Python Developer's Guide Documentation

Brett Cannon

Jun 17, 2022

CONTENTS

This guide is a comprehensive resource for *contributing* to Python – for both new and experienced contributors. It is *maintained* by the same community that maintains Python. We welcome your contributions to Python!

CHAPTER

ONE

QUICK REFERENCE

Here are the basic steps needed to get set up and contribute a patch. This is meant as a checklist, once you know the basics. For complete instructions please see the *setup guide*.

1. Install and set up *Git* and other dependencies (see the *Git Setup* page for detailed information).
2. Fork the CPython repository to your GitHub account and *get the source code* using:

   ```
   git clone https://github.com/<your_username>/cpython
   cd cpython
   ```

3. Build Python, on UNIX and Mac OS use:

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

   and on Windows use:

   ```
   PCbuild\build.bat -e -d
   ```

   See also *more detailed instructions*, *how to install and build dependencies*, and the platform-specific pages for *UNIX*, *Mac OS*, and *Windows*.

4. *Run the tests*:

   ```
   ./python -m test -j3
   ```

   On *most* Mac OS X systems, replace `./python` with `./python.exe`. On Windows, use `python.bat`.

5. Create a new branch where your work for the issue will go, e.g.:

   ```
   git checkout -b fix-issue-12345 main
   ```

   If an issue does not already exist, please create it. Trivial issues (e.g. typo fixes) do not require any issue to be created.

6. Once you fixed the issue, run the tests, run `make patchcheck`, and if everything is ok, commit.
7. Push the branch on your fork on GitHub and *create a pull request*. Include the issue number using `gh-NNNN` in the pull request description. For example:

   ```
   gh-12345: Fix some bug in spam module
   ```

8. Add a News entry into the `Misc/NEWS.d` directory as individual file. The news entry can be created by using blurb-it, or the blurb tool and its `blurb add` command. Please read more about `blurb` in *documentation*.

Note: First time contributors will need to sign the Contributor Licensing Agreement (CLA) as described in the *Licensing* section of this guide.

CHAPTER

TWO

QUICK LINKS

Here are some links that you probably will reference frequently while contributing to Python:

- Issue tracker
- Buildbot status
- *Where to Get Help*
- PEPs (Python Enhancement Proposals)
- *Git Bootcamp and Cheat Sheet*

CHAPTER

THREE

STATUS OF PYTHON BRANCHES

Branch	Schedule	Status	First release	End-of-life	Release manager
main	TBA	features	*2023-10-03*	*2028-10*	Thomas Wouters
3.11	**PEP 664**	bugfix	*2022-10-03*	*2027-10*	Pablo Galindo Salgado
3.10	**PEP 619**	bugfix	2021-10-04	*2026-10*	Pablo Galindo Salgado
3.9	**PEP 596**	security	2020-10-05	*2025-10*	Łukasz Langa
3.8	**PEP 569**	security	2019-10-14	*2024-10*	Łukasz Langa
3.7	**PEP 537**	security	2018-06-27	*2023-06-27*	Ned Deily

Dates in *italic* are scheduled and can be adjusted.

The main branch is currently the future Python 3.12, and is the only branch that accepts new features. The latest release for each Python version can be found on the download page.

Status:

features
new features, bugfixes, and security fixes are accepted.

prerelease
feature fixes, bugfixes, and security fixes are accepted for the upcoming feature release.

bugfix
bugfixes and security fixes are accepted, new binaries are still released. (Also called **maintenance** mode or **stable** release)

security
only security fixes are accepted and no more binaries are released, but new source-only versions can be released

end-of-life
release cycle is frozen; no further changes can be pushed to it.

See also the *Development Cycle* page for more information about branches.

By default, the end-of-life is scheduled 5 years after the first release, but can be adjusted by the release manager of each branch. All Python 2 versions have reached end-of-life.

CHAPTER

FOUR

CONTRIBUTING

We encourage everyone to contribute to Python and that's why we have put up this developer's guide. If you still have questions after reviewing the material in this guide, then the Core Python Mentorship group is available to help guide new contributors through the process.

A number of individuals from the Python community have contributed to a series of excellent guides at Open Source Guides.

Core developers and contributors alike will find the following guides useful:

- How to Contribute to Open Source
- Building Welcoming Communities

Guide for contributing to Python:

New Contributors	Documentarians	Triagers	Core Developers
Getting Started	*Helping with Documentation*	*Issue Tracking*	*How to Become a Core Developer*
Where to Get Help	*Documenting Python*	*Triaging an Issue*	*Developer Log*
Lifecycle of a Pull Request	*Style guide*	*Helping Triage Issues*	*Accepting Pull Requests*
Running & Writing Tests	*reStructuredText Primer*	*Experts Index*	*Development Cycle*
Fixing "easy" Issues (and Beyond)	*Translating*		*Core Developer Motivations and Affiliations*
Following Python's Development			*Core Developers Office Hours*
Git Bootcamp and Cheat Sheet			

Advanced tasks and topics for once you are comfortable:

- *Silence Warnings From the Test Suite*
- Fixing issues found by the *buildbots*
- *Coverity Scan*
- Helping out with reviewing open pull requests. See *how to review a Pull Request*.
- *Fixing "easy" Issues (and Beyond)*

It is **recommended** that the above documents be read as needed. New contributors will build understanding of the CPython workflow by reading the sections mentioned in this table. You can stop where you feel comfortable and begin contributing immediately without reading and understanding these documents all at once. If you do choose to skip around within the documentation, be aware that it is written assuming preceding documentation has been read so you may find it necessary to backtrack to fill in missing concepts and terminology.

CHAPTER

FIVE

PROPOSING CHANGES TO PYTHON ITSELF

Improving Python's code, documentation and tests are ongoing tasks that are never going to be "finished", as Python operates as part of an ever-evolving system of technology. An even more challenging ongoing task than these necessary maintenance activities is finding ways to make Python, in the form of the standard library and the language definition, an even better tool in a developer's toolkit.

While these kinds of change are much rarer than those described above, they do happen and that process is also described as part of this guide:

- *Adding to the Stdlib*
- *Changing the Python Language*

CHAPTER

SIX

OTHER INTERPRETER IMPLEMENTATIONS

This guide is specifically for contributing to the Python reference interpreter, also known as CPython (while most of the standard library is written in Python, the interpreter core is written in C and integrates most easily with the C and C++ ecosystems).

There are other Python implementations, each with a different focus. Like CPython, they always have more things they would like to do than they have developers to work on them. Some major examples that may be of interest are:

- PyPy: A Python interpreter focused on high speed (JIT-compiled) operation on major platforms
- Jython: A Python interpreter focused on good integration with the Java Virtual Machine (JVM) environment
- IronPython: A Python interpreter focused on good integration with the Common Language Runtime (CLR) provided by .NET and Mono
- Stackless: A Python interpreter focused on providing lightweight microthreads while remaining largely compatible with CPython specific extension modules

CHAPTER

SEVEN

KEY RESOURCES

- **Coding style guides**
  - **PEP 7** (Style Guide for C Code)
  - **PEP 8** (Style Guide for Python Code)
- **Issue tracker**
  - Meta tracker (issue tracker for the issue tracker)
  - *Experts Index*
- Buildbot status
- **Source code**
  - Browse online
  - Snapshot of the *main* branch
  - Daily OS X installer
- PEPs (Python Enhancement Proposals)
- *Where to Get Help*
- *Developer Log*

CHAPTER

EIGHT

ADDITIONAL RESOURCES

- Anyone can clone the sources for this guide. See *Helping with the Developer's Guide*.
- **Help with ...**
  - *Exploring CPython's Internals*
  - *Changing CPython's Grammar*
  - *Guide to CPython's Parser*
  - *Design of CPython's Compiler*
  - *Design of CPython's Garbage Collector*
- **Tool support**
  - *gdb Support*
  - *Dynamic Analysis with Clang*
  - Various tools with configuration files as found in the Misc directory
  - Information about editors and their configurations can be found in the wiki
- python.org maintenance
- Search this guide

CHAPTER

NINE

CODE OF CONDUCT

Please note that all interactions on Python Software Foundation-supported infrastructure is covered by the PSF Code of Conduct, which includes all infrastructure used in the development of Python itself (e.g. mailing lists, issue trackers, GitHub, etc.). In general this means everyone is expected to be open, considerate, and respectful of others no matter what their position is within the project.

CHAPTER

TEN

FULL TABLE OF CONTENTS

10.1 Getting Started

These instructions cover how to get a working copy of the source code and a compiled version of the CPython interpreter (CPython is the version of Python available from https://www.python.org/). It also gives an overview of the directory structure of the CPython source code.

Alternatively, if you have Docker installed you might want to use our official images. These contain the latest releases of several Python versions, along with git head, and are provided for development and testing purposes only.

See also:

The *Quick Reference* gives brief summary of the process from installing git to submitting a pull request.

10.1.1 Install `git`

CPython is developed using git for version control. The git command line program is named `git`; this is also used to refer to git itself. git is easily available for all common operating systems.

- **Install**

  As the CPython repo is hosted on GitHub, please refer to either the GitHub setup instructions or the git project instructions for step-by-step installation directions. You may also want to consider a graphical client such as TortoiseGit or GitHub Desktop.

- **Configure**

  Configure *your name and email* and create an SSH key as this will allow you to interact with GitHub without typing a username and password each time you execute a command, such as `git pull`, `git push`, or `git fetch`. On Windows, you should also *enable autocrlf* .

10.1.2 Get the source code

The CPython repo is hosted on GitHub. To get a copy of the source code you should *fork the Python repository on GitHub*, *create a local clone of your personal fork, and configure the remotes*.

You will only need to execute these steps once:

1. Go to https://github.com/python/cpython.
2. Press *Fork* on the top right.
3. When asked where to fork the repository, choose to fork it to your username.
4. Your fork will be created at `https://github.com/<username>/cpython`.

5. Clone your GitHub fork (replace `<username>` with your username):

   ```
   $ git clone git@github.com:<username>/cpython.git
   ```

   (You can use both SSH-based or HTTPS-based URLs.)

