Tutorial to contribute to the CPython project Documentation

Release 0.0

Victor Stinner

Jan 23, 2019

Contents:

CHAPTER 1

Getting Help

This tutorial is an early draft. It is likely that you will quickly be blocked by an incomplete section (like "XXX") or a missing section. Don't hesistate to complain, ask for help, and report issues!

1.1 Active Core Developers

While the CPython developers has more than 100 core developers, some of them "left" the project and are no more active.

GitHub statistics on contributors can help to identify active contributors *if* you select a recent time window, like last 6 or 12 months.

Developers available to mentor new contributors:

- Victor Stinner, lives in France, works for Red Hat, core dev since 2010. IRC nickname: `vstinner` on Freenode. Timezone: France (Paris) local clock.
- Zachary (Zach) Ware, living in the US (midwest), working as an independent consultant, core dev since 2013. IRC nickname: `zware` on Freenode. Timezone: US/Central.

1.2 #python-dev channel

The IRC `#python-dev` channel of the Freenode server.

Some nicknames:

- `vstinner`: Victor Stinner (previously known as `haypo`)
- `zware`: Zach Ware
- `matrixise`: Stéphane Wirtel

1.3 Core-mentorship mailing list

- Python Core Mentorship
- The core-mentorship mailing list

1.4 Developer Guide

- Python Developer's Guide

1.5 Contributors

Contributors are also encouraged to help other contributors since new comers are the most experienced in issues to start contributing for CPython :-)

- Stéphane Wirtel, living in Belgium, working as an independent consultant, contributor since 2014. IRC nickname: `matrixise` on Freenode. Timezone: Europe/Brussels.

1.6 Looking for help on a specific Python module

The Expert Index lists maintainers for the some but not all Python modules.

CHAPTER 2

Getting Started

Goals:

- Get a running Python executable built manually
- Run basic sanity checks to make sure that your executable is working

Requirements:

- Know CPython, Git and the C compiler of your operating system

2.1 Build CPython on Linux

See also *Build CPython on macOS* for specific instructions for macOS.

2.1.1 Get tools and dependencies

- Install Git:
  - Fedora: `sudo dnf install -y git-core`
  - Debian: `sudo apt-get install -y git-core`
- Clone the GitHub cpython project: `git clone https://github.com/python/cpython.git` which creates a `cpython/` directory. The history starts in 1991 so be patient, the clone can take several minutes!
- Get CPython dependencies:
  - Fedora:
    - version based on `yum`:

```
$ sudo yum install yum-utils
$ sudo yum-buildep python3
```

  * version based on `dnf`:

```
$ sudo dnf install dnf-plugins-core
$ sudo dnf builddep python3
```

- Debian: `sudo apt-get build-dep python3`
- Dependencies to get `ssl` and `readline` modules: OpenSSL headers (openssl-dev) and readline headers (libreadline-dev).

2.1.2 Build Python

Single command:

```
./configure --with-pydebug && make
```

Or detailed instructions:

- Configuration Python in debug mode: `./configure --with-pydebug`
- Build CPython: `make`
- There is no need to install Python.

For example, with a missing dependency, the *make* command will show the optional modules which do not compile

```
The necessary bits to build these optional modules were not found:
_lzma
```

In this case, you have to install the *xz* library. For Fedora 25 and 26, it's `xz-devel`, but normally, this dependency is installed with `builddep python3`. For Debian/Ubuntu, this dependency should be installed with `apt-get build-dep python3`

2.1.3 Test Python

Test the Python REPL:

- Run Python: type `./python`
- Import the ssl module: `import ssl`. If the import works, you are good. Otherwise, you missed some dependencies.
- Exit Python

Run the test_os unit test:

```
./python -m test test_os
```

Expected output (something like this):

```
Run tests sequentially
0:00:00 load avg: 0.51 [1/1] test_os
1 test OK.

Total duration: 2 sec
Tests result: SUCCESS
```

If everything is fine, you are done! Let's move to the next section: XXX!

If something goes wrong, see *Getting Help*.

