next` *function* with the iterator as an argument.

Python 3.x `Version ≥ 3.0`

```
g = (i for i in range(0, 3))
next(g)  # Yields 0
next(g)  # Yields 1
next(g)  # Yields 2
```

This code is portable across versions from 2.6 through to current releases.

Section 149.12: filter(), map() and zip() return iterators instead of sequences

Python 2.x `Version ≤ 2.7`

In Python 2 `filter`, `map` and `zip` built-in functions return a sequence. `map` and `zip` always return a list while with `filter` the return type depends on the type of given parameter:

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> s
'abc'
>>> s = map(lambda x: x * x, [0, 1, 2])
>>> s
[0, 1, 4]
>>> s = zip([0, 1, 2], [3, 4, 5])
>>> s
[(0, 3), (1, 4), (2, 5)]
```

Python 3.x `Version ≥ 3.0`

In Python 3 `filter`, `map` and `zip` return iterator instead:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> it
<filter object at 0x00000098A55C2518>
>>> ''.join(it)
'abc'
>>> it = map(lambda x: x * x, [0, 1, 2])
>>> it
<map object at 0x000000E0763C2D30>
>>> list(it)
[0, 1, 4]
>>> it = zip([0, 1, 2], [3, 4, 5])
>>> it
<zip object at 0x000000E0763C52C8>
>>> list(it)
[(0, 3), (1, 4), (2, 5)]
```

Since Python 2 `itertools.izip` is equivalent of Python 3 `zip` `izip` has been removed on Python 3.

Section 149.13: Renamed modules

A few modules in the standard library have been renamed:

Old name	New name
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
_markupbase	markupbase
repr	reprlib
test.test_support	test.support
Tkinter	tkinter
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Some modules have even been converted from files to libraries. Take tkinter and urllib from above as an example.

Compatibility

When maintaining compatibility between both Python 2.x and 3.x versions, you can use the `future` external package to enable importing top-level standard library packages with Python 3.x names on Python 2.x versions.

Section 149.14: Removed operators <> and ``, synonymous with != and repr()

In Python 2, `<>` is a synonym for `!=`; likewise, `` `foo` `` is a synonym for `repr(foo)`.

Python 2.x `Version ≤ 2.7`

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"'hello world'"
>>> `foo`
"'hello world'"
```

Python 3.x `Version ≥ 3.0`

```
>>> 1 <> 2
  File "<stdin>", line 1
    1 <> 2
       ^
SyntaxError: invalid syntax
>>> `foo`
  File "<stdin>", line 1
    `foo`
    ^
SyntaxError: invalid syntax
```

Section 149.15: long vs. int

In Python 2, any integer larger than a C `ssize_t` would be converted into the `long` data type, indicated by an `L` suffix on the literal. For example, on a 32 bit build of Python:

Python 2.x `Version ≤ 2.7`

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
<type 'int'>
>>> 2**31 - 1  # 2**31 is long and long - int is long
2147483647L
```

However, in Python 3, the `long` data type was removed; no matter how big the integer is, it will be an `int`.

Python 3.x `Version ≥ 3.0`

```
2**1024
# Output:
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602
11201138798713933576587897688144166224928474306394741243777678934248654852763022196012460941194530829
52085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224
137216
print(-(2**1024))
# Output:
-17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602
11201138798713933576587897688144166224928474306394741243777678934248654852763022196012460941194530829
52085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224
137216
type(2**1024)
# Output: <class 'int'>
```

Section 149.16: All classes are "new-style classes" in Python 3

In Python `3.x` all classes are *new-style classes*; when defining a new class python implicitly makes it inherit from `object`. As such, specifying `object` in a `class` definition is a completely optional:

Python 3.x `Version ≥ 3.0`

```
class X: pass
class Y(object): pass
```

Both of these classes now contain `object` in their `mro` (method resolution order):

Python 3.x `Version ≥ 3.0`

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

In Python `2.x` classes are, by default, old-style classes; they do not implicitly inherit from `object`. This causes the semantics of classes to differ depending on if we explicitly add `object` as a base `class`:

Python 2.x `Version ≥ 2.3`

```
class X: pass
class Y(object): pass
```

In this case, if we try to print the `__mro__` of `Y`, similar output as that in the Python `3.x` case will appear:

Python 2.x `Version ≥ 2.3`

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

This happens because we explicitly made `Y` inherit from object when defining it: `class Y(object): pass`. For class `X` which does *not* inherit from object the `__mro__` attribute does not exist, trying to access it results in an `AttributeError`.

In order to **ensure compatibility** between both versions of Python, classes can be defined with `object` as a base class:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

Alternatively, if the `__metaclass__` variable is set to `type` at global scope, all subsequently defined classes in a given module are implicitly new-style without needing to explicitly inherit from `object`:

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

Section 149.17: Reduce is no longer a built-in

In Python 2, `reduce` is available either as a built-in function or from the `functools` package (version 2.6 onwards), whereas in Python 3 `reduce` is available only from `functools`. However the syntax for `reduce` in both Python2 and Python3 is the same and is `reduce(function_to_reduce, list_to_reduce)`.

As an example, let us consider reducing a list to a single value by dividing each of the adjacent numbers. Here we use `truediv` function from the `operator` library.

In Python 2.x it is as simple as:

Python 2.x `Version ≥ 2.3`

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

In Python 3.x the example becomes a bit more complicated:

Python 3.x `Version ≥ 3.0`

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

We can also use `from functools import reduce` to avoid calling `reduce` with the namespace name.

Section 149.18: Absolute/Relative Imports

In Python 3, PEP 404 changes the way imports work from Python 2. *Implicit relative* imports are no longer allowed in packages and `from ... import *` imports are only allowed in module level code.

To achieve Python 3 behavior in Python 2:

- the absolute imports feature can be enabled with `from __future__ import absolute_import`
- *explicit relative* imports are encouraged in place of *implicit relative* imports

For clarification, in Python 2, a module can import the contents of another module located in the same directory as follows:

```
import foo
```

Notice the location of `foo` is ambiguous from the import statement alone. This type of implicit relative import is thus discouraged in favor of explicit relative imports, which look like the following:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

The dot `.` allows an explicit declaration of the module location within the directory tree.

More on Relative Imports

Consider some user defined package called `shapes`. The directory structure is as follows:

```
shapes
├── __init__.py
|
├── circle.py
|
├── square.py
|
└── triangle.py
```

`circle.py`, `square.py` and `triangle.py` all import `util.py` as a module. How will they refer to a module in the same level?

```
from . import util # use util.PI, util.sq(x), etc
```

OR

```
from .util import * #use PI, sq(x), etc to call functions
```

The `.` is used for same-level relative imports.

Now, consider an alternate layout of the `shapes` module:

```
shapes
├── __init__.py
|
├── circle
|   ├── __init__.py
|   └── circle.py
|
```

```
├── square
│   ├── __init__.py
│   └── square.py
│
├── triangle
│   ├── __init__.py
│   ├── triangle.py
│
└── util.py
```

Now, how will these 3 classes refer to util.py?

```
from .. import util # use util.PI, util.sq(x), etc
```

OR

```
from ..util import * # use PI, sq(x), etc to call functions
```

The `..` is used for parent-level relative imports. Add more `.`s with number of levels between the parent and child.

Section 149.19: map()

`map()` is a builtin that is useful for applying a function to elements of an iterable. In Python 2, `map` returns a list. In Python 3, `map` returns a *map object*, which is a generator.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x*>
>>> type(_)
<class 'map'>

# We need to apply map again because we "consumed" the previous map....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

In Python 2, you can pass `None` to serve as an identity function. This no longer works in Python 3.

Python 2.x `Version ≥ 2.3`

```
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

Python 3.x `Version ≥ 3.0`

```
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

Moreover, when passing more than one iterable as argument in Python 2, `map` pads the shorter iterables with `None` (similar to `itertools.izip_longest`). In Python 3, iteration stops after the shortest iterable.

In Python 2:

Python 2.x Version ≥ 2.3

```
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

In Python 3:

Python 3.x Version ≥ 3.0

```
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]

# to obtain the same padding as in Python 2 use zip_longest from itertools
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

Note: instead of `map` consider using list comprehensions, which are Python 2/3 compatible. Replacing `map(str, [1, 2, 3, 4, 5])`:

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

Section 149.20: The round() function tie-breaking and return type

round() tie breaking

In Python 2, using `round()` on a number equally close to two integers will return the one furthest from 0. For example:

Python 2.x Version ≤ 2.7

```
round(1.5)  # Out: 2.0
round(0.5)  # Out: 1.0
round(-0.5)  # Out: -1.0
round(-1.5)  # Out: -2.0
```

In Python 3 however, `round()` will return the even integer (aka *bankers' rounding*). For example:

Python 3.x Version ≥ 3.0

```
round(1.5)  # Out: 2
round(0.5)  # Out: 0
round(-0.5)  # Out: 0
round(-1.5)  # Out: -2
```

The round() function follows the *half to even rounding* strategy that will round half-way numbers to the nearest even integer (for example, `round(2.5)` now returns 2 rather than 3.0).

As per reference in Wikipedia, this is also known as *unbiased rounding*, *convergent rounding*, *statistician's rounding*, *Dutch rounding*, *Gaussian rounding*, or *odd-even rounding*.

Half to even rounding is part of the IEEE 754 standard and it's also the default rounding mode in Microsoft's .NET.

This rounding strategy tends to reduce the total rounding error. Since on average the amount of numbers that are rounded up is the same as the amount of numbers that are rounded down, rounding errors cancel out. Other

rounding methods instead tend to have an upwards or downwards bias in the average error.

round() return type

The `round()` function returns a `float` type in Python 2.7

Python 2.x `Version ≤ 2.7`

```
round(4.8)
# 5.0
```

Starting from Python 3.0, if the second argument (number of digits) is omitted, it returns an `int`.

Python 3.x `Version ≥ 3.0`

```
round(4.8)
# 5
```

Section 149.21: File I/O

`file` is no longer a builtin name in 3.x (`open` still works).

Internal details of file I/O have been moved to the standard library `io` module, which is also the new home of `StringIO`:

```
import io
assert io.open is open # the builtin is an alias
buffer = io.StringIO()
buffer.write('hello, ') # returns number of characters written
buffer.write('world!\n')
buffer.getvalue() # 'hello, world!\n'
```

The file mode (text vs binary) now determines the type of data produced by reading a file (and type required for writing):

```
with open('data.txt') as f:
    first_line = next(f)
    assert type(first_line) is str
with open('data.bin', 'rb') as f:
    first_kb = f.read(1024)
    assert type(first_kb) is bytes
```

The encoding for text files defaults to whatever is returned by `locale.getpreferredencoding(False)`. To specify an encoding explicitly, use the `encoding` keyword parameter:

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

Section 149.22: cmp function removed in Python 3

In Python 3 the `cmp` built-in function was removed, together with the `__cmp__` special method.

From the documentation:

> The `cmp()` function should be treated as gone, and the `__cmp__()` special method is no longer supported.

> Use `__lt__()` for sorting, `__eq__()` with `__hash__()`, and other rich comparisons as needed. (If you really need the `cmp()` functionality, you could use the expression `(a > b) - (a < b)` as the equivalent for `cmp(a, b)`.)

Moreover all built-in functions that accepted the `cmp` parameter now only accept the `key` keyword only parameter.

In the `functools` module there is also useful function `cmp_to_key(func)` that allows you to convert from a `cmp`-style function to a `key`-style function:

> Transform an old-style comparison function to a key function. Used with tools that accept key functions (such as `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.

Section 149.23: Octal Constants

In Python 2, an octal literal could be defined as

>>> 0755 # only Python 2

To ensure cross-compatibility, use

0o755 # both Python 2 and Python 3

Section 149.24: Return value when writing to a file object

In Python 2, writing directly to a file handle returns `None`:

Python 2.x `Version ≥ 2.3`

```
hi = sys.stdout.write('hello world\\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

In Python 3, writing to a handle will return the number of characters written when writing text, and the number of bytes written when writing bytes:

Python 3.x `Version ≥ 3.0`

```
import sys

char_count = sys.stdout.write('hello world ?\\n')
# Out: hello world ?
char_count
# Out: 14

byte_count = sys.stdout.buffer.write(b'hello world \\xf0\\x9f\\x90\\x8d\\n')
# Out: hello world ?
byte_count
# Out: 17
```

Section 149.25: exec statement is a function in Python 3

In Python 2, `exec` is a statement, with special syntax: `exec code [in globals[, locals]]`. In Python 3 `exec` is now

a function: `exec(code, [, globals[, locals]])`, and the Python 2 syntax will raise a `SyntaxError`.

As `print` was changed from statement into a function, a `__future__` import was also added. However, there is no `from __future__ import exec_function`, as it is not needed: the exec statement in Python 2 can be also used with syntax that looks exactly like the `exec` function invocation in Python 3. Thus you can change the statements

Python 2.x `Version ≥ 2.3`

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

to forms

Python 3.x `Version ≥ 3.0`

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

and the latter forms are guaranteed to work identically in both Python 2 and Python 3.

Section 149.26: encode/decode to hex no longer available

Python 2.x `Version ≤ 2.7`

```
"1deadbeef3".decode('hex')
# Out: '\x1d\xea\xdb\xee\xf3'
'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Out: 1deadbeef3
```

Python 3.x `Version ≥ 3.0`

```
"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'str' object has no attribute 'decode'

b"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.decode() to handle arbitrary codecs

'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.encode() to handle arbitrary codecs

b'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# AttributeError: 'bytes' object has no attribute 'encode'
```

However, as suggested by the error message, you can use the `codecs` module to achieve the same result:

```
import codecs
codecs.decode('1deadbeef4', 'hex')
# Out: b'\x1d\xea\xdb\xee\xf4'
codecs.encode(b'\x1d\xea\xdb\xee\xf4', 'hex')
# Out: b'1deadbeef4'
```

Note that `codecs.encode` returns a `bytes` object. To obtain a `str` object just `decode` to ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')
# Out: '1deadbeeff'
```

Section 149.27: Dictionary method changes

In Python 3, many of the dictionary methods are quite different in behaviour from Python 2, and many were removed as well: `has_key`, `iter*` and `view*` are gone. Instead of `d.has_key(key)`, which had been long deprecated, one must now use `key in d`.

In Python 2, dictionary methods `keys`, `values` and `items` return lists. In Python 3 they return *view* objects instead; the view objects are not iterators, and they differ from them in two ways, namely:

- they have size (one can use the `len` function on them)
- they can be iterated over many times

Additionally, like with iterators, the changes in the dictionary are reflected in the view objects.

Python 2.7 has backported these methods from Python 3; they're available as `viewkeys`, `viewvalues` and `viewitems`. To transform Python 2 code to Python 3 code, the corresponding forms are:

- `d.keys()`, `d.values()` and `d.items()` of Python 2 should be changed to `list(d.keys())`, `list(d.values())` and `list(d.items())`
- `d.iterkeys()`, `d.itervalues()` and `d.iteritems()` should be changed to `iter(d.keys())`, or even better, `iter(d)`; `iter(d.values())` and `iter(d.items())` respectively
- and finally Python 2.7 method calls `d.viewkeys()`, `d.viewvalues()` and `d.viewitems()` can be replaced with `d.keys()`, `d.values()` and `d.items()`.

Porting Python 2 code that *iterates* over dictionary keys, values or items while mutating it is sometimes tricky. Consider:

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

The code looks as if it would work similarly in Python 3, but there the `keys` method returns a view object, not a list, and if the dictionary changes size while being iterated over, the Python 3 code will crash with `RuntimeError: dictionary changed size during iteration`. The solution is of course to properly write `for key in list(d)`.

Similarly, view objects behave differently from iterators: one cannot use `next()` on them, and one cannot *resume* iteration; it would instead restart; if Python 2 code passes the return value of `d.iterkeys()`, `d.itervalues()` or `d.iteritems()` to a method that expects an iterator instead of an *iterable*, then that should be `iter(d)`, `iter(d.values())` or `iter(d.items())` in Python 3.

Section 149.28: Class Boolean Value

Python 2.x `Version ≤ 2.7`

In Python 2, if you want to define a class boolean value by yourself, you need to implement the `__nonzero__` method on your class. The value is True by default.

```
class MyClass:
    def __nonzero__(self):
        return False
```

```
my_instance = MyClass()
print bool(MyClass)       # True
print bool(my_instance)   # False
```

Python 3.x `Version ≥ 3.0`

In Python 3, `__bool__` is used instead of `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False

my_instance = MyClass()
print(bool(MyClass))       # True
print(bool(my_instance))   # False
```

Section 149.29: hasattr function bug in Python 2

In Python 2, when a property raise an error, `hasattr` will ignore this property, returning `False`.

```
class A(object):
    @property
    def get(self):
        raise IOError


class B(object):
    @property
    def get(self):
        return 'get in b'

a = A()
b = B()

print 'a hasattr get: ', hasattr(a, 'get')
# output False in Python 2 (fixed, True in Python 3)
print 'b hasattr get', hasattr(b, 'get')
# output True in Python 2 and Python 3
```

This bug is fixed in Python3. So if you use Python 2, use

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

or use `getattr` instead

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

Chapter 150: 2to3 tool

Parameter	Description
filename / directory_name	2to3 accepts a list of files or directories which is to be transformed as its argument. The directories are recursively traversed for Python sources.

Option	Option Description
-f FIX, --fix=FIX	Specify transformations to be applied; default: all. List available transformations with `--list-fixes`
-j PROCESSES, --processes=PROCESSES	Run 2to3 concurrently
-x NOFIX, --nofix=NOFIX	Exclude a transformation
-l, --list-fixes	List available transformations
-p, --print-function	Change the grammar so that `print()` is considered a function
-v, --verbose	More verbose output
--no-diffs	Do not output diffs of the refactoring
-w	Write back modified files
-n, --nobackups	Do not create backups of modified files
-o OUTPUT_DIR, --output-dir=OUTPUT_DIR	Place output files in this directory instead of overwriting input files. Requires the `-n` flag, as backup files are unnecessary when the input files are not modified.
-W, --write-unchanged-files	Write output files even is no changes were required. Useful with `-o` so that a complete source tree is translated and copied. Implies `-w`.
--add-suffix=ADD_SUFFIX	Specify a string to be appended to all output filenames. Requires `-n` if non-empty. Ex.: `--add-suffix='3'` will generate `.py3` files.

Section 150.1: Basic Usage

Consider the following Python2.x code. Save the file as `example.py`

Python 2.x `Version ≥ 2.0`

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

In the above file, there are several incompatible lines. The `raw_input()` method has been replaced with `input()` in Python 3.x and `print` is no longer a statement, but a function. This code can be converted to Python 3.x code using the 2to3 tool.

Unix

```
$ 2to3 example.py
```

Windows

```
> path/to/2to3.py example.py
```

Running the above code will output the differences against the original source file as shown below.

```
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
```

```
RefactoringTool: Refactored example.py
--- example.py    (original)
+++ example.py    (refactored)
@@ -1,5 +1,5 @@
 def greet(name):
-    print "Hello, {0}!".format(name)
-print "What's your name?"
-name = raw_input()
+    print("Hello, {0}!".format(name))
+print("What's your name?")
+name = input()
 greet(name)
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

The modifications can be written back to the source file using the -w flag. A backup of the original file called `example.py.bak` is created, unless the -n flag is given.

Unix

```
$ 2to3 -w example.py
```

Windows

```
> path/to/2to3.py -w example.py
```

Now the `example.py` file has been converted from Python 2.x to Python 3.x code.

Once finished, `example.py` will contain the following valid Python3.x code:

Python 3.x `Version ≥ 3.0`

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Chapter 151: Non-official Python implementations

Section 151.1: IronPython

Open-source implementation for .NET and Mono written in C#, licensed under Apache License 2.0. It relies on DLR (Dynamic Language Runtime). It supports only version 2.7, version 3 is currently being developed.

Differences with CPython:

- Tight integration with .NET Framework.
- Strings are Unicode by default.
- Does not support extensions for CPython written in C.
- Does not suffer from Global Interpreter Lock.
- Performance is usually lower, though it depends on tests.

Hello World

```
print "Hello World!"
```

You can also use .NET functions:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

External links

- Official website
- GitHub repository

Section 151.2: Jython

Open-source implementation for JVM written in Java, licensed under Python Software Foundation License. It supports only version 2.7, version 3 is currently being developed.

Differences with CPython:

- Tight integration with JVM.
- Strings are Unicode.
- Does not support extensions for CPython written in C.
- Does not suffer from Global Interpreter Lock.
- Performance is usually lower, though it depends on tests.

Hello World

```
print "Hello World!"
```

You can also use Java functions:

```
from java.lang import System
System.out.println("Hello World!")
```

External links

- Official website
- Mercurial repository

Section 151.3: Transcrypt

Transcrypt is a tool to precompile a fairly extensive subset of Python into compact, readable Javascript. It has the following characteristics:

- Allows for classical OO programming with multiple inheritance using pure Python syntax, parsed by CPython's native parser
- Seamless integration with the universe of high-quality web-oriented JavaScript libraries, rather than the desktop-oriented Python ones
- Hierarchical URL based module system allowing module distribution via PyPi
- Simple relation between Python source and generated JavaScript code for easy debugging
- Multi-level sourcemaps and optional annotation of target code with source references
- Compact downloads, kB's rather than MB's
- Optimized JavaScript code, using memoization (call caching) to optionally bypass the prototype lookup chain
- Operator overloading can be switched on and off locally to facilitate readable numerical math

Code size and speed

Experience has shown that 650 kB of Python sourcecode roughly translates in the same amount of JavaScript source code. The speed matches the speed of handwritten JavaScript and can surpass it if call memoizing is switched on.

Integration with HTML

```
<script src="__javascript__/hello.js"></script>
<h2>Hello demo</h2>

<p>
<div id = "greet">...</div>
<button onclick="hello.solarSystem.greet ()">Click me repeatedly!</button>

<p>
<div id = "explain">...</div>
<button onclick="hello.solarSystem.explain ()">And click me repeatedly too!</button>
```

Integration with JavaScript and DOM

```
from itertools import chain

class SolarSystem:
    planets = [list (chain (planet, (index + 1,))) for index, planet in enumerate ((
        ('Mercury', 'hot', 2240),
        ('Venus', 'sulphurous', 6052),
        ('Earth', 'fertile', 6378),
        ('Mars', 'reddish', 3397),
        ('Jupiter', 'stormy', 71492),
        ('Saturn', 'ringed', 60268),
        ('Uranus', 'cold', 25559),
        ('Neptune', 'very cold', 24766)
    ))]

    lines = (
        '{} is a {} planet',
        'The radius of {} is {} km',
        '{} is planet nr. {} counting from the sun'
    )

    def __init__ (self):
        self.lineIndex = 0
```

```
    def greet (self):
        self.planet = self.planets [int (Math.random () * len (self.planets))]
        document.getElementById ('greet') .innerHTML = 'Hello {}'.format (self.planet [0])
        self.explain ()

    def explain (self):
        document.getElementById ('explain').innerHTML = (
            self.lines [self.lineIndex] .format (self.planet [0], self.planet [self.lineIndex + 1])
        )
        self.lineIndex = (self.lineIndex + 1) % 3
         solarSystem = SolarSystem ()
```

Integration with other JavaScript libraries

Transcrypt can be used in combination with any JavaScript library without special measures or syntax. In the documentation examples are given for a.o. react.js, riot.js, fabric.js and node.js.

Relation between Python and JavaScript code

Python

```
class A:
    def __init__ (self, x):
        self.x = x

    def show (self, label):
        print ('A.show', label, self.x)

class B:
    def __init__ (self, y):
        alert ('In B constructor')
        self.y = y

    def show (self, label):
        print ('B.show', label, self.y)

class C (A, B):
    def __init__ (self, x, y):
        alert ('In C constructor')
        A.__init__ (self, x)
        B.__init__ (self, y)
        self.show ('constructor')

    def show (self, label):
        B.show (self, label)
        print ('C.show', label, self.x, self.y)

a = A (1001)
a.show ('america')

b = B (2002)
b.show ('russia')

c = C (3003, 4004)
c.show ('netherlands')

show2 = c.show
show2 ('copy')
```

JavaScript

```
var A = __class__ ('A', [object], {
    get __init__ () {return __get__ (this, function (self, x) {
        self.x = x;
    });},
    get show () {return __get__ (this, function (self, label) {
        print ('A.show', label, self.x);
    });}
});
var B = __class__ ('B', [object], {
    get __init__ () {return __get__ (this, function (self, y) {
        alert ('In B constructor');
        self.y = y;
    });},
    get show () {return __get__ (this, function (self, label) {
        print ('B.show', label, self.y);
    });}
});
var C = __class__ ('C', [A, B], {
    get __init__ () {return __get__ (this, function (self, x, y) {
        alert ('In C constructor');
        A.__init__ (self, x);
        B.__init__ (self, y);
        self.show ('constructor');
    });},
    get show () {return __get__ (this, function (self, label) {
        B.show (self, label);
        print ('C.show', label, self.x, self.y);
    });}
});
var a = A (1001);
a.show ('america');
var b = B (2002);
b.show ('russia');
var c = C (3003, 4004);
c.show ('netherlands');
var show2 = c.show;
show2 ('copy');
```

External links

- Official website: http://www.transcrypt.org/
- Repository: https://github.com/JdeH/Transcrypt

Chapter 152: Abstract syntax tree

Section 152.1: Analyze functions in a python script

This analyzes a python script and, for each defined function, reports the line number where the function began, where the signature ends, where the docstring ends, and where the function definition ends.

```
#!/usr/local/bin/python3

import ast
import sys

""" The data we collect.  Each key is a function name; each value is a dict
with keys: firstline, sigend, docend, and lastline and values of line numbers
where that happens. """
functions = {}

def process(functions):
    """ Handle the function data stored in functions. """
    for funcname,data in functions.items():
        print("function:",funcname)
        print("\tstarts at line:",data['firstline'])
        print("\tsignature ends at line:",data['sigend'])
        if ( data['sigend'] < data['docend'] ):
            print("\tdocstring ends at line:",data['docend'])
        else:
            print("\tno docstring")
        print("\tfunction ends at line:",data['lastline'])
        print()

class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        """ Recursively visit all functions, determining where each function
        starts, where its signature ends, where the docstring ends, and where
        the function ends. """
        functions[node.name] = {'firstline':node.lineno}
        sigend = max(node.lineno,lastline(node.args))
        functions[node.name]['sigend'] = sigend
        docstring = ast.get_docstring(node)
        docstringlength = len(docstring.split('\n')) if docstring else -1
        functions[node.name]['docend'] = sigend+docstringlength
        functions[node.name]['lastline'] = lastline(node)
        self.generic_visit(node)

def lastline(node):
    """ Recursively find the last line of a node """
    return max( [ node.lineno if hasattr(node,'lineno') else -1 , ]
                +[lastline(child) for child in ast.iter_child_nodes(node)] )

def readin(pythonfilename):
    """ Read the file name and store the function data into functions. """
    with open(pythonfilename) as f:
        code = f.read()
    FuncLister().visit(ast.parse(code))

def analyze(file,process):
    """ Read the file and process the function data. """
    readin(file)
    process(functions)
```

```
if __name__ == '__main__':
    if len(sys.argv)>1:
        for file in sys.argv[1:]:
            analyze(file,process)
    else:
        analyze(sys.argv[0],process)
```

Chapter 153: Unicode and bytes

Parameter	Details
encoding	The encoding to use, e.g. `'ascii'`, `'utf8'`, etc...
errors	The errors mode, e.g. `'replace'` to replace bad characters with question marks, `'ignore'` to ignore bad characters, etc...

Section 153.1: Encoding/decoding error handling

`.encode` and `.decode` both have error modes.

The default is `'strict'`, which raises exceptions on error. Other modes are more forgiving.

Encoding

```
>>> "£13.55".encode('ascii', errors='replace')
b'?13.55'
>>> "£13.55".encode('ascii', errors='ignore')
b'13.55'
>>> "£13.55".encode('ascii', errors='namereplace')
b'\\N{POUND SIGN}13.55'
>>> "£13.55".encode('ascii', errors='xmlcharrefreplace')
b'&#163;13.55'
>>> "£13.55".encode('ascii', errors='backslashreplace')
b'\\xa313.55'
```

Decoding

```
>>> b = "£13.55".encode('utf8')
>>> b.decode('ascii', errors='replace')
'��13.55'
>>> b.decode('ascii', errors='ignore')
'13.55'
>>> b.decode('ascii', errors='backslashreplace')
'\\xc2\\xa313.55'
```

Morale

It is clear from the above that it is vital to keep your encodings straight when dealing with unicode and bytes.

Section 153.2: File I/O

Files opened in a non-binary mode (e.g. `'r'` or `'w'`) deal with strings. The default encoding is `'utf8'`.

```
open(fn, mode='r')                    # opens file for reading in utf8
open(fn, mode='r', encoding='utf16')  # opens file for reading utf16

# ERROR: cannot write bytes when a string is expected:
open("foo.txt", "w").write(b"foo")
```

Files opened in a binary mode (e.g. `'rb'` or `'wb'`) deal with bytes. No encoding argument can be specified as there is no encoding.

```
open(fn, mode='wb')  # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

Section 153.3: Basics

In Python 3 `str` is the type for unicode-enabled strings, while `bytes` is the type for sequences of raw bytes.

```
type("f") == type(u"f")  # True, <class 'str'>
type(b"f")               # <class 'bytes'>
```

In Python 2 a casual string was a sequence of raw bytes by default and the unicode string was every string with "u" prefix.