6. Configure an `upstream` remote:

   ```
   $ cd cpython
   $ git remote add upstream git@github.com:python/cpython.git
   ```

7. Verify that your setup is correct:

   ```
   $ git remote -v
   origin  git@github.com:<your-username>/cpython.git (fetch)
   origin  git@github.com:<your-username>/cpython.git (push)
   upstream        git@github.com:python/cpython.git (fetch)
   upstream        git@github.com:python/cpython.git (push)
   ```

If you did everything correctly, you should now have a copy of the code in the `cpython` directory and two remotes that refer to your own GitHub fork (`origin`) and the official CPython repository (`upstream`).

If you want a working copy of an already-released version of Python, i.e., a version in *maintenance mode*, you can checkout a release branch. For instance, to checkout a working copy of Python 3.8, do `git checkout 3.8`.

You will need to re-compile CPython when you do such an update.

Do note that CPython will notice that it is being run from a working copy. This means that if you edit CPython's source code in your working copy, changes to Python code will be picked up by the interpreter for immediate use and testing. (If you change C code, you will need to recompile the affected files as described below.)

Patches for the documentation can be made from the same repository; see *Documenting Python*.

10.1.3 Compile and build

CPython provides several compilation flags which help with debugging various things. While all of the known flags can be found in the `Misc/SpecialBuilds.txt` file, the most critical one is the `Py_DEBUG` flag which creates what is known as a "pydebug" build. This flag turns on various extra sanity checks which help catch common issues. The use of the flag is so common that turning on the flag is a basic compile option.

You should always develop under a pydebug build of CPython (the only instance of when you shouldn't is if you are taking performance measurements). Even when working only on pure Python code the pydebug build provides several useful checks that one should not skip.

See also:

The effects of various configure and build flags are documented in the Python configure docs.

UNIX

The core CPython interpreter only needs a C compiler to be built, however, some of the extension modules will need development headers for additional libraries (such as the `zlib` library for compression). Depending on what you intend to work on, you might need to install these additional requirements so that the compiled interpreter supports the desired features.

If you want to install these optional dependencies, consult the *Install dependencies* section below.

If you don't need to install them, the basic steps for building Python for development is to configure it and then compile it.

Configuration is typically:

```
./configure --with-pydebug
```

More flags are available to `configure`, but this is the minimum you should do to get a pydebug build of CPython.

Note: You might need to run `make clean` before or after re-running `configure` in a particular build directory.

Once `configure` is done, you can then compile CPython with:

```
make -s -j2
```

This will build CPython with only warnings and errors being printed to stderr and utilize up to 2 CPU cores. If you are using a multi-core machine with more than 2 cores (or a single-core machine), you can adjust the number passed into the `-j` flag to match the number of cores you have (or if your version of Make supports it, you can use `-j` without a number and Make will not limit the number of steps that can run simultaneously.).

At the end of the build you should see a success message, possibly followed by a list of extension modules that haven't been built because their dependencies were missing:

```
Python build finished successfully!
The necessary bits to build these optional modules were not found:
_bz2                  _dbm                  _gdbm
_lzma                 _sqlite3              _ssl
_tkinter              _uuid                 readline
zlib
To find the necessary bits, look in setup.py in detect_modules()
for the module's name.
```

If the build failed and you are using a C89 or C99-compliant compiler, please open a bug report on the issue tracker.

If you decide to *Install dependencies*, you will need to re-run both `configure` and `make`.

Once CPython is done building you will then have a working build that can be run in-place; `./python` on most machines (and what is used in all examples), `./python.exe` wherever a case-insensitive filesystem is used (e.g. on OS X by default), in order to avoid conflicts with the `Python` directory. There is normally no need to install your built copy of Python! The interpreter will realize where it is being run from and thus use the files found in the working copy. If you are worried you might accidentally install your working copy build, you can add `--prefix=/tmp/python` to the configuration step. When running from your working directory, it is best to avoid using the `--enable-shared` flag to `configure`; unless you are very careful, you may accidentally run with code from an older, installed shared Python library rather than from the interpreter you just built.

Clang

If you are using clang to build CPython, some flags you might want to set to quiet some standard warnings which are specifically superfluous to CPython are `-Wno-unused-value -Wno-empty-body -Qunused-arguments`. You can set your `CFLAGS` environment variable to these flags when running `configure`.

If you are using clang with ccache, turn off the noisy `parentheses-equality` warnings with the `-Wno-parentheses-equality` flag. These warnings are caused by clang not having enough information to detect that extraneous parentheses in expanded macros are valid, because the preprocessing is done separately by ccache.

If you are using LLVM 2.8, also use the `-no-integrated-as` flag in order to build the `ctypes` module (without the flag the rest of CPython will still build properly).

Windows

For a quick guide to building you can read this documentation from Victor Stinner.

All current versions of Python can be built using Microsoft Visual Studio 2017 or later. You can download and use any of the free or paid versions of Visual Studio 2017.

When installing Visual Studio 2017, select the **Python development** workload and the optional **Python native development tools** component to obtain all of the necessary build tools. If you do not already have git installed, you can find git for Windows on the **Individual components** tab of the installer.

Note: If you want to build MSI installers, be aware that the build toolchain for them has a dependency on the Microsoft .NET Framework Version 3.5 (which may not be configured on recent versions of Windows, such as Windows 10). If you are building on a recent Windows version, use the Control Panel (Programs | Programs and Features | Turn Windows Features on or off) and ensure that the entry ".NET Framework 3.5 (includes .NET 2.0 and 3.0)" is enabled.

Your first build should use the command line to ensure any external dependencies are downloaded:

```
PCbuild\build.bat
```

After this build succeeds, you can open the `PCbuild\pcbuild.sln` solution in Visual Studio to continue development.

See the readme for more details on what other software is necessary and how to build.

Note: If you are using the Windows Subsystem for Linux (WSL), clone the repository from a native Windows terminal program like cmd.exe command prompt or PowerShell as well as use a build of git targeted for Windows, e.g., the official one from https://git-scm.com. Otherwise, Visual Studio will not be able to find all the project's files and will fail the build.

10.1.4 Install dependencies

This section explains how to install additional extensions (e.g. `zlib`) on *Linux* and *macOs/OS X*. On Windows, extensions are already included and built automatically.

Linux

For UNIX based systems, we try to use system libraries whenever available. This means optional components will only build if the relevant system headers are available. The best way to obtain the appropriate headers will vary by distribution, but the appropriate commands for some popular distributions are below.

On **Fedora**, **Red Hat Enterprise Linux** and other `yum` based systems:

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

On **Fedora** and other `DNF` based systems:

```
$ sudo dnf install dnf-plugins-core  # install this to use 'dnf builddep'
$ sudo dnf builddep python3
```

On **Debian**, **Ubuntu**, and other `apt` based systems, try to get the dependencies for the Python you're working on by using the `apt` command.

First, make sure you have enabled the source packages in the sources list. You can do this by adding the location of the source packages, including URL, distribution name and component name, to `/etc/apt/sources.list`. Take Ubuntu 22.04 LTS (Jammy Jellyfish) for example:

```
deb-src http://archive.ubuntu.com/ubuntu/ jammy main
```

Alternatively, uncomment lines with `deb-src` using an editor, e.g.:

```
sudo nano /etc/apt/sources.list
```

For other distributions, like Debian, change the URL and names to correspond with the specific distribution.

Then you should update the packages index:

```
$ sudo apt-get update
```

Now you can install the build dependencies via `apt`:

```
$ sudo apt-get build-dep python3
$ sudo apt-get install pkg-config
```

If you want to build all optional modules, install the following packages and their dependencies:

```
$ sudo apt-get install build-essential gdb lcov pkg-config \
      libbz2-dev libffi-dev libgdbm-dev libgdbm-compat-dev liblzma-dev \
      libncurses5-dev libreadline6-dev libsqlite3-dev libssl-dev \
      lzma lzma-dev tk-dev uuid-dev zlib1g-dev
```

macOS and OS X

For **macOS systems** (versions 10.12+) and **OS X 10.9 and later**, the Developer Tools can be downloaded and installed automatically; you do not need to download the complete Xcode application.

If necessary, run the following:

```
$ xcode-select --install
```

This will also ensure that the system header files are installed into `/usr/include`.

On **Mac OS X systems** (versions 10.0 - 10.7) and **OS X 10.8**, use the C compiler and other development utilities provided by Apple's Xcode Developer Tools. The Developer Tools are not shipped with Mac OS X.

For these **older releases (versions 10.0 - 10.8)**, you will need to download either the correct version of the Command Line Tools, if available, or install them from the full Xcode app or package for that OS X release. Older versions may be available either as a no-cost download through Apple's App Store or from the Apple Developer web site.

Also note that OS X does not include several libraries used by the Python standard library, including `libzma`, so expect to see some extension module build failures unless you install local copies of them. As of OS X 10.11, Apple no longer provides header files for the deprecated system version of OpenSSL which means that you will not be able to build the `_ssl` extension. One solution is to install these libraries from a third-party package manager, like Homebrew or MacPorts, and then add the appropriate paths for the header and library files to your `configure` command. For example,

with **Homebrew**:

```
$ brew install pkg-config openssl xz gdbm tcl-tk
```

For Python 3.10 and newer:

```
$ PKG_CONFIG_PATH="$(brew --prefix tcl-tk)/lib/pkgconfig" \
  ./configure --with-pydebug --with-openssl=$(brew --prefix openssl)
```

For Python versions 3.9 through 3.7:

```
$ export PKG_CONFIG_PATH="$(brew --prefix tcl-tk)/lib/pkgconfig"
$ ./configure --with-pydebug \
              --with-openssl=$(brew --prefix openssl) \
              --with-tcltk-libs="$(pkg-config --libs tcl tk)" \
              --with-tcltk-includes="$(pkg-config --cflags tcl tk)"
```

and `make`:

```
$ make -s -j2
```

or **MacPorts**:

```
$ sudo port install pkgconfig openssl xz gdbm
```

and `configure`:

```
$ CPPFLAGS="-I/opt/local/include" \
  LDFLAGS="-L/opt/local/lib" \
  ./configure --with-pydebug
```

and `make`:

```
$ make -s -j2
```

There will sometimes be optional modules added for a new release which won't yet be identified in the OS level build dependencies. In those cases, just ask for assistance on the core-mentorship list.