2.2 Build CPython on Windows

Guide to build the master branch of CPython. Other Python versions need a different Visual Studio version.

To develop on CPython, the best is to enable all debug checks and so compile Python in "debug mode".

2.2.1 Install tools and dependencies

- Install Visual Studio 2015: XXX
- Install Git: Download Git for Windows
- Open a terminal: run `cmd.exe`
- Go the Python directory
- Download source code of dependencies (OpenSSL, Tk, etc.): type `PCbuild\get_externals.bat`

2.2.2 Build CPython

- Open Visual Studio
- Open `PCbuild/pcbuild.sln` solution
- Select `Debug` and `x64` (64-bit)
- Right click on the solution: Build, or just "F10"
- Close Visual Studio

2.2.3 Run Python

- Open a terminal: run `cmd.exe`
- Go to the Python directory
- Run `PCbuild\amd64\python_d.exe`
- Try to import the `ssl` module: `import ssl`. If the ssl module is available, you are good. Otherwise, you missed maybe the "get_externals.bat" step?

Your `python_d.exe` works? Great! Let's move to the next section.

2.3 Build CPython on macOS

Guide written for brew.

See also *Build CPython on Linux*.

2.3.1 Get tools and dependencies

- Install homebrew
- Install Git: `brew install git`
- XXX install OpenSSL

- XXX install readline
- XXX install gcc, make, autotools?

2.3.2 Build CPython

Single command:

```
./configure --with-pydebug && make
```

Or detailed instructions:

- Configuration Python in debug mode: `./configure --with-pydebug`
- Build CPython: `make`
- There is no need to install Python.

XXX how to check "setup" output and detect missing dependencies?

2.3.3 Test Python

Test the Python REPL:

- Run Python: type `./python.exe`.
- Import the ssl module: `import ssl`. If the import works, you are good. Otherwise, you missed some dependencies.
- Exit Python

Run the test_os unit test:

```
./python -m test test_os
```

Expected output (something like this):

```
Run tests sequentially
0:00:00 load avg: 0.51 [1/1] test_os
1 test OK.

Total duration: 2 sec
Tests result: SUCCESS
```

If everything is fine, you are done! Let's move to the next section: XXX!

If something goes wrong, see *Getting Help*.

For more information about the compilation, you can read the compiling section of the devguide.

Badge: "getting_started".

CHAPTER 3

Contribute to CPython: where should I start?

CPython core isn't the easiest place to start. The development process is rather enterprisy with long release cycles and rigid backwards compatibility policy. CPython also support a lot of platforms and CPU architectures.

How can you start? Where? That's an hard question. CPython is old and widely used: any change must be carefully discussed to remain Python homogenous. For bug fixes, the most complex part is the backward compatibility.

It's very hard to find "easy issue". First, just **watch** the current activity to get some ideas.

- To start, the best is maybe to look at recent commits to see what is currently done to get some ideas: https://github.com/python/cpython/commits/master
- Look also at active bugs: see https://bugs.python.org/ homepage, and maybe the second and third pages
- You may also look at ideas currently discussed on the https://mail.python.org/mailman/listinfo/python-ideas mailing list

To find an easy issue, persistence is the key :-)

Contribute to CPython? What is CPython? CPython is made of many parts:

- CPython source code: basically made of 50% Python and 50% C code
- Build system: complex tools to build Python on all platforms, Visual Studio project for Windows
- Windows and macOS installer
- 233,000 lines of documentation written with Sphinx
- Documentation translated to multiple languages: french, japanese, and other languages
- Bug tracker: bugs.python.org which has its own "meta" bug tracker for bugs in the bug tracker :-)
- GitHub project: pull requests, host the Git repository
- Travis CI to run the test suite on Linux and check the documentation
- AppVeyor CI to run the test suite on Windows
- Buildbot master and many workers to run the test suite as post-commit on many variables architectures and operating systems

There are also many things around Python:

- Devguide: documentation of the CPython development workflow
- python-dev and python-committers mailing lists
- #python-dev IRC channel
- PyPI

Contributing to CPython is hard and it can take up to 2 years until your work is released. Are you sure that CPython itself is the best project for you? Depending on your interests and skills, you may enjoy better to contribute to another popular project like Django or requests.