```
type("f") == type(b"f")  # True, <type 'str'>
type(u"f")               # <type 'unicode'>
```

Unicode to bytes

Unicode strings can be converted to bytes with `.encode(encoding)`.

Python 3

```
>>> "£13.55".encode('utf8')
b'\xc2\xa313.55'
>>> "£13.55".encode('utf16')
b'\xff\xfe\xa3\x001\x003\x00.\x005\x005\x00'
```

Python 2

in py2 the default console encoding is `sys.getdefaultencoding() == 'ascii'` and not `utf-8` as in py3, therefore printing it as in the previous example is not directly possible.

```
>>> print type(u"£13.55".encode('utf8'))
<type 'str'>
>>> print u"£13.55".encode('utf8')
SyntaxError: Non-ASCII character '\xc2' in...

# with encoding set inside a file

# -*- coding: utf-8 -*-
>>> print u"£13.55".encode('utf8')
┬ú13.55
```

If the encoding can't handle the string, a `UnicodeEncodeError` is raised:

```
>>> "£13.55".encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xa3' in position 0: ordinal not in
range(128)
```

Bytes to unicode

Bytes can be converted to unicode strings with `.decode(encoding)`.

A sequence of bytes can only be converted into a unicode string via the appropriate encoding!

```
>>> b'\xc2\xa313.55'.decode('utf8')
```

```
'£13.55'
```

If the encoding can't handle the string, a `UnicodeDecodeError` is raised:

```
>>> b'\xc2\xa313.55'.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/csaftoiu/csaftoiu-github/yahoo-groups-
backup/.virtualenv/bin/../lib/python3.5/encodings/utf_16.py", line 16, in decode
    return codecs.utf_16_decode(input, errors, True)
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x35 in position 6: truncated data
```

Chapter 154: Python Serial Communication (pyserial)

parameter	details
port	Device name e.g. /dev/ttyUSB0 on GNU/Linux or COM3 on Windows.
baudrate	baudrate type: int default: 9600 standard values: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

Section 154.1: Initialize serial device

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

Section 154.2: Read from serial port

Initialize serial device

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

to read single byte from serial device

```
data = ser.read()
```

to read given number of bytes from the serial device

```
data = ser.read(size=5)
```

to read one line from serial device.

```
data = ser.readline()
```

to read the data from serial device while something is being written over it.

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

Section 154.3: Check what serial ports are available on your machine

To get a list of available serial ports use

```
python -m serial.tools.list_ports
```

at a command prompt or

```
from serial.tools import list_ports
```

```
list_ports.comports()  # Outputs list of available serial ports
```

from the Python shell.

Chapter 155: Neo4j and Cypher using Py2Neo

Section 155.1: Adding Nodes to Neo4j Graph

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    article.properties["title"] = results[r]['news_title']
    article.properties["timestamp"] = results[r]['news_timestamp']
    article.push()
    [...]
```

Adding nodes to the graph is pretty simple,`graph.merge_one` is important as it prevents duplicate items. (If you run the script twice, then the second time it would update the title and not create new nodes for the same articles)

`timestamp` should be an integer and not a date string as neo4j doesn't really have a date datatype. This causes sorting issues when you store date as '05-06-1989'

`article.push()` is an the call that actually commits the operation into neo4j. Don't forget this step.

Section 155.2: Importing and Authenticating

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

You have to make sure your Neo4j Database exists at localhost:7474 with the appropriate credentials.

the `graph` object is your interface to the neo4j instance in the rest of your python code. Rather than making this a global variable, you should keep it in a class's `__init__` method.

Section 155.3: Adding Relationships to Neo4j Graph

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    if 'LOCATION' in results[r].keys():
        for loc in results[r]['LOCATION']:
            loc = graph.merge_one("Location", "name", loc)
            try:
                rel = graph.create_unique(Relationship(article, "about_place", loc))
            except Exception, e:
                print e
```

`create_unique` is important for avoiding duplicates. But otherwise it's a pretty straightforward operation. The relationship name is also important as you would use it in advanced cases.

Section 155.4: Query 1 : Autocomplete on News Titles

```
def get_autocomplete(text):
    query = """
    start n = node(*) where n.name =~ '(?i)%s.*' return n.name,labels(n) limit 10;
    """
```

```
    query = query % (text)
    obj = []
    for res in graph.cypher.execute(query):
        # print res[0],res[1]
        obj.append({'name':res[0],'entity_type':res[1]})
    return res
```

This is a sample cypher query to get all nodes with the property `name` that starts with the argument `text`.

Section 155.5: Query 2 : Get News Articles by Location on a particular date

```
def search_news_by_entity(location,timestamp):
    query = """
    MATCH (n)-[]->(l)
    where l.name='%s' and n.timestamp='%s'
    RETURN n.news_id limit 10
    """

    query = query % (location,timestamp)

    news_ids = []
    for res in graph.cypher.execute(query):
        news_ids.append(str(res[0]))

    return news_ids
```

You can use this query to find all news articles `(n)` connected to a location `(l)` by a relationship.

Section 155.6: Cypher Query Samples

Count articles connected to a particular person over time

```
MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date
```

Search for other People / Locations connected to the same news articles as Trump with at least 5 total relationship nodes.

```
MATCH (n:NewsArticle)-[]->(l)
where l.name='Donald Trump'
MATCH (n:NewsArticle)-[]->(m)
with m,count(n) as num where num>5
return labels(m)[0],(m.name), num order by num desc limit 10
```

Chapter 156: Basic Curses with Python

Section 156.1: The wrapper() helper function

While the basic invocation above is easy enough, the curses package provides the `wrapper(func, ...)` helper function. The example below contains the equivalent of above:

```
main(scr, *args):
    # -- Perform an action with Screen --
    scr.border(0)
    scr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    scr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = scr.getch()
        if ch == ord('q'):

curses.wrapper(main)
```

Here, wrapper will initialize curses, create `stdscr`, a WindowObject and pass both stdscr, and any further arguments to `func`. When `func` returns, `wrapper` will restore the terminal before the program exits.

Section 156.2: Basic Invocation Example

```
import curses
import traceback

try:
    # -- Initialize --
    stdscr = curses.initscr()   # initialize curses screen
    curses.noecho()             # turn off auto echoing of keypress on to screen
    curses.cbreak()             # enter break mode where pressing Enter key
                                #  after keystroke is not required for it to register
    stdscr.keypad(1)            # enable special Key values such as curses.KEY_LEFT etc

    # -- Perform an action with Screen --
    stdscr.border(0)
    stdscr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    stdscr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = stdscr.getch()
        if ch == ord('q'):
            break

    # -- End of user code --

except:
    traceback.print_exc()     # print trace back log of the error

finally:
    # --- Cleanup on exit ---
    stdscr.keypad(0)
    curses.echo()
    curses.nocbreak()
    curses.endwin()
```

Chapter 157: Templates in python

Section 157.1: Simple data output program using template

```
from string import Template

data = dict(item = "candy", price = 8, qty = 2)

# define the template
t = Template("Simon bought $qty $item for $price dollar")
print(t.substitute(data))
```

Output:

```
Simon bought 2 candy for 8 dollar
```

Templates support $-based substitutions instead of %-based substitution. **Substitute** (mapping, keywords) performs template substitution, returning a new string.

Mapping is any dictionary-like object with keys that match with the template placeholders. In this example, price and qty are placeholders. Keyword arguments can also be used as placeholders. Placeholders from keywords take precedence if both are present.

Section 157.2: Changing delimiter

You can change the "$" delimiter to any other. The following example:

```
from string import Template

class MyOtherTemplate(Template):
    delimiter = "#"


data = dict(id = 1, name = "Ricardo")
t = MyOtherTemplate("My name is #name and I have the id: #id")
print(t.substitute(data))
```

You can read de docs here

Chapter 158: Pillow

Section 158.1: Read Image File

```
from PIL import Image

im = Image.open("Image.bmp")
```

Section 158.2: Convert files to JPEG

```
from __future__ import print_function
import os, sys
from PIL import Image

for infile in sys.argv[1:]:
    f, e = os.path.splitext(infile)
    outfile = f + ".jpg"
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print("cannot convert", infile)
```

Chapter 159: The pass statement

Section 159.1: Ignore an exception

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

Section 159.2: Create a new Exception that can be caught

```
class CompileError(Exception):
    pass
```

Chapter 160: CLI subcommands with precise help output

Different ways to create subcommands like in `hg` or `svn` with the exact command line interface and help output as shown in Remarks section.

Parsing Command Line arguments covers broader topic of arguments parsing.

Section 160.1: Native way (no libraries)

```
"""
usage: sub <command>

commands:

  status -  show status
  list   -  print list
"""

import sys

def check():
    print("status")
    return 0

if sys.argv[1:] == ['status']:
    sys.exit(check())
elif sys.argv[1:] == ['list']:
    print("list")
else:
    print(__doc__.strip())
```

Output without arguments:

```
usage: sub

commands:

status - show status
list - print list
```

Pros:

- no deps
- everybody should be able to read that
- complete control over help formatting

Section 160.2: argparse (default help formatter)

```
import argparse
import sys

def check():
    print("status")
    return 0
```

```
parser = argparse.ArgumentParser(prog="sub", add_help=False)
subparser = parser.add_subparsers(dest="cmd")

subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())
```

Output without arguments:

```
usage: sub {status,list} ...

positional arguments:
  {status,list}
    status       show status
    list         print list
```

Pros:

- comes with Python
- option parsing is included

Section 160.3: argparse (custom help formatter)

```
import argparse
import sys

class CustomHelpFormatter(argparse.HelpFormatter):
    def _format_action(self, action):
        if type(action) == argparse._SubParsersAction:
            # inject new class variable for subcommand formatting
            subactions = action._get_subactions()
            invocations = [self._format_action_invocation(a) for a in subactions]
            self._subcommand_max_length = max(len(i) for i in invocations)

        if type(action) == argparse._SubParsersAction._ChoicesPseudoAction:
            # format subcommand help line
            subcommand = self._format_action_invocation(action) # type: str
            width = self._subcommand_max_length
            help_text = ""
            if action.help:
                help_text = self._expand_help(action)
            return "  {:{width}} -  {}\n".format(subcommand, help_text, width=width)

        elif type(action) == argparse._SubParsersAction:
            # process subcommand help section
            msg = '\n'
            for subaction in action._get_subactions():
                msg += self._format_action(subaction)
            return msg
```

```
        else:
            return super(CustomHelpFormatter, self)._format_action(action)


def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(usage="sub <command>", add_help=False,
             formatter_class=CustomHelpFormatter)

subparser = parser.add_subparsers(dest="cmd")
subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# custom help message
parser._positionals.title = "commands"

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())
```

Output without arguments:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Chapter 161: Database Access

Section 161.1: SQLite

SQLite is a lightweight, disk-based database. Since it does not require a separate database server, it is often used for prototyping or for small applications that are often used by a single user or by one user at a given time.

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
print(c.fetchall())

conn.close()
```

The code above connects to the database stored in the file named `users.db`, creating the file first if it doesn't already exist. You can interact with the database via SQL statements.

The result of this example should be:

```
[(u'User A', 42), (u'User B', 43)]
```

The SQLite Syntax: An in-depth analysis
Getting started

1. Import the sqlite module using

   ```
   >>> import sqlite3
   ```

2. To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:

   ```
   >>> conn = sqlite3.connect('users.db')
   ```

   Alternatively, you can also supply the special name `:memory:` to create a temporary database in RAM, as follows:

   ```
   >>> conn = sqlite3.connect(':memory:')
   ```

3. Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

   ```
   c = conn.cursor()
   ```

```
# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

Important Attributes and Functions of `Connection`

1. `isolation_level`

   It is an attribute used to get or set the current isolation level. None for autocommit mode or one of `DEFERRED`, `IMMEDIATE` or `EXCLUSIVE`.

2. `cursor`

   The cursor object is used to execute SQL commands and queries.

3. `commit()`

   Commits the current transaction.

4. `rollback()`

   Rolls back any changes made since the previous call to `commit()`

5. `close()`

   Closes the database connection. It does not call `commit()` automatically. If `close()` is called without first calling `commit()` (assuming you are not in autocommit mode) then all changes made will be lost.

6. `total_changes`

   An attribute that logs the total number of rows modified, deleted or inserted since the database was opened.

7. `execute`, `executemany`, and `executescript`

   These functions perform the same way as those of the cursor object. This is a shortcut since calling these functions through the connection object results in the creation of an intermediate cursor object and calls the corresponding method of the cursor object

8. `row_factory`

   You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row.

```
def dict_factory(cursor, row):
    d = {}
    for i, col in enumerate(cursor.description):
        d[col[0]] = row[i]
    return d

conn = sqlite3.connect(":memory:")
conn.row_factory = dict_factory
```

Important Functions of `Cursor`

1. `execute(sql[, parameters])`

   Executes a *single* SQL statement. The SQL statement may be parametrized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks `?` ("qmark style") and named placeholders `:name` ("named style").

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.execute("create table people (name, age)")

who = "Sophia"
age = 37
# This IS the qmark STYLE:
cur.execute("insert into people values (?, ?)",
            (who, age))

# AND this IS the named STYLE:
cur.execute("select * from people where name=:who and age=:age",
            {"who": who, "age": age})  # the KEYS correspond TO the placeholders IN SQL

print(cur.fetchone())
```

> Beware: don't use `%s` for inserting strings into SQL commands as it can make your program vulnerable to an SQL injection attack (see SQL Injection).

2. `executemany(sql, seq_of_parameters)`

   Executes an SQL command against all parameter sequences or mappings found in the sequence sql. The sqlite3 module also allows using an iterator yielding parameters instead of a sequence.

```
L = [(1, 'abcd', 'dfj', 300),     # A list OF tuples TO be inserted INTO the DATABASE
     (2, 'cfgd', 'dyfj', 400),
     (3, 'sdd', 'dfjh', 300.50)]

conn = sqlite3.connect("test1.db")
conn.execute("create table if not exists book (id int, name text, author text, price real)")
conn.executemany("insert into book values (?, ?, ?, ?)", L)
```

```
FOR ROW IN conn.execute("select * from book"):
    print(ROW)
```

You can also pass iterator objects as a parameter to executemany, and the function will iterate over the each tuple of values that the iterator returns. The iterator must return a tuple of values.

```
import sqlite3

class IterChars:
    def __init__(SELF):
        SELF.count = ord('a')

    def __iter__(SELF):
        RETURN SELF

    def __next__(SELF):            # (USE NEXT(SELF) FOR Python 2)
        IF SELF.count > ord('z'):
            raise StopIteration
        SELF.count += 1
        RETURN (chr(SELF.count - 1),)

conn = sqlite3.connect("abc.db")
cur = conn.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

ROWS = cur.execute("select c from characters")
FOR ROW IN ROWS:
    print(ROW[0]),
```

3. `executescript(sql_script)`

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a `COMMIT` statement first, then executes the SQL script it gets as a parameter.

`sql_script` can be an instance of `str` or `bytes`.

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.executescript("""
     create table person(
         firstname,
         lastname,
         age
     );

     create table book(
         title,
         author,
         published
     );

     insert into book(title, author, published)
     values (
         'Dirk Gently''s Holistic Detective Agency',
         'Douglas Adams',
         1987
```

```
    );
    """)
```

The next set of functions are used in conjunction with `SELECT` statements in SQL. To retrieve data after executing a `SELECT` statement, you can either treat the cursor as an iterator, call the cursor's `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows.

Example of the iterator form:

```
import sqlite3
stocks = [('2006-01-05', 'BUY', 'RHAT', 100, 35.14),
          ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.0),
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
conn = sqlite3.connect(":memory:")
conn.execute("create table stocks (date text, buysell text, symb text, amount int, price
real)")
conn.executemany("insert into stocks values (?, ?, ?, ?, ?)", stocks)
cur = conn.cursor()

FOR ROW IN cur.execute('SELECT * FROM stocks ORDER BY price'):
    print(ROW)

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

4. `fetchone()`

Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
i = cur.fetchone()
while(i):
    print(i)
    i = cur.fetchone()

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

5. `fetchmany(size=cursor.arraysize)`

Fetches the next set of rows of a query result (specified by size), returning a list. If size is omitted, fetchmany returns a single row. An empty list is returned when no more rows are available.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchmany(2))

# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)]
```

6. `fetchall()`

   Fetches all (remaining) rows of a query result, returning a list.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchall())

# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
('2006-04-06', 'SELL', 'IBM', 500, 53.0), ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

SQLite and Python data types

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

This is how the data types are converted when moving from SQL to Python or vice versa.

```
None     <->     NULL
int      <->     INTEGER/INT
float    <->     REAL/FLOAT
str      <->     TEXT/VARCHAR(n)
bytes    <->     BLOB
```

Section 161.2: Accessing MySQL database using MySQLdb

The first thing you need to do is create a connection to the database using the connect method. After that, you will need a cursor that will operate with that connection.

Use the execute method of the cursor to interact with the database, and every once in a while, commit the changes using the commit method of the connection object.

Once everything is done, don't forget to close the cursor and the connection.

Here is a Dbconnect class with everything you'll need.

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconection = MySQLdb.connect(host='host_example',
                                           port=int('port_example'),
                                           user='user_example',
                                           passwd='pass_example',
                                           db='schema_example')
        self.dbcursor = self.dbconection.cursor()

    def commit_db(self):
        self.dbconection.commit()

    def close_db(self):
        self.dbcursor.close()
        self.dbconection.close()
```

Interacting with the database is simple. After creating the object, just use the execute method.

```
db = Dbconnect()
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

If you want to call a stored procedure, use the following syntax. Note that the parameters list is optional.

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters] )
```

After the query is done, you can access the results multiple ways. The cursor object is a generator that can fetch all the results or be looped.

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

If you want a loop using directly the generator:

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

If you want to commit changes to the database:

```
db.commit_db()
```

If you want to close the cursor and the connection:

```
db.close_db()
```

Section 161.3: Connection

Creating a connection

According to PEP 249, the connection to a database should be established using a `connect()` constructor, which returns a `Connection` object. The arguments for this constructor are database dependent. Refer to the database specific topics for the relevant arguments.

```
import MyDBAPI

con = MyDBAPI.connect(*database_dependent_args)
```

This connection object has four methods:

1: close

```
con.close()
```

Closes the connection instantly. Note that the connection is automatically closed if the `Connection.__del__` method is called. Any pending transactions will implicitly be rolled back.

2: commit

```
con.commit()
```

Commits any pending transaction to the database.

3: rollback

```
con.rollback()
```

Rolls back to the start of any pending transaction. In other words: this cancels any non-committed transaction to the database.

4: cursor

```
cur = con.cursor()
```

Returns a `Cursor` object. This is used to do transactions on the database.

Section 161.4: PostgreSQL Database access using psycopg2

psycopg2 is the most popular PostgreSQL database adapter that is both lightweight and efficient. It is the current implementation of the PostgreSQL adapter.

> Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection)

Establishing a connection to the database and creating a table

```
import psycopg2

# Establish a connection to the database.
# Replace parameter values with database credentials.
conn = psycopg2.connect(database="testpython",
                        user="postgres",
                        host="localhost",
                        password="abc123",
                        port="5432")

# Create a cursor. The cursor allows you to execute database queries.
cur = conn.cursor()

# Create a table. Initialise the table name, the column names and data type.
cur.execute("""CREATE TABLE FRUITS (
                    id          INT ,
                    fruit_name  TEXT,
                    color       TEXT,
                    price       REAL
            )""")
conn.commit()
conn.close()
```

Inserting data into the table:

```
# After creating the table as shown above, insert values into it.
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Apples', 'green', 1.00)""")

cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Bananas', 'yellow', 0.80)""")
```

Retrieving table data:

```
# Set up a query and execute it
cur.execute("""SELECT id, fruit_name, color, price
             FROM fruits""")

# Fetch the data
rows = cur.fetchall()

# Do stuff with the data
for row in rows:
    print "ID = {} ".format(row[0])
    print "FRUIT NAME = {}".format(row[1])
    print("COLOR = {}".format(row[2]))
    print("PRICE = {}".format(row[3]))
```

The output of the above would be:

```
ID = 1
NAME = Apples
COLOR = green
PRICE = 1.0

ID = 2
NAME = Bananas
COLOR = yellow
PRICE = 0.8
```

And so, there you go, you now know half of all you need to know about ***psycopg2***! :)

Section 161.5: Oracle database

Pre-requisites:

- cx_Oracle package - See here for all versions
- Oracle instant client - For Windows x64, Linux x64

Setup:

- Install the cx_Oracle package as:

```
sudo rpm -i <YOUR_PACKAGE_FILENAME>
```

- Extract the Oracle instant client and set environment variables as:

```
ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH
```

Creating a connection:

```
import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):
```

```
        self._db_connection =
            cx_Oracle.connect('<USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME>')
        self._db_cur = self._db_connection.cursor()
```

Get database version:

```
ver = con.version.split(".")
print ver
```

Sample Output: ['12', '1', '0', '2', '0']

Execute query: SELECT

```
_db_cur.execute("select * from employees order by emp_id")
for result in _db_cur:
    print result
```

Output will be in Python tuples:

(10, 'SYSADMIN', 'IT-INFRA', 7)

(23, 'HR ASSOCIATE', 'HUMAN RESOURCES', 6)

Execute query: INSERT

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)
                values (31, 'MTS', 'ENGINEERING', 7)
_db_connection.commit()
```

When you perform insert/update/delete operations in an Oracle Database, the changes are only available within your session until `commit` is issued. When the updated data is committed to the database, it is then available to other users and sessions.

Execute query: INSERT using Bind variables

Reference

Bind variables enable you to re-execute statements with new values, without the overhead of re-parsing the statement. Bind variables improve code re-usability, and can reduce the risk of SQL Injection attacks.

```
rows = [ (1, "First" ),
     (2, "Second" ),
     (3, "Third" ) ]
_db_cur.bindarraysize = 3
_db_cur.setinputsizes(int, 10)
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)
_db_connection.commit()
```

Close connection:

```
_db_connection.close()
```

The close() method closes the connection. Any connections not explicitly closed will be automatically released when the script ends.

Section 161.6: Using sqlalchemy

To use sqlalchemy for database:

```
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL


url = URL(drivername='mysql',
          username='user',
          password='passwd',
          host='host',
          database='db')

engine = create_engine(url)  # sqlalchemy engine
```

Now this engine can be used: e.g. with pandas to fetch dataframes directly from mysql

```
import pandas as pd

con = engine.connect()
dataframe = pd.read_sql(sql=query, con=con)
```

Chapter 162: Connecting Python to SQL Server

Section 162.1: Connect to Server, Create Table, Query Data

Install the package:

```
$ pip install pymssql
```

```
import pymssql

SERVER = "servername"
USER = "username"
PASSWORD = "password"
DATABASE = "dbname"

connection = pymssql.connect(server=SERVER, user=USER,
                password=PASSWORD, database=DATABASE)

cursor = connection.cursor() # to access field as dictionary use cursor(as_dict=True)
cursor.execute("SELECT TOP 1 * FROM TableName")
row = cursor.fetchone()

######## CREATE TABLE ########
cursor.execute("""
CREATE TABLE posts (
    post_id INT PRIMARY KEY NOT NULL,
    message TEXT,
    publish_date DATETIME
)
""")

######## INSERT DATA IN TABLE ########
cursor.execute("""
    INSERT INTO posts VALUES(1, "Hey There", "11.23.2016")
""")
# commit your work to database
connection.commit()

######## ITERATE THROUGH RESULTS  ########
cursor.execute("SELECT TOP 10 * FROM posts ORDER BY publish_date DESC")
for row in cursor:
    print("Message: " + row[1] + " | " + "Date: " + row[2])
    # if you pass as_dict=True to cursor
    # print(row["message"])

connection.close()
```

You can do anything if your work is related with SQL expressions, just pass this expressions to the execute method(CRUD operations).

For with statement, calling stored procedure, error handling or more example check: pymssql.org

Chapter 163: PostgreSQL

Section 163.1: Getting Started

PostgreSQL is an actively developed and mature open source database. Using the `psycopg2` module, we can execute queries on the database.

Installation using pip

```
pip install psycopg2
```

Basic usage

Let's assume we have a table `my_table` in the database `my_database` defined as follows.

id	first_name	last_name
1	John	Doe

We can use the `psycopg2` module to run queries on the database in the following fashion.

```
import psycopg2

# Establish a connection to the existing database 'my_database' using
# the user 'my_user' with password 'my_password'
con = psycopg2.connect("host=localhost dbname=my_database user=my_user password=my_password")

# Create a cursor
cur = con.cursor()

# Insert a record into 'my_table'
cur.execute("INSERT INTO my_table(id, first_name, last_name) VALUES (2, 'Jane', 'Doe');")

# Commit the current transaction
con.commit()

# Retrieve all records from 'my_table'
cur.execute("SELECT * FROM my_table;")
results = cur.fetchall()

# Close the database connection
con.close()

# Print the results
print(results)

# OUTPUT: [(1, 'John', 'Doe'), (2, 'Jane', 'Doe')]
```

Chapter 164: Python and Excel

Section 164.1: Read the excel data using xlrd module

Python xlrd library is to extract data from Microsoft Excel (tm) spreadsheet files.

Installation:

```
pip install xlrd
```

Or you can use setup.py file from pypi

https://pypi.python.org/pypi/xlrd

Reading an excel sheet: Import xlrd module and open excel file using open_workbook() method.

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Check number of sheets in the excel

```
print book.nsheets
```

Print the sheet names

```
print book.sheet_names()
```

Get the sheet based on index

```
sheet=book.sheet_by_index(1)
```

Read the contents of a cell

```
cell = sheet.cell(row,col) #where row=row number and col=column number
print cell.value #to print the cell contents
```

Get number of rows and number of columns in an excel sheet

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

Get excel sheet by name

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

Section 164.2: Format Excel files with xlsxwriter

```
import xlsxwriter

# create a new file
workbook = xlsxwriter.Workbook('your_file.xlsx')

# add some new formats to be used by the workbook
```

```
percent_format = workbook.add_format({'num_format': '0%'})
percent_with_decimal = workbook.add_format({'num_format': '0.0%'})
bold = workbook.add_format({'bold': True})
red_font = workbook.add_format({'font_color': 'red'})
remove_format = workbook.add_format()

# add a new sheet
worksheet = workbook.add_worksheet()

# set the width of column A
worksheet.set_column('A:A', 30, )

# set column B to 20 and include the percent format we created earlier
worksheet.set_column('B:B', 20, percent_format)

# remove formatting from the first row (change in height=None)
worksheet.set_row('0:0', None, remove_format)

workbook.close()
```

Section 164.3: Put list data into a Excel's file

```
import os, sys
from openpyxl import Workbook
from datetime import datetime

dt = datetime.now()
list_values = [["01/01/2016", "05:00:00", 3], \
               ["01/02/2016", "06:00:00", 4], \
               ["01/03/2016", "07:00:00", 5], \
               ["01/04/2016", "08:00:00", 6], \
               ["01/05/2016", "09:00:00", 7]]

# Create a Workbook on Excel:
wb = Workbook()
sheet = wb.active
sheet.title = 'data'

# Print the titles into Excel Workbook:
row = 1
sheet['A'+str(row)] = 'Date'
sheet['B'+str(row)] = 'Hour'
sheet['C'+str(row)] = 'Value'

# Populate with data
for item in list_values:
    row += 1
    sheet['A'+str(row)] = item[0]
    sheet['B'+str(row)] = item[1]
    sheet['C'+str(row)] = item[2]

# Save a file by date:
filename = 'data_' + dt.strftime("%Y%m%d_%I%M%S") + '.xlsx'
wb.save(filename)

# Open the file for the user:
os.chdir(sys.path[0])
os.system('start excel.exe "%s\\%s"' % (sys.path[0], filename, ))
```

Section 164.4: OpenPyXL

OpenPyXL is a module for manipulating and creating `xlsx/xlsm/xltx/xltm` workbooks in memory.

Manipulating and reading an existing workbook:

```
import openpyxl as opx
#To change an existing workbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

`load_workbook()` contains the parameter `read_only`, setting this to `True` will load the workbook as read_only, this is helpful when reading larger `xlsx` files:

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

Once you have loaded the workbook into memory, you can access the individual sheets using `workbook.sheets`

```
first_sheet = workbook.worksheets[0]
```

If you want to specify the name of an available sheets, you can use `workbook.get_sheet_names()`.

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Finally, the rows of the sheet can be accessed using `sheet.rows`. To iterate over the rows in a sheet, use:

```
for row in sheet.rows:
    print row[0].value
```

Since each `row` in `rows` is a list of `Cells`, use `Cell.value` to get the contents of the Cell.