Explaining how to build optional dependencies on a UNIX based system without root access is beyond the scope of this guide.

For more details on various options and considerations for building, refer to the macOS README.

Note: While you need a C compiler to build CPython, you don't need any knowledge of the C language to contribute! Vast areas of CPython are written completely in Python: as of this writing, CPython contains slightly more Python code than C.

10.1.5 Regenerate `configure`

If a change is made to Python which relies on some POSIX system-specific functionality (such as using a new system call), it is necessary to update the `configure` script to test for availability of the functionality.

Python's `configure` script is generated from `configure.ac` using Autoconf. Instead of editing `configure`, edit `configure.ac` and then run `autoreconf` to regenerate `configure` and a number of other files (such as `pyconfig.h`).

When submitting a patch with changes made to `configure.ac`, you should also include the generated files.

Note that running `autoreconf` is not the same as running `autoconf`. For example, `autoconf` by itself will not regenerate `pyconfig.h.in`. `autoreconf` runs `autoconf` and a number of other tools repeatedly as is appropriate.

Python's `configure.ac` script typically requires a specific version of Autoconf. At the moment, this reads: `AC_PREREQ(2.69)`. It also requires to have the `autoconf-archive` and `pkg-config` utilities installed in the system and the `pkg.m4` macro file located in the appropriate `alocal` location. You can easily check if this is correctly configured by running:

```
ls $(aclocal --print-ac-dir) | grep pkg.m4
```

If the system copy of Autoconf does not match this version, you will need to install your own copy of Autoconf.

10.1.6 Troubleshoot the build

This section lists some of the common problems that may arise during the compilation of Python, with proposed solutions.

Avoid recreating auto-generated files

Under some circumstances you may encounter Python errors in scripts like `Parser/asdl_c.py` or `Python/makeopcodetargets.py` while running `make`. Python auto-generates some of its own code, and a full build from scratch needs to run the auto-generation scripts. However, this makes the Python build require an already installed Python interpreter; this can also cause version mismatches when trying to build an old (2.x) Python with a new (3.x) Python installed, or vice versa.

To overcome this problem, auto-generated files are also checked into the Git repository. So if you don't touch the auto-generation scripts, there's no real need to auto-generate anything.

10.1.7 Editors and Tools

Python is used widely enough that practically all code editors have some form of support for writing Python code. Various coding tools also include Python support.

For editors and tools which the core developers have felt some special comment is needed for coding *in* Python, see *Additional Resources*.

10.1.8 Directory structure

There are several top-level directories in the CPython source tree. Knowing what each one is meant to hold will help you find where a certain piece of functionality is implemented. Do realize, though, there are always exceptions to every rule.

`Doc`
: The official documentation. This is what https://docs.python.org/ uses. See also *Building the documentation*.

`Grammar`
: Contains the EBNF (Extended Backus-Naur Form) grammar file for Python.

`Include`
: Contains all interpreter-wide header files.

`Lib`
: The part of the standard library implemented in pure Python.

`Mac`
: Mac-specific code (e.g., using IDLE as an OS X application).

`Misc`
: Things that do not belong elsewhere. Typically this is varying kinds of developer-specific documentation.

`Modules`
: The part of the standard library (plus some other code) that is implemented in C.

`Objects`
: Code for all built-in types.

`PC`
: Windows-specific code.

`PCbuild`
: Build files for the version of MSVC currently used for the Windows installers provided on python.org.

`Parser`
: Code related to the parser. The definition of the AST nodes is also kept here.

`Programs`
: Source code for C executables, including the main function for the CPython interpreter (in versions prior to Python 3.5, these files are in the Modules directory).

`Python`
: The code that makes up the core CPython runtime. This includes the compiler, eval loop and various built-in modules.

`Tools`
: Various tools that are (or have been) used to maintain Python.

10.2 Where to Get Help

If you are working on Python it is very possible you will come across an issue where you need some assistance to solve it (this happens to core developers all the time).

Should you require help, there are a *variety of options available* to seek assistance. If the question involves process or tool usage then please check the rest of this guide first as it should answer your question.

10.2.1 Discourse

Python has a hosted Discourse instance. This forum has many different categories and most core development discussions take place in the open forum categories for PEPs and Core Development . Most categories are open for all users to read and post with the exception of Committers and Core Development categories. Be sure to visit the related Core categories, such as Core Development and Core Workflow.

See also:

Discourse on how to get started.

10.2.2 Mailing Lists

Further options for seeking assistance include the python-ideas and python-dev mailing lists. Python-ideas contains discussion of speculative Python language ideas for possible inclusion into the language. If an idea gains traction it can then be discussed and honed to the point of becoming a solid proposal and presented on python-dev. Python-dev contains discussion of current Python design issues, release mechanics, and maintenance of existing releases. These mailing lists are for questions involving the development *of* Python, **not** for development *with* Python.

10.2.3 Ask #python-dev

If you are comfortable with IRC you can try asking on `#python-dev` (on the Libera.Chat network). Typically there are a number of experienced developers, ranging from triagers to core developers, who can answer questions about developing for Python. As with the mailing lists, `#python-dev` is for questions involving the development *of* Python whereas `#python` is for questions concerning development *with* Python.

Note: You may not be able to access the history of this channel, so it cannot be used as a "knowledge base" of sorts.

10.2.4 Zulip

An alternative to IRC is our own Zulip instance. There are different streams for asking help with core development, as well as core developers' office hour stream. It is preferred that you ask questions here first or schedule an office hour, before posting to python-dev mailing list or filing bugs.

Warning: This is no longer actively monitored by core devs. Consider asking your questions on Discourse or on the python-dev mailing list.

10.2.5 Core Mentorship

If you are interested in improving Python and contributing to its development, but don't yet feel entirely comfortable with the public channels mentioned above, Python Mentors are here to help you. Python is fortunate to have a community of volunteer core developers willing to mentor anyone wishing to contribute code, work on bug fixes or improve documentation. Everyone is welcomed and encouraged to contribute.

10.2.6 Core Developers Office Hours

Several core developers have set aside time to host mentorship office hours. During the office hour, core developers are available to help contributors with our process, answer questions, and help lower the barrier of contributing and becoming Python core developers.

The PSF's code of conduct applies for interactions with core developers during office hours.

Core Developer	Schedule	Details
Zachary Ware	See details link	Schedule at https://calendly.com/zware
Mariatta Wijaya	Thursdays 7PM - 8PM Pacific (Vancouver, Canada Timezone)	In Python's Zulip Chat, Core > Office Hour stream. A reminder will be posted to both Zulip and Mariatta's twitter account 24 hours before the start.

10.2.7 File a Bug

If you strongly suspect you have stumbled on a bug (be it in the build process, in the test suite, or in other areas), then open an issue on the issue tracker. As with every bug report it is strongly advised that you detail which conditions triggered it (including the OS name and version, and what you were trying to do), as well as the exact error message you encountered.

10.3 Lifecycle of a Pull Request

10.3.1 Introduction

CPython uses a workflow based on pull requests. What this means is that you create a branch in Git, make your changes, push those changes to your fork on GitHub (`origin`), and then create a pull request against the official CPython repository (`upstream`).

10.3.2 Quick Guide

Clear communication is key to contributing to any project, especially an Open Source project like CPython.

Here is a quick overview of how you can contribute to CPython:

1. Create an issue that describes your change*[0]
2. *Create a new branch in Git*
3. Work on changes (e.g. fix a bug or add a new feature)

[0] If an issue is trivial (e.g. typo fixes), or if an issue already exists, you can skip this step.

4. *Run tests* and `make patchcheck`
5. *Commit* and *push* changes to your GitHub fork
6. Create Pull Request on GitHub to merge a branch from your fork
7. Make sure the continuous integration checks on your Pull Request are green (i.e. successful)
8. Review and address comments on your Pull Request
9. When your changes are merged, you can *delete the PR branch*
10. Celebrate contributing to CPython! :)

Note: In order to keep the commit history intact, please avoid squashing or amending history and then force-pushing to the PR. Reviewers often want to look at individual commits.

10.3.3 Step-by-step Guide

You should have already *set up your system*, *got the source code*, and *built Python*.

- Update data from your `upstream` repository:

  ```
  git fetch upstream
  ```

- Create a new branch in your local clone:

  ```
  git checkout -b <branch-name> upstream/main
  ```

- Make changes to the code, and use `git status` and `git diff` to see them.

  (Learn more about *Making Good PRs*)

- Make sure the changes are fine and don't cause any test failure:

  ```
  make patchcheck
  ./python -m test
  ```

  (Learn more about *patchcheck* and about *Running & Writing Tests*)

- Once you are satisfied with the changes, add the files and commit them:

  ```
  git add <filenames>
  git commit -m '<message>'
  ```

  (Learn more about *Making Good Commits*)

- Then push your work to your GitHub fork:

  ```
  git push origin <branch-name>
  ```

- Finally go on `https://github.com/<your-username>/cpython`: you will see a box with the branch you just pushed and a green button that allows you to create a pull request against the official CPython repository.

- When people start adding review comments, you can address them by switching to your branch, making more changes, committing them, and pushing them to automatically update your PR:

```
git checkout <branch-name>
# make changes and run tests
git add <filenames>
git commit -m '<message>'
git push origin <branch-name>
```

  - If a core developer reviewing your PR pushed one or more commits to your PR branch, then after checking out your branch and before editing, run:

    ```
    git pull origin <branch-name>  # pull = fetch + merge
    ```

    If you have made local changes that have not been pushed to your fork and there are merge conflicts, git will warn you about this and enter conflict resolution mode. See *Resolving Merge Conflicts* below.