Contributing to CPython is harder because CPython developers require a very high quality for contributions: every change must be carefully documented, tested and implemented. The implementation can take several rounds until it reachs the expected quality and respect a long list of requirements.

Ok, I understand and I am well aware of all these things. But I want to contribute, how should I start?

- Bug triage: complete bug report to adjust the title, complete the description, identify impacted Python version and impacted platforms, help to reproduce the bug, or even help to analyze and find the **root bug**.
- Review pull requests: https://github.com/python/cpython/pulls/
  - Make sure that a change is carefully documented. For example, behaviour changes and enhancements must have the ".. versionchanged:: x.y" markup in the documentation of the module. New features must be have the ".. versionadded:: x.y" tag and maybe also documented in "What's New in Python X.Y?" documentation.
  - The NEWS entry and commit message must first explain the solved problem, then maybe explain the change itself. While the commit message can be technical and very detailed, the NEWS entry should be short and written for end-users who don't care of technical details.
  - Every change must be tested: exceptions are rare and should usually be justified. Example of exception which doesn't have to be justified: changes only impacting the documentation, changes fixing a typo in a comment.
  - The backward compatibility is very important. A change must be carefully reviewed to make sure that it doesn't break the backward compatibility. If it does break it, the change must be discussed with enough people to make sure that there is a smooth transition plan.
  - Feature removals require a DeprecationWarning in Python version N to remove it in version N+1. It's common that a feature is kept longer to avoid breaking the world just being a developer wants the perfection. Python developers must be pragmatic: balance between perfection and usability, very hard choices should be made and usually it takes a a long to time to discuss them :-)
  - While Python 2 end of line is near (2020), Python 3 changes which make writing code working on Python 2 and Python 3 harder are usually rejected.
- Propose to write a bugfix change (a pull request) for an existing bug report. This task is very hard since usually if a bug report doesn't have a patch, it's because there is a disagrement on the bug itself, or because the bugfix is very hard to implement.

CHAPTER 4

Python Community

Goals:

- Know the CPython community, code of conduct, the diversity statement and communication channels

4.1 Code Of Conduct

The Python community wants to be welcome and fair with everyone and so has written down a code of conduct to decide how to handle conflicts: Python Community Code of Conduct.

The Code of Conduct applies to most, if not all, Python mailing lists, to the bug tracker, etc.

Questions/comments/reports? See the contact information at the end of Python Community Code of Conduct.

See also Proposal for procedures regarding CoC actions.

Action: [Click to confirm that you read the code of conduct :-)]

4.2 Diversity

Diversity Statement:

> The Python Software Foundation and the global Python community welcome and encourage participation by everyone. Our community is based on mutual respect, tolerance, and encouragement, and we are working to help each other live up to these principles. We want our community to be more diverse: whoever you are, and whatever your background, we welcome you.

- https://www.python.org/community/diversity/
- http://wiki.python.org/moin/DiversityInPython

For example, don't say "hey *guys*!" but "hey **everyone**!".

Action: [Click to confirm that you read the diversity statement :-)]

4.3 CPython Communication Channels

4.3.1 Communication Channels

- IRC: #python-dev on Freenode
- Main Mailing lists:
  - python-ideas
  - python-dev

4.3.2 Physical meetings

- Pycon US: https://us.pycon.org/
- Language Summit, during Pycon US
  - 2017: https://us.pycon.org/2017/events/language-summit/
  - LWN report, 2017: https://lwn.net/Articles/723251/
  - LWN report, 2016: https://lwn.net/Articles/688969/
  - LWN report, 2015: https://lwn.net/Articles/639773/
  - XXX: [recent group photo]
- EuroPython: no formal meeting, discussions in the corridor usually
- CPython sprints:
  - 2016: http://blog.python.org/2016/09/python-core-development-sprint-2016-36.html

Action: [Click to confirm that you read everything :-)]

Badge: “community”.

CHAPTER 5

Run tests

5.1 Introduce a Bug

Instead of trying to write new code, let's start by writing a bug!