Creating a new Workbook in memory:

```
#Calling the Workbook() function creates a new book in memory
wb = opx.Workbook()

#We can then create a new sheet in the wb
ws = wb.create_sheet('Sheet Name', 0) #0 refers to the index of the sheet order in the wb
```

Several tab properties may be changed through openpyxl, for example the `tabColor`:

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

To save our created workbook we finish with:

```
wb.save('filename.xlsx')
```

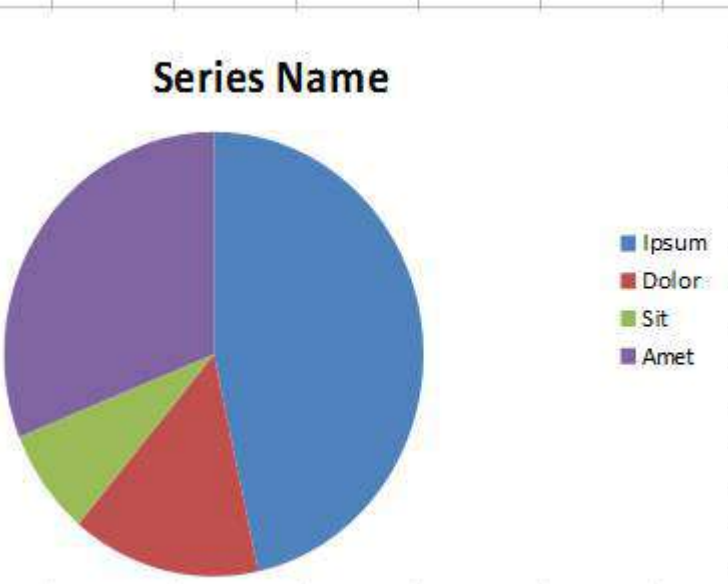
Section 164.5: Create excel charts with xlsxwriter

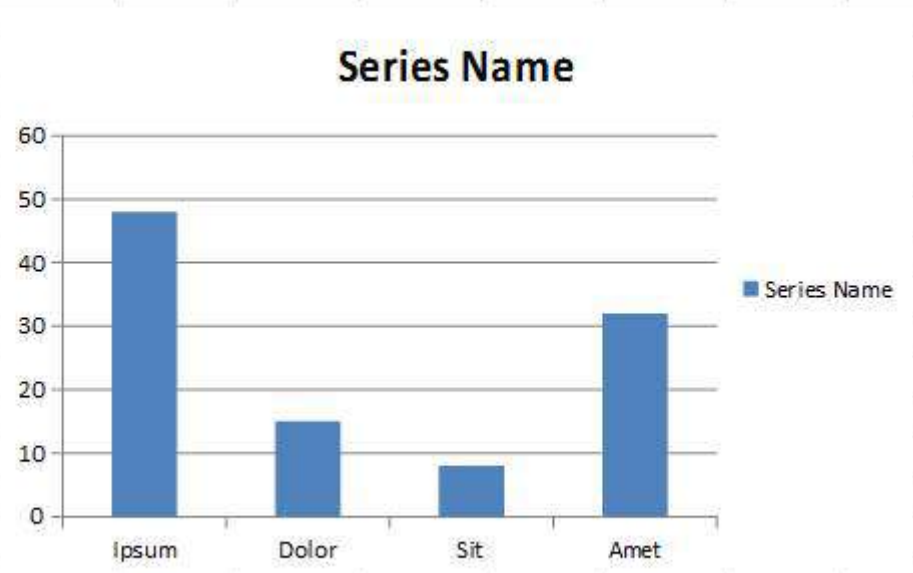
```
import xlsxwriter

# sample data
chart_data = [
    {'name': 'Lorem', 'value': 23},
    {'name': 'Ipsum', 'value': 48},
    {'name': 'Dolor', 'value': 15},
```

```
    {'name': 'Sit', 'value': 8},
    {'name': 'Amet', 'value': 32}
]

# excel file path
xls_file = 'chart.xlsx'

# the workbook
workbook = xlsxwriter.Workbook(xls_file)

# add worksheet to workbook
worksheet = workbook.add_worksheet()

row_ = 0
col_ = 0

# write headers
worksheet.write(row_, col_, 'NAME')
col_ += 1
worksheet.write(row_, col_, 'VALUE')
row_ += 1

# write sample data
for item in chart_data:
    col_ = 0
    worksheet.write(row_, col_, item['name'])
    col_ += 1
    worksheet.write(row_, col_, item['value'])
    row_ += 1

# create pie chart
pie_chart = workbook.add_chart({'type': 'pie'})

# add series to pie chart
pie_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert pie chart
worksheet.insert_chart('D2', pie_chart)

# create column chart
column_chart = workbook.add_chart({'type': 'column'})

# add serie to column chart
column_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert column chart
worksheet.insert_chart('D20', column_chart)

workbook.close()
```

Result:

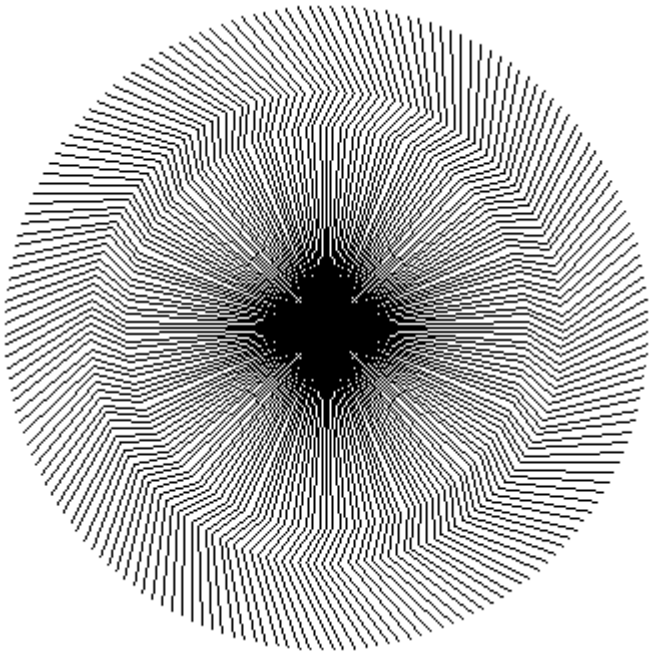
	A	B
1	NAME	VALUE
2	Lorem	23
3	Ipsum	48
4	Dolor	15
5	Sit	8
6	Amet	32





Chapter 165: Turtle Graphics

Section 165.1: Ninja Twist (Turtle Graphics)



Here a Turtle Graphics Ninja Twist:

```
import turtle

ninja = turtle.Turtle()

ninja.speed(10)

for i in range(180):
    ninja.forward(100)
    ninja.right(30)
    ninja.forward(20)
    ninja.left(60)
    ninja.forward(50)
    ninja.right(30)

    ninja.penup()
    ninja.setposition(0, 0)
    ninja.pendown()

    ninja.right(2)

turtle.done()
```

Chapter 166: Python Persistence

Parameter	Details
obj	pickled representation of obj to the open file object file
protocol	an integer, tells the pickler to use the given protocol,`0`-ASCII, `1`- old binary format
file	The file argument must have a write() method `wb` for *dump* method and for loading read() method `rb`

Section 166.1: Python Persistence

Objects like numbers, lists, dictionaries,nested structures and class instance objects live in your computer's memory and are lost as soon as the script ends.

pickle stores data persistently in separate file.

pickled representation of an object is always a bytes object in all cases so one must open files in `wb` to store data and `rb` to load data from pickle.

the data may be of any kind, for example,

```
data={'a':'some_value',
     'b':[9,4,7],
     'c':['some_str','another_str','spam','ham'],
     'd':{'key':'nested_dictionary'},
     }
```

Store data

```
import pickle
file=open('filename','wb')  #file object in binary write mode
pickle.dump(data,file)      #dump the data in the file object
file.close()                #close the file to write into the file
```

Load data

```
import pickle
file=open('filename','rb')  #file object in binary read mode
data=pickle.load(file)      #load the data back
file.close()

>>>data
{'b': [9, 4, 7], 'a': 'some_value', 'd': {'key': 'nested_dictionary'},
 'c': ['some_str', 'another_str', 'spam', 'ham']}
```

The following types can be pickled

1. None, True, and False
2. integers, floating point numbers, complex numbers
3. strings, bytes, bytearrays
4. tuples, lists, sets, and dictionaries containing only picklable objects
5. functions defined at the top level of a module (using def, not lambda)
6. built-in functions defined at the top level of a module
7. classes that are defined at the top level of a module
8. instances of such classes whose **dict** or the result of calling **getstate**()

Section 166.2: Function utility for save and load

Save data to and from file

```
import pickle
def save(filename,object):
    file=open(filename,'wb')
    pickle.dump(object,file)
    file.close()

def load(filename):
    file=open(filename,'rb')
    object=pickle.load(file)
    file.close()
    return object


>>>list_object=[1,1,2,3,5,8,'a','e','i','o','u']
>>>save(list_file,list_object)
>>>new_list=load(list_file)
>>>new_list
[1, 1, 2, 3, 5, 8, 'a', 'e', 'i', 'o', 'u'
```

Chapter 167: Design Patterns

A design pattern is a general solution to a commonly occurring problem in software development. This documentation topic is specifically aimed at providing examples of common design patterns in Python.

Section 167.1: Introduction to design patterns and Singleton Pattern

Design Patterns provide solutions to the `commonly occurring problems` in software design. The design patterns were first introduced by `GoF(Gang of Four)` where they described the common patterns as problems which occur over and over again and solutions to those problems.

Design patterns have four essential elements:

1. `The pattern name` is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. `The problem` describes when to apply the pattern.
3. `The solution` describes the elements that make up the design, their relationships, responsibilities, and collaborations.
4. `The consequences` are the results and trade-offs of applying the pattern.

Advantages of design patterns:

1. They are reusable across multiple projects.
2. The architectural level of problems can be solved
3. They are time-tested and well-proven, which is the experience of developers and architects
4. They have reliability and dependence

Design patterns can be classified into three categories:

1. Creational Pattern
2. Structural Pattern
3. Behavioral Pattern

`Creational Pattern` - They are concerned with how the object can be created and they isolate the details of object creation.

`Structural Pattern` - They design the structure of classes and objects so that they can compose to achieve larger results.

`Behavioral Pattern` - They are concerned with interaction among objects and responsibility of objects.

Singleton Pattern:

It is a type of `creational pattern` which provides a mechanism to have only one and one object of a given type and provides a global point of access.

e.g. Singleton can be used in database operations, where we want database object to maintain data consistency.

Implementation

We can implement Singleton Pattern in Python by creating only one instance of Singleton class and serving the same object again.

```
class Singleton(object):
    def __new__(cls):
        # hasattr method checks if the class object an instance property or not.
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

s = Singleton()
print ("Object created", s)

s1 = Singleton()
print ("Object2 created", s1)
```

Output:

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

Note that in languages like C++ or Java, this pattern is implemented by making the constructor private and creating a static method that does the object initialization. This way, one object gets created on the first call and class returns the same object thereafter. But in Python, we do not have any way to create private constructors.

Factory Pattern

Factory pattern is also a `Creational pattern`. The term `factory` means that a class is responsible for creating objects of other types. There is a class that acts as a factory which has objects and methods associated with it. The client creates an object by calling the methods with certain parameters and factory creates the object of the desired type and return it to the client.

```
from abc import ABCMeta, abstractmethod

class Music():
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_play(self):
        pass

class Mp3(Music):
    def do_play(self):
        print ("Playing .mp3 music!")

class Ogg(Music):
    def do_play(self):
        print ("Playing .ogg music!")

class MusicFactory(object):
    def play_sound(self, object_type):
        return eval(object_type)().do_play()

if __name__ == "__main__":
    mf = MusicFactory()
    music = input("Which music you want to play Mp3 or Ogg")
    mf.play_sound(music)
```

Output:

```
Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!
```

`MusicFactory` is the factory class here that creates either an object of type `Mp3` or `Ogg` depending on the choice user provides.

Section 167.2: Strategy Pattern

This design pattern is called Strategy Pattern. It is used to define a family of algorithms, encapsulates each one, and make them interchangeable. Strategy design pattern lets an algorithm vary independently from clients that use it.

For example, animals can "walk" in many different ways. Walking could be considered a strategy that is implemented by different types of animals:

```
from types import MethodType


class Animal(object):

    def __init__(self, *args, **kwargs):
        self.name = kwargs.pop('name', None) or 'Animal'
        if kwargs.get('walk', None):
            self.walk = MethodType(kwargs.pop('walk'), self)

    def walk(self):
        """
        Cause animal instance to walk

        Walking functionality is a strategy, and is intended to
        be implemented separately by different types of animals.
        """
        message = '{} should implement a walk method'.format(
            self.__class__.__name__)
        raise NotImplementedError(message)


# Here are some different walking algorithms that can be used with Animal
def snake_walk(self):
    print('I am slithering side to side because I am a {}.'.format(self.name))

def four_legged_animal_walk(self):
    print('I am using all four of my legs to walk because I am a(n) {}.'.format(
        self.name))

def two_legged_animal_walk(self):
    print('I am standing up on my two legs to walk because I am a {}.'.format(
        self.name))
```

Running this example would produce the following output:

```
generic_animal = Animal()
king_cobra = Animal(name='King Cobra', walk=snake_walk)
elephant = Animal(name='Elephant', walk=four_legged_animal_walk)
kangaroo = Animal(name='Kangaroo', walk=two_legged_animal_walk)

kangaroo.walk()
elephant.walk()
king_cobra.walk()
# This one will Raise a NotImplementedError to let the programmer
# know that the walk method is intended to be used as a strategy.
generic_animal.walk()
```

```
    # OUTPUT:
    #
    # I am standing up on my two legs to walk because I am a Kangaroo.
    # I am using all four of my legs to walk because I am a(n) Elephant.
    # I am slithering side to side because I am a King Cobra.
    # Traceback (most recent call last):
    #   File "./strategy.py", line 56, in <module>
    #     generic_animal.walk()
    #   File "./strategy.py", line 30, in walk
    #     raise NotImplementedError(message)
    # NotImplementedError: Animal should implement a walk method
```

Note that in languages like C++ or Java, this pattern is implemented using an abstract class or an interface to define a strategy. In Python it makes more sense to just define some functions externally that can be added dynamically to a class using `types.MethodType`.

Section 167.3: Proxy

Proxy object is often used to ensure guarded access to another object, which internal business logic we don't want to pollute with safety requirements.

Suppose we'd like to guarantee that only user of specific permissions can access resource.

Proxy definition: (it ensure that only users which actually can see reservations will be able to consumer reservation_service)

```
from datetime import date
from operator import attrgetter

class Proxy:
    def __init__(self, current_user, reservation_service):
        self.current_user = current_user
        self.reservation_service = reservation_service

    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        if self.current_user.can_see_reservations:
            return self.reservation_service.highest_total_price_reservations(
                date_from,
                date_to,
                reservations_count
              )
        else:
            return []

#Models and ReservationService:

class Reservation:
    def __init__(self, date, total_price):
        self.date = date
        self.total_price = total_price

class ReservationService:
    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        # normally it would be read from database/external service
        reservations = [
            Reservation(date(2014, 5, 15), 100),
            Reservation(date(2017, 5, 15), 10),
            Reservation(date(2017, 1, 15), 50)
        ]
```

```
        filtered_reservations = [r for r in reservations if (date_from <= r.date <= date_to)]

        sorted_reservations = sorted(filtered_reservations, key=attrgetter('total_price'),
reverse=True)

        return sorted_reservations[0:reservations_count]


class User:
    def __init__(self, can_see_reservations, name):
        self.can_see_reservations = can_see_reservations
        self.name = name

#Consumer service:

class StatsService:
    def __init__(self, reservation_service):
        self.reservation_service = reservation_service

    def year_top_100_reservations_average_total_price(self, year):
        reservations = self.reservation_service.highest_total_price_reservations(
            date(year, 1, 1),
            date(year, 12, 31),
            1
        )

        if len(reservations) > 0:
            total = sum(r.total_price for r in reservations)

            return total / len(reservations)
        else:
            return 0

#Test:
def test(user, year):
    reservations_service = Proxy(user, ReservationService())
    stats_service = StatsService(reservations_service)
    average_price = stats_service.year_top_100_reservations_average_total_price(year)
    print("{0} will see: {1}".format(user.name, average_price))

test(User(True, "John the Admin"), 2017)
test(User(False, "Guest"),         2017)
```

BENEFITS

- **we're avoiding any changes in `ReservationService` when access restrictions are changed.**
- **we're not mixing business related data (`date_from`, `date_to`, `reservations_count`) with domain unrelated concepts (user permissions) in service.**
- **Consumer (`StatsService`) is free from permissions related logic as well**

CAVEATS

- Proxy interface is always exactly the same as the object it hides, so that user that consumes service wrapped by proxy wasn't even aware of proxy presence.

Chapter 168: hashlib

hashlib implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512.

Section 168.1: MD5 hash of a string

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet RFC 1321).

There is one constructor method named for each type of hash. All return a hash object with the same simple interface. For example: use `sha1()` to create a SHA1 hash object.

```
hash.sha1()
```

Constructors for hash algorithms that are always present in this module are `md5()`, `sha1()`, `sha224()`, `sha256()`, `sha384()`, and `sha512()`.

You can now feed this object with arbitrary strings using the `update()` method. At any point you can ask it for the digest of the concatenation of the strings fed to it so far using the `digest()` or `hexdigest()` methods.

```
hash.update(arg)
```

> Update the hash object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments: m.update(a); m.update(b) is equivalent to m.update(a+b).

```
hash.digest()
```

> Return the digest of the strings passed to the update() method so far. This is a string of digest_size bytes which may contain non-ASCII characters, including null bytes.

```
hash.hexdigest()
```

> Like digest() except the digest is returned as a string of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

Here is an example:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
>>> m.digest_size
16
```

```
>>> m.block_size
64
```

or:

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()
    'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

Section 168.2: algorithm provided by OpenSSL

A generic `new()` constructor that takes the string name of the desired algorithm as its first parameter also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than `new()` and should be preferred.

Using `new()` with an algorithm provided by OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Chapter 169: Creating a Windows service using Python

Headless processes (with no UI) in Windows are called Services. They can be controlled (started, stopped, etc) using standard Windows controls such as the command console, PowerShell or the Services tab in Task Manager. A good example might be an application that provides network services, such as a web application, or maybe a backup application that performs various background archival tasks. There are several ways to create and install a Python application as a Service in Windows.

Section 169.1: A Python script that can be run as a service

The modules used in this example are part of pywin32 (Python for Windows extensions). Depending on how you installed Python, you might need to install this separately.

```
import win32serviceutil
import win32service
import win32event
import servicemanager
import socket


class AppServerSvc (win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"

    def __init__(self,args):
        win32serviceutil.ServiceFramework.__init__(self,args)
        self.hWaitStop = win32event.CreateEvent(None,0,0,None)
        socket.setdefaulttimeout(60)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self):
        servicemanager.LogMsg(servicemanager.EVENTLOG_INFORMATION_TYPE,
                              servicemanager.PYS_SERVICE_STARTED,
                              (self._svc_name_,''))
        self.main()

    def main(self):
        pass

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(AppServerSvc)
```

This is just boilerplate. Your application code, probably invoking a separate script, would go in the main() function.

You will also need to install this as a service. The best solution for this at the moment appears to be to use Non-sucking Service Manager. This allows you to install a service and provides a GUI for configuring the command line the service executes. For Python you can do this, which creates the service in one go:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

Where my_script.py is the boilerplate script above, modified to invoke your application script or code in the main() function. Note that the service doesn't run the Python script directly, it runs the Python interpreter and passes it the

main script on the command line.

Alternatively you can use tools provided in the Windows Server Resource Kit for your operating system version so create the service.

Section 169.2: Running a Flask web application as a service

This is a variation on the generic example. You just need to import your app script and invoke it's `run()` method in the service's `main()` function. In this case we're also using the multiprocessing module due to an issue accessing `WSGIRequestHandler`.

```
import win32serviceutil
import win32service
import win32event
import servicemanager
from multiprocessing import Process

from app import app


class Service(win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"
    _svc_description_ = "Tests Python service framework by receiving and echoing messages over a
named pipe"

    def __init__(self, *args):
        super().__init__(*args)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        self.process.terminate()
        self.ReportServiceStatus(win32service.SERVICE_STOPPED)

    def SvcDoRun(self):
        self.process = Process(target=self.main)
        self.process.start()
        self.process.run()

    def main(self):
        app.run()


if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(Service)
```

Adapted from http://stackoverflow.com/a/25130524/318488

Chapter 170: Mutable vs Immutable (and Hashable) in Python

Section 170.1: Mutable vs Immutable

There are two kind of types in Python. Immutable types and mutable types.

Immutables

An object of an immutable type cannot be changed. Any attempt to modify the object will result in a copy being created.

This category includes: integers, floats, complex, strings, bytes, tuples, ranges and frozensets.

To highlight this property, let's play with the `id` builtin. This function returns the unique identifier of the object passed as parameter. If the id is the same, this is the same object. If it changes, then this is another object. *(Some say that this is actually the memory address of the object, but beware of them, they are from the dark side of the force...)*

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Okay, 1 is not 3... Breaking news... Maybe not. However, this behaviour is often forgotten when it comes to more complex types, especially strings.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Aha! See? We can modify it!

```
>>> id(stack)
140128123911472
```

No. While it seems we can change the string named by the variable `stack`, what we actually do, is creating a new object to contain the result of the concatenation. We are fooled because in the process, the old object goes nowhere, so it is destroyed. In another situation, that would have been more obvious:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

In this case it is clear that if we want to retain the first string, we need a copy. But is that so obvious for other types?

Exercise

Now, knowing how immutable types work, what would you say with the below piece of code? Is it wise?

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ","
```

Mutables

An object of a mutable type can be changed, and it is changed *in-situ*. No implicit copies are done.

This category includes: lists, dictionaries, bytearrays and sets.

Let's continue to play with our little `id` function.

```
>>> b = bytearray(b'Stack')
>>> b
bytearray(b'Stack')
>>> b = bytearray(b'Stack')
>>> id(b)
140128030688288
>>> b += b'Overflow'
>>> b
bytearray(b'StackOverflow')
>>> id(b)
140128030688288
```

(As a side note, I use bytes containing ascii data to make my point clear, but remember that bytes are not designed to hold textual data. May the force pardon me.)

What do we have? We create a bytearray, modify it and using the `id`, we can ensure that this is the same object, modified. Not a copy of it.

Of course, if an object is going to be modified often, a mutable type does a much better job than an immutable type. Unfortunately, the reality of this property is often forgotten when it hurts the most.

```
>>> c = b
>>> c += b' rocks!'
>>> c
bytearray(b'StackOverflow rocks!')
```

Okay...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Waiiit a second...

```
>>> id(c) == id(b)
True
```

Indeed. `c` is not a copy of `b`. `c` is `b`.

Exercise

Now you better understand what side effect is implied by a mutable type, can you explain what is going wrong in this example?

```
>>> ll = [ [] ]*4 # Create a list of 4 lists to contain our results
>>> ll
[[], [], [], []]
>>> ll[0].append(23) # Add result 23 to first list
>>> ll
[[23], [23], [23], [23]]
>>> # Oops...
```

Section 170.2: Mutable and Immutable as Arguments

One of the major use case when a developer needs to take mutability into account is when passing arguments to a function. This is very important, because this will determine the ability for the function to modify objects that doesn't belong to its scope, or in other words if the function has side effects. This is also important to understand where the result of a function has to be made available.

```
>>> def list_add3(lin):
    lin += [3]
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
>>> b
[1, 2, 3, 3]
>>> a
[1, 2, 3, 3]
```

Here, the mistake is to think that `lin`, as a parameter to the function, can be modified locally. Instead, `lin` and `a` reference the same object. As this object is mutable, the modification is done in-place, which means that the object referenced by both `lin` and `a` is modified. `lin` doesn't really need to be returned, because we already have a reference to this object in the form of `a`. `a` and `b` end referencing the same object.

This doesn't go the same for tuples.

```
>>> def tuple_add3(tin):
    tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

At the beginning of the function, `tin` and `a` reference the same object. But this is an immutable object. So when the function tries to modify it, `tin` receive a new object with the modification, while `a` keeps a reference to the original object. In this case, returning `tin` is mandatory, or the new object would be lost.

Exercise

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue
```

```
>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

Note: `reverse` operates in-place.

What do you think of this function? Does it have side effects? Is the return necessary? After the call, what is the value of `saying`? Of `focused`? What happens if the function is called again with the same parameters?

Chapter 171: configparser

This module provides the ConfigParser class which implements a basic configuration language in INI files. You can use this to write Python programs which can be customized by end users easily.

Section 171.1: Creating configuration file programmatically

Configuration file contains sections, each section contains keys and values. configparser module can be used to read and write config files. Creating the configuration file:

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                    'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

The output file contains below structure

```
[settings]
resolution = 320x240
color = blue
```

If you want to change particular field ,get the field and assign the value

```
settings=config['settings']
settings['color']='red'
```

Section 171.2: Basic usage

In config.ini:

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

In Python:

```
from ConfigParser import ConfigParser
config = ConfigParser()

#Load configuration file
config.read("config.ini")

# Access the key "debug" in "DEFAULT" section
config.get("DEFAULT", "debug")
# Return 'True'

# Access the key "path" in "FILES" destion
config.get("FILES", "path")
# Return '/path/to/file'
```

Chapter 172: Optical Character Recognition

Optical Character Recognition is converting images of text into actual text. In these examples find ways of using OCR in python.

Section 172.1: PyTesseract

PyTesseract is an in-development python package for OCR.

Using PyTesseract is pretty easy:

```
try:
    import Image
except ImportError:
    from PIL import Image

import pytesseract

    #Basic OCR
    print(pytesseract.image_to_string(Image.open('test.png')))

    #In French
    print(pytesseract.image_to_string(Image.open('test-european.jpg'), lang='fra'))
```

PyTesseract is open source and can be found here.

Section 172.2: PyOCR

Another module of some use is `PyOCR`, source code of which is here.

Also simple to use and has more features than `PyTesseract`.

To initialize:

```
from PIL import Image
import sys

import pyocr
import pyocr.builders

tools = pyocr.get_available_tools()
# The tools are returned in the recommended order of usage
tool = tools[0]

langs = tool.get_available_languages()
lang = langs[0]
# Note that languages are NOT sorted in any way. Please refer
# to the system locale settings for the default language
# to use.
```

And some examples of usage:

```
txt = tool.image_to_string(
    Image.open('test.png'),
    lang=lang,
```

```
    builder=pyocr.builders.TextBuilder()
)
# txt is a Python string

word_boxes = tool.image_to_string(
    Image.open('test.png'),
    lang="eng",
    builder=pyocr.builders.WordBoxBuilder()
)
# list of box objects. For each box object:
#   box.content is the word in the box
#   box.position is its position on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

line_and_word_boxes = tool.image_to_string(
    Image.open('test.png'), lang="fra",
    builder=pyocr.builders.LineBoxBuilder()
)
# list of line objects. For each line object:
#   line.word_boxes is a list of word boxes (the individual words in the line)
#   line.content is the whole text of the line
#   line.position is the position of the whole line on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

# Digits - Only Tesseract (not 'libtesseract' yet !)
digits = tool.image_to_string(
    Image.open('test-digits.png'),
    lang=lang,
    builder=pyocr.tesseract.DigitBuilder()
)
# digits is a python string
```

Chapter 173: Virtual environments

A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the "Project X depends on version 1.x but, Project Y needs 4.x" dilemma, and keeps your global site-packages directory clean and manageable.

This helps isolate your environments for different projects from each other and from your system libraries.

Section 173.1: Creating and using a virtual environment

`virtualenv` is a tool to build isolated Python environments. This program creates a folder which contains all the necessary executables to use the packages that a Python project would need.

Installing the virtualenv tool

This is only required once. The `virtualenv` program may be available through your distribution. On Debian-like distributions, the package is called `python-virtualenv` or `python3-virtualenv`.

You can alternatively install `virtualenv` using pip:

```
$ pip install virtualenv
```

Creating a new virtual environment

This only required once per project. When starting a project for which you want to isolate dependencies, you can setup a new virtual environment for this project:

```
$ virtualenv foo
```

This will create a `foo` folder containing tooling scripts and a copy of the `python` binary itself. The name of the folder is not relevant. Once the virtual environment is created, it is self-contained and does not require further manipulation with the `virtualenv` tool. You can now start using the virtual environment.

Activating an existing virtual environment

To *activate* a virtual environment, some shell magic is required so your Python is the one inside `foo` instead of the system one. This is the purpose of the `activate` file, that you must source into your current shell:

```
$ source foo/bin/activate
```

Windows users should type:

```
$ foo\Scripts\activate.bat
```

Once a virtual environment has been activated, the `python` and `pip` binaries and all scripts installed by third party modules are the ones inside `foo`. Particularly, all modules installed with `pip` will be deployed to the virtual environment, allowing for a contained development environment. Activating the virtual environment should also add a prefix to your prompt as seen in the following commands.

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

Saving and restoring dependencies

To save the modules that you have installed via `pip`, you can list all of those modules (and the corresponding versions) into a text file by using the `freeze` command. This allows others to quickly install the Python modules needed for the application by using the install command. The conventional name for such a file is `requirements.txt`:

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

Please note that `freeze` lists all the modules, including the transitive dependencies required by the top-level modules you installed manually. As such, you may prefer to craft the `requirements.txt` file by hand, by putting only the top-level modules you need.

Exiting a virtual environment

If you are done working in the virtual environment, you can deactivate it to get back to your normal shell:

```
(foo)$ deactivate
```

Using a virtual environment in a shared host

Sometimes it's not possible to `$ source bin/activate` a virtualenv, for example if you are using mod_wsgi in shared host or if you don't have access to a file system, like in Amazon API Gateway, or Google AppEngine. For those cases you can deploy the libraries you installed in your local virtualenv and patch your `sys.path`.

Luckily virtualenv ships with a script that updates both your `sys.path` and your `sys.prefix`

```
import os

mydir = os.path.dirname(os.path.realpath(__file__))
activate_this = mydir + '/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

You should append these lines at the very beginning of the file your server will execute.

This will find the `bin/activate_this.py` that `virtualenv` created file in the same dir you are executing and add your `lib/python2.7/site-packages` to `sys.path`

If you are looking to use the `activate_this.py` script, remember to deploy with, at least, the `bin` and `lib/python2.7/site-packages` directories and their content.

Python 3.x `Version ≥ 3.3`

Built-in virtual environments

From Python 3.3 onwards, the venv module will create virtual environments. The `pyvenv` command does not need installing separately:

```
$ pyvenv foo
$ source foo/bin/activate
```

or

```
$ python3 -m venv foo
$ source foo/bin/activate
```

Section 173.2: Specifying specific python version to use in script on Unix/Linux

In order to specify which version of python the Linux shell should use the first line of Python scripts can be a shebang line, which starts with `#!`:

```
#!/usr/bin/python
```

If you are in a virtual environment, then `python myscript.py` will use the Python from your virtual environment, but `./myscript.py` will use the Python interpreter in the `#!` line. To make sure the virtual environment's Python is used, change the first line to:

```
#!/usr/bin/env python
```

After specifying the shebang line, remember to give execute permissions to the script by doing:

```
chmod +x myscript.py
```

Doing this will allow you to execute the script by running `./myscript.py` (or provide the absolute path to the script) instead of `python myscript.py` or `python3 myscript.py`.