- If time passes and there are merge conflicts with the main branch, GitHub will show a warning to this end and you may be asked to address this. Merge the changes from the main branch while resolving the conflicts locally:

```
git checkout <branch-name>
git pull upstream main  # pull = fetch + merge
# resolve conflicts: see "Resolving Merge Conflicts" below
git push origin <branch-name>
```

- After your PR has been accepted and merged, you can *delete the branch*:

```
git branch -D <branch-name>  # delete local branch
git push origin -d <branch-name>  # delete remote branch
```

Resolving Merge Conflicts

When merging changes from different branches (or variants of a branch on different repos), the two branches may contain incompatible changes to one or more files. These are called "merge conflicts" and need to be manually resolved as follows:

1. Check which files have merge conflicts:

```
git status
```

2. Edit the affected files and bring them to their intended final state. Make sure to remove the special "conflict markers" inserted by git.

3. Commit the affected files:

```
git add <filenames>
git merge --continue
```

When running the final command, git may open an editor for writing a commit message. It is usually okay to leave that as-is and close the editor.

See the merge command's documentation for a detailed technical explanation.

10.3.4 Making Good PRs

When creating a pull request for submission, there are several things that you should do to help ensure that your pull request is accepted.

First, make sure to follow Python's style guidelines. For Python code you should follow **PEP 8**, and for C code you should follow **PEP 7**. If you have one or two discrepancies those can be fixed by the core developer who merges your pull request. But if you have systematic deviations from the style guides your pull request will be put on hold until you fix the formatting issues.

Note: Pull requests with only code formatting changes are usually rejected. On the other hand, fixes for typos and grammar errors in documents and docstrings are welcome.

Second, be aware of backwards-compatibility considerations. While the core developer who eventually handles your pull request will make the final call on whether something is acceptable, thinking about backwards-compatibility early will help prevent having your pull request rejected on these grounds. Put yourself in the shoes of someone whose code will be broken by the change(s) introduced by the pull request. It is quite likely that any change made will break someone's code, so you need to have a good reason to make a change as you will be forcing someone to update their code. (This obviously does not apply to new classes or functions; new arguments should be optional and have default values which maintain the existing behavior.) If in doubt, have a look at **PEP 387** or *discuss* the issue with experienced developers.

Third, make sure you have proper tests to verify your pull request works as expected. Pull requests will not be accepted without the proper tests!

Fourth, make sure the entire test suite *runs* **without failure** because of your changes. It is not sufficient to only run whichever test seems impacted by your changes, because there might be interferences unknown to you between your changes and some other part of the interpreter.

Fifth, proper *documentation* additions/changes should be included.

10.3.5 `patchcheck`

`patchcheck` is a simple automated patch checklist that guides a developer through the common patch generation checks. To run `patchcheck`:

On *UNIX* (including Mac OS X):

```
make patchcheck
```

On *Windows* (after any successful build):

```
python.bat Tools\scripts\patchcheck.py
```

The automated patch checklist runs through:

- Are there any whitespace problems in Python files? (using `Tools/scripts/reindent.py`)
- Are there any whitespace problems in C files?
- Are there any whitespace problems in the documentation? (using `Tools/scripts/reindent-rst.py`)
- Has the documentation been updated?
- Has the test suite been updated?
- Has an entry under `Misc/NEWS.d/next` been added? (using blurb-it, or the blurb tool)

- Has `Misc/ACKS` been updated?
- Has `configure` been regenerated, if necessary?
- Has `pyconfig.h.in` been regenerated, if necessary?

The automated patch check doesn't actually *answer* all of these questions. Aside from the whitespace checks, the tool is a memory aid for the various elements that can go into making a complete patch.

10.3.6 Making Good Commits

Each feature or bugfix should be addressed by a single pull request, and for each pull request there may be several commits. In particular:

- Do **not** fix more than one issue in the same commit (except, of course, if one code change fixes all of them).
- Do **not** do cosmetic changes to unrelated code in the same commit as some feature/bugfix.

Commit messages should follow the following structure:

```
Make the spam module more spammy

The spam module sporadically came up short on spam. This change
raises the amount of spam in the module by making it more spammy.
```

The first line or sentence is meant to be a dense, to-the-point explanation of what the purpose of the commit is. The imperative form (used in the example above) is strongly preferred to a descriptive form such as 'the spam module is now more spammy'. Use `git log --oneline` to see existing title lines. Furthermore, the first line should not end in a period.

If this is not enough detail for a commit, a new paragraph(s) can be added to explain in proper depth what has happened (detail should be good enough that a core developer reading the commit message understands the justification for the change).

Check *the git bootcamp* for further instructions on how the commit message should look like when merging a pull request.

Note: How to Write a Git Commit Message is a nice article that describes how to write a good commit message.

10.3.7 Licensing

To accept your change we must have your formal approval for distributing your work under the PSF license. Therefore, you need to sign a contributor agreement which allows the Python Software Foundation to license your code for use with Python (you retain the copyright).

Note: You only have to sign this document once, it will then apply to all your further contributions to Python.

Here are the steps needed in order to sign the CLA:

1. Create a change and submit it as a pull request.
2. When `cpython-cla-bot` comments on your pull request that commit authors are required to sign a Contributor License Agreement, click on the button in the comment to sign it. It's enough to log in through GitHub. The process is automatic.

3. After signing, the comment by `cpython-cla-bot` will update to indicate that "all commit authors signed the Contributor License Agreement.

10.3.8 Submitting

Once you are satisfied with your work you will want to commit your changes to your branch. In general you can run `git commit -a` and that will commit everything. You can always run `git status` to see what changes are outstanding.

When all of your changes are committed (i.e. `git status` doesn't list anything), you will want to push your branch to your fork:

```
git push origin <branch name>
```

This will get your changes up to GitHub.

Now you want to create a pull request from your fork. If this is a pull request in response to a pre-existing issue on the issue tracker, please make sure to reference the issue number using `gh-NNNNN:` prefix in the pull request title and `#NNNNN` in the description.

If this is a pull request for an unreported issue (assuming you already performed a search on the issue tracker for a pre-existing issue), create a new issue and reference it in the pull request. Please fill in as much relevant detail as possible to prevent reviewers from having to delay reviewing your pull request because of lack of information.

If this issue is so simple that there's no need for an issue to track any discussion of what the pull request is trying to solve (e.g. fixing a spelling mistake), then the pull request needs to have the "skip issue" label added to it by someone with commit access.

Your pull request may involve several commits as a result of addressing code review comments. Please keep the commit history in the pull request intact by not squashing, amending, or anything that would require a force push to GitHub. A detailed commit history allows reviewers to view the diff of one commit to another so they can easily verify whether their comments have been addressed. The commits will be squashed when the pull request is merged.

10.3.9 Converting an Existing Patch from b.p.o to GitHub

When a patch exists in the issue tracker that should be converted into a GitHub pull request, please first ask the original patch author to prepare their own pull request. If the author does not respond after a week, it is acceptable for another contributor to prepare the pull request based on the existing patch. In this case, both parties should sign the *CLA*. When creating a pull request based on another person's patch, provide attribution to the original patch author by adding "Co-authored-by: Author Name <email_address> ." to the pull request description and commit message. See the GitHub article on how to properly add the co-author info.

See also *Applying a Patch to Git*.

10.3.10 Reviewing

To begin with, please be patient! There are many more people submitting pull requests than there are people capable of reviewing your pull request. Getting your pull request reviewed requires a reviewer to have the spare time and motivation to look at your pull request (we cannot force anyone to review pull requests and no one is employed to look at pull requests). If your pull request has not received any notice from reviewers (i.e., no comment made) after one month, first "ping" the issue on the issue tracker to remind the nosy list that the pull request needs a review. If you don't get a response within a week after pinging the issue, then you can try emailing python-dev@python.org to ask for someone to review your pull request.

When someone does manage to find the time to look at your pull request they will most likely make comments about how it can be improved (don't worry, even core developers of Python have their pull requests sent back to them for changes). It is then expected that you update your pull request to address these comments, and the review process will thus iterate until a satisfactory solution has emerged.

How to Review a Pull Request

One of the bottlenecks in the Python development process is the lack of code reviews. If you browse the bug tracker, you will see that numerous issues have a fix, but cannot be merged into the main source code repository, because no one has reviewed the proposed solution. Reviewing a pull request can be just as informative as providing a pull request and it will allow you to give constructive comments on another developer's work. This guide provides a checklist for submitting a code review. It is a common misconception that in order to be useful, a code review has to be perfect. This is not the case at all! It is helpful to just test the pull request and/or play around with the code and leave comments in the pull request or issue tracker.

1. If you have not already done so, get a copy of the CPython repository by following the *setup guide*, build it and run the tests.
2. Check the bug tracker to see what steps are necessary to reproduce the issue and confirm that you can reproduce the issue in your version of the Python REPL (the interactive shell prompt), which you can launch by executing ./python inside the repository.
3. Checkout and apply the pull request (Please refer to the instruction *Downloading Other's Patches*)
4. If the changes affect any C file, run the build again.
5. Launch the Python REPL (the interactive shell prompt) and check if you can reproduce the issue. Now that the pull request has been applied, the issue should be fixed (in theory, but mistakes do happen! A good review aims to catch these before the code is merged into the Python repository). You should also try to see if there are any corner cases in this or related issues that the author of the fix may have missed.
6. If you have time, run the entire test suite. If you are pressed for time, run the tests for the module(s) where changes were applied. However, please be aware that if you are recommending a pull request as 'merge-ready', you should always make sure the entire test suite passes.

10.3.11 Leaving a Pull Request Review on GitHub

When you review a pull request, you should provide additional details and context of your review process.

Instead of simply "approving" the pull request, leave comments. For example:

1. If you tested the PR, report the result and the system and version tested on, such as 'Windows 10', 'Ubuntu 16.4', or 'Mac High Sierra'.
2. If you request changes, try to suggest how.
3. Comment on what is "good" about the pull request, not just the "bad". Doing so will make it easier for the PR author to find the good in your comments.

10.3.12 Dismissing Review from Another Core Developer

A core developer can dismiss another core developer's review if they confirmed that the requested changes have been made. When a core developer has assigned the PR to themselves, then it is a sign that they are actively looking after the PR, and their review should not be dismissed.

10.3.13 Committing/Rejecting

Once your pull request has reached an acceptable state (and thus considered "accepted"), it will either be merged or rejected. If it is rejected, please do not take it personally! Your work is still appreciated regardless of whether your pull request is merged. Balancing what *does* and *does not* go into Python is tricky and we simply cannot accept everyone's contributions.