Diff:

```
diff --git a/Lib/http/server.py b/Lib/http/server.py
index b1151a2..f132be4 100644
--- a/Lib/http/server.py
+++ b/Lib/http/server.py
@@ -965,6 +965,7 @@ class CGIHTTPRequestHandler(SimpleHTTPRequestHandler):
     rbufsize = 0

     def do_POST(self):
+        bug
         """Serve a POST request.

         This is only implemented for CGI scripts.
```

5.2 Run tests

Type:

```
$ ./python -m test -v test_httpservers
```

We expect that tests fail:

```
(...)
======================================================================
ERROR: test_post (test.test_httpservers.CGIHTTPServerTestCase)
----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File ".../Lib/test/test_httpservers.py", line 726, in test_post
    res = self.request('/cgi-bin/file2.py', 'POST', params, headers)
  File ".../Lib/test/test_httpservers.py", line 79, in request
    return self.connection.getresponse()
  File ".../Lib/http/client.py", line 1322, in getresponse
    response.begin()
  File ".../Lib/http/client.py", line 296, in begin
    version, status, reason = self._read_status()
  File ".../Lib/http/client.py", line 265, in _read_status
    raise RemoteDisconnected("Remote end closed connection without"
http.client.RemoteDisconnected: Remote end closed connection without response

----------------------------------------------------------------------
(...)
FAILED (errors=1)
(...)
```

Re-run only the failing method:

```
$ ./python -m test -v -m test_post test_httpservers
```

Find the code of the failing tests, see most recent call in the traceback:

```
File ".../Lib/test/test_httpservers.py", line 726, in test_post
  res = self.request('/cgi-bin/file2.py', 'POST', params, headers)
```

Open Lib/test/test_httpservers.py line 726:

```
def test_post(self):
    params = urllib.parse.urlencode(
        {'spam' : 1, 'eggs' : 'python', 'bacon' : 123456})
    headers = {'Content-type' : 'application/x-www-form-urlencoded'}
    res = self.request('/cgi-bin/file2.py', 'POST', params, headers)

    self.assertEqual(res.read(), b'1, python, 123456' + self.linesep)
```

5.3 Fix the bug

- Modify Lib/http/server.py to remove `bug`.
- Re-run test_post() test

5.4 Run the whole test suite

- Re-run all test_httpservers tests
- Run the whole test suite

Run the whole test suite:

```
./python -m test -j0 -rW
```

Flags:

- -j0: run tests in parallel using as many test processes than your CPUs
- -r: randomize tests
- -W: hide test output on success, but write tests output on failure

CHAPTER 6

Write your first Python fix

Goal: be able to write a Python fix with an unit test, but not publish it.

6.1 How to find a bug or feature to write

XXX

6.2 Write the code

XXX

6.3 Add an unit test

XXX

6.4 Check for non-regression

XXX

CHAPTER 7

Write your first C fix

Goal: be able to write a C fix with an unit test (but not publish it).

7.1 How to find a bug or feature to write

XXX

7.2 Write the code

XXX

7.3 Add an unit test

XXX

7.4 Check for non-regression

XXX

7.5 Check for reference leaks

Run your test using:: `./python -m test -R 3:3 test_xxx`.

CHAPTER 8

Open your first issue on bugs.python.org

Required badge: “community”.

8.1 Create your account

- http://bugs.python.org/
- Click on [Register]
- Fill the form
- XXX
- Confirm your email
- XXX

8.2 Open your issue

- Log in
- Click on [Create New]
- Write a short title
- Describe the bug or feature request
- [Submit New Entry]

8.3 Bug fix

Complete your issue for a bug fix:

- Specify the OS
- Specify your Python version
- Write a script to reproduce the bug, the best is if the script doesn't use third-party modules

8.4 Feature request

XXX

Congrats! Next section: XXX

Badge: "first bpo created".

CHAPTER 9

Write your first Pull Request

Required badge: “community”.