Section 173.3: Creating a virtual environment for a different version of python

Assuming `python` and `python3` are both installed, it is possible to create a virtual environment for Python 3 even if `python3` is not the default Python:

```
virtualenv -p python3 foo
```

or

```
virtualenv --python=python3 foo
```

or

```
python3 -m venv foo
```

or

```
pyvenv foo
```

Actually you can create virtual environment based on any version of working python of your system. You can check different working python under your `/usr/bin/` or `/usr/local/bin/` (In Linux) OR in `/Library/Frameworks/Python.framework/Versions/X.X/bin/` (OSX), then figure out the name and use that in the `--python` or `-p` flag while creating virtual environment.

Section 173.4: Making virtual environments using Anaconda

A powerful alternative to `virtualenv` is Anaconda - a cross-platform, `pip`-like package manager bundled with features for quickly making and removing virtual environments. After installing Anaconda, here are some commands to get started:

Create an environment

```
conda create --name <envname> python=<version>
```

where **<envname>** in an arbitrary name for your virtual environment, and **<version>** is a specific Python version you wish to setup.

Activate and deactivate your environment

```
# Linux, Mac
source activate <envname>
source deactivate
```

or

```
# Windows
activate <envname>
deactivate
```

View a list of created environments

```
conda env list
```

Remove an environment

```
conda env remove -n <envname>
```

Find more commands and features in the official conda documentation.

Section 173.5: Managing multiple virtual environments with virtualenvwrapper

The `virtualenvwrapper` utility simplifies working with virtual environments and is especially useful if you are dealing with many virtual environments/projects.

Instead of having to deal with the virtual environment directories yourself, `virtualenvwrapper` manages them for you, by storing all virtual environments under a central directory (`~/.virtualenvs` by default).

Installation

Install `virtualenvwrapper` with your system's package manager.

Debian/Ubuntu-based:

```
apt-get install virtualenvwrapper
```

Fedora/CentOS/RHEL:

```
yum install python-virtualenvrwapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

Or install it from PyPI using `pip`:

```
pip install virtualenvwrapper
```

Under Windows you can use either `virtualenvwrapper-win` or `virtualenvwrapper-powershell` instead.

Usage

Virtual environments are created with `mkvirtualenv`. All arguments of the original `virtualenv` command are accepted as well.

```
mkvirtualenv my-project
```

or e.g.

```
mkvirtualenv --system-site-packages my-project
```

The new virtual environment is automatically activated. In new shells you can enable the virtual environment with `workon`

```
workon my-project
```

The advantage of the `workon` command compared to the traditional `. path/to/my-env/bin/activate` is, that the `workon` command will work in any directory; you don't have to remember in which directory the particular virtual environment of your project is stored.

Project Directories

You can even specify a project directory during the creation of the virtual environment with the `-a` option or later with the `setvirtualenvproject` command.

```
mkvirtualenv -a /path/to/my-project my-project
```

or

```
workon my-project
cd /path/to/my-project
setvirtualenvproject
```

Setting a project will cause the `workon` command to switch to the project automatically and enable the `cdproject` command that allows you to change to project directory.

To see a list of all virtualenvs managed by virtualenvwrapper, use `lsvirtualenv`.

To remove a virtualenv, use `rmvirtualenv`:

```
rmvirtualenv my-project
```

Each virtualenv managed by virtualenvwrapper includes 4 empty bash scripts: `preactivate`, `postactivate`, `predeactivate`, and `postdeactivate`. These serve as hooks for executing bash commands at certain points in the life cycle of the virtualenv; for example, any commands in the `postactivate` script will execute just after the virtualenv is activated. This would be a good place to set special environment variables, aliases, or anything else relevant. All 4 scripts are located under `.virtualenvs/<virtualenv_name>/bin/`.

For more details read the virtualenvwrapper documentation.

Section 173.6: Installing packages in a virtual environment

Once your virtual environment has been activated, any package that you install will now be installed in the `virtualenv` & not globally. Hence, new packages can be without needing root privileges.

To verify that the packages are being installed into the `virtualenv` run the following command to check the path of the executable that is being used:

```
(<Virtualenv Name) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

Any package then installed using pip will be installed in the `virtualenv` itself in the following directory:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

Alternatively, you may create a file listing the needed packages.

requirements.txt:

```
requests==2.10.0
```

Executing:

```
# Install packages from requirements.txt
pip install -r requirements.txt
```

will install version 2.10.0 of the package `requests`.

You can also get a list of the packages and their versions currently installed in the active virtual environment:

```
# Get a list of installed packages
pip freeze

# Output list of packages and versions into a requirement.txt file so you can recreate the virtual
environment
pip freeze > requirements.txt
```

Alternatively, you do not have to activate your virtual environment each time you have to install a package. You can directly use the pip executable in the virtual environment directory to install packages.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

More information about using pip can be found on the PIP topic.

Since you're installing without root in a virtual environment, this is *not* a global install, across the entire system - the installed package will only be available in the current virtual environment.

Section 173.7: Discovering which virtual environment you are using

If you are using the default `bash` prompt on Linux, you should see the name of the virtual environment at the start of your prompt.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Section 173.8: Checking if running inside a virtual environment

Sometimes the shell prompt doesn't display the name of the virtual environment and you want to be sure if you are in a virtual environment or not.

Run the python interpreter and try:

```
import sys
sys.prefix
sys.real_prefix
```

- Outside a virtual, environment `sys.prefix` will point to the system python installation and `sys.real_prefix` is not defined.
- Inside a virtual environment, `sys.prefix` will point to the virtual environment python installation and `sys.real_prefix` will point to the system python installation.

For virtual environments created using the standard library venv module there is no `sys.real_prefix`. Instead, check whether `sys.base_prefix` is the same as `sys.prefix`.

Section 173.9: Using virtualenv with fish shell

Fish shell is friendlier yet you might face trouble while using with `virtualenv` or `virtualenvwrapper`. Alternatively `virtualfish` exists for the rescue. Just follow the below sequence to start using Fish shell with virtualenv.

- Install virtualfish to the global space

  ```
  sudo pip install virtualfish
  ```

- Load the python module virtualfish during the fish shell startup

  ```
  $ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
  ```

- Edit this function `fish_prompt` by `$ funced fish_prompt --editor vim` and add the below lines and close the vim editor

  ```
  if set -q VIRTUAL_ENV
      echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color
  normal) " "
  end
  ```

  Note: If you are unfamiliar with vim, simply supply your favorite editor like this `$ funced fish_prompt --editor nano` or `$ funced fish_prompt --editor gedit`

- Save changes using `funcsave`

  ```
  funcsave fish_prompt
  ```

- To create a new virtual environment use `vf new`

  ```
  vf new my_new_env # Make sure $HOME/.virtualenv exists
  ```

- If you want create a new python3 environment specify it via `-p` flag

```
vf new -p python3 my_new_env
```

- To switch between virtualenvironments use `vf deactivate` & `vf activate another_env`

Official Links:

- https://github.com/adambrenecki/virtualfish
- http://virtualfish.readthedocs.io/en/latest/

Chapter 174: Python Virtual Environment - virtualenv

A Virtual Environment ("virtualenv") is a tool to create isolated Python environments. It keeps the dependencies required by different projects in separate places, by creating virtual Python env for them. It solves the “project A depends on version 2.xxx but, project B needs 2.xxx” dilemma, and keeps your global site-packages directory clean and manageable.

"virtualenv" creates a folder which contains all the necessary libs and bins to use the packages that a Python project would need.

Section 174.1: Installation

Install virtualenv via pip / (apt-get):

```
pip install virtualenv
```

OR

```
apt-get install python-virtualenv
```

Note: In case you are getting permission issues, use sudo.

Section 174.2: Usage

```
$ cd test_proj
```

Create virtual environment:

```
$ virtualenv test_proj
```

To begin using the virtual environment, it needs to be activated:

```
$ source test_project/bin/activate
```

To exit your virtualenv just type “deactivate”:

```
$ deactivate
```

Section 174.3: Install a package in your Virtualenv

If you look at the bin directory in your virtualenv, you’ll see easy_install which has been modified to put eggs and packages in the virtualenv’s site-packages directory. To install an app in your virtual environment:

```
$ source test_project/bin/activate
$ pip install flask
```

At this time, you don't have to use sudo since the files will all be installed in the local virtualenv site-packages directory. This was created as your own user account.

Section 174.4: Other useful virtualenv commands

lsvirtualenv : List all of the environments.

cdvirtualenv : Navigate into the directory of the currently activated virtual environment, so you can browse its site-packages, for example.

cdsitepackages : Like the above, but directly into site-packages directory.

lssitepackages : Shows contents of site-packages directory.

Chapter 175: Virtual environment with virtualenvwrapper

Suppose you need to work on three different projects project A, project B and project C. project A and project B need python 3 and some required libraries. But for project C you need python 2.7 and dependent libraries.

So best practice for this is to separate those project environments. To create virtual environment you can use below technique:

Virtualenv, Virtualenvwrapper and Conda

Although we have several options for virtual environment but virtualenvwrapper is most recommended.

Section 175.1: Create virtual environment with virtualenvwrapper

Suppose you need to work on three different projects project A, project B and project C. project A and project B need python 3 and some required libraries. But for project C you need python 2.7 and dependent libraries.

So best practice for this is to separate those project environments. To create virtual environment you can use below technique:

Virtualenv, Virtualenvwrapper and Conda

Although we have several options for virtual environment but virtualenvwrapper is most recommended.

Although we have several options for virtual environment but I always prefer virtualenvwrapper because it has more facility then others.

```
$ pip install virtualenvwrapper

$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ printf '\n%s\n%s\n%s' '# virtualenv' 'export WORKON_HOME=~/virtualenvs' 'source
/home/salayhin/bin/virtualenvwrapper.sh' >> ~/.bashrc
$ source ~/.bashrc

$ mkvirtualenv python_3.5
Installing
setuptools.....................................
.................................................
.................................................
.............................done.
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postactivate New python
executable in python_3.5/bin/python

(python_3.5)$ ls $WORKON_HOME
python_3.5 hook.log
```

Now we can install some software into the environment.

```
(python_3.5)$ pip install django
Downloading/unpacking django
Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
Running setup.py egg_info for package django
Installing collected packages: django
Running setup.py install for django
changing mode of build/scripts-2.6/django-admin.py from 644 to 755
changing mode of /Users/salayhin/Envs/env1/bin/django-admin.py to 755
Successfully installed django
```

We can see the new package with lssitepackages:

```
(python_3.5)$ lssitepackages
Django-1.1.1-py2.6.egg-info easy-install.pth
setuptools-0.6.10-py2.6.egg pip-0.6.3-py2.6.egg
django setuptools.pth
```

We can create multiple virtual environment if we want.

Switch between environments with workon:

```
(python_3.6)$ workon python_3.5
(python_3.5)$ echo $VIRTUAL_ENV
/Users/salayhin/Envs/env1
(python_3.5)$
```

To exit the virtualenv

```
$ deactivate
```

Chapter 176: Create virtual environment with virtualenvwrapper in windows

Section 176.1: Virtual environment with virtualenvwrapper for windows

Suppose you need to work on three different projects project A, project B and project C. project A and project B need python 3 and some required libraries. But for project C you need python 2.7 and dependent libraries.

So best practice for this is to separate those project environments. For creating separate python virtual environment need to follow below steps:

Step 1: Install pip with this command: `python -m pip install -U pip`

Step 2: Then install "virtualenvwrapper-win" package by using command (command can be executed windows power shell):

```
pip install virtualenvwrapper-win
```

Step 3: Create a new virtualenv environment by using command: `mkvirtualenv python_3.5`

Step 4: Activate the environment by using command:

```
workon < environment name>
```

Main commands for virtualenvwrapper:

```
mkvirtualenv <name>
Create a new virtualenv environment named <name>. The environment will be created in WORKON_HOME.

lsvirtualenv
List all of the environments stored in WORKON_HOME.

rmvirtualenv <name>
Remove the environment <name>. Uses folder_delete.bat.

workon [<name>]
If <name> is specified, activate the environment named <name> (change the working virtualenv to
<name>). If a project directory has been defined, we will change into it. If no argument is
specified, list the available environments. One can pass additional option -c after virtualenv name
to cd to virtualenv directory if no projectdir is set.

deactivate
Deactivate the working virtualenv and switch back to the default system Python.

add2virtualenv <full or relative path>
If a virtualenv environment is active, appends <path> to virtualenv_path_extensions.pth inside the
environment's site-packages, which effectively adds <path> to the environment's PYTHONPATH. If a
virtualenv environment is not active, appends <path> to virtualenv_path_extensions.pth inside the
default Python's site-packages. If <path> doesn't exist, it will be created.
```

Chapter 177: sys

The **sys** module provides access to functions and values concerning the program's runtime environment, such as the command line parameters in `sys.argv` or the function `sys.exit()` to end the current process from any point in the program flow.

While cleanly separated into a module, it's actually built-in and as such will always be available under normal circumstances.

Section 177.1: Command line arguments

```
if len(sys.argv) != 4:         # The script name needs to be accounted for as well.
    raise RuntimeError("expected 3 command line arguments")

f = open(sys.argv[1], 'rb')    # Use first command line argument.
start_line = int(sys.argv[2])  # All arguments come as strings, so need to be
end_line = int(sys.argv[3])    # converted explicitly if other types are required.
```

Note that in larger and more polished programs you would use modules such as click to handle command line arguments instead of doing it yourself.

Section 177.2: Script name

```
# The name of the executed script is at the beginning of the argv list.
print('usage:', sys.argv[0], '<filename> <start> <end>')

# You can use it to generate the path prefix of the executed program
# (as opposed to the current module) to access files relative to that,
# which would be good for assets of a game, for instance.
program_file = sys.argv[0]

import pathlib
program_path = pathlib.Path(program_file).resolve().parent
```

Section 177.3: Standard error stream

```
# Error messages should not go to standard output, if possible.
print('ERROR: We have no cheese at all.', file=sys.stderr)

try:
    f = open('nonexistent-file.xyz', 'rb')
except OSError as e:
    print(e, file=sys.stderr)
```

Section 177.4: Ending the process prematurely and returning an exit code

```
def main():
    if len(sys.argv) != 4 or '--help' in sys.argv[1:]:
        print('usage: my_program <arg1> <arg2> <arg3>', file=sys.stderr)

        sys.exit(1)    # use an exit code to signal the program was unsuccessful

    process_data()
```

Chapter 178: ChemPy - python package

ChemPy is a python package designed mainly to solve and address problems in physical, analytical and inorganic Chemistry. It is a free, open-source Python toolkit for chemistry, chemical engineering, and materials science applications.

Section 178.1: Parsing formulae

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)6-3')
ferricyanide.composition == {0: -3, 26: 1, 6: 6, 7: 6}
True
print(ferricyanide.unicode_name)
Fe(CN)₆³⁻
 print(ferricyanide.latex_name + ", " + ferricyanide.html_name)
Fe(CN)_{6}^{3-}, Fe(CN)<sub>6</sub><sup>3-</sup>
 print('%.3f' % ferricyanide.mass)
211.955
```

In composition, the atomic numbers (and 0 for charge) is used as keys and the count of each kind became respective value.

Section 178.2: Balancing stoichiometry of a chemical reaction

```
 from chempy import balance_stoichiometry  # Main reaction in NASA's booster rockets:
 reac, prod = balance_stoichiometry({'NH4ClO4', 'Al'}, {'Al2O3', 'HCl', 'H2O', 'N2'})
 from pprint import pprint
 pprint(reac)
{'Al': 10, 'NH4ClO4': 6}
 pprint(prod)
{'Al2O3': 5, 'H2O': 9, 'HCl': 6, 'N2': 3}
 from chempy import mass_fractions
 for fractions in map(mass_fractions, [reac, prod]):
...     pprint({k: '{0:.3g} wt%'.format(v*100) for k, v in fractions.items()})
...
{'Al': '27.7 wt%', 'NH4ClO4': '72.3 wt%'}
{'Al2O3': '52.3 wt%', 'H2O': '16.6 wt%', 'HCl': '22.4 wt%', 'N2': '8.62 wt%'}
```

Section 178.3: Balancing reactions

```
from chempy import Equilibrium
 from sympy import symbols
 K1, K2, Kw = symbols('K1 K2 Kw')
 e1 = Equilibrium({'MnO4-': 1, 'H+': 8, 'e-': 5}, {'Mn+2': 1, 'H2O': 4}, K1)
 e2 = Equilibrium({'O2': 1, 'H2O': 2, 'e-': 4}, {'OH-': 4}, K2)
 coeff = Equilibrium.eliminate([e1, e2], 'e-')
 coeff
[4, -5]
 redox = e1*coeff[0] + e2*coeff[1]
 print(redox)
20 OH- + 32 H+ + 4 MnO4- = 26 H2O + 4 Mn+2 + 5 O2; K1**4/K2**5
 autoprot = Equilibrium({'H2O': 1}, {'H+': 1, 'OH-': 1}, Kw)
 n = redox.cancel(autoprot)
 n
20
 redox2 = redox + n*autoprot
 print(redox2)
```

```
12 H+ + 4 MnO4- = 4 Mn+2 + 5 O2 + 6 H2O; K1**4*Kw**20/K2**5
```

Section 178.4: Chemical equilibria

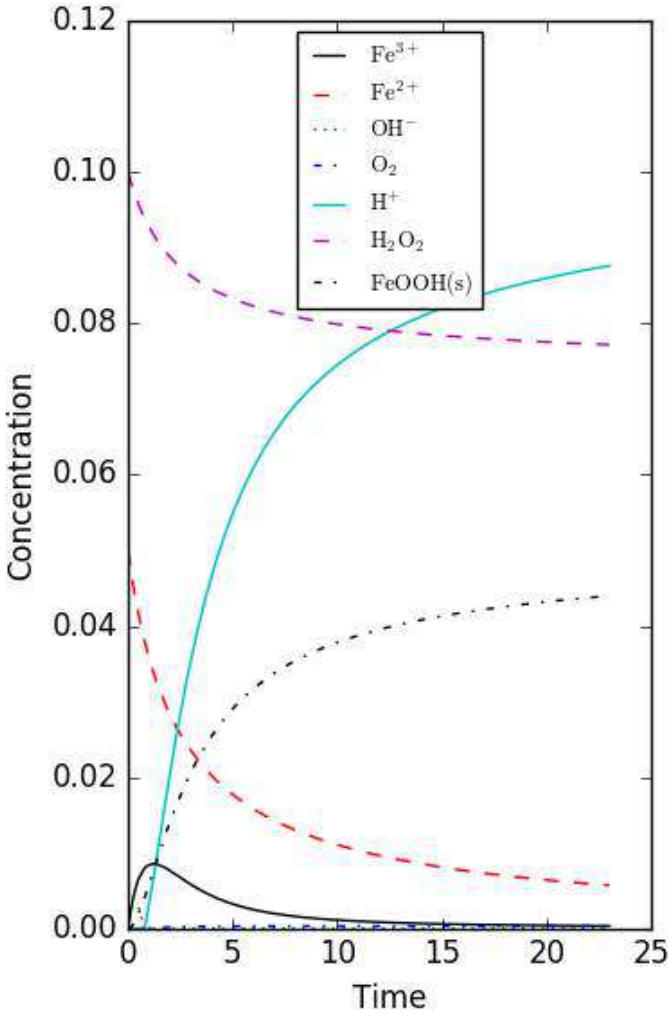
```
 from chempy import Equilibrium
 from chempy.chemistry import Species
 water_autop = Equilibrium({'H2O'}, {'H+', 'OH-'}, 10**-14)  # unit "molar" assumed
 ammonia_prot = Equilibrium({'NH4+'}, {'NH3', 'H+'}, 10**-9.24)  # same here
 from chempy.equilibria import EqSystem
 substances = map(Species.from_formula, 'H2O OH- H+ NH3 NH4+'.split())
 eqsys = EqSystem([water_autop, ammonia_prot], substances)
 print('\n'.join(map(str, eqsys.rxns)))  # "rxns" short for "reactions"
H2O = H+ + OH-; 1e-14
NH4+ = H+ + NH3; 5.75e-10
 from collections import defaultdict
 init_conc = defaultdict(float, {'H2O': 1, 'NH3': 0.1})
 x, sol, sane = eqsys.root(init_conc)
 assert sol['success'] and sane
 print(sorted(sol.keys()))  # see package "pyneqsys" for more info
['fun', 'intermediate_info', 'internal_x_vecs', 'nfev', 'njev', 'success', 'x', 'x_vecs']
 print(', '.join('%.2g' % v for v in x))
1, 0.0013, 7.6e-12, 0.099, 0.0013
```

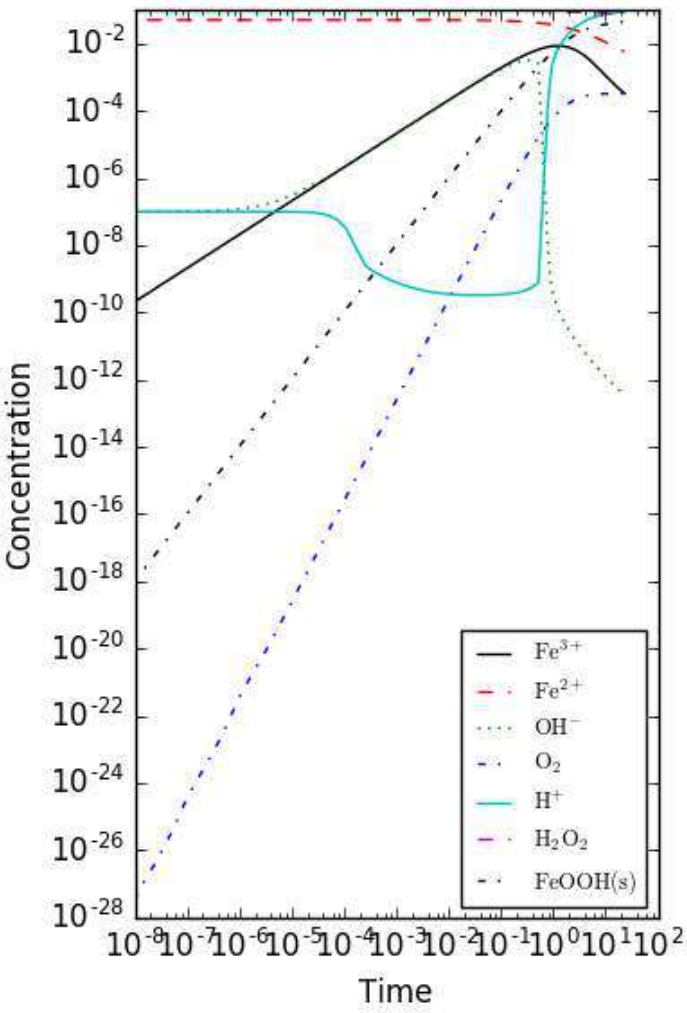
Section 178.5: Ionic strength

```
 from chempy.electrolytes import ionic_strength
     ionic_strength({'Fe+3': 0.050, 'ClO4-': 0.150}) == .3
    True
```

Section 178.6: Chemical kinetics (system of ordinary differential equations)

```
from chempy import ReactionSystem  # The rate constants below are arbitrary
 rsys = ReactionSystem.from_string("""2 Fe+2 + H2O2 -> 2 Fe+3 + 2 OH-; 42
     2 Fe+3 + H2O2 -> 2 Fe+2 + O2 + 2 H+; 17
     H+ + OH- -> H2O; 1e10
     H2O -> H+ + OH-; 1e-4
     Fe+3 + 2 H2O -> FeOOH(s) + 3 H+; 1
     FeOOH(s) + 3 H+ -> Fe+3 + 2 H2O; 2.5""")  # "[H2O]" = 1.0 (actually 55.4 at RT)
 from chempy.kinetics.ode import get_odesys
 odesys, extra = get_odesys(rsys)
 from collections import defaultdict
 import numpy as np
 tout = sorted(np.concatenate((np.linspace(0, 23), np.logspace(-8, 1))))
 c0 = defaultdict(float, {'Fe+2': 0.05, 'H2O2': 0.1, 'H2O': 1.0, 'H+': 1e-7, 'OH-': 1e-7})
 result = odesys.integrate(tout, c0, atol=1e-12, rtol=1e-14)
 import matplotlib.pyplot as plt
 _ = plt.subplot(1, 2, 1)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'])
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.subplot(1, 2, 2)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'], xscale='log', yscale='log')
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.tight_layout()
 plt.show()
```


0.12
0.10
0.08
0.06
0.04
0.02
0.00
Concentration
0
5
10
15
20
25
Time
Fe^{3+}
Fe^{2+}
OH^-
O_2
H^+
H_2O_2
FeOOH(s)


10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
10^{-18}
10^{-20}
10^{-22}
10^{-24}
10^{-26}
10^{-28}
Concentration
10^{-8} 10^{-7} 10^{-6} 10^{-5} 10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1} 10^{2}
Time
Fe^{3+}
Fe^{2+}
OH^-
O_2
H^+
H_2O_2
FeOOH(s)

Chapter 179: pygame

Parameter	Details
count	A positive integer that represents something like the number of channels needed to be reserved.
force	A boolean value (`False` or `True`) that determines whether `find_channel()` has to return a channel (inactive or not) with `True` or not (if there are no inactive channels) with `False`

Pygame is the go-to library for making multimedia applications, especially games, in Python. The official website is http://www.pygame.org/.

Section 179.1: Pygame's mixer module

The `pygame.mixer` module helps control the music used in `pygame` programs. As of now, there are 15 different functions for the `mixer` module.

Initializing

Similar to how you have to initialize `pygame` with `pygame.init()`, you must initialize `pygame.mixer` as well.

By using the first option, we initialize the module using the default values. You can though, override these default options. By using the second option, we can initialize the module using the values we manually put in ourselves. Standard values:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

To check whether we have initialized it or not, we can use `pygame.mixer.get_init()`, which returns `True` if it is and `False` if it is not. To quit/undo the initializing, simply use `pygame.mixer.quit()`. If you want to continue playing sounds with the module, you might have to reinitialize the module.

Possible Actions

As your sound is playing, you can pause it temporarily with `pygame.mixer.pause()`. To resume playing your sounds, simply use `pygame.mixer.unpause()`. You can also fadeout the end of the sound by using `pygame.mixer.fadeout()`. It takes an argument, which is the number of milliseconds it takes to finish fading out the music.

Channels

You can play as many songs as needed as long there are enough open channels to support them. By default, there are 8 channels. To change the number of channels there are, use `pygame.mixer.set_num_channels()`. The argument is a non-negative integer. If the number of channels are decreased, any sounds playing on the removed channels will immediately stop.

To find how many channels are currently being used, call `pygame.mixer.get_channels(count)`. The output is the number of channels that are not currently open. You can also reserve channels for sounds that must be played by using `pygame.mixer.set_reserved(count)`. The argument is also a non-negative integer. Any sounds playing on the newly reserved channels will not be stopped.

You can also find out which channel isn't being used by using `pygame.mixer.find_channel(force)`. Its argument is a bool: either True or False. If there are no channels that are idle and `force` is False, it will return `None`. If `force` is true, it will return the channel that has been playing for the longest time.

Section 179.2: Installing pygame

With `pip`:

```
pip install pygame
```

With `conda`:

```
conda install -c tlatorre pygame=1.9.2
```

Direct download from website : http://www.pygame.org/download.shtml

You can find the suitable installers for Windows and other operating systems.

Projects can also be found at http://www.pygame.org/

Chapter 180: Pyglet

Pyglet is a Python module used for visuals and sound. It has no dependencies on other modules. See [pyglet.org][1] for the official information. [1]: http://pyglet.org

Section 180.1: Installation of Pyglet

Install Python, go into the command line and type:

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

Section 180.2: Hello World in Pyglet

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                          font_name='Times New Roman',
                          font_size=36,
                          x=window.width//2, y=window.height//2,
                          anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

Section 180.3: Playing Sound in Pyglet

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

Section 180.4: Using Pyglet for OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()

@win.event()
def on_draw():
    #OpenGL goes here. Use OpenGL as normal.

pyglet.app.run()
```

Section 180.5: Drawing Points Using Pyglet and OpenGL

```
import pyglet
from pyglet.gl import *
```

```
win = pyglet.window.Window()
glClear(GL_COLOR_BUFFER_BIT)

@win.event
def on_draw():
    glBegin(GL_POINTS)
    glVertex2f(x, y) #x is desired distance from left side of window, y is desired distance from
bottom of window
    #make as many vertexes as you want
    glEnd
```

To connect the points, replace `GL_POINTS` with `GL_LINE_LOOP`.

Chapter 181: Audio

Section 181.1: Working with WAV files

winsound

- Windows environment

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

wave

- Support mono/stereo
- Doesn't support compression/decompression

```
import wave
with wave.open("path_to_wav_file.wav", "rb") as wav_file:    # Open WAV file in read-only mode.
    # Get basic information.
    n_channels = wav_file.getnchannels()      # Number of channels. (1=Mono, 2=Stereo).
    sample_width = wav_file.getsampwidth()    # Sample width in bytes.
    framerate = wav_file.getframerate()       # Frame rate.
    n_frames = wav_file.getnframes()          # Number of frames.
    comp_type = wav_file.getcomptype()        # Compression type (only supports "NONE").
    comp_name = wav_file.getcompname()        # Compression name.

    # Read audio data.
    frames = wav_file.readframes(n_frames)    # Read n_frames new frames.
    assert len(frames) == sample_width * n_frames

# Duplicate to a new WAV file.
with wave.open("path_to_new_wav_file.wav", "wb") as wav_file:    # Open WAV file in write-only
mode.
    # Write audio data.
    params = (n_channels, sample_width, framerate, n_frames, comp_type, comp_name)
    wav_file.setparams(params)
    wav_file.writeframes(frames)
```

Section 181.2: Convert any soundfile with python and ffmpeg

```
from subprocess import check_call

ok = check_call(['ffmpeg','-i','input.mp3','output.wav'])
if ok:
    with open('output.wav', 'rb') as f:
        wav_file = f.read()
```

note:

- http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference
- What are the differences and similarities between ffmpeg, libav, and avconv?