But if your pull request is merged it will then go into Python's VCS (version control system) to be released with the next major release of Python. It may also be backported to older versions of Python as a bugfix if the core developer doing the merge believes it is warranted.

10.3.14 Crediting

Non-trivial contributions are credited in the `Misc/ACKS` file (and, most often, in a contribution's news entry as well). You may be asked to make these edits on the behalf of the core developer who accepts your pull request.

10.4 Running & Writing Tests

Note: This document assumes you are working from an *in-development* checkout of Python. If you are not then some things presented here may not work as they may depend on new features not available in earlier versions of Python.

10.4.1 Running

The shortest, simplest way of running the test suite is the following command from the root directory of your checkout (after you have *built Python*):

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

You may need to change this command as follows throughout this section. On *most* Mac OS X systems, replace `./python` with `./python.exe`. On Windows, use `python.bat`. If using Python 2.7, replace `test` with `test.regrtest`.

This will run the majority of tests, but exclude a small portion of them; these excluded tests use special kinds of resources: for example, accessing the Internet, or trying to play a sound or to display a graphical interface on your desktop. They are disabled by default so that running the test suite is not too intrusive. To enable some of these additional tests (and for other flags which can help debug various issues such as reference leaks), read the help text:

```
./python -m test -h
```

If you want to run a single test file, simply specify the test file name (without the extension) as an argument. You also probably want to enable verbose mode (using `-v`), so that individual failures are detailed:

```
./python -m test -v test_abc
```

To run a single test case, use the `unittest` module, providing the import path to the test case:

```
./python -m unittest -v test.test_abc.TestABC_Py
```

Some test modules also support direct invocation, which might be useful for IDEs and local debugging:

```
./python Lib/test/test_typing.py
```

But, there are several important notes:

1. This way of running tests exists only for local developer needs and is discouraged for anything else
2. Some modules do not support it at all. One example is``test_importlib``. In other words: if some module does not have `unittest.main()`, then most likely it does not support direct invocation.

If you have a multi-core or multi-CPU machine, you can enable parallel testing using several Python processes so as to speed up things:

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

If you are running a version of Python prior to 3.3 you must specify the number of processes to run simultaneously (e.g. `-j2`).

Finally, if you want to run tests under a more strenuous set of settings, you can run `test` as:

```
./python -bb -E -Wd -m test -r -w -uall
```

The various extra flags passed to Python cause it to be much stricter about various things (the `-Wd` flag should be `-W error` at some point, but the test suite has not reached a point where all warnings have been dealt with and so we cannot guarantee that a bug-free Python will properly complete a test run with `-W error`). The `-r` flag to the test runner causes it to run tests in a more random order which helps to check that the various tests do not interfere with each other. The `-w` flag causes failing tests to be run again to see if the failures are transient or consistent. The `-uall` flag allows the use of all available resources so as to not skip tests requiring, e.g., Internet access.

To check for reference leaks (only needed if you modified C code), use the `-R` flag. For example, `-R 3:2` will first run the test 3 times to settle down the reference count, and then run it 2 more times to verify if there are any leaks.

You can also execute the `Tools/scripts/run_tests.py` script as found in a CPython checkout. The script tries to balance speed with thoroughness. But if you want the most thorough tests you should use the strenuous approach shown above.

Unexpected Skips

Sometimes when running the test suite, you will see "unexpected skips" reported. These represent cases where an entire test module has been skipped, but the test suite normally expects the tests in that module to be executed on that platform.

Often, the cause is that an optional module hasn't been built due to missing build dependencies. In these cases, the missing module reported when the test is skipped should match one of the modules reported as failing to build when *Compile and build*.

In other cases, the skip message should provide enough detail to help figure out and resolve the cause of the problem (for example, the default security settings on some platforms will disallow some tests)

10.4.2 Writing

Writing tests for Python is much like writing tests for your own code. Tests need to be thorough, fast, isolated, consistently repeatable, and as simple as possible. We try to have tests both for normal behaviour and for error conditions. Tests live in the `Lib/test` directory, where every file that includes tests has a `test_` prefix.

One difference with ordinary testing is that you are encouraged to rely on the `test.support` module. It contains various helpers that are tailored to Python's test suite and help smooth out common problems such as platform differences, resource consumption and cleanup, or warnings management. That module is not suitable for use outside of the standard library.

When you are adding tests to an existing test file, it is also recommended that you study the other tests in that file; it will teach you which precautions you have to take to make your tests robust and portable.

10.4.3 Benchmarks

Benchmarking is useful to test that a change does not degrade performance.

The Python Benchmark Suite has a collection of benchmarks for all Python implementations. Documentation about running the benchmarks is in the README.txt of the repo.

10.5 Increase Test Coverage

Python development follows a practice that all semantic changes and additions to the language and STDLIB (standard library) are accompanied by appropriate unit tests. Unfortunately Python was in existence for a long time before the practice came into effect. This has left chunks of the stdlib untested which is not a desirable situation to be in.

A good, easy way to become acquainted with Python's code and to help out is to help increase the test coverage for Python's stdlib. Ideally we would like to have 100% coverage, but any increase is a good one. Do realize, though, that getting 100% coverage is not always possible. There could be platform-specific code that simply will not execute for you, errors in the output, etc. You can use your judgement as to what should and should not be covered, but being conservative and assuming something should be covered is generally a good rule to follow.

Choosing what module you want to increase test coverage for can be done in a couple of ways. You can simply run the entire test suite yourself with coverage turned on and see what modules need help. This has the drawback of running the entire test suite under coverage measuring which takes some time to complete, but you will have an accurate, up-to-date notion of what modules need the most work.

Another is to follow the examples below and simply see what coverage your favorite module has. This is "stabbing in the dark", though, and so it might take some time to find a module that needs coverage help.

Do make sure, though, that for any module you do decide to work on that you run coverage for just that module. This will make sure you know how good the explicit coverage of the module is from its own set of tests instead of from implicit testing by other code that happens to use the module.

10.5.1 Common Gotchas

Please realize that coverage reports on modules already imported before coverage data starts to be recorded will be wrong. Typically you can tell a module falls into this category by the coverage report saying that global statements that would obviously be executed upon import have gone unexecuted while local statements have been covered. In these instances you can ignore the global statement coverage and simply focus on the local statement coverage.

When writing new tests to increase coverage, do take note of the style of tests already provided for a module (e.g., whitebox, blackbox, etc.). As some modules are primarily maintained by a single core developer they may have a specific preference as to what kind of test is used (e.g., whitebox) and prefer that other types of tests not be used (e.g., blackbox). When in doubt, stick with whitebox testing in order to properly exercise the code.

10.5.2 Measuring Coverage

It should be noted that a quirk of running coverage over Python’s own stdlib is that certain modules are imported as part of interpreter startup. Those modules required by Python itself will not be viewed as executed by the coverage tools and thus look like they have very poor coverage (e.g., the `stat` module). In these instances the module will appear to not have any coverage of global statements but will have proper coverage of local statements (e.g., function definitions will not be traced, but the function bodies will). Calculating the coverage of modules in this situation will simply require manually looking at what local statements were not executed.

Using coverage.py

One of the most popular third-party coverage tools is coverage.py which provides very nice HTML output along with advanced features such as *branch coverage*. If you prefer to stay with tools only provided by the stdlib then you can *use test.regrtest*.

Install Coverage

By default, pip will not install into the in-development version of Python you just built, and this built version of Python will not see packages installed into your default version of Python. One option is to use a virtual environment to install coverage.

On Unix run:

```
./python -m venv ../cpython-venv
source ../cpython-venv/bin/activate
pip install coverage
```

On *most* Mac OS X systems run:

```
./python.exe -m venv ../cpython-venv
source ../cpython-venv/bin/activate
pip install coverage
```

On Windows run:

```
python.bat -m venv ..\\cpython-venv
..\\cpython-venv\\Scripts\\activate.bat
pip install coverage
```

You can now use python without the ./ for the rest of these instructions, as long as your venv is activated. For more info on venv see Virtual Environment documentation.

If this does not work for you for some reason, you should try using the in-development version of coverage.py to see if it has been updated as needed. To do this you should clone/check out the development version of coverage.py:

git clone https://github.com/nedbat/coveragepy

You will need to use the full path to the installation.

Another option is to use an installed copy of coverage.py, if you already have it. For this, you will again need to use the full path to that installation.

Basic Usage

The following command will tell you if your copy of coverage works (substitute `COVERAGEDIR` with the directory where your clone exists, e.g. `../coveragepy`):

```
./python COVERAGEDIR
```

Coverage.py will print out a little bit of helper text verifying that everything is working. If you are using an installed copy, you can do the following instead (note this must be installed using the built copy of Python, such as by venv):

```
./python -m coverage
```

The rest of the examples on how to use coverage.py will assume you are using a cloned copy, but you can substitute the above and all instructions should still be valid.

To run the test suite under coverage.py, do the following:

```
./python COVERAGEDIR run --pylib Lib/test/regrtest.py
```

To run only a single test, specify the module/package being tested in the `--source` flag (so as to prune the coverage reporting to only the module/package you are interested in) and then append the name of the test you wish to run to the command:

```
./python COVERAGEDIR run --pylib --source=abc Lib/test/regrtest.py test_abc
```

To see the results of the coverage run, you can view a text-based report with:

```
./python COVERAGEDIR report
```

You can use the `--show-missing` flag to get a list of lines that were not executed:

```
./python COVERAGEDIR report --show-missing
```

But one of the strengths of coverage.py is its HTML-based reports which let you visually see what lines of code were not tested:

```
./python COVERAGEDIR html -i --include=`pwd`/Lib/* --omit="Lib/test/*,Lib/*/tests/*"
```

This will generate an HTML report in a directory named `htmlcov` which ignores any errors that may arise and ignores modules for which test coverage is unimportant (e.g. tests, temp files, etc.). You can then open the `htmlcov/index.html` file in a web browser to view the coverage results along with pages that visibly show what lines of code were or were not executed.

Branch Coverage

For the truly daring, you can use another powerful feature of coverage.py: branch coverage. Testing every possible branch path through code, while a great goal to strive for, is a secondary goal to getting 100% line coverage for the entire stdlib (for now).