9.1 Create your GitHub account

If you already have a GitHub account, skip this section ;-)

- https://github.com/
- Click on [Sign up]
- Fill the form
- XXX
- Confirm your email
- XXX

9.2 Link your bugs.python.org account to your GitHub account

- Go to your bugs.python.org account
- Go to “Your Details”
- Set the **GitHub name** field
- Submit your changes

9.3 Sign the CLA

Sign the Python Contributor Agreement.

- Sign https://www.python.org/psf/contrib/contrib-form/
- Wait... the confirmation is done manually by humans, so be prepared to wait up to one week (usually 3 business days)

For legal question, you can send an email to legal@python.org .

9.4 Create a local Git branch

Once you have your patch (see XXX Python or XXX C fix sections), make sure that your changes are in a local branch:

```
git checkout -b mybranch
```

Template of a commit message:

```
bpo-12345: fix a bug in the email module

Fix a race condition in the email module. Before xxx failed. It now works
well!
```

9.5 Git remotes

- origin: your repository
- upstream: cpython/python

9.6 Publish your pull request

9.6.1 Manually

- git push origin HEAD
- Open https://github.com/python/cpython and click on "New pull request"
- Select the base fork: `python/cpython` and base branch: `master`
- Select the head fork: `<username>/cpython` and base branch: the branch
- Press `Create Pull Request` button

For more information: please, you could read this section of the devguide

Given badge: "first pr".

CHAPTER 10

Lifetime of a change: bugs.python.org and PR updates

10.1 First, nothing happens

xxx

10.2 First feedback

xxx

10.3 Update your PR

xxx

CHAPTER 11

Review bugs and pull requests

Reading code written by others can be a good practice to learn from others, and learn the Python workflow.

The repository on GitHub has a page for the Pull Requests (PR) and in this case, you could go on the page and review the proposed PR.

For example, we prefer to review the PR where the CLA is signed

Usually, if the CLA is not signed by the contributor, we prefer to ask to sign the document and wait for 3 business days, its bugs.python.org account will be updated.

CHAPTER 12

Propose new idea

To propose new ideas, the python-ideas mailing list is the good place.

- The python-idea mailing list
- python-ideas archives
- LWN: Things that won't change in Python
- LWN: Ideas versus implementation

CHAPTER 13

python-dev mailing list

To discuss more in depth the actual implementation of a feature, the python-dev mailing list is the good place.

- https://mail.python.org/mailman/listinfo/python-dev
- https://mail.python.org/pipermail/python-dev/

CHAPTER 14

Continuous Integration (CI)

14.1 Travis CI

XXX

Configuration: `.travis.yml` file.

14.2 AppVeyor

XXX

Configuration: `.github/appveyor.yml` file.

14.3 Buildbots

- python-buildbots mailing list
- buildbot-status mailing list

CPython is running a Buildbot server for continuous integration, but tests are run as post-commit: see Python buildbots.

Buildbot "waterfall" views:

- Python master ("3.x")
- Python 3.6
- Python 3.5
- Python 2.7

The buildbot configuration can be found in the buildmaster-config project (start with the `master/master.cfg` file).

CHAPTER 15

What is a CPython core developer?

Devguide: How to Become a Core Developer

See [python-committers] What is a CPython core developer? (Sept 2017).

15.1 python-committers

While the python-committers mailing list is public (anyone can be archives), only core developers are allowed to subscribe and to send messages.

15.2 Who are core developers?

- Devguide Developer Log
- GitHub contributors statistics gives an idea of which core developers were active recently: click on the graph to select a recent period, like last 6 months

15.3 Different stages of core developers

- Newcomer
- Contributor: as soon as you post a comment, send an email, ask a question, you become an active contributor!
- *Permission for bug triage*: once peers estimated that your behaviour is correct and that you are active, you may be proposed to be promoted to "bug triage"
- *Followed by a mentor*: spotted active contributors can be asked to get a mentor to speedup their learning
- *Core developer*, allowed to merge a pull request: once other core developers consider that a contributor is ready to be promoted, a core dev opens a vote on python-committers

15.4 New powers but also new responsabilities

The main power of a core developer is to be allowed to merge a pull request.