Section 181.3: Playing Windows' beeps

Windows provides an explicit interface through which the `winsound` module allows you to play raw beeps at a given frequency and duration.

```
import winsound
```

```
freq = 2500 # Set frequency To 2500 Hertz
dur = 1000 # Set duration To 1000 ms == 1 second
winsound.Beep(freq, dur)
```

Section 181.4: Audio With Pyglet

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

For further information, see pyglet

Chapter 182: pyaudio

PyAudio provides Python bindings for PortAudio, the cross-platform audio I/O library. With PyAudio, you can easily use Python to play and record audio on a variety of platforms. PyAudio is inspired by:

1.pyPortAudio/fastaudio: Python bindings for PortAudio v18 API.

2.tkSnack: cross-platform sound toolkit for Tcl/Tk and Python.

Section 182.1: Callback Mode Audio I/O

```
"""PyAudio Example: Play a wave file (callback version)."""

import pyaudio
import wave
import time
import sys

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()

# define callback (2)
def callback(in_data, frame_count, time_info, status):
    data = wf.readframes(frame_count)
    return (data, pyaudio.paContinue)

# open stream using callback (3)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True,
                stream_callback=callback)

# start the stream (4)
stream.start_stream()

# wait for stream to finish (5)
while stream.is_active():
    time.sleep(0.1)

# stop stream (6)
stream.stop_stream()
stream.close()
wf.close()

# close PyAudio (7)
p.terminate()
```

In callback mode, PyAudio will call a specified callback function (2) whenever it needs new audio data (to play) and/or when there is new (recorded) audio data available. Note that PyAudio calls the callback function in a separate thread. The function has the following signature `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` and must return a tuple containing `frame_count` frames of audio data and a flag

signifying whether there are more frames to play/record.

Start processing the audio stream using **pyaudio.Stream.start_stream()** (4), which will call the callback function repeatedly until that function returns **pyaudio.paComplete**.

To keep the stream active, the main thread must not terminate, e.g., by sleeping (5).

Section 182.2: Blocking Mode Audio I/O

"""PyAudio Example: Play a wave file."""

```
import pyaudio
import wave
import sys

CHUNK = 1024

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()

# open stream (2)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True)

# read data
data = wf.readframes(CHUNK)

# play stream (3)
while len(data) > 0:
    stream.write(data)
    data = wf.readframes(CHUNK)

# stop stream (4)
stream.stop_stream()
stream.close()

# close PyAudio (5)
p.terminate()
```

To use PyAudio, first instantiate PyAudio using **pyaudio.PyAudio()** (1), which sets up the portaudio system.

To record or play audio, open a stream on the desired device with the desired audio parameters using **pyaudio.PyAudio.open()** (2). This sets up a **pyaudio.Stream** to play or record audio.

Play audio by writing audio data to the stream using **pyaudio.Stream.write()**, or read audio data from the stream using **pyaudio.Stream.read()**. (3)

Note that in "*blocking mode*", each **pyaudio.Stream.write()** or **pyaudio.Stream.read()** blocks until all the given/requested frames have been played/recorded. Alternatively, to generate audio data on the fly or immediately process recorded audio data, use the "callback mode"(*refer the example on call back mode*)

Use pyaudio.Stream.stop_stream() to pause playing/recording, and **pyaudio.Stream.close()** to terminate the stream. (4)

Finally, terminate the portaudio session using **pyaudio.PyAudio.terminate()** (5)

Chapter 183: shelve

Shelve is a python module used to store objects in a file. The shelve module implements persistent storage for arbitrary Python objects which can be pickled, using a dictionary-like API. The shelve module can be used as a simple persistent storage option for Python objects when a relational database is overkill. The shelf is accessed by keys, just as with a dictionary. The values are pickled and written to a database created and managed by anydbm.

Section 183.1: Creating a new Shelf

The simplest way to use shelve is via the **DbfilenameShelf** class. It uses anydbm to store the data. You can use the class directly, or simply call **shelve.open()**:

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```

To access the data again, open the shelf and use it like a dictionary:

```
    import shelve

    s = shelve.open('test_shelf.db')
    try:
        existing = s['key1']
    finally:
        s.close()

print existing
```

If you run both sample scripts, you should see:

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

The **dbm** module does not support multiple applications writing to the same database at the same time. If you know your client will not be modifying the shelf, you can tell shelve to open the database read-only.

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

If your program tries to modify the database while it is opened read-only, an access error exception is generated. The exception type depends on the database module selected by anydbm when the database was created.

Section 183.2: Sample code for shelve

To shelve an object, first import the module and then assign the object value as follows:

```
import shelve
 database = shelve.open(filename.suffix)
 object = Object()
 database['key'] = object
```

Section 183.3: To summarize the interface (key is a string, data is an arbitrary object):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

Section 183.4: Write-back

Shelves do not track modifications to volatile objects, by default. That means if you change the contents of an item stored in the shelf, you must update the shelf explicitly by storing the item again.

```
import shelve

s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
```

```
    print s['key1']
finally:
    s.close()
```

In this example, the dictionary at 'key1' is not stored again, so when the shelf is re-opened, the changes have not been preserved.

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

To automatically catch changes to volatile objects stored in the shelf, open the shelf with writeback enabled. The writeback flag causes the shelf to remember all of the objects retrieved from the database using an in-memory cache. Each cache object is also written back to the database when the shelf is closed.

```
import shelve

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

Although it reduces the chance of programmer error, and can make object persistence more transparent, using writeback mode may not be desirable in every situation. The cache consumes extra memory while the shelf is open, and pausing to write every cached object back to the database when it is closed can take extra time. Since there is no way to tell if the cached objects have been modified, they are all written back. If your application reads data more than it writes, writeback will add more overhead than you might want.

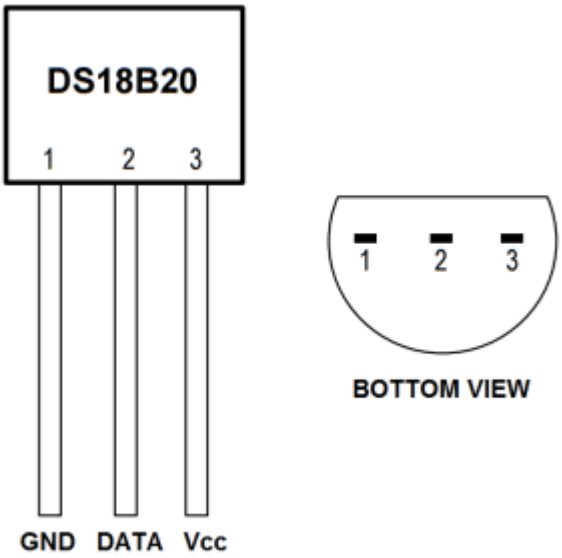
```
$ python shelve_create.py
$ python shelve_writeback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

Chapter 184: IoT Programming with Python and Raspberry PI

Section 184.1: Example - Temperature sensor

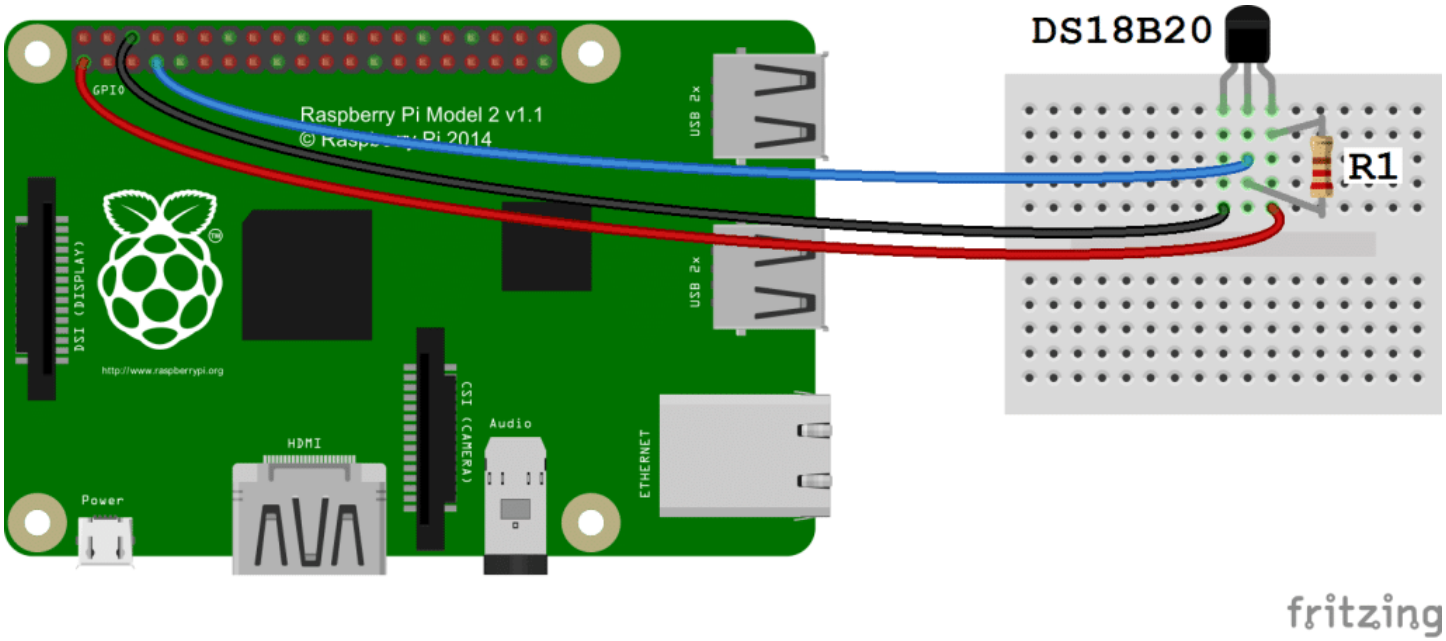
Interfacing of DS18B20 with Raspberry pi

Connection of DS18B20 with Raspberry pi



You can see there are three terminal

1. Vcc
2. Gnd
3. Data (One wire protocol)



R1 is 4.7k ohm resistance for pulling up the voltage level

1. **Vcc** should be connected to any of the 5v or 3.3v pins of Raspberry pi (PIN : 01, 02, 04, 17).
2. **Gnd** should be connected to any of the Gnd pins of Raspberry pi (PIN : 06, 09, 14, 20, 25).

3. **DATA** should be connected to (PIN : 07)

Enabling the one-wire interface from the RPi side

4. Login to Raspberry pi using putty or any other linux/unix terminal.
5. After login, open the /boot/config.txt file in your favourite browser.

   nano /boot/config.txt

6. Now add the this line `dtoverlay=w1-gpio` to the end of the file.
7. Now reboot the Raspberry pi `sudo reboot`.
8. Log in to Raspberry pi, and run `sudo modprobe g1-gpio`
9. Then run `sudo modprobe w1-therm`
10. Now go to the directory /sys/bus/w1/devices `cd /sys/bus/w1/devices`
11. Now you will found out a virtual directory created of your temperature sensor starting from 28-********.
12. Go to this directory `cd 28-********`
13. Now there is a file name **w1-slave**, This file contains the temperature and other information like CRC. `cat w1-slave`.

Now write a module in python to read the temperature

```
import glob
import time

RATE = 30
sensor_dirs = glob.glob("/sys/bus/w1/devices/28*")

if len(sensor_dirs) != 0:
    while True:
        time.sleep(RATE)
        for directories in sensor_dirs:
            temperature_file = open(directories + "/w1_slave")
            # Reading the files
            text = temperature_file.read()
            temperature_file.close()
            # Split the text with new lines (\n) and select the second line.
            second_line = text.split("\n")[1]
            # Split the line into words, and select the 10th word
            temperature_data = second_line.split(" ")[9]
            # We will read after ignoring first two character.
            temperature = float(temperature_data[2:])
            # Now normalise the temperature by dividing 1000.
            temperature = temperature / 1000
            print 'Address : '+str(directories.split('/')[-1])+', Temperature : '+str(temperature)
```

Above python module will print the temperature vs address for infinite time. RATE parameter is defined to change or adjust the frequency of temperature query from the sensor.

GPIO pin diagram

1. [https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.pn

g][3]

Chapter 185: kivy - Cross-platform Python Framework for NUI Development

NUI : A natural user interface (NUI) is a system for human-computer interaction that the user operates through intuitive actions related to natural, everyday human behavior.

Kivy is a Python library for development of multi-touch enabled media rich applications which can be installed on different devices. Multi-touch refers to the ability of a touch-sensing surface (usually a touch screen or a trackpad) to detect or sense input from two or more points of contact simultaneously.

Section 185.1: First App

To create an kivy application

1. sub class the **app** class
2. Implement the **build** method, which will return the widget.
3. Instantiate the class an invoke the **run**.

```
from kivy.app import App
from kivy.uix.label import Label

class Test(App):
    def build(self):
        return Label(text='Hello world')

if __name__ == '__main__':
    Test().run()
```

Explanation

```
from kivy.app import App
```

The above statement will import the parent class **app**. This will be present in your installation directory your_installtion_directory/kivy/app.py

```
from kivy.uix.label import Label
```

The above statement will import the ux element **Label**. All the ux element are present in your installation directory your_installation_directory/kivy/uix/.

```
class Test(App):
```

The above statement is for to create your app and class name will be your app name. This class is inherited the parent app class.

```
def build(self):
```

The above statement override the build method of app class. Which will return the widget that needs to be shown when you will start the app.

```
return Label(text='Hello world')
```

The above statement is the body of the build method. It is returning the Label with its text **Hello world**.

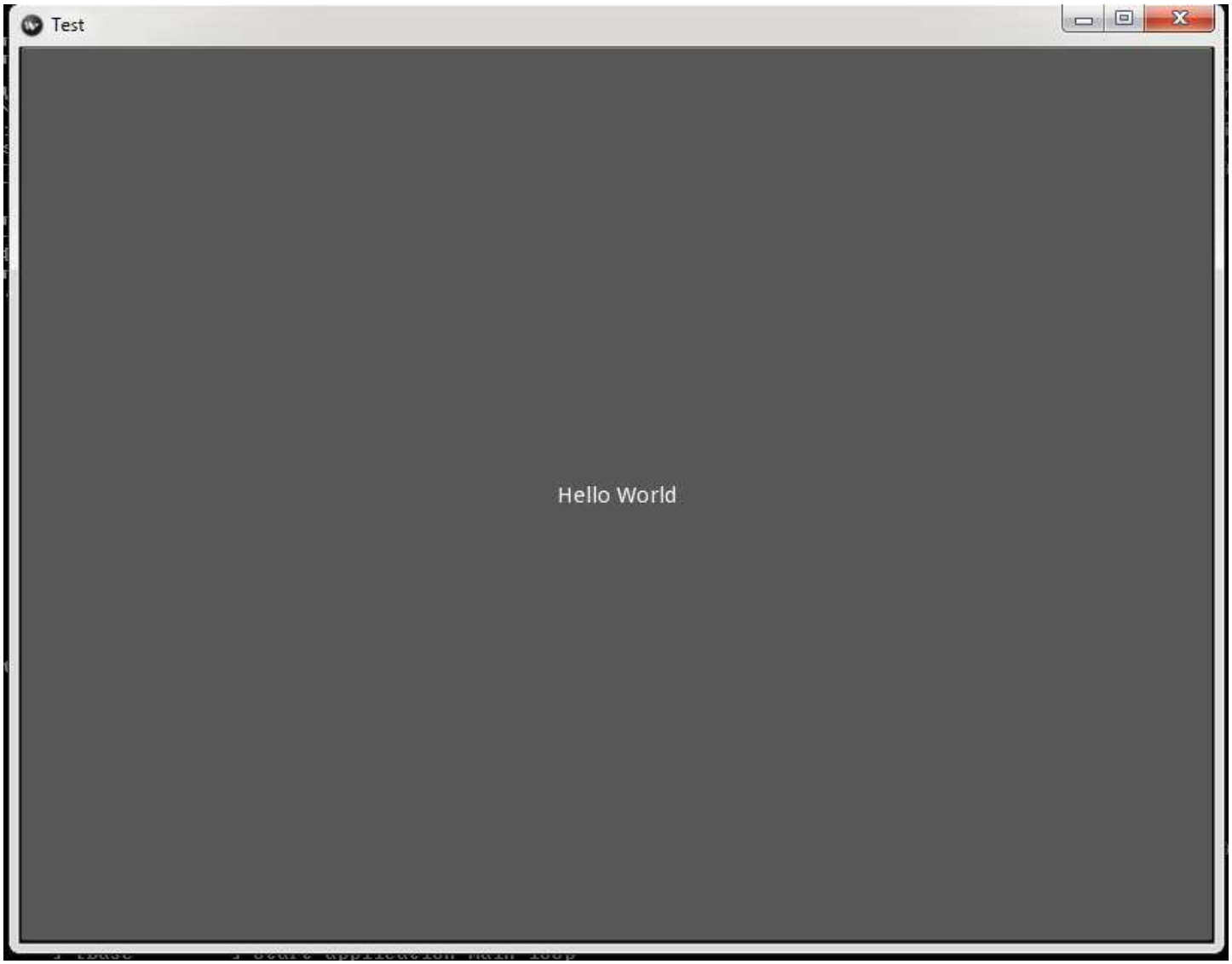
```
if __name__ == '__main__':
```

The above statement is the entry point from where python interpreter start executing your app.

```
Test().run()
```

The above statement Initialise your Test class by creating its instance. And invoke the app class function run().

Your app will look like the below picture.



Chapter 186: Pandas Transform: Preform operations on groups and concatenate the results

Section 186.1: Simple transform

First, Let's create a dummy dataframe

We assume that a customer can have n orders, an order can have m items, and items can be ordered more multiple times

```
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry', 'strawberry']

# And this is how the dataframe looks like:
print(orders_df)
#     customer_id  order_id        item
# 0             1         1      apples
# 1             1         1   chocolate
# 2             1         1   chocolate
# 3             1         2      coffee
# 4             1         2      coffee
# 5             2         3      apples
# 6             2         3     bananas
# 7             3         4      coffee
# 8             3         5   milkshake
# 9             3         6   chocolate
# 10            3         6  strawberry
# 11            3         6  strawberry
```

.

.

Now, we will use pandas `transform` function to count the number of orders per customer

```
# First, we define the function that will be applied per customer_id
count_number_of_orders = lambda x: len(x.unique())

# And now, we can transform each group using the logic defined above
orders_df['number_of_orders_per_cient'] = (               # Put the results into a new column that
is called 'number_of_orders_per_cient'
                     orders_df                            # Take the original dataframe
                    .groupby(['customer_id'])['order_id'] # Create a separate group for each
customer_id & select the order_id
                    .transform(count_number_of_orders))   # Apply the function to each group
separately

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  number_of_orders_per_cient
# 0             1         1      apples                           2
# 1             1         1   chocolate                           2
# 2             1         1   chocolate                           2
# 3             1         2      coffee                           2
# 4             1         2      coffee                           2
```

```
# 5             2         3      apples                          1
# 6             2         3     bananas                          1
# 7             3         4      coffee                          3
# 8             3         5   milkshake                          3
# 9             3         6   chocolate                          3
# 10            3         6  strawberry                          3
# 11            3         6  strawberry                          3
```

Section 186.2: Multiple results per group

Using `transform` functions that return sub-calculations per group

In the previous example, we had one result per client. However, functions returning different values for the group can also be applied.

```
# Create a dummy dataframe
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry', 'strawberry']


# Let's try to see if the items were ordered more than once in each orders

# First, we define a function that will be applied per group
def multiple_items_per_order(_items):
    # Apply .duplicated, which will return True is the item occurs more than once.
    multiple_item_bool = _items.duplicated(keep=False)
    return(multiple_item_bool)

# Then, we transform each group according to the defined function
orders_df['item_duplicated_per_order'] = (                    # Put the results into a new column
                        orders_df                             # Take the orders dataframe
                        .groupby(['order_id'])['item']        # Create a separate group for each
order_id & select the item
                        .transform(multiple_items_per_order)) # Apply the defined function to each
group separately

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  item_duplicated_per_order
# 0             1         1      apples                      False
# 1             1         1   chocolate                       True
# 2             1         1   chocolate                       True
# 3             1         2      coffee                       True
# 4             1         2      coffee                       True
# 5             2         3      apples                      False
# 6             2         3     bananas                      False
# 7             3         4      coffee                      False
# 8             3         5   milkshake                      False
# 9             3         6   chocolate                      False
# 10            3         6  strawberry                       True
# 11            3         6  strawberry                       True
```

Chapter 187: Similarities in syntax, Differences in meaning: Python vs. JavaScript

It sometimes happens that two languages put different meanings on the same or similar syntax expression. When the both languages are of interest for a programmer, clarifying these bifurcation points helps to better understand the both languages in their basics and subtleties.

Section 187.1: `in` with lists

```
2 in [2, 3]
```

In Python this evaluates to True, but in JavaScript to false. This is because in Python in checks if a value is contained in a list, so 2 is in [2, 3] as its first element. In JavaScript in is used with objects and checks if an object contains the property with the name expressed by the value. So JavaScript considers [2, 3] as an object or a key-value map like this:

```
{'0': 2, '1': 3}
```

and checks if it has a property or a key '2' in it. Integer 2 is silently converted to string '2'.

Chapter 188: Call Python from C#

The documentation provides a sample implementation of the inter-process communication between C# and Python scripts.

Section 188.1: Python script to be called by C# application

```
import sys
import json

# load input arguments from the text file
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
    input_args = json.loads( data_file.read() )

# cast strings to floats
x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]

print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

Section 188.2: C# code calling Python script

```
using MongoDB.Bson;
using System;
using System.Diagnostics;
using System.IO;

namespace python_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // full path to .py file
            string pyScriptPath = "...../sum.py";
            // convert input arguments to JSON string
            BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");

            bool saveInputFile = false;

            string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());

            string outputString = null;
            // create new process start info
            ProcessStartInfo prcStartInfo = new ProcessStartInfo
            {
                // full path of the Python interpreter 'python.exe'
                FileName = "python.exe", // string.Format(@"""{0}""", "python.exe"),
                UseShellExecute = false,
                RedirectStandardOutput = true,
                CreateNoWindow = false
            };

            try
            {
                // write input arguments to .txt file
                using (StreamWriter sw = new StreamWriter(argsFile))
```

```
                {
                    sw.WriteLine(argsBson);
                    prcStartInfo.Arguments = string.Format("{0} {1}", string.Format(@"""{0}""",
pyScriptPath), string.Format(@"""{0}""", argsFile));
                }
                // start process
                using (Process process = Process.Start(prcStartInfo))
                {
                    // read standard output JSON string
                    using (StreamReader myStreamReader = process.StandardOutput)
                    {
                        outputString = myStreamReader.ReadLine();
                        process.WaitForExit();
                    }
                }
            }
            finally
            {
                // delete/save temporary .txt file
                if (!saveInputFile)
                {
                    File.Delete(argsFile);
                }
            }
            Console.WriteLine(outputString);
        }
    }
}
```

Chapter 189: ctypes

`ctypes` is a python built-in library that invokes exported functions from native compiled libraries.

Note: Since this library handles compiled code, it is relatively OS dependent.

Section 189.1: ctypes arrays

As any good C programmer knows, a single value won't get you that far. What will really get us going are arrays!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

This is not an actual array, but it's pretty darn close! We created a class that denotes an array of 16 `int`s.

Now all we need to do is to initialize it:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

Now `arr` is an actual array that contains the numbers from 0 to 15.

They can be accessed just like any list:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

And just like any other `ctypes` object, it also has a size and a location:

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc000l0ff'
```

Section 189.2: Wrapping functions for ctypes

In some cases, a C function accepts a function pointer. As avid `ctypes` users, we would like to use those functions, and even pass python function as arguments.

Let's define a function:

```
>>> def max(x, y):
        return x if x >= y else y
```

Now, that function takes two arguments and returns a result of the same type. For the sake of the example, let's assume that type is an int.

Like we did on the array example, we can define an object that denotes that prototype:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
```

```
<CFunctionType object at 0xdeadbeef>
```

That prototype denotes a function that returns an `c_int` (the first argument), and accepts two `c_int` arguments (the other arguments).

Now let's wrap the function:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Function prototypes have on more usage: They can wrap `ctypes` function (like `libc.ntohl`) and verify that the correct arguments are used when invoking the function.

```
>>> libc.ntohl() # garbage in - garbage out
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: this function takes at least 1 argument (0 given)
```

Section 189.3: Basic usage

Let's say we want to use `libc`'s `ntohl` function.

First, we must load `libc.so`:

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Then, we get the function object:

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

And now, we can simply invoke the function:

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

Which does exactly what we expect it to do.

Section 189.4: Common pitfalls

Failing to load a file

The first possible error is failing to load the library. In that case an OSError is usually raised.

This is either because the file doesn't exists (or can't be found by the OS):

```
>>> cdll.LoadLibrary("foobar.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: foobar.so: cannot open shared object file: No such file or directory
```

As you can see, the error is clear and pretty indicative.

The second reason is that the file is found, but is not of the correct format.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

In this case, the file is a script file and not a `.so` file. This might also happen when trying to open a `.dll` file on a Linux machine or a 64bit file on a 32bit python interpreter. As you can see, in this case the error is a bit more vague, and requires some digging around.

Failing to access a function

Assuming we successfully loaded the `.so` file, we then need to access our function like we've done on the first example.

When a non-existing function is used, an `AttributeError` is raised:

```
>>> libc.foo
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 360, in __getattr__
    func = self.__getitem__(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 365, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: undefined symbol: foo
```

Section 189.5: Basic ctypes object

The most basic object is an int:

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

Now, `obj` refers to a chunk of memory containing the value 12.

That value can be accessed directly, and even modified:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

Since `obj` refers to a chunk of memory, we can also find out its size and location:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

Section 189.6: Complex usage

Let's combine all of the examples above into one complex scenario: using `libc`'s `lfind` function.

For more details about the function, read the man page. I urge you to read it before going on.

First, we'll define the proper prototypes:

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint, compar_proto)
```

Then, let's create the variables:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

And now we define the comparison function:

```
>>> def compar(x, y):
        return x.contents.value - y.contents.value
```

Notice that `x`, and `y` are `POINTER(c_int)`, so we need to dereference them and take their values in order to actually compare the value stored in the memory.

Now we can combine everything together:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

`ptr` is the returned void pointer. If `key` wasn't found in `arr`, the value would be `None`, but in this case we got a valid value.

Now we can convert it and access the value:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

Also, we can see that `ptr` points to the correct value inside `arr`:

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

Chapter 190: Writing extensions

Section 190.1: Hello World with C Extension

The following C source file (which we will call `hello.c` for demonstration purposes) produces an extension module named `hello` that contains a single function `greet()`:

```
#include <Python.h>
#include <stdio.h>

#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif

static PyObject *hello_greet(PyObject *self, PyObject *args)
{
    const char *input;
    if (!PyArg_ParseTuple(args, "s", &input)) {
        return NULL;
    }
    printf("%s", input);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
    { "greet", hello_greet, METH_VARARGS, "Greet the user" },
    { NULL, NULL, 0, NULL }
};

#ifdef IS_PY3K
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT, "hello", NULL, -1, HelloMethods
};

PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&hellomodule);
}
#else
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", HelloMethods);
}
#endif
```

To compile the file with the `gcc` compiler, run the following command in your favourite terminal:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

To execute the `greet()` function that we wrote earlier, create a file in the same directory, and call it `hello.py`

```
import hello          # imports the compiled library
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

Section 190.2: C Extension Using c++ and Boost

This is a basic example of a *C Extension* using C++ and Boost.

C++ Code

C++ code put in hello.cpp:

```
#include <boost/python/module.hpp>
#include <boost/python/list.hpp>
#include <boost/python/class.hpp>
#include <boost/python/def.hpp>

// Return a hello world string.
std::string get_hello_function()
{
   return "Hello world!";
}

// hello class that can return a list of count hello world strings.
class hello_class
{
public:

   // Taking the greeting message in the constructor.
   hello_class(std::string message) : _message(message) {}

   // Returns the message count times in a python list.
   boost::python::list as_list(int count)
   {
      boost::python::list res;
      for (int i = 0; i < count; ++i) {
         res.append(_message);
      }
      return res;
   }

private:
   std::string _message;
};


// Defining a python module naming it to "hello".
BOOST_PYTHON_MODULE(hello)
{
   // Here you declare what functions and classes that should be exposed on the module.

   // The get_hello_function exposed to python as a function.
   boost::python::def("get_hello", get_hello_function);

   // The hello_class exposed to python as a class.
   boost::python::class_<hello_class>("Hello", boost::python::init<std::string>())
      .def("as_list", &hello_class::as_list)
      ;
}
```

To compile this into a python module you will need the python headers and the boost libraries. This example was made on Ubuntu 12.04 using python 3.4 and gcc. Boost is supported on many platforms. In case of Ubuntu the needed packages was installed using:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Compiling the source file into a .so-file that can later be imported as a module provided it is on the python path:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -lboost_system
-l:libpython3.4m.so
```

The python code in the file example.py:

```
import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))
```

Then `python3 example.py` will give the following output:

```
Hello world!
['World hello!', 'World hello!', 'World hello!']
```

Section 190.3: Passing an open file to C Extensions

Pass an open file object from Python to C extension code.