If you decide you want to try to improve branch coverage, simply add the `--branch` flag to your coverage run:

```
./python COVERAGEDIR run --pylib --branch <arguments to run test(s)>
```

This will lead to the report stating not only what lines were not covered, but also what branch paths were not executed.

Coverage Results For Modules Imported Early On

For the *truly truly* daring, you can use a hack to get coverage.py to include coverage for modules that are imported early on during CPython's startup (e.g. the encodings module). Do not worry if you can't get this to work or it doesn't make any sense; it's entirely optional and only important for a small number of modules.

If you still choose to try this, the first step is to make sure coverage.py's C extension is installed. You can check this with:

```
./python COVERAGEDIR --version
```

If it says 'without C extension', then you will need to build the C extension. Assuming that coverage.py's clone is at `COVERAGEDIR` and your clone of CPython is at `CPYTHONDIR`, you can do this by executing the following in your coverage.py clone:

```
CPPFLAGS="-I CPYTHONDIR -I CPYTHONDIR/Include" CPYTHONDIR/python setup.py build_ext --
↪inplace
```

This will build coverage.py's C extension code in-place, allowing the previous instructions on how to gather coverage to continue to work.

To get coverage.py to be able to gather the most accurate coverage data on as many modules as possible **with a HORRIBLE HACK that you should NEVER use in your own code**, run the following from your CPython clone:

```
PYTHONPATH=COVERAGEDIR/coverage/fullcoverage ./python COVERAGEDIR run --pylib Lib/test/
↪regrtest.py
```

This will give you the most complete coverage possible for CPython's standard library.

Using test.regrtest

If you prefer to rely solely on the stdlib to generate coverage data, you can do so by passing the appropriate flags to `test` (along with any other flags you want to):

```
./python -m test --coverage -D `pwd`/coverage_data <test arguments>
```

Do note the argument to `-D`; if you do not specify an absolute path to where you want the coverage data to end up it will go somewhere you don't expect.

Note: If you are running coverage over the entire test suite, make sure to add `-x test_importlib test_runpy test_trace` to exclude those tests as they trigger exceptions during coverage; see https://bugs.python.org/issue10541 and https://bugs.python.org/issue10991.

Once the tests are done you will find the directory you specified contains files for each executed module along with which lines were executed how many times.

10.5.3 Filing the Issue

Once you have increased coverage, you need to create an issue on the issue tracker and submit a *pull request*. On the issue set the "Components" to "Test" and "Versions" to the version of Python you worked on (i.e., the in-development version).

10.5.4 Measuring coverage of C code with gcov and lcov

It's also possible to measure the function, line and branch coverage of Python's C code. Right now only GCC with gcov is supported. In order to create an instrumented build of Python with gcov, run:

```
make coverage
```

Then run some code and gather coverage data with the `gcov` command. In order to create a HTML report you can install lcov. The command:

```
make coverage-lcov
```

assembles coverage data, removes 3rd party and system libraries and finally creates a report. You can skip both steps and just run:

```
make coverage-report
```

if you like to generate a coverage report for Python's stdlib tests. It takes about 20 to 30 minutes on a modern computer.

Note: Multiple test jobs may not work properly. C coverage reporting has only been tested with a single test process.

10.6 Helping with Documentation

Python is known for having well-written documentation. Maintaining the documentation's accuracy and keeping a high level of quality takes a lot of effort. Community members, like you, help with writing, editing, and updating content, and these contributions are appreciated and welcomed.

This high-level **Helping with Documentation** section provides:

- an overview of Python's documentation
- how to help with documentation issues
- information on proofreading
- guidance on contributing to this Developer's Guide

The next chapter, *Documenting Python*, gives extensive, detailed information on how to write documentation and submit changes.

10.6.1 Python Documentation

The *Documenting Python* section covers the details of how Python's documentation works. It includes information about the markup language used, specific formats, and style recommendations. Looking at pre-existing documentation source files can be very helpful when getting started. *How to build the documentation* walks you through the steps to create a draft build which lets you see how your changes will look and validates that your new markup is correct.

You can view the documentation built from *in-development* and *maintenance* branches at https://docs.python.org/dev/. The in-development and recent maintenance branches are rebuilt once per day.

If you would like to be more involved with documentation, consider subscribing to the docs@python.org mailing list. The issue tracker sends new documentation issues to this mailing list, and, less frequently, the list receives some directly mailed bug reports. The docs-sig@python.org mailing list discusses the documentation toolchain, projects, and standards.

10.6.2 Helping with documentation issues

If you look at documentation issues on the issue tracker, you will find various documentation problems that may need work. Issues vary from typos to unclear documentation and items lacking documentation.

If you see a documentation issue that you would like to tackle, you can:

- check to see if there is a paperclip or octocat icon at the end of the issue's title column. If there is, then someone has already created a pull request for the issue.
- leave a comment on the issue saying you are going to try and create a pull request and roughly how long you think you will take to do so (this allows others to take on the issue if you happen to forget or lose interest).
- submit a *pull request* for the issue.

By following the steps in the *Quick Guide to Pull Requests*, you will learn the workflow for documentation pull requests.

10.6.3 Proofreading

While an issue filed on the issue tracker means there is a known issue somewhere, that does not mean there are not other issues lurking about in the documentation. Proofreading a part of the documentation, such as a "How to" or OS specific document, can often uncover problems (e.g., documentation that needs updating for Python 3).

If you decide to proofread, read a section of the documentation from start to finish, filing issues in the issue tracker for each major type of problem you find. Simple typos don't require issues of their own, but, instead, submit a pull request directly. It's best to avoid filing a single issue for an entire section containing multiple problems; instead, file several issues so that it is easier to break the work up for multiple people and more efficient review.

10.6.4 Helping with the Developer's Guide

The Developer's Guide (what you're reading now) uses the same process as the main Python documentation, except for some small differences. The source lives in a separate repository and bug reports should be submitted to the devguide GitHub tracker.

Our devguide workflow uses continuous integration and deployment so changes to the devguide are normally published when the pull request is merged. Changes to CPython documentation follow the workflow of a CPython release and are published in the release.

10.6.5 Developer's Guide workflow

To submit a *pull request*, you can fork the devguide repo to your GitHub account and clone it using:

```
$ git clone https://github.com/<your_username>/devguide
```

For your PR to be accepted, you will also need to sign the *contributor agreement*.

To build the devguide, some additional dependencies are required (most importantly, Sphinx), and the standard way to install dependencies in Python projects is to create a virtualenv, and then install dependencies from a `requirements.txt` file. For your convenience, this is all *automated for you*. To build the devguide on a Unix-like system use:

```
$ make html
```

in the checkout directory. On Windows use:

```
> .\make html
```

You will find the generated files in `_build/html` or, if you use `make htmlview`, the docs will be opened in a browser once the build completes. Note that `make check` runs automatically when you submit a *pull request*. You may wish to run `make check` and `make linkcheck` to make sure that it runs without errors.

10.7 Documenting Python

The Python language has a substantial body of documentation, much of it contributed by various authors. The markup used for the Python documentation is reStructuredText, developed by the docutils project, amended by custom directives and using a toolset named Sphinx to post-process the HTML output.

This document describes the style guide for our documentation as well as the custom reStructuredText markup introduced by Sphinx to support Python documentation and how it should be used.

The documentation in HTML, PDF or EPUB format is generated from text files written using the reStructuredText format and contained in the *CPython Git repository*.

Note: If you're interested in contributing to Python's documentation, there's no need to write reStructuredText if you're not so inclined; plain text contributions are more than welcome as well. Send an e-mail to docs@python.org or open an issue on the tracker.

10.7.1 Introduction

Python's documentation has long been considered to be good for a free programming language. There are a number of reasons for this, the most important being the early commitment of Python's creator, Guido van Rossum, to providing documentation on the language and its libraries, and the continuing involvement of the user community in providing assistance for creating and maintaining documentation.

The involvement of the community takes many forms, from authoring to bug reports to just plain complaining when the documentation could be more complete or easier to use.

This document is aimed at authors and potential authors of documentation for Python. More specifically, it is for people contributing to the standard documentation and developing additional documents using the same tools as the standard documents. This guide will be less useful for authors using the Python documentation tools for topics other than Python, and less useful still for authors not using the tools at all.

If your interest is in contributing to the Python documentation, but you don't have the time or inclination to learn reStructuredText and the markup structures documented here, there's a welcoming place for you among the Python contributors as well. Any time you feel that you can clarify existing documentation or provide documentation that's missing, the existing documentation team will gladly work with you to integrate your text, dealing with the markup for you. Please don't let the material in this document stand between the documentation and your desire to help out!

10.7.2 Style guide

Use of whitespace

All reST files use an indentation of 3 spaces; no tabs are allowed. The maximum line length is 80 characters for normal text, but tables, deeply indented code samples and long links may extend beyond that. Code example bodies should use normal Python 4-space indentation.

Make generous use of blank lines where applicable; they help group things together.

A sentence-ending period may be followed by one or two spaces; while reST ignores the second space, it is customarily put in by some users, for example to aid Emacs' auto-fill mode.

Footnotes

Footnotes are generally discouraged, though they may be used when they are the best way to present specific information. When a footnote reference is added at the end of the sentence, it should follow the sentence-ending punctuation. The reST markup should appear something like this:

```
This sentence has a footnote reference. [#]_ This is the next sentence.
```

Footnotes should be gathered at the end of a file, or if the file is very long, at the end of a section. The docutils will automatically create backlinks to the footnote reference.

Footnotes may appear in the middle of sentences where appropriate.

Capitalization

Sentence case

Sentence case is a set of capitalization rules used in English sentences: the first word is always capitalized and other words are only capitalized if there is a specific rule requiring it.

In the Python documentation, the use of sentence case in section titles is preferable, but consistency within a unit is more important than following this rule. If you add a section to a chapter where most sections are in title case, you can either convert all titles to sentence case or use the dominant style in the new section title.

Sentences that start with a word for which specific rules require starting it with a lower case letter should be avoided.

Note: Sections that describe a library module often have titles in the form of "modulename — Short description of the module." In this case, the description should be capitalized as a stand-alone sentence.