New powers comes with new responsabilities. Merging a pull request indirectly means becoming responsible of the added and modified code.

15.5 Bug tracker

See Helping Triage Issues and Triaging an Issue.

15.5.1 Requirements to get the bug triage permission

Requirements:

- Be nice, polite and respectful
- Know what they are talking about, and explain their reasoning well. Contributions to the bug tracker that demonstrate an understanding of our code culture... specifically, commenting that a bug should be closed (and why) and demonstrating an understanding of what python versions a bug applies to (enhancement versus bug fix, when to backport a bug fix and when not). See the Status of Python branches
- Don't close a bug if it's not well understood to not "lose" information (closed bugs are ignored in search by default, and hidden from the main page).

A good triager must be familiar with our codebase, our bug tracker, and our "code culture", in particular:

- being able to find (or remember) duplicate and related issues, link them to each other, and closing the duplicates as necessary;
- being able to correctly select the versions affected by the issue, the components, the stage, and other fields;
- being able to verify if the issue can be reproduced and if the report is valid or not;
- being able to recognize commonly reported issues and link to the appropriate FAQ or other existing issues/explanations;
- being able to identify and specify the next step, possibly suggesting which files should be updated to fix the issue;
- being able to locate the commits that might have introduced the issue, and the reason for the change;
- being able to leave meaningful opinions on the issue (e.g. whether it should be addressed or closed and why);

It usually takes some time and a few contributions before people can do these things.

Responsability for bug tracker:

- Request more information if a report is incomplete
- Ping original reporters if they don't reply
- Adjust Python version, component, bug type, etc.
- Rewrite the issue title if needed
- Close duplicated bugs as DUPLICATE
- Close irrevelant bugs as NOTABUG

Nice to have, but not a strong requirement:

- Signed the CLA
- At least one commit merged into CPython

15.5.2 Give bug triage permission to a contributor

A core developer has to identify an active contributor on the bug tracker, or a contributor can also ask for more power themselves.

If the contributor reachs the requirements listed in the previous section, the core developer give them triage.

15.6 Mentoring

XXX see *Getting help*.

15.7 Requirements to become a core developer

> "*A core dev should know not only what changes should be made, but also (and this is more important) what changes should not be made.*" – Serhiy Storchaka

The historical definition is that CPython core developer has the *commit bit*, is able to push a change to *upstream*, to the GitHub cpython project.

Officially, other core developers don't expect and cannot expect anything from a developer.

The unwritten definition is that other core developers expect from a core developer to maintain their contributions: long term *commitement*, since CPython requires long term support: 5 years or longer. If a core developer writes a new large chunk of code but then disappears for whatever reasons, there is a risk that the code dies slowly.

For example, Python branches are supported for 5 years. See the Status of Python branches.

Most of the following core developer requirements are already expected from regular contributors.

15.7.1 Be nice and respectful

Know to be nice and respectful to the others, at least to the extent they're nice and respectful to yourself :-) We don't have a rock-star (or "bro", "wizard", "ninja", whatever the hyperbole of the day is) culture here.

15.7.2 Humility

Show a bit of humility towards existing work and try to understand the decisions behind something before deciding to change it all. That said, given Python's current position on the technical evolution and adoption curve, we get less and less proposals for sweeping changes (perhaps not enough, actually, since even when rejected, they help challenge the status quo).

15.7.3 Long term commitement

When someone lands a big chunk of code, we need someone to maintain it for at least the next 2 years. Maybe for the next 10 years.

It is not strictly a requirement, it's more a whish from other core developers.

15.7.4 Reviews

Review patches and pull requests. While we don't require not expect newcomers to review, we expect that core developers dedicate a part of their time on reviews. What it means is that core developers care about the quality of the whole code base (and also the non-code parts), not only their own contributions to it.

15.7.5 CPython workflow

Know the CPython workflow. Be aware of the pre-commit and post-commits CIs. How ideas are discussed. It's not only about writing and pushing patches. This part is also required from regular contributors, at least the experienced ones.