You can convert the file to an integer file descriptor using `PyObject_AsFileDescriptor` function:

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

To convert an integer file descriptor back into a python object, use `PyFile_FromFd`.

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

Chapter 191: Python Lex-Yacc

PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc.

Section 191.1: Getting Started with PLY

To install PLY on your machine for python2/3, follow the steps outlined below:

1. Download the source code from here.
2. Unzip the downloaded zip file
3. Navigate into the unzipped `ply-3.10` folder
4. Run the following command in your terminal: `python setup.py install`

If you completed all the above, you should now be able to use the PLY module. You can test it out by opening a python interpreter and typing `import ply.lex`.

Note: Do *not* use `pip` to install PLY, it will install a broken distribution on your machine.

Section 191.2: The "Hello, World!" of PLY - A Simple Calculator

Let's demonstrate the power of PLY with a simple example: this program will take an arithmetic expression as a string input, and attempt to solve it.

Open up your favourite editor and copy the following code:

```
from ply import lex
import ply.yacc as yacc

tokens = (
    'PLUS',
    'MINUS',
    'TIMES',
    'DIV',
    'LPAREN',
    'RPAREN',
    'NUMBER',
)

t_ignore = ' \t'

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIV     = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER( t ) :
    r'[0-9]+'
    t.value = int( t.value )
    return t

def t_newline( t ):
  r'\n+'
  t.lexer.lineno += len( t.value )

def t_error( t ):
```

```
  print("Invalid Token:",t.value[0])
  t.lexer.skip( 1 )

lexer = lex.lex()

precedence = (
    ( 'left', 'PLUS', 'MINUS' ),
    ( 'left', 'TIMES', 'DIV' ),
    ( 'nonassoc', 'UMINUS' )
)

def p_add( p ) :
    'expr : expr PLUS expr'
    p[0] = p[1] + p[3]

def p_sub( p ) :
    'expr : expr MINUS expr'
    p[0] = p[1] - p[3]

def p_expr2uminus( p ) :
    'expr : MINUS expr %prec UMINUS'
    p[0] = - p[2]

def p_mult_div( p ) :
    '''expr : expr TIMES expr
            | expr DIV expr'''

    if p[2] == '*' :
        p[0] = p[1] * p[3]
    else :
        if p[3] == 0 :
            print("Can't divide by 0")
            raise ZeroDivisionError('integer division by 0')
        p[0] = p[1] / p[3]

def p_expr2NUM( p ) :
    'expr : NUMBER'
    p[0] = p[1]

def p_parens( p ) :
    'expr : LPAREN expr RPAREN'
    p[0] = p[2]

def p_error( p ):
    print("Syntax error in input!")

parser = yacc.yacc()

res = parser.parse("-4*-(3-5)") # the input
print(res)
```

Save this file as `calc.py` and run it.

Output:

```
-8
```

Which is the right answer for `-4 * - (3 - 5)`.

Section 191.3: Part 1: Tokenizing Input with Lex

There are two steps that the code from example 1 carried out: one was *tokenizing* the input, which means it looked for symbols that constitute the arithmetic expression, and the second step was *parsing*, which involves analysing the extracted tokens and evaluating the result.

This section provides a simple example of how to *tokenize* user input, and then breaks it down line by line.

```
    import ply.lex as lex

    # List of token names. This is always required
    tokens = [
       'NUMBER',
       'PLUS',
       'MINUS',
       'TIMES',
       'DIVIDE',
       'LPAREN',
       'RPAREN',
    ]

    # Regular expression rules for simple tokens
    t_PLUS    = r'\+'
    t_MINUS   = r'-'
    t_TIMES   = r'\*'
    t_DIVIDE  = r'/'
    t_LPAREN  = r'\('
    t_RPAREN  = r'\)'

    # A regular expression rule with some action code
    def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t

    # Define a rule so we can track line numbers
    def t_newline(t):
        r'\n+'
        t.lexer.lineno += len(t.value)

    # A string containing ignored characters (spaces and tabs)
    t_ignore  = ' \t'

    # Error handling rule
    def t_error(t):
        print("Illegal character '%s'" % t.value[0])
        t.lexer.skip(1)

    # Build the lexer
    lexer = lex.lex()

    # Give the lexer some input
    lexer.input(data)

    # Tokenize
    while True:
        tok = lexer.token()
        if not tok:
            break      # No more input
        print(tok)
```

Save this file as `calclex.py`. We'll be using this when building our Yacc parser.

Breakdown

1. Import the module using `import ply.lex`
2. All lexers must provide a list called `tokens` that defines all of the possible token names that can be produced by the lexer. This list is always required.

```
tokens = [
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
]
```

`tokens` could also be a tuple of strings (rather than a string), where each string denotes a token as before.

3. The regex rule for each string may be defined either as a string or as a function. In either case, the variable name should be prefixed by t_ to denote it is a rule for matching tokens.

   - For simple tokens, the regular expression can be specified as strings: `t_PLUS = r'\+'`
   - If some kind of action needs to be performed, a token rule can be specified as a function.

     ```
     def t_NUMBER(t):
         r'\d+'
         t.value = int(t.value)
         return t
     ```

     Note, the rule is specified as a doc string within the function. The function accepts one argument which is an instance of `LexToken`, performs some action and then returns back the argument.

     If you want to use an external string as the regex rule for the function instead of specifying a doc string, consider the following example:

     ```
     @TOKEN(identifier)          # identifier is a string holding the regex
     def t_ID(t):
         ...      # actions
     ```

   - An instance of `LexToken` object (let's call this object `t`) has the following attributes:

     1. `t.type` which is the token type (as a string) (eg: `'NUMBER'`, `'PLUS'`, etc). By default, `t.type` is set to the name following the `t_` prefix.
     2. `t.value` which is the lexeme (the actual text matched)
     3. `t.lineno` which is the current line number (this is not automatically updated, as the lexer knows nothing of line numbers). Update lineno using a function called `t_newline`.

     ```
     def t_newline(t):
         r'\n+'
         t.lexer.lineno += len(t.value)
     ```

4. `t.lexpos` which is the position of the token relative to the beginning of the input text.

- If nothing is returned from a regex rule function, the token is discarded. If you want to discard a token, you can alternatively add t_ignore_ prefix to a regex rule variable instead of defining a function for the same rule.

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

...Is the same as:

```
t_ignore_COMMENT = r'\#.*'
```

This is of course invalid if you're carrying out some action when you see a comment. In which case, use a function to define the regex rule.

If you haven't defined a token for some characters but still want to ignore it, use `t_ignore = "<characters to ignore>"` (these prefixes are necessary):

```
t_ignore_COMMENT = r'\#.*'
t_ignore  = ' \t'    # ignores spaces and tabs
```

- When building the master regex, lex will add the regexes specified in the file as follows:

1. Tokens defined by functions are added in the same order as they appear in the file.
2. Tokens defined by strings are added in decreasing order of the string length of the string defining the regex for that token.

If you are matching `==` and `=` in the same file, take advantage of these rules.

- Literals are tokens that are returned as they are. Both `t.type` and `t.value` will be set to the character itself. Define a list of literals as such:

```
literals = [ '+', '-', '*', '/' ]
```

or,

```
literals = "+-*/"
```

It is possible to write token functions that perform additional actions when literals are matched. However, you'll need to set the token type appropriately. For example:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{'  # Set token type to the expected literal (ABSOLUTE MUST if this is a
literal)
    return t
```

- Handle errors with t_error function.

```
# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1) # skip the illegal token (don't process it)
```

In general, `t.lexer.skip(n)` skips n characters in the input string.

4. Final preparations:

Build the lexer using `lexer = lex.lex()`.

You can also put everything inside a class and call use instance of the class to define the lexer. Eg:

```
import ply.lex as lex
class MyLexer(object):
      ...     # everything relating to token rules and error handling comes here as usual

      # Build the lexer
      def build(self, **kwargs):
          self.lexer = lex.lex(module=self, **kwargs)

      def test(self, data):
          self.lexer.input(data)
          for token in self.lexer.token():
              print(token)

      # Build the lexer and try it out

m = MyLexer()
m.build()           # Build the lexer
m.test("3 + 4")     #
```

Provide input using `lexer.input(data)` where data is a string

To get the tokens, use `lexer.token()` which returns tokens matched. You can iterate over lexer in a loop as in:

```
for i in lexer:
    print(i)
```

Section 191.4: Part 2: Parsing Tokenized Input with Yacc

This section explains how the tokenized input from Part 1 is processed - it is done using Context Free Grammars (CFGs). The grammar must be specified, and the tokens are processed according to the grammar. Under the hood, the parser uses an LALR parser.

```
# Yacc example

import ply.yacc as yacc

# Get the token map from the lexer. This is required.
from calclex import tokens
```

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print(result)
```

Breakdown

- Each grammar rule is defined by a function where the docstring to that function contains the appropriate context-free grammar specification. The statements that make up the function body implement the semantic actions of the rule. Each function accepts a single argument p that is a sequence containing the values of each grammar symbol in the corresponding rule. The values of `p[i]` are mapped to grammar symbols as shown here:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]
```

```
    p[0] = p[1] + p[3]
```

- For tokens, the "value" of the corresponding `p[i]` is the same as the `p.value` attribute assigned in the lexer module. So, `PLUS` will have the value `+`.
- For non-terminals, the value is determined by whatever is placed in `p[0]`. If nothing is placed, the value is None. Also, `p[-1]` is not the same as `p[3]`, since `p` is not a simple list (`p[-1]` can specify embedded actions (not discussed here)).

Note that the function can have any name, as long as it is preceded by `p_`.

- The `p_error(p)` rule is defined to catch syntax errors (same as `yyerror` in yacc/bison).
- Multiple grammar rules can be combined into a single function, which is a good idea if productions have a similar structure.

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

- Character literals can be used instead of tokens.

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
       term       : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

  Of course, the literals must be specified in the lexer module.

- Empty productions have the form `'''symbol : '''`
- To explicitly set the start symbol, use `start = 'foo'`, where `foo` is some non-terminal.
- Setting precedence and associativity can be done using the precedence variable.

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),  # Nonassociative operators
```

```
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

Tokens are ordered from lowest to highest precedence. `nonassoc` means that those tokens do not associate. This means that something like `a < b < c` is illegal whereas `a < b` is still legal.

- `parser.out` is a debugging file that is created when the yacc program is executed for the first time. Whenever a shift/reduce conflict occurs, the parser always shifts.

Chapter 192: Unit Testing

Section 192.1: Test Setup and Teardown within a unittest.TestCase

Sometimes we want to prepare a context for each test to be run under. The `setUp` method is run prior to each test in the class. `tearDown` is run at the end of every test. These methods are optional. Remember that TestCases are often used in cooperative multiple inheritance so you should be careful to always call `super` in these methods so that base class's `setUp` and `tearDown` methods also get called. The base implementation of `TestCase` provides empty `setUp` and `tearDown` methods so that they can be called without raising exceptions:

```
import unittest


class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1,2,3,4,5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []


if __name__ == '__main__':
    unittest.main()
```

Note that in python2.7+, there is also the `addCleanup` method that registers functions to be called after the test is run. In contrast to `tearDown` which only gets called if `setUp` succeeds, functions registered via `addCleanup` will be called even in the event of an unhandled exception in `setUp`. As a concrete example, this method can frequently be seen removing various mocks that were registered while the test was running:

```
import unittest
import some_module


class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

        # Replace `some_module.method` with a `mock.Mock`
        my_patch = mock.patch.object(some_module, 'method')
        my_patch.start()

        # When the test finishes running, put the original method back.
        self.addCleanup(my_patch.stop)
```

Another benefit of registering cleanups this way is that it allows the programmer to put the cleanup code next to the setup code and it protects you in the event that a subclasser forgets to call `super` in `tearDown`.

Section 192.2: Asserting on Exceptions

You can test that a function throws an exception with the built-in unittest through two different methods.

Using a context manager

```
def division_function(dividend, divisor):
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

This will run the code inside of the context manager and, if it succeeds, it will fail the test because the exception was not raised. If the code raises an exception of the correct type, the test will continue.

You can also get the content of the raised exception if you want to execute additional assertions against it.

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
            x = division_function(1, 0)

        self.assertEqual(ex.message, 'integer division or modulo by zero')
```

By providing a callable function

```
def division_function(dividend, divisor):
    """
    Dividing two numbers.

    :type dividend: int
    :type divisor: int

    :raises: ZeroDivisionError if divisor is zero (0).
    :rtype: int
    """
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

The exception to check for must be the first parameter, and a callable function must be passed as the second parameter. Any other parameters specified will be passed directly to the function that is being called, allowing you to specify the parameters that trigger the exception.

Section 192.3: Testing Exceptions

Programs throw errors when for instance wrong input is given. Because of this, one needs to make sure that an error is thrown when actual wrong input is given. Because of that we need to check for an exact exception, for this example we will use the following exception:

```
class WrongInputException(Exception):
    pass
```

This exception is raised when wrong input is given, in the following context where we always expect a number as text input.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input
```

To check whether an exception has been raised, we use `assertRaises` to check for that exception. `assertRaises` can be used in two ways:

1. Using the regular function call. The first argument takes the exception type, second a callable (usually a function) and the rest of arguments are passed to this callable.
2. Using a `with` clause, giving only the exception type to the function. This has as advantage that more code can be executed, but should be used with care since multiple functions can use the same exception which can be problematic. An example: with self.assertRaises(WrongInputException): convert2number("not a number")

This first has been implemented in the following test case:

```
import unittest

class ExceptionTestCase(unittest.TestCase):

    def test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "not a number")

    def test_correct_input(self):
        try:
            result = convert2number("56")
            self.assertIsInstance(result, int)
        except WrongInputException:
            self.fail()
```

There also may be a need to check for an exception which should not have been thrown. However, a test will automatically fail when an exception is thrown and thus may not be necessary at all. Just to show the options, the second test method shows a case on how one can check for an exception not to be thrown. Basically, this is catching the exception and then failing the test using the `fail` method.

Section 192.4: Choosing Assertions Within Unittests

While Python has an `assert` statement, the Python unit testing framework has better assertions specialized for tests: they are more informative on failures, and do not depend on the execution's debug mode.

Perhaps the simplest assertion is `assertTrue`, which can be used like this:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

This will run fine, but replacing the line above with

```
        self.assertTrue(1 + 1 == 3)
```

will fail.

The `assertTrue` assertion is quite likely the most general assertion, as anything tested can be cast as some boolean condition, but often there are better alternatives. When testing for equality, as above, it is better to write

```
        self.assertEqual(1 + 1, 3)
```

When the former fails, the message is

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

File "stuff.py", line 6, in test

self.assertTrue(1 + 1 == 3)

AssertionError: False is not true
```

but when the latter fails, the message is

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

File "stuff.py", line 6, in test

self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

which is more informative (it actually evaluated the result of the left hand side).

You can find the list of assertions in the standard documentation. In general, it is a good idea to choose the assertion that is the most specifically fitting the condition. Thus, as shown above, for asserting that `1 + 1 == 2` it is better to use `assertEqual` than `assertTrue`. Similarly, for asserting that `a is None`, it is better to use `assertIsNone` than `assertEqual`.

Note also that the assertions have negative forms. Thus `assertEqual` has its negative counterpart `assertNotEqual`, and `assertIsNone` has its negative counterpart `assertIsNotNone`. Once again, using the negative counterparts when appropriate, will lead to clearer error messages.

Section 192.5: Unit tests with pytest

installing pytest:

```
pip install pytest
```

getting the tests ready:

```
mkdir tests
touch tests/test_docker.py
```

Functions to test in `docker_something/helpers.py`:

```
from subprocess import Popen, PIPE
# this Popen is monkeypatched with the fixture `all_popens`

def copy_file_to_docker(src, dest):
    try:
        result = Popen(['docker','cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE,
stderr=PIPE)
        err = result.stderr.read()
        if err:
            raise Exception(err)
    except Exception as e:
        print(e)
    return result

def docker_exec_something(something_file_string):
    fl = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE, stdout=PIPE,
stderr=PIPE)
    fl.stdin.write(something_file_string)
    fl.stdin.close()
    err = fl.stderr.read()
    fl.stderr.close()
    if err:
        print(err)
        exit()
    result = fl.stdout.read()
    print(result)
```

The test imports `test_docker.py`:

```
import os
from tempfile import NamedTemporaryFile
import pytest
from subprocess import Popen, PIPE

from docker_something import helpers
copy_file_to_docker = helpers.copy_file_to_docker
docker_exec_something = helpers.docker_exec_something
```

mocking a file like object in `test_docker.py`:

```
class MockBytes():
    '''Used to collect bytes
    '''
    all_read = []
    all_write = []
    all_close = []

    def read(self, *args, **kwargs):
        # print('read', args, kwargs, dir(self))
        self.all_read.append((self, args, kwargs))

    def write(self, *args, **kwargs):
        # print('wrote', args, kwargs)
        self.all_write.append((self, args, kwargs))
```

```
    def close(self, *args, **kwargs):
        # print('closed', self, args, kwargs)
        self.all_close.append((self, args, kwargs))

    def get_all_mock_bytes(self):
        return self.all_read, self.all_write, self.all_close
```

Monkey patching with pytest in `test_docker.py`:

```
@pytest.fixture
def all_popens(monkeypatch):
    '''This fixture overrides / mocks the builtin Popen
        and replaces stdin, stdout, stderr with a MockBytes object

        note: monkeypatch is magically imported
    '''
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
            pass
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens
```

Example tests, must start with the prefix `test_` in the `test_docker.py` file:

```
def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popens):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']


def test_docker_exec_something(all_popens):

    docker_exec_something(something_file_string)

    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm',
'col_a', 'col_b', '/tmp/test.tsv']
    assert all([x in something_template_stdin for x in these])
```

running the tests one at a time:

```
py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

running all the tests in the `tests` folder:

```
py.test -k test_ tests
```

Section 192.6: Mocking functions with unittest.mock.create_autospec

One way to mock a function is to use the `create_autospec` function, which will mock out an object according to its specs. With functions, we can use this to ensure that they are called appropriately.

With a function `multiply` in `custom_math.py`:

```
def multiply(a, b):
    return a * b
```

And a function `multiples_of` in `process_math.py`:

```
from custom_math import multiply


def multiples_of(integer, *args, num_multiples=0, **kwargs):
    """
    :rtype: list
    """
    multiples = []

    for x in range(1, num_multiples + 1):
        """
        Passing in args and kwargs here will only raise TypeError if values were
        passed to multiples_of function, otherwise they are ignored. This way we can
        test that multiples_of is used correctly. This is here for an illustration
        of how create_autospec works. Not recommended for production code.
        """
        multiple = multiply(integer,x, *args, **kwargs)
        multiples.append(multiple)

    return multiples
```

We can test `multiples_of` alone by mocking out `multiply`. The below example uses the Python standard library unittest, but this can be used with other testing frameworks as well, like pytest or nose:

```
from unittest.mock import create_autospec
import unittest

# we import the entire module so we can mock out multiply
import custom_math
custom_math.multiply = create_autospec(custom_math.multiply)
from process_math import multiples_of


class TestCustomMath(unittest.TestCase):
    def test_multiples_of(self):
        multiples = multiples_of(3, num_multiples=1)
        custom_math.multiply.assert_called_with(3, 1)
```

```
    def test_multiples_of_with_bad_inputs(self):
        with self.assertRaises(TypeError) as e:
            multiples_of(1, "extra arg",  num_multiples=1) # this should raise a TypeError
```

Chapter 193: py.test

Section 193.1: Setting up py.test

`py.test` is one of several third party testing libraries that are available for Python. It can be installed using `pip` with

```
pip install pytest
```

The Code to Test

Say we are testing an addition function in `projectroot/module/code.py`:

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

The Testing Code

We create a test file in `projectroot/tests/test_code.py`. The file **must begin with `test_`** to be recognized as a testing file.

```
# projectroot/tests/test_code.py
from module import code


def test_add():
    assert code.add(1, 2) == 3
```

Running The Test

From `projectroot` we simply run `py.test`:

```
# ensure we have the modules
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
=========================================================== test session starts
============================================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py .

================================================== 1 passed in 0.01 seconds
================================================
```

Section 193.2: Intro to Test Fixtures

More complicated tests sometimes need to have things set up before you run the code you want to test. It is possible to do this in the test function itself, but then you end up with large test functions doing so much that it is difficult to tell where the setup stops and the test begins. You can also get a lot of duplicate setup code between your various test functions.

Our code file:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2
```

Our test file:

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

These are pretty simple examples, but if our `Stuff` object needed a lot more setup, it would get unwieldy. We see that there is some duplicated code between our test cases, so let's refactor that into a separate function first.

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def get_prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff


def test_foo_updates():
    my_stuff = get_prepped_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = get_prepped_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

This looks better but we still have the `my_stuff = get_prepped_stuff()` call cluttering up our test functions.

py.test fixtures to the rescue!

Fixtures are much more powerful and flexible versions of test setup functions. They can do a lot more than we're leveraging here, but we'll take it one step at a time.

First we change `get_prepped_stuff` to a fixture called `prepped_stuff`. You want to name your fixtures with nouns rather than verbs because of how the fixtures will end up being used in the test functions themselves later. The `@pytest.fixture` indicates that this specific function should be handled as a fixture rather than a regular function.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Now we should update the test functions so that they use the fixture. This is done by adding a parameter to their definition that exactly matches the fixture name. When py.test executes, it will run the fixture before running the test, then pass the return value of the fixture into the test function through that parameter. (Note that fixtures don't **need** to return a value; they can do other setup things instead, like calling an external resource, arranging things on the filesystem, putting values in a database, whatever the tests need for setup)

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Now you can see why we named it with a noun. but the `my_stuff = prepped_stuff` line is pretty much useless, so let's just use `prepped_stuff` directly instead.

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 30000
    assert prepped_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

Now we're using fixtures! We can go further by changing the scope of the fixture (so it only runs once per test module or test suite execution session instead of once per test function), building fixtures that use other fixtures, parametrizing the fixture (so that the fixture and all tests using that fixture are run multiple times, once for each parameter given to the fixture), fixtures that read values from the module that calls them... as mentioned earlier, fixtures have a lot more power and flexibility than a normal setup function.

Cleaning up after the tests are done.

Let's say our code has grown and our Stuff object now needs special clean up.

```
# projectroot/module/stuff.py
```

```
class Stuff(object):
def prep(self):
    self.foo = 1
    self.bar = 2

def finish(self):
    self.foo = 0
    self.bar = 0
```

We could add some code to call the clean up at the bottom of every test function, but fixtures provide a better way to do this. If you add a function to the fixture and register it as a **finalizer**, the code in the finalizer function will get called after the test using the fixture is done. If the scope of the fixture is larger than a single function (like module or session), the finalizer will be executed after all the tests in scope are completed, so after the module is done running or at the end of the entire test running session.

```
@pytest.fixture
def prepped_stuff(request):  # we need to pass in the request to use finalizers
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin():  # finalizer function
        # do all the cleanup here
        my_stuff.finish()
    request.addfinalizer(fin)  # register fin() as a finalizer
    # you can do more setup here if you really want to
    return my_stuff
```

Using the finalizer function inside a function can be a bit hard to understand at first glance, especially when you have more complicated fixtures. You can instead use a **yield fixture** to do the same thing with a more human readable execution flow. The only real difference is that instead of using `return` we use a `yield` at the part of the fixture where the setup is done and control should go to a test function, then add all the cleanup code after the `yield`. We also decorate it as a `yield_fixture` so that py.test knows how to handle it.

```
@pytest.yield_fixture
def prepped_stuff():  # it doesn't need request now!
    # do setup
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    # setup is done, pass control to the test functions
    yield my_stuff
    # do cleanup
    my_stuff.finish()
```

And that concludes the Intro to Test Fixtures!

For more information, see the official py.test fixture documentation and the official yield fixture documentation

Section 193.3: Failing Tests

A failing test will provide helpful output as to what went wrong:

```
# projectroot/tests/test_code.py
from module import code


def test_add__failing():
    assert code.add(10, 11) == 33
```

Results:

```
$ py.test
============================================== test session starts
==============================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py F

=================================================== FAILURES
===================================================
_____________________________________________ test_add__failing
_____________________________________________

    def test_add__failing():
>       assert code.add(10, 11) == 33
E       assert 21 == 33
E        +  where 21 = <function add at 0x105d4d6e0>(10, 11)
E        +    where <function add at 0x105d4d6e0> = code.add

tests/test_code.py:5: AssertionError
============================================ 1 failed in 0.01 seconds
============================================
```

Chapter 194: Profiling

Section 194.1: %%timeit and %timeit in IPython

Profiling string concatenation:

```
In [1]: import string

In [2]: %%timeit s=""; long_list=list(string.ascii_letters)*50
  ....: for substring in long_list:
  ....:   s+=substring
  ....:
1000 loops, best of 3: 570 us per loop

In [3]: %%timeit long_list=list(string.ascii_letters)*50
  ....: s="".join(long_list)
  ....:
100000 loops, best of 3: 16.1 us per loop
```

Profiling loops over iterables and lists:

```
In [4]: %timeit for i in range(100000):pass
100 loops, best of 3: 2.82 ms per loop

In [5]: %timeit for i in list(range(100000)):pass
100 loops, best of 3: 3.95 ms per loop
```

Section 194.2: Using cProfile (Preferred Profiler)

Python includes a profiler called cProfile. This is generally preferred over using timeit.

It breaks down your entire script and for each method in your script it tells you:

- `ncalls`: The number of times a method was called
- `tottime`: Total time spent in the given function (excluding time made in calls to sub-functions)
- `percall`: Time spent per call. Or the quotient of tottime divided by ncalls
- `cumtime`: The cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.
- `percall`: is the quotient of cumtime divided by primitive calls
- `filename:lineno(function)`: provides the respective data of each function

The cProfiler can be easily called on Command Line using:

```
$ python -m cProfile main.py
```

To sort the returned list of profiled methods by the time taken in the method:

```
$ python -m cProfile -s time main.py
```

Section 194.3: timeit() function

Profiling repetition of elements in an array

```
>>> import timeit
```

```
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

Section 194.4: timeit command line

Profiling concatenation of numbers

```
python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 3: 29.2 usec per loop

python -m timeit "'-'.join(map(str,range(100)))"
100000 loops, best of 3: 19.4 usec per loop
```

Section 194.5: line_profiler in command line

The source code with @profile directive before the function we want to profile:

```
import requests

@profile
def slow_func():
    s = requests.session()
    html=s.get("https://en.wikipedia.org/").text
    sum([pow(ord(x),3.1) for x in list(html)])

for i in range(50):
    slow_func()
```

Using kernprof command to calculate profiling line by line

```
$ kernprof -lv so6.py

Wrote profile results to so6.py.lprof
Timer unit: 4.27654e-07 s

Total time: 22.6427 s
File: so6.py
Function: slow_func at line 4

Line # Hits Time Per Hit % Time Line Contents
==============================================================
4 @profile
5 def slow_func():
6 50 20729 414.6 0.0 s = requests.session()
7 50 47618627 952372.5 89.9 html=s.get("https://en.wikipedia.org/").text
8 50 5306958 106139.2 10.0 sum([pow(ord(x),3.1) for x in list(html)])
```

Page request is almost always slower than any calculation based on the information on the page.

Chapter 195: Python speed of program

Section 195.1: Deque operations

A deque is a double-ended queue.