Many special names are used in the Python documentation, including the names of operating systems, programming languages, standards bodies, and the like. Most of these entities are not assigned any special markup, but the preferred spellings are given here to aid authors in maintaining the consistency of presentation in the Python documentation.

Other terms and words deserve special mention as well; these conventions should be used to ensure consistency throughout the documentation:

CPU
For "central processing unit." Many style guides say this should be spelled out on the first use (and if you must use it, do so!). For the Python documentation, this abbreviation should be avoided since there's no reasonable way to predict which occurrence will be the first seen by the reader. It is better to use the word "processor" instead.

POSIX
The name assigned to a particular group of standards. This is always uppercase.

Python
The name of our favorite programming language is always capitalized.

reST
For "reStructuredText," an easy to read, plaintext markup syntax used to produce Python documentation. When spelled out, it is always one word and both forms start with a lower case 'r'.

Unicode
The name of a character coding system. This is always written capitalized.

Unix
The name of the operating system developed at AT&T Bell Labs in the early 1970s.

Affirmative Tone

The documentation focuses on affirmatively stating what the language does and how to use it effectively.

Except for certain security or segfault risks, the docs should avoid wording along the lines of "feature x is dangerous" or "experts only". These kinds of value judgments belong in external blogs and wikis, not in the core documentation.

Bad example (creating worry in the mind of a reader):

> Warning: failing to explicitly close a file could result in lost data or excessive resource consumption. Never rely on reference counting to automatically close a file.

Good example (establishing confident knowledge in the effective use of the language):

> A best practice for using files is use a try/finally pair to explicitly close a file after it is used. Alternatively, using a with-statement can achieve the same effect. This assures that files are flushed and file descriptor resources are released in a timely manner.

Economy of Expression

More documentation is not necessarily better documentation. Err on the side of being succinct.

It is an unfortunate fact that making documentation longer can be an impediment to understanding and can result in even more ways to misread or misinterpret the text. Long descriptions full of corner cases and caveats can create the impression that a function is more complex or harder to use than it actually is.

Security Considerations (and Other Concerns)

Some modules provided with Python are inherently exposed to security issues (e.g. shell injection vulnerabilities) due to the purpose of the module (e.g. `ssl`). Littering the documentation of these modules with red warning boxes for problems that are due to the task at hand, rather than specifically to Python's support for that task, doesn't make for a good reading experience.

Instead, these security concerns should be gathered into a dedicated "Security Considerations" section within the module's documentation, and cross-referenced from the documentation of affected interfaces with a note similar to `"Please refer to the :ref:`security-considerations` section for important information on how to avoid common mistakes."`.

Similarly, if there is a common error that affects many interfaces in a module (e.g. OS level pipe buffers filling up and stalling child processes), these can be documented in a "Common Errors" section and cross-referenced rather than repeated for every affected interface.

Code Examples

Short code examples can be a useful adjunct to understanding. Readers can often grasp a simple example more quickly than they can digest a formal description in prose.

People learn faster with concrete, motivating examples that match the context of a typical use case. For instance, the `str.rpartition()` method is better demonstrated with an example splitting the domain from a URL than it would be with an example of removing the last word from a line of Monty Python dialog.

The ellipsis for the `sys.ps2` secondary interpreter prompt should only be used sparingly, where it is necessary to clearly differentiate between input lines and output lines. Besides contributing visual clutter, it makes it difficult for readers to cut-and-paste examples so they can experiment with variations.

Code Equivalents

Giving pure Python code equivalents (or approximate equivalents) can be a useful adjunct to a prose description. A documenter should carefully weigh whether the code equivalent adds value.

A good example is the code equivalent for `all()`. The short 4-line code equivalent is easily digested; it re-emphasizes the early-out behavior; and it clarifies the handling of the corner-case where the iterable is empty. In addition, it serves as a model for people wanting to implement a commonly requested alternative where `all()` would return the specific object evaluating to False whenever the function terminates early.

A more questionable example is the code for `itertools.groupby()`. Its code equivalent borders on being too complex to be a quick aid to understanding. Despite its complexity, the code equivalent was kept because it serves as a model to alternative implementations and because the operation of the "grouper" is more easily shown in code than in English prose.

An example of when not to use a code equivalent is for the `oct()` function. The exact steps in converting a number to octal doesn't add value for a user trying to learn what the function does.

Audience

The tone of the tutorial (and all the docs) needs to be respectful of the reader's intelligence. Don't presume that the readers are stupid. Lay out the relevant information, show motivating use cases, provide glossary links, and do your best to connect-the-dots, but don't talk down to them or waste their time.

The tutorial is meant for newcomers, many of whom will be using the tutorial to evaluate the language as a whole. The experience needs to be positive and not leave the reader with worries that something bad will happen if they make a misstep. The tutorial serves as guide for intelligent and curious readers, saving details for the how-to guides and other sources.

Be careful accepting requests for documentation changes from the rare but vocal category of reader who is looking for vindication for one of their programming errors ("I made a mistake, therefore the docs must be wrong …"). Typically, the documentation wasn't consulted until after the error was made. It is unfortunate, but typically no documentation edit would have saved the user from making false assumptions about the language ("I was surprised by …").

10.7.3 reStructuredText Primer

This section is a brief introduction to reStructuredText (reST) concepts and syntax, intended to provide authors with enough information to author documents productively. Since reST was designed to be a simple, unobtrusive markup language, this will not take too long.

See also:

The authoritative reStructuredText User Documentation.

Paragraphs

The paragraph is the most basic block in a reST document. Paragraphs are simply chunks of text separated by one or more blank lines. As in Python, indentation is significant in reST, so all lines of the same paragraph must be left-aligned to the same level of indentation.

Inline markup

The standard reST inline markup is quite simple: use

- one asterisk: `*text*` for emphasis (italics),
- two asterisks: `**text**` for strong emphasis (boldface), and
- backquotes: ``` ``text`` ``` for code samples.

If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, they have to be escaped with a backslash.

Be aware of some restrictions of this markup:

- it may not be nested,
- content may not start or end with whitespace: `* text*` is wrong,
- it must be separated from surrounding text by non-word characters. Use a backslash escaped space to work around that: `thisis\ *one*\ word`.

These restrictions may be lifted in future versions of the docutils.

reST also allows for custom "interpreted text roles"', which signify that the enclosed text should be interpreted in a specific way. Sphinx uses this to provide semantic markup and cross-referencing of identifiers, as described in the appropriate section. The general syntax is ``:rolename:`content` ``.

Lists and Quotes

List markup is natural: just place an asterisk at the start of a paragraph and indent properly. The same goes for numbered lists; they can also be automatically numbered using a # sign:

```
* This is a bulleted list.
* It has two items, the second
  item uses two lines.

1. This is a numbered list.
2. It has two items too.

#. This is a numbered list.
#. It has two items too.
```

Nested lists are possible, but be aware that they must be separated from the parent list items by blank lines:

```
* this is
* a list

  * with a nested list
  * and some subitems

* and here the parent list continues
```

Definition lists are created as follows:

```
term (up to a line of text)
   Definition of the term, which must be indented

   and can even consist of multiple paragraphs

next term
   Description.
```

Paragraphs are quoted by just indenting them more than the surrounding paragraphs.

Source Code

Literal code blocks are introduced by ending a paragraph with the special marker `::`. The literal block must be indented:

```
This is a normal text paragraph. The next paragraph is a code sample::

   It is not processed in any way, except
   that the indentation is removed.

   It can span multiple lines.

This is a normal text paragraph again.
```

The handling of the `::` marker is smart:

- If it occurs as a paragraph of its own, that paragraph is completely left out of the document.
- If it is preceded by whitespace, the marker is removed.

- If it is preceded by non-whitespace, the marker is replaced by a single colon.

That way, the second sentence in the above example's first paragraph would be rendered as "The next paragraph is a code sample:".

Hyperlinks

External links

Use `` `Link text <http://target>`_ `` for inline web links. If the link text should be the web address, you don't need special markup at all, the parser finds links and mail addresses in ordinary text.

Internal links

Internal linking is done via a special reST role, see the section on specific markup, *Cross-linking markup*.

Sections

Section headers are created by underlining (and optionally overlining) the section title with a punctuation character, at least as long as the text:

```
=================
This is a heading
=================
```

Normally, there are no heading levels assigned to certain characters as the structure is determined from the succession of headings. However, for the Python documentation, here is a suggested convention:

- `#` with overline, for parts
- `*` with overline, for chapters
- `=`, for sections
- `-`, for subsections
- `^`, for subsubsections
- `"`, for paragraphs

Explicit Markup

"Explicit markup" is used in reST for most constructs that need special handling, such as footnotes, specially-highlighted paragraphs, comments, and generic directives.

An explicit markup block begins with a line starting with `..` followed by whitespace and is terminated by the next paragraph at the same level of indentation. (There needs to be a blank line between explicit markup and normal paragraphs. This may all sound a bit complicated, but it is intuitive enough when you write it.)

Directives

A directive is a generic block of explicit markup. Besides roles, it is one of the extension mechanisms of reST, and Sphinx makes heavy use of it.

Basically, a directive consists of a name, arguments, options and content. (Keep this terminology in mind, it is used in the next chapter describing custom directives.) Looking at this example,

```
.. function:: foo(x)
              foo(y, z)
   :bar: no

   Return a line of text input from the user.
```

`function` is the directive name. It is given two arguments here, the remainder of the first line and the second line, as well as one option `bar` (as you can see, options are given in the lines immediately following the arguments and indicated by the colons).

The directive content follows after a blank line and is indented relative to the directive start.

Footnotes

For footnotes, use `[#]_` to mark the footnote location, and add the footnote body at the bottom of the document after a "Footnotes" rubric heading, like so:

```
Lorem ipsum [#]_ dolor sit amet ... [#]_

.. rubric:: Footnotes

.. [#] Text of the first footnote.
.. [#] Text of the second footnote.
```

You can also explicitly number the footnotes for better context.

Comments

Every explicit markup block which isn't a valid markup construct (like the footnotes above) is regarded as a comment.