15.7.6 CPython lifecycle

Know the project's lifecycle: Python has multiple maitained branches, some of them accept bugfixes, others only security fixes. Deciding if a fix can or cannot be backported is a complex question.

15.7.7 Python C API specific issues

For C developer: know CPython specific issues like reference leaks.

15.7.8 Good quality patches

Good quality patches: proposed changes are good (or almost good) at the first iteration. Or, if the code isn't good at the first iteration, the author is able to figure it out by themselves and doesn't rush merge it. Of course, nobody is perfect, which is why non-trivial code written by core developers ideally goes through a review phase anyway. But a general sense of what is "in good state for review/merging" vs. "just a draft I'm working on" is indeed preferrable.

15.7.9 Maintain pushed code

Pushing core means becoming responsible for this code. For regressions, backward compatibility, security, etc.

15.7.10 Backward compatibility

CPython has a long history and many unwritten strict rules. For example, backward compatibility is taken very seriously. We don't remove public functions in a minor release (3.x), but start with a deprecation period. It's not only about removing features, but also *changing* the behaviour. Even if Python has a wide test suite with a good code coverage, some functions are still untested, or not fully tested.

15.8 Be patient

Be patient, being aware of Python complex code and workflow can take between 6 months and 2 years.

15.9 Vote to promote a contributor as a core developer

15.9.1 Identify a potential candidate

Usually the contributor doesn't ask for themselves to become a core developer, but another core developer proposes to promote them.

The main questions about a potential new core developer are:

- Would gaining core developer privileges improve their ability to contribute effectively (in my opinion or the opinion of another core developer)?
- Do a core developer that is willing to mentor them trust their judgment on when things should be escalated for further review & discussion (or even rejected outright) versus just going ahead and merging them?

15.9.2 Ask the candidate permission

The first step is to ask the contributor if they want to become a core developer. Since great power comes with great responsibilities, it's not uncommon that some contributors prefer to remain contributors. Technically, it doesn't prevent to propose pull requests, reviews, etc.

15.9.3 Private vote

If the contributor agrees, usually a *private* discussion starts. Sadly, the discussion is private because it's tricky to discuss someone skills in public. A negative vote can be harmful, whereas it isn't the intend.

Sometimes, the contributor is evaluated as "too green" and someone can propose to become their mentor to help them to learn the workflow, give advices on pull requests, etc.

15.9.4 Public vote

The real vote occurs on the python-committers list where only core developers are allow to post. The developer who proposes to promote someone has to write a very short biography, list previous contributions and evaluates the contributor skills.

A negative vote can still happen at this point. It doesn't mean that the contributor will never become a core dev, just that they need more practice.

15.9.5 Vote result

If the vote is positive, the contributor sends their SSH key and will be subscribed to the python-committers mailing list.

15.9.6 Examples of public votes

- Proposing Carol Willing to become a core developer (Brett Cannon, May 2017)
- Proposed new core developer – Mariatta Wijaya (Raymond Hettinger, January 2017)
- Promote Xiang Zhang as a core developer (Victor Stinner, Nov 2016)
- commit privileges for INADA Naoki (Yury Selivanov, Sept 2016)

Other votes:

- commit privs given to Maciej Szulik for bugs.python.org work (Brett Cannon, December 2016): https://hg.python.org/tracker/ repository

15.9.7 Gaining Commit Privileges

See DevGuide: Gaining Commit Privileges.

15.10 We Are All Volunteers

Except one or two exceptions, no core developer is paid to contribute to CPython: *we are all volunteers*. Don't be surprised to not get any kind of feedback in next hours, or sometimes even before one week.

Most core developers have a specific interest in specific areas of the code, and so not all core developers are interested by your specific issue.

CHAPTER 16

Documentation of CPython internals

- The official Python Developer's Guide
- Prashanth Raghu's documentation:
  - Internals of CPython 2.7
  - Internals of CPython 3.6
  - Advanced Internals of CPython 3.6
- Python Development Documentation by Victor Stinner
- CPython internals: A ten-hour codewalk through the Python interpreter source code (October 2014) by Philip Guo

CHAPTER 17

Indices and tables

- genindex
- modindex
- search