```
class Deque:
def __init__(self):
    self.items = []

def isEmpty(self):
    return self.items == []

def addFront(self, item):
    self.items.append(item)

def addRear(self, item):
    self.items.insert(0,item)

def removeFront(self):
    return self.items.pop()

def removeRear(self):
    return self.items.pop(0)

def size(self):
    return len(self.items)
```

Operations : Average Case (assumes parameters are randomly generated)

Append : O(1)

Appendleft : O(1)

Copy : O(n)

Extend : O(k)

Extendleft : O(k)

Pop : O(1)

Popleft : O(1)

Remove : O(n)

Rotate : O(k)

Section 195.2: Algorithmic Notations

There are certain principles that apply to optimization in any computer language, and Python is no exception. **Don't optimize as you go**: Write your program without regard to possible optimizations, concentrating instead on making sure that the code is clean, correct, and understandable. If it's too big or too slow when you've finished, then you can consider optimizing it.

Remember the 80/20 rule: In many fields you can get 80% of the result with 20% of the effort (also called the

90/10 rule - it depends on who you talk to). Whenever you're about to optimize code, use profiling to find out where that 80% of execution time is going, so you know where to concentrate your effort.

Always run "before" and "after" benchmarks: How else will you know that your optimizations actually made a difference? If your optimized code turns out to be only slightly faster or smaller than the original version, undo your changes and go back to the original, clear code.

Use the right algorithms and data structures: Don't use an O(n2) bubble sort algorithm to sort a thousand elements when there's an O(n log n) quicksort available. Similarly, don't store a thousand items in an array that requires an O(n) search when you could use an O(log n) binary tree, or an O(1) Python hash table.

For more visit the link below... Python Speed Up

The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1. Θ **Notation**: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression. 3n3 + 6n2 + 6000 = Θ(n3) Dropping lower order terms is always fine because there will always be a n0 after which Θ(n3) has higher values than Θn2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions. Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1*g(n)* <= *f(n)* <= *c2*g(n) for all n >= n0} The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.
2. **Big O Notation**: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time. If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm. O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 <= f(n) <= cg(n) for all n >= n0}

0. Ω **Notation**: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three. For a given function g(n), we denote by Ω(g(n)) the set of functions. Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= cg(n) <= f(n) for all n >= n0}. Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Section 195.3: Notation

Basic Idea

The notation used when describing the speed of your Python program is called Big-O notation. Let's say you have a function:

```
def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
          return True
    return False
```

This is a simple function to check if an item is in a list. To describe the complexity of this function, you will say O(n). This means "Order of n" as the O function is known as the Order function.

O(n) - generally n is the number of items in container

O(k) - generally k is the value of the parameter or the number of elements in the parameter

Section 195.4: List operations

Operations : Average Case (assumes parameters are randomly generated)

Append : O(1)

Copy : O(n)

Del slice : O(n)

Delete item : O(n)

Insert : O(n)

Get item : O(1)

Set item : O(1)

Iteration : O(n)

Get slice : O(k)

Set slice : O(n + k)

Extend : O(k)

Sort : O(n log n)

Multiply : O(nk)

x in s : O(n)

min(s), max(s) :O(n)

Get length : O(1)

Section 195.5: Set operations

Operation : Average Case (assumes parameters generated randomly) : Worst case

x in s : O(1)

Difference s - t : O(len(s))

Intersection s&t : O(min(len(s), len(t))) : O(len(s) * len(t)

Multiple intersection s1&s2&s3&...&sn : : (n-1) * O(l) where l is max(len(s1),...,len(sn))

s.difference_update(t) : O(len(t)) : O(len(t) * len(s))

s.symetric_difference_update(t) : O(len(t))

Symetric difference s^t : O(len(s)) : O(len(s) * len(t))

Union s|t : O(len(s) + len(t))

Chapter 196: Performance optimization

Section 196.1: Code profiling

First and foremost you should be able to find the bottleneck of your script and note that no optimization can compensate for a poor choice in data structure or a flaw in your algorithm design. Secondly do not try to optimize too early in your coding process at the expense of readability/design/quality. Donald Knuth made the following statement on optimization:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

To profile your code you have several tools: `cProfile` (or the slower `profile`) from the standard library, `line_profiler` and `timeit`. Each of them serve a different purpose.

`cProfile` is a deterministic profiler: function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (up to 0.001s). The library documentation ([https://docs.python.org/2/library/profile.html][1]) provides us with a simple use case

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

Or if you prefer to wrap parts of your existing code:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
# ... long ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

This will create outputs looking like the table below, where you can quickly see where your program spends most of its time and identify the functions to optimize.

```
 3 function calls in 0.000 seconds

Ordered by: standard name
ncalls tottime percall cumtime percall filename:lineno(function)
1 0.000 0.000 0.000 0.000 :1(f)
1 0.000 0.000 0.000 0.000 :1()
1 0.000 0.000 0.000 0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

The module `line_profiler` ([https://github.com/rkern/line_profiler][1]) is useful to have a line by line analysis of your code. This is obviously not manageable for long scripts but is aimed at snippets. See the documentation for more details. The easiest way to get started is to use the kernprof script as explained one the package page, note that you will need to specify manually the function(s) to profile.

```
$ kernprof -l script_to_profile.py
```

kernprof will create an instance of LineProfiler and insert it into the `__builtins__` namespace with the name profile. It has been written to be used as a decorator, so in your script, you decorate the functions you want to profile with `@profile`.

```
@profile
def slow_function(a, b, c):
    ...
```

The default behavior of kernprof is to put the results into a binary file `script_to_profile.py.lprof`. You can tell kernprof to immediately view the formatted results at the terminal with the [-v/--view] option. Otherwise, you can view the results later like so:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Finally `timeit` provides a simple way to test one liners or small expression both from the command line and the python shell. This module will answer question such as, is it faster to do a list comprehension or use the built-in `list()` when transforming a set into a list. Look for the `setup` keyword or `-s` option to add setup code.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

from a terminal

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Chapter 197: Security and Cryptography

Python, being one of the most popular languages in computer and network security, has great potential in security and cryptography. This topic deals with the cryptographic features and implementations in Python from its uses in computer and network security to hashing and encryption/decryption algorithms.

Section 197.1: Secure Password Hashing

The PBKDF2 algorithm exposed by `hashlib` module can be used to perform secure password hashing. While this algorithm cannot prevent brute-force attacks in order to recover the original password from the stored hash, it makes such attacks very expensive.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 can work with any digest algorithm, the above example uses SHA256 which is usually recommended. The random salt should be stored along with the hashed password, you will need it again in order to compare an entered password to the stored hash. It is essential that each password is hashed with a different salt. As to the number of rounds, it is recommended to set it as high as possible for your application.

If you want the result in hexadecimal, you can use the `binascii` module:

```
import binascii
hexhash = binascii.hexlify(hash)
```

Note: While PBKDF2 isn't bad, bcrypt and especially scrypt are considered stronger against brute-force attacks. Neither is part of the Python standard library at the moment.

Section 197.2: Calculating a Message Digest

The `hashlib` module allows creating message digest generators via the `new` method. These generators will turn an arbitrary string into a fixed-length digest:

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==> b'.\xdf\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9eF-='
```

Note that you can call `update` an arbitrary number of times before calling `digest` which is useful to hash a large file chunk by chunk. You can also get the digest in hexadecimal format by using `hexdigest`:

```
h.hexdigest()
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

Section 197.3: Available Hashing Algorithms

`hashlib.new` requires the name of an algorithm when you call it to produce a generator. To find out what algorithms are available in the current Python interpreter, use `hashlib.algorithms_available`:

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-SHA1',
'sha1', 'SHA384', 'md5', 'SHA1', 'MD5', 'MD4', 'SHA256', 'sha384', 'md4', 'ripemd160', 'sha224',
'sha512', 'DSA', 'dsaEncryption', 'sha', 'whirlpool'}
```

The returned list will vary according to platform and interpreter; make sure you check your algorithm is available.

There are also some algorithms that are *guaranteed* to be available on all platforms and interpreters, which are available using `hashlib.algorithms_guaranteed`:

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

Section 197.4: File Hashing

A hash is a function that converts a variable length sequence of bytes to a fixed length sequence. Hashing files can be advantageous for many reasons. Hashes can be used to check if two files are identical or verify that the contents of a file haven't been corrupted or changed.

You can use `hashlib` to generate a hash for a file:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

For larger files, a buffer of fixed length can be used:

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    buffer = f.read(SIZE)
    while len(buffer) > 0:
        hasher.update(buffer)
        buffer = f.read(SIZE)
print(hasher.hexdigest())
```

Section 197.5: Generating RSA signatures using pycrypto

RSA can be used to create a message signature. A valid signature can only be generated with access to the private RSA key, validating on the other hand is possible with merely the corresponding public key. So as long as the other side knows your public key they can verify the message to be signed by you and unchanged - an approach used for email for example. Currently, a third-party module like pycrypto is required for this functionality.

```
import errno

from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

```
message = b'This message is from me, I promise.'

try:
    with open('privkey.pem', 'r') as f:
        key = RSA.importKey(f.read())
except IOError as e:
    if e.errno != errno.ENOENT:
        raise
    # No private key, generate a new one. This can take a few seconds.
    key = RSA.generate(4096)
    with open('privkey.pem', 'wb') as f:
        f.write(key.exportKey('PEM'))
    with open('pubkey.pem', 'wb') as f:
        f.write(key.publickey().exportKey('PEM'))

hasher = SHA256.new(message)
signer = PKCS1_v1_5.new(key)
signature = signer.sign(hasher)
```

Verifying the signature works similarly but uses the public key rather than the private key:

```
with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
hasher = SHA256.new(message)
verifier = PKCS1_v1_5.new(key)
if verifier.verify(hasher, signature):
    print('Nice, the signature is valid!')
else:
    print('No, the message was signed with the wrong private key or modified')
```

Note: The above examples use PKCS#1 v1.5 signing algorithm which is very common. pycrypto also implements the newer PKCS#1 PSS algorithm, replacing `PKCS1_v1_5` by `PKCS1_PSS` in the examples should work if you want to use that one. Currently there seems to be little reason to use it however.

Section 197.6: Asymmetric RSA encryption using pycrypto

Asymmetric encryption has the advantage that a message can be encrypted without exchanging a secret key with the recipient of the message. The sender merely needs to know the recipients public key, this allows encrypting the message in such a way that only the designated recipient (who has the corresponding private key) can decrypt it. Currently, a third-party module like pycrypto is required for this functionality.

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'This is a very secret message.'

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
encrypted = cipher.encrypt(message)
```

The recipient can decrypt the message then if they have the right private key:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Note: The above examples use PKCS#1 OAEP encryption scheme. pycrypto also implements PKCS#1 v1.5 encryption scheme, this one is not recommended for new protocols however due to known caveats.

Section 197.7: Symmetric encryption using pycrypto

Python's built-in crypto functionality is currently limited to hashing. Encryption requires a third-party module like pycrypto. For example, it provides the AES algorithm which is considered state of the art for symmetric encryption. The following code will encrypt a given message using a passphrase:

```
import hashlib
import math
import os

from Crypto.Cipher import AES

IV_SIZE = 16    # 128 bit, fixed for the AES algorithm
KEY_SIZE = 32   # 256 bit meaning AES-256, can also be 128 or 192 bits
SALT_SIZE = 16  # This size is arbitrary

cleartext = b'Lorem ipsum'
password = b'highly secure encryption password'
salt = os.urandom(SALT_SIZE)
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)
```

The AES algorithm takes three parameters: encryption key, initialization vector (IV) and the actual message to be encrypted. If you have a randomly generated AES key then you can use that one directly and merely generate a random initialization vector. A passphrase doesn't have the right size however, nor would it be recommendable to use it directly given that it isn't truly random and thus has comparably little entropy. Instead, we use the built-in implementation of the PBKDF2 algorithm to generate a 128 bit initialization vector and 256 bit encryption key from the password.

Note the random salt which is important to have a different initialization vector and key for each message encrypted. This ensures in particular that two equal messages won't result in identical encrypted text, but it also prevents attackers from reusing work spent guessing one passphrase on messages encrypted with another passphrase. This salt has to be stored along with the encrypted message in order to derive the same initialization vector and key for decrypting.

The following code will decrypt our message again:

```
salt = encrypted[0:SALT_SIZE]
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]
cleartext = AES.new(key, AES.MODE_CFB, iv).decrypt(encrypted[SALT_SIZE:])
```

Chapter 198: Secure Shell Connection in Python

Parameter	Usage
hostname	This parameter tells the host to which the connection needs to be established
username	username required to access the host
port	host port
password	password for the account

Section 198.1: ssh connection

```
from paramiko import client
ssh = client.SSHClient() # create a new SSHClient object
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) #auto-accept unknown host keys
ssh.connect(hostname, username=username, port=port, password=password) #connect with a host
stdin, stdout, stderr = ssh.exec_command(command) # submit a command to ssh
print stdout.channel.recv_exit_status() #tells the status  1 - job failed
```

Chapter 199: Python Anti-Patterns

Section 199.1: Overzealous except clause

Exceptions are powerful, but a single overzealous except clause can take it all away in a single line.

try: res = get_result() res = res[0] log('got result: %r' % res) except: if not res: res = '' print('got exception')

This example demonstrates 3 symptoms of the antipattern:

1. The `except` with no exception type (line 5) will catch even healthy exceptions, including `KeyboardInterrupt`. That will prevent the program from exiting in some cases.
2. The except block does not reraise the error, meaning that we won't be able to tell if the exception came from within `get_result` or because `res` was an empty list.
3. Worst of all, if we were worried about result being empty, we've caused something much worse. If `get_result` fails, `res` will stay completely unset, and the reference to `res` in the except block, will raise `NameError`, completely masking the original error.

Always think about the type of exception you're trying to handle. Give the exceptions page a read and get a feel for what basic exceptions exist.

Here is a fixed version of the example above:

import traceback try: res = get_result() except Exception: log_exception(traceback.format_exc()) raise try: res = res[0] except IndexError: res = '' log('got result: %r' % res)

We catch more specific exceptions, reraising where necessary. A few more lines, but infinitely more correct.

Section 199.2: Looking before you leap with processor-intensive function

A program can easily waste time by calling a processor-intensive function multiple times.

For example, take a function which looks like this: it returns an integer if the input `value` can produce one, else `None`:

```
def intensive_f(value): # int -> Optional[int]
   # complex, and time-consuming code
   if process_has_failed:
       return None
   return integer_output
```

And it could be used in the following way:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Whilst this will work, it has the problem of calling `intensive_f`, which doubles the length of time for the code to run. A better solution would be to get the return value of the function beforehand.

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "could not be processed")
```

However, a clearer and possibly more pythonic way is to use exceptions, for example:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Here no temporary variable is needed. It may often be preferable to use a `assert` statement, and to catch the `AssertionError` instead.

Dictionary keys

A common example of where this may be found is accessing dictionary keys. For example compare:

```
bird_speeds = get_very_long_dictionary()

if "european swallow" in bird_speeds:
    speed = bird_speeds["european swallow"]
else:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

with:

```
bird_speeds = get_very_long_dictionary()

try:
    speed = bird_speeds["european swallow"]
except KeyError:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

The first example has to look through the dictionary twice, and as this is a long dictionary, it may take a long time to do so each time. The second only requires one search through the dictionary, and thus saves a lot of processor time.

An alternative to this is to use `dict.get(key, default)`, however many circumstances may require more complex operations to be done in the case that the key is not present

Chapter 200: Common Pitfalls

Python is a language meant to be clear and readable without any ambiguities and unexpected behaviors. Unfortunately, these goals are not achievable in all cases, and that is why Python does have a few corner cases where it might do something different than what you were expecting.

This section will show you some issues that you might encounter when writing Python code.

Section 200.1: List multiplication and common references

Consider the case of creating a nested list structure by multiplying:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

At first glance we would think we have a list of containing 3 different nested lists. Let's try to append `1` to the first one:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

`1` got appended to all of the lists in `li`.

The reason is that `[[]] * 3` doesn't create a `list` of 3 different `list`s. Rather, it creates a `list` holding 3 references to the same `list` object. As such, when we append to `li[0]` the change is visible in all sub-elements of `li`. This is equivalent of:

```
li = []
element = [[]]
li = element + element + element
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

This can be further corroborated if we print the memory addresses of the contained `list` by using `id`:

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

The solution is to create the inner lists with a loop:

```
li = [[] for _ in range(3)]
```

Instead of creating a single `list` and then making 3 references to it, we now create 3 different distinct lists. This, again, can be verified by using the `id` function:

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

You can also do this. It causes a new empty list to be created in each `append` call.

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
```

Don't use index to loop over a sequence.

Don't:

```
for i in range(len(tab)):
    print(tab[i])
```

Do:

```
for elem in tab:
    print(elem)
```

`for` will automate most iteration operations for you.

Use enumerate if you really need both the index and the element.

```
for i, elem in enumerate(tab):
     print((i, elem))
```

Be careful when using "==" to check against True or False

```
if (var == True):
    # this will execute if var is True or 1, 1.0, 1L

if (var != True):
    # this will execute if var is neither True nor 1

if (var == False):
    # this will execute if var is False or 0 (or 0.0, 0L, 0j)

if (var == None):
    # only execute if var is None

if var:
    # execute if var is a non-empty string/list/dictionary/tuple, non-0, etc

if not var:
    # execute if var is "", {}, [], (), 0, None, etc.

if var is True:
    # only execute if var is boolean True, not 1

if var is False:
    # only execute if var is boolean False, not 0

if var is None:
```

```
    # same as var == None
```

Do not check if you can, just do it and handle the error

Pythonistas usually say "It's easier to ask for forgiveness than permission".

Don't:

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something
```

Do:

```
try:
    file = open(file_path)
except OSError as e:
    # do something
```

Or even better with `Python 2.6+`:

```
with open(file_path) as file:
```

It is much better because it is much more generic. You can apply `try/except` to almost anything. You don't need to care about what to do to prevent it, just care about the error you are risking.

Do not check against type

Python is dynamically typed, therefore checking for type makes you lose flexibility. Instead, use duck typing by checking behavior. If you expect a string in a function, then use `str()` to convert any object to a string. If you expect a list, use `list()` to convert any iterable to a list.

Don't:

```
def foo(name):
    if isinstance(name, str):
        print(name.lower())

def bar(listing):
    if isinstance(listing, list):
        listing.extend((1, 2, 3))
        return ", ".join(listing)
```

Do:

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

Using the last way, `foo` will accept any object. `bar` will accept strings, tuples, sets, lists and much more. Cheap DRY.

Don't mix spaces and tabs

Use *object* as first parent

This is tricky, but it will bite you as your program grows. There are old and new classes in `Python 2.x`. The old ones are, well, old. They lack some features, and can have awkward behavior with inheritance. To be usable, any of your class must be of the "new style". To do so, make it inherit from `object`.

Don't:

```
class Father:
    pass

class Child(Father):
    pass
```

Do:

```
class Father(object):
    pass


class Child(Father):
    pass
```

In `Python 3.x` all classes are new style so you don't need to do that.

Don't initialize class attributes outside the init method

People coming from other languages find it tempting because that is what you do in Java or PHP. You write the class name, then list your attributes and give them a default value. It seems to work in Python, however, this doesn't work the way you think. Doing that will setup class attributes (static attributes), then when you will try to get the object attribute, it will gives you its value unless it's empty. In that case it will return the class attributes. It implies two big hazards:

- If the class attribute is changed, then the initial value is changed.
- If you set a mutable object as a default value, you'll get the same object shared across instances.

Don't (unless you want static):

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Do:

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Section 200.2: Mutable default argument

```
def foo(li=[]):
    li.append(1)
    print(li)
```

```
foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

This code behaves as expected, but what if we don't pass an argument?

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1]  Not as expected...
```

This is because default arguments of functions and methods are evaluated at **definition** time rather than run time. So we only ever have a single instance of the `li` list.

The way to get around it is to use only immutable types for default arguments:

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]

foo()
# Out: [1]
```

While an improvement and although `if not li` correctly evaluates to `False`, many other objects do as well, such as zero-length sequences. The following example arguments can cause unintended results:

```
x = []
foo(li=x)
# Out: [1]

foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

The idiomatic approach is to directly check the argument against the `None` object:

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

Section 200.3: Changing the sequence you are iterating over

A `for` loop iterates over a sequence, so **altering this sequence inside the loop could lead to unexpected results** (especially when adding or removing elements):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

Note: `list.pop()` is being used to remove elements from the list.

The second element was not deleted because the iteration goes through the indices in order. The above loop iterates twice, with the following results:

```
# Iteration #1
index = 0
alist = [0, 1, 2]
alist.pop(0) # removes '0'

# Iteration #2
index = 1
alist = [1, 2]
alist.pop(1) # removes '2'

# loop terminates, but alist is not empty:
alist = [1]
```

This problem arises because the indices are changing while iterating in the direction of increasing index. To avoid this problem, you can **iterate through the loop backwards**:

```
alist = [1,2,3,4,5,6,7]
for index, item in reversed(list(enumerate(alist))):
    # delete all even items
    if item % 2 == 0:
        alist.pop(index)
print(alist)
# Out: [1, 3, 5, 7]
```

By iterating through the loop starting at the end, as items are removed (or added), it does not affect the indices of items earlier in the list. So this example will properly remove all items that are even from `alist`.

A similar problem arises when **inserting or appending elements to a list that you are iterating over**, which can result in an infinite loop:

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    # break to avoid infinite loop:
    if index == 20:
        break
    alist.insert(index, 'a')
print(alist)
# Out (abbreviated): ['a', 'a', ..., 'a', 'a',  0,   1,   2]
```

Without the `break` condition the loop would insert `'a'` as long as the computer does not run out of memory and the program is allowed to continue. In a situation like this, it is usually preferred to create a new list, and add items to the new list as you loop through the original list.

When using a `for` loop, **you cannot modify the list elements with the placeholder variable**:

```
alist = [1,2,3,4]
```

```
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

In the above example, **changing `item` doesn't actually change anything in the original list**. You need to use the list index (`alist[2]`), and `enumerate()` works well for this:

```
alist = [1,2,3,4]
for index, item in enumerate(alist):
    if item % 2 == 0:
        alist[index] = 'even'
print(alist)
# Out: [1, 'even', 3, 'even']
```

A `while` **loop** might be a better choice in some cases:

If you are going to **delete all the items** in the list:

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

Although simply resetting `zlist` will accomplish the same result;

```
zlist = []
```

The above example can also be combined with `len()` to stop after a certain point, or to delete all but `x` items in the list:

```
zlist = [0, 1, 2]
x = 1
while len(zlist) > x:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
# After: zlist = [2]
```

Or to **loop through a list while deleting elements that meet a certain condition** (in this case deleting all even elements):

```
zlist = [1,2,3,4,5]
i = 0
while i < len(zlist):
    if zlist[i] % 2 == 0:
        zlist.pop(i)
    else:
```

```
        i += 1
print(zlist)
# Out: [1, 3, 5]
```

Notice that you don't increment `i` after deleting an element. By deleting the element at `zlist[i]`, the index of the next item has decreased by one, so by checking `zlist[i]` with the same value for `i` on the next iteration, you will be correctly checking the next item in the list.

A contrary way to think about removing unwanted items from a list, is to **add wanted items to a new list**. The following example is an alternative to the latter `while` loop example:

```
zlist = [1,2,3,4,5]

z_temp = []
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Out: [1, 3, 5]
```

Here we are funneling desired results into a new list. We can then optionally reassign the temporary list to the original variable.

With this trend of thinking, you can invoke one of Python's most elegant and powerful features, **list comprehensions**, which eliminates temporary lists and diverges from the previously discussed in-place list/index mutation ideology.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

Section 200.4: Integer and String identity

Python uses internal caching for a range of integers to reduce unnecessary overhead from their repeated creation.

In effect, this can lead to confusing behavior when comparing integer identities:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

and, using another example:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Wait what?

We can see that the identity operation `is` yields `True` for some integers (`-3`, `256`) but no for others (`-8`, `257`).

To be more specific, integers in the range `[-5, 256]` are internally cached during interpreter startup and are only created once. As such, they are **identical** and comparing their identities with `is` yields `True`; integers outside this

range are (usually) created on-the-fly and their identities compare to `False`.

This is a common pitfall since this is a common range for testing, but often enough, the code fails in the later staging process (or worse - production) with no apparent reason after working perfectly in development.

The solution is to **always compare values using the equality** (`==`) operator and **not** the identity (`is`) operator.

Python also keeps references to commonly used strings and can result in similarly confusing behavior when comparing identities (i.e. using `is`) of strings.

```
>>> 'python' is 'py' + 'thon'
True
```

The string `'python'` is commonly used, so Python has one object that all references to the string `'python'` use.

For uncommon strings, comparing identity fails even when the strings are equal.

```
>>> 'this is not a common string' is 'this is not' + ' a common string'
False
>>> 'this is not a common string' == 'this is not' + ' a common string'
True
```

So, just like the rule for Integers, **always compare string values using the equality** (`==`) operator and **not** the identity (`is`) operator.

Section 200.5: Dictionaries are unordered

You might expect a Python dictionary to be sorted by keys like, for example, a C++ `std::map`, but this is not the case:

```
myDict = {'first': 1, 'second': 2, 'third': 3}
print(myDict)
# Out: {'first': 1, 'second': 2, 'third': 3}

print([k for k in myDict])
# Out: ['second', 'third', 'first']
```

Python doesn't have any built-in class that automatically sorts its elements by key.

However, if sorting is not a must, and you just want your dictionary to remember the order of insertion of its key/value pairs, you can use `collections.OrderedDict`:

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Out: ['first', 'second', 'third']
```

Keep in mind that initializing an `OrderedDict` with a standard dictionary won't sort in any way the dictionary for you. All that this structure does is to *preserve* the order of key insertion.

The implementation of dictionaries was changed in Python 3.6 to improve their memory consumption. A side effect of this new implementation is that it also preserves the order of keyword arguments passed to a function:

Python 3.x `Version ≥ 3.6`

```
def func(**kw): print(kw.keys())

func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # expected order
```

> **Caveat**: beware that *"the order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon"*, as it may change in the future.

Section 200.6: Variable leaking in list comprehensions and for loops

Consider the following list comprehension

Python 2.x `Version ≤ 2.7`

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

This occurs only in Python 2 due to the fact that the list comprehension "leaks" the loop control variable into the surrounding scope (source). This behavior can lead to hard-to-find bugs and ***it has been fixed in Python 3***.

Python 3.x `Version ≥ 3.0`

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

Similarly, for loops have no private scope for their iteration variable

```
i = 0
for i in range(3):
    pass
print(i) # Outputs 2
```

This type of behavior occurs both in Python 2 and Python 3.

To avoid issues with leaking variables, use new variables in list comprehensions and for loops as appropriate.

Section 200.7: Chaining of or operator

When testing for any of several equality comparisons:

```
if a == 3 or b == 3 or c == 3:
```

it is tempting to abbreviate this to

```
if a or b or c == 3: # Wrong
```

This is wrong; the `or` operator has lower precedence than `==`, so the expression will be evaluated as `if (a) or (b) or (c == 3):`. The correct way is explicitly checking all the conditions:

```
if a == 3 or b == 3 or c == 3:  # Right Way
```

Alternately, the built-in `any()` function may be used in place of chained `or` operators:

```
if any([a == 3, b == 3, c == 3]): # Right
```

Or, to make it more efficient:

```
if any(x == 3 for x in (a, b, c)): # Right
```

Or, to make it shorter:

```
if 3 in (a, b, c): # Right
```

Here, we use the `in` operator to test if the value is present in a tuple containing the values we want to compare against.

Similarly, it is incorrect to write

```
if a == 1 or 2 or 3:
```

which should be written as

```
if a in (1, 2, 3):
```

Section 200.8: sys.argv[0] is the name of the file being executed

The first element of `sys.argv[0]` is the name of the python file being executed. The remaining elements are the script arguments.

```
# script.py
import sys

print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']

$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']

$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

Section 200.9: Accessing int literals' attributes

You might have heard that everything in Python is an object, even literals. This means, for example, `7` is an object as well, which means it has attributes. For example, one of these attributes is the `bit_length`. It returns the amount of bits needed to represent the value it is called upon.

```
x = 7
x.bit_length()
# Out: 3
```

Seeing the above code works, you might intuitively think that `7.bit_length()` would work as well, only to find out it raises a `SyntaxError`. Why? because the interpreter needs to differentiate between an attribute access and a floating number (for example `7.2` or `7.bit_length()`). It can't, and that's why an exception is raised.

There are a few ways to access an `int` literals' attributes:

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

Using two dots (like this `7..bit_length()`) doesn't work in this case, because that creates a `float` literal and floats don't have the `bit_length()` method.

This problem doesn't exist when accessing `float` literals' attributes since the interpreter is "smart" enough to know that a `float` literal can't contain two `.`, for example:

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

Section 200.10: Global Interpreter Lock (GIL) and blocking threads

Plenty has been written about Python's GIL. It can sometimes cause confusion when dealing with multi-threaded (not to be confused with multiprocess) applications.

Here's an example:

```
import math
from threading import Thread

def calc_fact(num):
    math.factorial(num)

num = 600000
t = Thread(target=calc_fact, daemon=True, args=[num])
print("About to calculate: {}!".format(num))
t.start()
print("Calculating...")
t.join()
print("Calculated")
```

You would expect to see `Calculating...` printed out immediately after the thread is started, we wanted the calculation to happen in a new thread after all! But in actuality, you see it get printed after the calculation is complete. That is because the new thread relies on a C function (`math.factorial`) which will lock the GIL while it executes.

There are a couple ways around this. The first is to implement your factorial function in native Python. This will allow the main thread to grab control while you are inside your loop. The downside is that this solution will be **a lot** slower, since we're not using the C function anymore.

```
def calc_fact(num):
    """ A slow version of factorial in native Python """
    res = 1
    while num >= 1:
        res = res * num
```

```
        num -= 1
    return res
```

You can also `sleep` for a period of time before starting your execution. Note: this won't actually allow your program to interrupt the computation happening inside the C function, but it will allow your main thread to continue after the spawn, which is what you may expect.

```
def calc_fact(num):
    sleep(0.001)
    math.factorial(num)
```

Section 200.11: Multiple return

Function xyz returns two values a and b:

```
def xyz():
  return a, b
```

Code calling xyz stores result into one variable assuming xyz returns only one value:

```
t = xyz()
```

Value of `t` is actually a tuple (a, b) so any action on `t` assuming it is not a tuple may fail **deep** in the code with a an unexpected **error** about tuples.

> TypeError: type tuple doesn't define ... method

The fix would be to do:

```
a, b = xyz()
```

Beginners will have trouble finding the reason of this message by only reading the tuple error message !

Section 200.12: Pythonic JSON keys

```
my_var = 'bla';
api_key = 'key';
...lots of code here...
params = {"language": "en", my_var: api_key}
```

If you are used to JavaScript, variable evaluation in Python dictionaries won't be what you expect it to be. This statement in JavaScript would result in the `params` object as follows:

```
{
    "language": "en",
    "my_var": "key"
}
```

In Python, however, it would result in the following dictionary:

```
{
    "language": "en",
    "bla": "key"
```

```
}
```

`my_var` is evaluated and its value is used as the key.

Chapter 201: Hidden Features

Section 201.1: Operator Overloading

Everything in Python is an object. Each object has some special internal methods which it uses to interact with other objects. Generally, these methods follow the `__action__` naming convention. Collectively, this is termed as the Python Data Model.

You can overload *any* of these methods. This is commonly used in operator overloading in Python. Below is an example of operator overloading using Python's data model. The `Vector` class creates a simple vector of two variables. We'll add appropriate support for mathematical operations of two vectors using operator overloading.

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        # Addition with another vector.
        return Vector(self.x + v.x, self.y + v.y)

    def __sub__(self, v):
        # Subtraction with another vector.
        return Vector(self.x - v.x, self.y - v.y)

    def __mul__(self, s):
        # Multiplication with a scalar.
        return Vector(self.x * s, self.y * s)

    def __div__(self, s):
        # Division with a scalar.
        float_s = float(s)
        return Vector(self.x / float_s, self.y / float_s)

    def __floordiv__(self, s):
        # Division with a scalar (value floored).
        return Vector(self.x // s, self.y // s)

    def __repr__(self):
        # Print friendly representation of Vector class. Else, it would
        # show up like, <__main__.Vector instance at 0x01DDDDC8>.
        return '<Vector (%f, %f)>' % (self.x, self.y, )

a = Vector(3, 5)
b = Vector(2, 7)

print a + b # Output: <Vector (5.000000, 12.000000)>
print b - a # Output: <Vector (-1.000000, 2.000000)>
print b * 1.3 # Output: <Vector (2.600000, 9.100000)>
print a // 17 # Output: <Vector (0.000000, 0.000000)>
print a / 17 # Output: <Vector (0.176471, 0.294118)>
```

The above example demonstrates overloading of basic numeric operators. A comprehensive list can be found here.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Çağatay Uslu	Chapter 20
2Cubed	Chapters 39, 122 and 147
4444	Chapter 21
A. Ciclet	Chapter 196
A. Raza	Chapter 1
Aaron Christiansen	Chapters 40 and 109
Aaron Critchley	Chapter 1
Aaron Hall	Chapter 38
Abhishek Kumar	Chapter 149
abukaj	Chapter 200
acdr	Chapter 21
Adam Brenecki	Chapter 85
Adam Matan	Chapters 84 and 104
Adam_92	Chapter 40
adeora	Chapter 197
Aditya	Chapters 40, 53, 134, 195 and 200
Adrian Antunez	Chapter 88
Adriano	Chapter 33
afeique	Chapter 1
Aidan	Chapter 75
Ajean	Chapter 5
Akshat Mahajan	Chapters 33, 67, 120 and 146
aldanor	Chapter 40
Aldo	Chapter 103
Alec	Chapter 200
alecxe	Chapters 5, 40, 87, 92 and 93
alejosocorro	Chapters 1 and 75
Alex Gaynor	Chapter 55
Alex L	Chapter 16
Alex Logan	Chapter 1
AlexV	Chapter 33
Alfe	Chapter 88
alfonso.kim	Chapter 16
ALinuxLover	Chapter 1
Alireza Savand	Chapter 192
Alleo	Chapters 21 and 149
Alon Alexander	Chapter 116
amblina	Chapters 83, 85 and 129
Ami Tavory	Chapter 192
amin	Chapter 9
Amir Rachum	Chapters 19 and 39
Amitay Stern	Chapters 41 and 91
Anaphory	Chapter 159
anatoly techtonik	Chapter 160
Andrea	Chapter 1
andrew	Chapter 131
Andrew Schade	Chapter 84

Andrii Abramov	Chapter 1
Andrzej Pronobis	Chapters 8, 38 and 55
Andy	Chapters 1, 5, 7, 17, 33, 51, 82 and 88
Andy Hayden	Chapters 8, 33, 67, 73, 75, 77, 98 and 149
angussidney	Chapter 1
Ani Menon	Chapters 1, 4, 40, 149 and 154
Annonymous	Chapters 78, 87 and 199
Anthony Pham	Chapters 13, 15, 16, 19, 20, 30, 33, 43, 45, 55, 60, 70 and 179
Antoine Bolvy	Chapters 1 and 149
Antoine Pinsard	Chapter 39
Antti Haapala	Chapters 5, 16, 72, 87 and 106
Antwan	Chapter 149
APerson	Chapters 21, 69 and 72
Aquib Javed Khan	Chapter 1
Ares	Chapters 1, 15, 20 and 41
Arkady	Chapter 33
Arpit Solanki	Chapters 1, 82 and 125
Artem Kolontay	Chapter 173
ArtOfCode	Chapters 67, 173 and 197
Arun	Chapters 65 and 108
Aryaman Arora	Chapter 179
ashes999	Chapter 75
asmeurer	Chapters 45 and 47
atayenel	Chapter 124
Athari	Chapter 151
Avantol13	Chapter 38
avb	Chapter 30
Axe	Chapters 21 and 149
B8vrede	Chapters 1, 40, 75 and 103
Baaing Cow	Chapters 1 and 200
Bahrom	Chapter 8
Bakuriu	Chapter 149
balki	Chapter 53
Barry	Chapter 20
Bastian	Chapter 86
bbayles	Chapter 128
Beall619	Chapters 74 and 101
bee	Chapters 161 and 164
Benedict Bunting	Chapter 99
Bharel	Chapters 30 and 149
Bhargav	Chapter 40
Bhargav Rao	Chapters 41, 149 and 200
bignose	Chapter 149
Billy	Chapter 200
Biswa_9937	Chapters 178, 182 and 183
bitchaser	Chapter 149
bixel	Chapter 200
blubberdiblub	Chapter 177
blueberryfields	Chapter 181
blueenvelope	Chapters 9 and 20
Bluethon	Chapter 149
boboquack	Chapters 10, 11 and 18
bogdanciobanu	Chapter 111

Bonifacio2	Chapter 64
BoppreH	Chapter 19
Bosoneando	Chapter 46
Božo Stojković	Chapters 1, 14, 21, 33 and 149
bpachev	Chapter 53
Brendan Abel	Chapter 84
brennan	Chapter 173
Brett Cannon	Chapters 11 and 43
Brian C	Chapter 1
Brien	Chapter 41
Bryan P	Chapter 1
BSL	Chapters 1 and 197
Burhan Khalid	Chapter 19
BusyAnt	Chapters 1, 20, 43, 60, 78 and 88
Buzz	Chapter 18
Cache Staheli	Chapter 41
CamelBackNotation	Chapter 33
Cameron Gagnon	Chapter 149
Camsbury	Chapters 9 and 33
caped114	Chapter 41
carrdelling	Chapter 77
Cbeb24404	Chapter 1
ceruleus	Chapter 1
cfi	Chapters 21, 26 and 69
Chandan Purohit	Chapters 20 and 33
ChaoticTwist	Chapters 21, 33, 41 and 107
Charles	Chapters 10, 21, 30, 41, 149 and 200
Charul	Chapter 167
Chinmay Hegde	Chapters 50, 94, 132, 164 and 171
Chong Tang	Chapter 21
Chris Hunt	Chapter 16
Chris Larson	Chapter 33
Chris Midgley	Chapter 1
Chris Mueller	Chapter 19
Christian Ternus	Chapters 1, 16, 43, 51 and 78
Christofer Ohlsson	Chapter 45
Christophe Roussy	Chapter 200
Chromium	Chapter 134
Cilyan	Chapter 170
Cimbali	Chapter 8
cizixs	Chapters 19, 20 and 128
cjds	Chapters 110 and 134
Clíodhna	Chapter 1
Claudiu	Chapters 1, 31, 67, 75, 80, 83, 88, 111, 118, 153, 192 and 193
Clayton Wahlstrom	Chapter 149
cledoux	Chapter 108
CodenameLambda	Chapters 1 and 67
Cody Piersall	Chapter 8
Colin Yang	Chapter 149
Comrade SparklePony	Chapters 180 and 181
Conrad.Dean	Chapters 1, 5, 21, 38 and 43
crhodes	Chapter 30
c□ι□s□□□□	Chapters 1, 149, 161 and 191

Dair	Chapters 11, 120 and 173
Daksh Gupta	Chapters 1 and 38
Dania	Chapter 1
danidee	Chapter 33
Daniel	Chapter 43
Daniil Ryzhkov	Chapter 173
Darkade	Chapter 173
Darth Kotik	Chapter 16
Darth Shadow	Chapters 1, 40, 75, 97, 149 and 173
Dartmouth	Chapters 1, 62, 149 and 150
Dave J	Chapters 40 and 149
David	Chapters 9, 33, 124 and 161
David Cullen	Chapters 30 and 121
David Heyman	Chapters 41 and 149
davidism	Chapter 13
DawnPaladin	Chapter 33
Dee	Chapter 186
deeenes	Chapters 1, 20 and 67
deepakkt	Chapter 14
DeepSpace	Chapters 9, 16, 77, 100, 149 and 200
Delgan	Chapters 1, 16, 20 and 40
denvaar	Chapter 167
depperm	Chapters 1, 3, 38, 41 and 99
DevD	Chapter 1
Devesh Saini	Chapter 115
DhiaTN	Chapter 16
dhimanta	Chapters 184 and 185
Dilettant	Chapter 200
Dima Tisnek	Chapter 21
djaszczurowski	Chapter 167
Doc	Chapter 143
dodell	Chapter 1
Doraemon	Chapter 29
Doug Henderson	Chapter 41
Douglas Starnes	Chapter 1
Dov	Chapter 30
dreftymac	Chapter 40
driax	Chapters 39, 75 and 88
Duh	Chapter 149
Dunatotatos	Chapter 171
dwanderson	Chapter 149
eandersson	Chapter 127
edwinksl	Chapter 173
eenblam	Chapter 21
Elazar	Chapters 1, 7, 8, 13, 15, 16, 20, 21, 28, 38, 41, 67, 87, 111, 144 and 149
Eleftheria	Chapter 113
elegent	Chapter 33
Ellis	Chapters 20 and 45
Elodin	Chapters 33 and 195
Emma	Chapters 20 and 21
Enamul Hassan	Chapter 16
engineercoding	Chapter 192
Enrico Maria De Angelis	Chapter 1

enrico.bacis	Chapters 21, 56 and 149
erewok	Chapter 149
Eric	Chapter 107
Eric Finn	Chapter 16
Eric Zhang	Chapter 110
Erica	Chapter 1
ericdwang	Chapter 149
ericmarkmartin	Chapters 67, 98, 110 and 149
Erik Godard	Chapter 1
EsmaeelE	Chapter 1
Esteis	Chapters 13 and 21
ettanany	Chapters 27 and 149
Everyone_Else	Chapter 149
evuez	Chapters 7, 8, 14, 20, 40, 113 and 149
exhuma	Chapter 20
Fábio Perez	Chapter 31
Faiz Halde	Chapters 21, 114 and 119
FazeL	Chapters 111 and 148
Felix D.	Chapter 16
Felk	Chapter 21
Fermi paradox	Chapter 21
Fernando	Chapter 173
Ffisegydd	Chapters 14, 38, 98, 108 and 193
Filip Haglund	Chapter 1
Firix	Chapters 1 and 150
flamenco	Chapter 56
Flickerlight	Chapter 20
Florian Bender	Chapter 21
FMc	Chapter 43
Francisco Guimaraes	Chapter 94
Franck Dernoncourt	Chapter 1
FrankBr	Chapter 36
frankyjuang	Chapter 181
Fred Barclay	Chapters 1 and 157
Freddy	Chapter 1
fredley	Chapter 45
freidrichen	Chapter 21
Frustrated	Chapter 74
Gal Dreiman	Chapters 16, 20, 21, 33, 40, 136 and 137
ganesh gadila	Chapters 20 and 41
Ganesh K	Chapter 148
Gareth Latty	Chapter 19
garg10may	Chapters 21 and 149
Gavin	Chapter 149
Geeklhem	Chapters 14, 110 and 124
Generic Snake	Chapter 16
geoffspear	Chapter 149
Gerard Roche	Chapter 1
gerrit	Chapter 40
ghostarbeiter	Chapters 16, 21, 33, 41, 45, 88, 132 and 149
Giannis Spiliopoulos	Chapter 40
GiantsLoveDeathMetal	Chapters 40 and 164
girish946	Chapters 31 and 154

giucal	Chapter 55
GoatsWearHats	Chapters 1, 16, 29, 41 and 72
goodmami	Chapter 75
Greg	Chapter 22
greut	Chapter 37
Guy	Chapters 16, 19 and 156
H. Pauwelyn	Chapter 1
hackvan	Chapter 164
Hannele	Chapter 21
Hannes Karppila	Chapters 14 and 134
Harrison	Chapter 40
hashcode55	Chapter 76
ha_1694	Chapter 173
heyhey2k	Chapter 94
hiro protagonist	Chapters 54 and 200
HoverHell	Chapter 12
Hriddhi Dey	Chapter 105
Hurkyl	Chapters 21 and 33
hxysayhi	Chapters 66 and 168
Ian	Chapter 1
IanAuld	Chapters 1 and 21
iankit	Chapters 21 and 37
iBelieve	Chapter 19
idjaw	Chapter 41
ifma	Chapter 108
Igor Raush	Chapters 1, 19, 20, 38, 39, 45, 67 and 98
Ilia Barahovski	Chapters 32, 41 and 88
ilse2005	Chapter 30
Inbar Rose	Chapter 16
Indradhanush Gupta	Chapter 49
Infinity	Chapters 19 and 115
InitializeSahib	Chapters 38, 39 and 82
intboolstring	Chapters 10, 21 and 45
iScrE4m	Chapter 149
J_F	Chapters 1, 9, 15, 20, 29, 33, 38, 88, 97, 111, 119, 125 and 149
Jörn Hees	Chapter 137
J0HN	Chapters 21 and 67
j3485	Chapter 20
jackskis	Chapters 53 and 67
Jacques de Hooge	Chapters 97 and 151
JakeD	Chapter 22
James	Chapters 8, 14, 19, 20, 33 and 100
James Elderfield	Chapters 20, 40, 45 and 149
James Taylor	Chapter 1
JamesS	Chapter 21
Jan	Chapter 75
jani	Chapter 20
japborst	Chapter 86
Jean	Chapters 1 and 40
jedwards	Chapters 1, 102 and 149
Jeff Hutchins	Chapter 110
Jeffrey Lin	Chapters 1, 16, 75, 132 and 200
JelmerS	Chapter 102

JGreenwell	Chapters 3, 9, 33 and 37
JHS	Chapter 21
Jim Fasarakis Hilliard	Chapters 1, 16, 33, 41, 55, 87, 110, 149 and 200
jim opleydulven	Chapter 1
Jimmy Song	Chapter 149
jimsug	Chapters 1 and 20
jkdev	Chapters 20 and 38
jkitchen	Chapter 33
JL Peyret	Chapters 40 and 51
jlarsch	Chapter 38
jmunsch	Chapters 1, 47, 92, 125, 181 and 192
JNat	Chapters 10, 20 and 29
joel3000	Chapters 21, 103 and 192
Johan Lundberg	Chapter 1
John Slegers	Chapters 1 and 149
John Zwinck	Chapter 11
Jonatan	Chapters 40, 64 and 87
jonrsharpe	Chapters 1, 75, 78 and 98
Joseph True	Chapter 1
Josh	Chapter 149
Jossie Calderon	Chapter 86
jrast	Chapter 16
JRodDynamite	Chapters 1, 21 and 40
jtbandes	Chapter 70
Juan T	Chapters 1, 67, 90 and 149
JuanPablo	Chapter 110
Julien Spronck	Chapters 53 and 75
Julij Jegorov	Chapter 188
Justin	Chapters 30, 33, 40, 74 and 149
Justin Chadwell	Chapter 125
Justin M. Ucar	Chapter 149
j__	Chapter 41
Kabie	Chapter 149
Kallz	Chapter 38
Kamran Mackey	Chapter 1
Karl Knechtel	Chapters 67, 69 and 149
KartikKannapur	Chapters 20 and 38
kdopen	Chapters 19, 21 and 110
keiv.fly	Chapter 194
Kevin Brown	Chapters 1, 5, 14, 30, 53, 75, 146, 149 and 192
KeyWeeUsr	Chapters 80 and 153
KIDJourney	Chapter 21
Kinifwyne	Chapters 43 and 193
Kiran Vemuri	Chapter 1
kisanme	Chapter 1
knight	Chapter 40
kollery	Chapter 156
kon psych	Chapter 47
krato	Chapter 83
Kunal Marwaha	Chapter 149
Kwarrtz	Chapter 21
L3viathan	Chapters 33 and 98
Lafexlos	Chapters 1, 9 and 20

LDP	Chapter 20
Lee Netherton	Chapters 21, 33 and 114
Leo	Chapters 49 and 97
Leo Thumma	Chapter 20
Leon	Chapter 1
Leon Z.	Chapter 74
Liteye	Chapters 21 and 38
loading...	Chapters 83 and 114
Locane	Chapter 21
lorenzofeliz	Chapter 2
LostAvatar	Chapters 1 and 161
Luca Van Oort	Chapter 165
Luke Taylor	Chapter 70
lukewrites	Chapter 20
Lyndsy Simon	Chapter 21
machine yearning	Chapters 19, 53 and 67
magu_	Chapter 77
Mahdi	Chapters 21, 82 and 120
Mahmoud Hashemi	Chapter 199
Majid	Chapters 19, 28, 77, 82, 98, 112 and 173
Malt	Chapter 200
manu	Chapter 1
MANU	Chapter 1
Marco Pashkov	Chapters 29, 39, 40, 83 and 173
Mario Corchero	Chapter 192
Mark	Chapter 125
Mark Miller	Chapter 130
Mark Omo	Chapter 168
Markus Meskanen	Chapter 21
MarkyPython	Chapter 41
Marlon Abeykoon	Chapter 79
Martijn Pieters	Chapters 1, 67, 77 and 97
Martin Valgur	Chapter 104
Math	Chapter 16
Mathias711	Chapter 1
matsjoyce	Chapters 1 and 9
Matt Dodge	Chapters 149 and 200
Matt Giltaji	Chapters 33, 40, 173 and 193
Matt Rowland	Chapter 149
MattCorr	Chapters 4 and 121
Mattew Whitt	Chapters 21, 37, 39, 110, 146, 173 and 192
mattgathu	Chapters 19, 117, 124 and 190
Matthew	Chapter 77
max	Chapter 67
Max Feng	Chapter 14
mbrig	Chapter 21
mbsingh	Chapter 161
Md Sifatul Islam	Chapters 28 and 75
mdegis	Chapter 1
Mechanic	Chapters 1, 9, 19 and 28
mertyildiran	Chapter 1
MervS	Chapter 125
metmirr	Chapter 162

mezzode	Chapter 28
mgilson	Chapter 192
Michael Recachinas	Chapter 149
Michel Touw	Chapter 161
Mike Driscoll	Chapters 1 and 13
Miljen Mikic	Chapter 1
Mimouni	Chapter 1
Mirec Miskuf	Chapter 21
mnoronha	Chapters 1 and 149
Mohammad Julfikar	Chapter 123
moshemeirelles	Chapter 1
MrP01	Chapter 20
mrtuovinen	Chapter 92
MSD	Chapter 1
MSeifert	Chapters 9, 16, 19, 21, 26, 33, 37, 41, 43, 45, 48, 55, 64, 67, 68, 69, 70, 71, 72, 73 and 200
muddyfish	Chapters 1, 20, 21, 33, 37, 57, 111 and 149
Mukunda Modell	Chapter 37
Muntasir Alam	Chapter 1
Murphy4	Chapter 33
MYGz	Chapter 40
Naga2Raja	Chapter 150
Nander Speerstra	Chapters 40, 75 and 116
naren	Chapters 42 and 89
Nathan Osman	Chapter 190
Nathaniel Ford	Chapters 1, 29 and 41
ncmathsadist	Chapter 200
nd.	Chapter 33
nehemiah	Chapter 173
nemesisfixx	Chapter 10
Ni.	Chapters 1 and 92
Nick Humrich	Chapter 139
Nicolás	Chapter 29
Nicole White	Chapter 5
niemmi	Chapter 149
niyasc	Chapters 1, 43, 45 and 149
nlsdfnbch	Chapters 5, 19, 28, 30, 53, 67, 117, 120 and 121
Nour Chawich	Chapter 40
noɯʎ□ʎzɐɹƆ	Chapters 1, 19, 21, 28, 88 and 149
Nuhil Mehdy	Chapter 173
numbermaniac	Chapters 1 and 9
obust	Chapter 54
Ohad Eytan	Chapter 5
ojas mohril	Chapter 38
omgimanerd	Chapter 200
Or East	Chapters 8, 21, 75, 112 and 189
OrangeTux	Chapter 149
Ortomala Lokni	Chapter 173
orvi	Chapters 1, 25, 41, 125, 133 and 134
Oz Bar	Chapter 20
Ozair Kafray	Chapter 30
Panda	Chapter 21
Parousia	Chapters 23 and 69

Pasha	Chapters 3, 20, 21, 33, 37, 38, 67, 70, 77, 83 and 149
Patrick Haugh	Chapters 1 and 200
Paul	Chapter 5
Paul Weaver	Chapter 88
paulmorriss	Chapter 5
Paulo Freitas	Chapter 149
Paulo Scardine	Chapter 39
Pavan Nath	Chapters 1, 20 and 179
pcurry	Chapters 29, 110 and 149
Peter Brittain	Chapter 77
Peter Mølgaard Pallesen	Chapter 73
Peter Shinners	Chapter 109
petrs	Chapter 77
philngo	Chapter 16
Pigman168	Chapter 96
pistache	Chapter 38
pktangyue	Chapter 149
poke	Chapter 20
PolyGeo	Chapter 145
poppie	Chapter 149
ppperry	Chapter 55
Prem Narain	Chapter 59
Preston	Chapters 138 and 173
proprefenetre	Chapter 147
proprius	Chapter 5
PSN	Chapter 1
pylang	Chapters 1, 8, 21, 33, 38, 43, 53, 54, 73, 80, 128, 147, 173, 192 and 200
PYPL	Chapter 79
Pythonista	Chapters 32 and 149
pzp	Chapters 1, 29, 30, 33, 41, 49, 64 and 67
Quill	Chapter 1
qwertyuip9	Chapters 161, 173 and 181
R Colmenares	Chapter 10
R Nar	Chapter 21
Rápli András	Chapter 82
Régis B.	Chapter 173
Raghav	Chapter 125
Rahul Nair	Chapters 1, 21, 43, 88 and 143
RahulHP	Chapters 5, 8, 45, 53, 64 and 112
rajah9	Chapters 14, 16 and 45
Ram Grandhi	Chapter 1
RandomHash	Chapter 95
rassar	Chapter 172
ravigadila	Chapters 20, 60, 85, 91 and 104
Razik	Chapter 158
Rednivrug	Chapters 2, 35 and 63
regnarg	Chapter 75
Reut Sharabani	Chapters 64 and 200
rfkortekaas	Chapters 1 and 115
Ricardo	Chapter 157
Riccardo Petraglia	Chapters 21, 84 and 117
Richard Fitzhugh	Chapter 38
rlee827	Chapter 62

rll	Chapter 21
Rob H	Chapter 121
Rob Murray	Chapter 94
Ronen Ness	Chapter 30
ronrest	Chapters 19 and 41
Roy Iacob	Chapter 19
rrao	Chapter 1
rrawat	Chapter 161
Ryan Smith	Chapters 21 and 120
ryanyuyu	Chapter 21
Ry□	Chapter 149
Sachin Kalkur	Chapter 125
sagism	Chapter 5
Saiful Azad	Chapter 43
Sam Krygsheld	Chapter 1
Sam Whited	Chapter 111
Sangeeth Sudheer	Chapter 1
Saqib Shamsi	Chapter 92
Sardathrion	Chapter 103
Sardorbek Imomaliev	Chapter 103
sarvajeetsuman	Chapters 9 and 16
SashaZd	Chapters 12, 14, 29, 64, 86, 134, 143, 144, 146 and 194
satsumas	Chapter 67
sayan	Chapters 1 and 96
Scott Mermelstein	Chapters 110, 135 and 149
Sebastian Schrader	Chapter 173
Selcuk	Chapters 1, 28, 149 and 201
Sempoo	Chapter 38
Serenity	Chapters 20, 30, 40, 43, 149 and 173
SerialDev	Chapter 82
serv	Chapter 40
Seth M. Larson	Chapters 54 and 87
Seven	Chapters 1, 11, 20 and 33
sevenforce	Chapters 16 and 67
ShadowRanger	Chapter 149
Shantanu Alshi	Chapter 173
Shawn Mehan	Chapters 10, 15, 19, 20 and 47
Shihab Shahriar	Chapter 200
Shijo	Chapter 198
Shoe	Chapter 21
Shrey Gupta	Chapters 41, 56 and 173
Shreyash S Sarnayak	Chapter 12
Shuo	Chapter 77
SiggyF	Chapters 16 and 111
Simon Fraser	Chapter 173
Simon Hibbs	Chapter 169
Simplans	Chapters 1, 9, 10, 14, 16, 19, 21, 28, 33, 37, 38, 41, 43, 44, 45, 50, 69, 75, 77, 82, 88, 99, 147, 149 and 173
Sirajus Salayhin	Chapters 5, 175 and 176
sisanared	Chapter 39
skrrgwasme	Chapters 16 and 99
SN Ravichandran KR	Chapter 75
solarc	Chapters 20 and 149

Soumendra Kumar Sahoo	Chapters 14 and 38
SouvikMaji	Chapter 1
sricharan	Chapter 149
StardustGogeta	Chapter 43
stark	Chapter 1
Stephen Nyamweya	Chapter 161
Steve Barnes	Chapters 33 and 82
Steven Maude	Chapters 11, 33, 47 and 92
sth	Chapters 91, 92 and 141
strpeter	Chapters 85 and 192
StuxCrystal	Chapters 21, 37, 43, 56, 76, 82, 121 and 142
Sudip Bhandari	Chapter 88
Sun Qingyao	Chapter 101
Sunny Patel	Chapter 21
SuperBiasedMan	Chapters 1, 4, 15, 16, 20, 21, 26, 29, 30, 33, 41, 43, 64, 69, 77, 80, 88, 132, 149 and 200
supersam654	Chapter 70
surfthecity	Chapter 6
Symmitchry	Chapters 47 and 53
sytech	Chapter 92
Sнаđошfa□	Chapters 1 and 134
Tadhg McDonald	Chapter 149
talhasch	Chapters 92, 93 and 164
Tasdik Rahman	Chapter 30
taylor swift	Chapter 1
techydesigner	Chapters 1, 38, 43 and 190
Teepeemm	Chapter 152
Tejas Jadhav	Chapter 201
Tejus Prasad	Chapter 1
TemporalWolf	Chapter 90
textshell	Chapters 16, 28, 33 and 121
TheGenie OfTruth	Chapter 1
theheadofabroom	Chapters 41, 49 and 64
the_cat_lady	Chapter 43
The_Curry_Man	Chapter 16
Thomas Ahle	Chapters 60 and 134
Thomas Crowley	Chapter 105
Thomas Gerot	Chapters 30, 43, 58, 61, 126 and 181
Thomas Moreau	Chapter 119
Thtu	Chapter 80
Tim	Chapter 149
Tim D	Chapter 200
tjohnson	Chapter 44
tlo	Chapter 82
tobias_k	Chapters 28 and 40
Tom	Chapter 16
Tom Barron	Chapters 1 and 21
Tom de Geus	Chapter 1
Tony Meyer	Chapter 43
Tony Suffolk 66	Chapters 9, 10, 38 and 40
tox123	Chapter 38
TuringTux	Chapter 4
Tyler Crompton	Chapter 86

Tyler Gubala	Chapters 119 and 122
Udi	Chapter 54
UltraBob	Chapter 38
Umibozu	Chapter 30
Underyx	Chapter 49
Undo	Chapter 9
unutbu	Chapter 116
user2027202827	Chapter 163
user2314737	Chapters 8, 9, 13, 16, 20, 28, 30, 33, 38, 40, 41, 57, 60, 64, 69, 75, 88, 110, 134, 142, 149, 154, 161, 196 and 200
user2683246	Chapters 43 and 187
user2728397	Chapter 166
user2853437	Chapter 1
user312016	Chapters 1, 40 and 77
user3333708	Chapter 33
user405	Chapter 33
user6457549	Chapter 20
Utsav T	Chapter 20
vaichidrewar	Chapter 1
valeas	Chapter 161
Valentin Lorentz	Chapters 20, 21, 43, 110 and 149
Valor Naram	Chapter 43
vaultah	Chapters 20, 33, 43 and 77
Veedrac	Chapters 21, 33, 41 and 110
Vikash Kumar Jain	Chapter 174
Vin	Chapters 1 and 17
Vinayak	Chapter 149
vinzee	Chapters 99, 116 and 120
viveksyngh	Chapters 10 and 19
VJ Magar	Chapter 31
Vlad Bezden	Chapter 103
weewooquestionaire	Chapter 1
WeizhongTu	Chapters 41 and 149
Wickramaranga	Chapter 53
Will	Chapters 5, 15, 16, 21, 30, 33, 91, 116, 117, 121 and 164
Wingston Sharon	Chapter 155
Wladimir Palant	Chapters 21, 37 and 197
wnnmaw	Chapters 41, 43 and 53
Wolf	Chapter 149
WombatPM	Chapter 30
Wombatz	Chapter 200
wrwrwr	Chapter 173
wwii	Chapter 14
wythagoras	Chapter 9
Xavier Combelle	Chapters 15, 19, 111 and 120
XCoder Real	Chapter 47
xgord	Chapters 14 and 30
XonAether	Chapter 52
xtreak	Chapters 67 and 149
Y0da	Chapter 85
ygram	Chapter 190
Yogendra Sharma	Chapters 1 and 117
yurib	Chapter 64

Zach Janicki	Chapter 1
Zags	Chapter 1
Zaid Ajaj	Chapter 67
zarak	Chapter 149
Zaz	Chapter 21
zenlc2000	Chapter 34
Zhanping Shi	Chapter 145
zmo	Chapters 83, 140 and 141
zondo	Chapters 75 and 83
zopieux	Chapter 173
zvone	Chapters 13, 37, 39 and 112
zxxz	Chapter 33
Zydnar	Chapter 81
K□ÌSTƠF	Chapter 117
λuser	Chapters 67 and 77
Некто	Chapters 24, 37 and 128

You may also like