Source encoding

Since the easiest way to include special characters like em dashes or copyright signs in reST is to directly write them as Unicode characters, one has to specify an encoding:

All Python documentation source files must be in UTF-8 encoding, and the HTML documents written from them will be in that encoding as well.

Gotchas

There are some problems one commonly runs into while authoring reST documents:

- **Separation of inline markup:** As said above, inline markup spans must be separated from the surrounding text by non-word characters, you have to use an escaped space to get around that.

10.7.4 Additional Markup Constructs

Sphinx adds a lot of new directives and interpreted text roles to standard reST markup. This section contains the reference material for these facilities. Documentation for "standard" reST constructs is not included here, though they are used in the Python documentation.

Note: This is just an overview of Sphinx' extended markup capabilities; full coverage can be found in its own documentation.

Meta-information markup

`sectionauthor`

Identifies the author of the current section. The argument should include the author's name such that it can be used for presentation (though it isn't) and email address. The domain name portion of the address should be lower case. Example:

```
.. sectionauthor:: Guido van Rossum <guido@python.org>
```

Currently, this markup isn't reflected in the output in any way, but it helps keep track of contributions.

Module-specific markup

The markup described in this section is used to provide information about a module being documented. Each module should be documented in its own file. Normally this markup appears after the title heading of that file; a typical file might start like this:

```
:mod:`parrot` -- Dead parrot access
===================================

.. module:: parrot
   :platform: Unix, Windows
   :synopsis: Analyze and reanimate dead parrots.
.. moduleauthor:: Eric Cleese <eric@python.invalid>
.. moduleauthor:: John Idle <john@python.invalid>
```

As you can see, the module-specific markup consists of two directives, the `module` directive and the `moduleauthor` directive.

`module`

This directive marks the beginning of the description of a module, package, or submodule. The name should be fully qualified (i.e. including the package name for submodules).

The `platform` option, if present, is a comma-separated list of the platforms on which the module is available (if it is available on all platforms, the option should be omitted). The keys are short identifiers; examples that are in use include "IRIX", "Mac", "Windows", and "Unix". It is important to use a key which has already been used when applicable.

The `synopsis` option should consist of one sentence describing the module's purpose – it is currently only used in the Global Module Index.

The `deprecated` option can be given (with no value) to mark a module as deprecated; it will be designated as such in various locations then.

`moduleauthor`

The `moduleauthor` directive, which can appear multiple times, names the authors of the module code, just like `sectionauthor` names the author(s) of a piece of documentation. It too does not result in any output currently.

Note: It is important to make the section title of a module-describing file meaningful since that value will be inserted in the table-of-contents trees in overview files.

Information units

There are a number of directives used to describe specific features provided by modules. Each directive requires one or more signatures to provide basic information about what is being described, and the content should be the description. The basic version makes entries in the general index; if no index entry is desired, you can give the directive option flag `:noindex:`. The following example shows all of the features of this directive type:

```
.. function:: spam(eggs)
              ham(eggs)
   :noindex:

   Spam or ham the foo.
```

The signatures of object methods or data attributes should not include the class name, but be nested in a class directive. The generated files will reflect this nesting, and the target identifiers (for HTML output) will use both the class and method name, to enable consistent cross-references. If you describe methods belonging to an abstract protocol such as context managers, use a class directive with a (pseudo-)type name too to make the index entries more informative.

The directives are:

`c:function`

Describes a C function. The signature should be given as in C, e.g.:

```
.. c:function:: PyObject* PyType_GenericAlloc(PyTypeObject *type, Py_ssize_t nitems)
```

This is also used to describe function-like preprocessor macros. The names of the arguments should be given so they may be used in the description.

Note that you don't have to backslash-escape asterisks in the signature, as it is not parsed by the reST inliner.

`c:member`

Describes a C struct member. Example signature:

```
.. c:member:: PyObject* PyTypeObject.tp_bases
```

The text of the description should include the range of values allowed, how the value should be interpreted, and whether the value can be changed. References to structure members in text should use the `member` role.

`c:macro`

Describes a "simple" C macro. Simple macros are macros which are used for code expansion, but which do not take arguments so cannot be described as functions. This is not to be used for simple constant definitions. Examples of its use in the Python documentation include `PyObject_HEAD` and `Py_BEGIN_ALLOW_THREADS`.

`c:type`

Describes a C type. The signature should just be the type name.

`c:var`

Describes a global C variable. The signature should include the type, such as:

```
.. c:var:: PyObject* PyClass_Type
```

`data`

Describes global data in a module, including both variables and values used as "defined constants." Class and object attributes are not documented using this directive.

`exception`

Describes an exception class. The signature can, but need not include parentheses with constructor arguments.

`function`

Describes a module-level function. The signature should include the parameters, enclosing optional parameters in brackets. Default values can be given if it enhances clarity. For example:

```
.. function:: repeat([repeat=3[, number=1000000]])
```

Object methods are not documented using this directive. Bound object methods placed in the module namespace as part of the public interface of the module are documented using this, as they are equivalent to normal functions for most purposes.

The description should include information about the parameters required and how they are used (especially whether mutable objects passed as parameters are modified), side effects, and possible exceptions. A small example may be provided.

`coroutinefunction`

Describes a module-level coroutine. The description should include similar information to that described for `function`.

`decorator`

Describes a decorator function. The signature should *not* represent the signature of the actual function, but the usage as a decorator. For example, given the functions

```
def removename(func):
    func.__name__ = ''
    return func

def setnewname(name):
    def decorator(func):
        func.__name__ = name
        return func
    return decorator
```

the descriptions should look like this:

```
.. decorator:: removename

   Remove name of the decorated function.

.. decorator:: setnewname(name)

   Set name of the decorated function to *name*.
```

There is no `deco` role to link to a decorator that is marked up with this directive; rather, use the `:func:` role.

`class`

Describes a class. The signature can include parentheses with parameters which will be shown as the constructor arguments.

`attribute`

Describes an object data attribute. The description should include information about the type of the data to be expected and whether it may be changed directly. This directive should be nested in a class directive, like in this example:

```
.. class:: Spam

   Description of the class.

   .. attribute:: ham

      Description of the attribute.
```

If is also possible to document an attribute outside of a class directive, for example if the documentation for different attributes and methods is split in multiple sections. The class name should then be included explicitly:

```
.. attribute:: Spam.eggs
```

`method`

Describes an object method. The parameters should not include the `self` parameter. The description should include similar information to that described for `function`. This directive should be nested in a class directive, like in the example above.

`coroutinemethod`

Describes an object coroutine method. The parameters should not include the `self` parameter. The description should include similar information to that described for `function`. This directive should be nested in a `class` directive.

`decoratormethod`

Same as `decorator`, but for decorators that are methods.

Refer to a decorator method using the `:meth:` role.

`staticmethod`

Describes an object static method. The description should include similar information to that described for `function`. This directive should be nested in a `class` directive.

`classmethod`

Describes an object class method. The parameters should not include the `cls` parameter. The description should include similar information to that described for `function`. This directive should be nested in a `class` directive.

`abstractmethod`

Describes an object abstract method. The description should include similar information to that described for `function`. This directive should be nested in a `class` directive.

`opcode`

Describes a Python bytecode instruction.

`cmdoption`

Describes a Python command line option or switch. Option argument names should be enclosed in angle brackets. Example:

```
.. cmdoption:: -m <module>

   Run a module as a script.
```

`envvar`

Describes an environment variable that Python uses or defines.

There is also a generic version of these directives:

`describe`

This directive produces the same formatting as the specific ones explained above but does not create index entries or cross-referencing targets. It is used, for example, to describe the directives in this document. Example:

```
.. describe:: opcode

   Describes a Python bytecode instruction.
```

Showing code examples

Examples of Python source code or interactive sessions are represented using standard reST literal blocks. They are started by a `::` at the end of the preceding paragraph and delimited by indentation.

Representing an interactive session requires including the prompts and output along with the Python code. No special markup is required for interactive sessions. After the last line of input or output presented, there should not be an "unused" primary prompt; this is an example of what *not* to do:

```
>>> 1 + 1
2
>>>
```

Syntax highlighting is handled in a smart way:

- There is a "highlighting language" for each source file. By default, this is `'python'` as the majority of files will have to highlight Python snippets.
- Within Python highlighting mode, interactive sessions are recognized automatically and highlighted appropriately.
- The highlighting language can be changed using the `highlight` directive, used as follows:

  ```
  .. highlight:: c
  ```

  This language is used until the next `highlight` directive is encountered.
- The `code-block` directive can be used to specify the highlight language of a single code block, e.g.:

```
.. code-block:: c

   #include <stdio.h>

   void main() {
       printf("Hello world!\n");
   }
```

- The values normally used for the highlighting language are:
  - `python` (the default)
  - `c`
  - `rest`
  - `none` (no highlighting)
- If highlighting with the current language fails, the block is not highlighted in any way.

Longer displays of verbatim text may be included by storing the example text in an external file containing only plain text. The file may be included using the `literalinclude` directive.[1] For example, to include the Python source file `example.py`, use:

```
.. literalinclude:: example.py
```

The file name is relative to the current file's path. Documentation-specific include files should be placed in the `Doc/includes` subdirectory.

Inline markup

As said before, Sphinx uses interpreted text roles to insert semantic markup in documents.

Names of local variables, such as function/method arguments, are an exception, they should be marked simply with `*var*`.

For all other roles, you have to write `:rolename:`content``.

There are some additional facilities that make cross-referencing roles more versatile:

- You may supply an explicit title and reference target, like in reST direct hyperlinks: `:role:`title <target>`` will refer to *target*, but the link text will be *title*.
- If you prefix the content with `!`, no reference/hyperlink will be created.
- For the Python object roles, if you prefix the content with `~`, the link text will only be the last component of the target. For example, `:meth:`~Queue.Queue.get`` will refer to `Queue.Queue.get` but only display `get` as the link text.

  In HTML output, the link's `title` attribute (that is e.g. shown as a tool-tip on mouse-hover) will always be the full target name.

The following roles refer to objects in modules and are possibly hyperlinked if a matching identifier is found:

mod
: The name of a module; a dotted name may be used. This should also be used for package names.

[1] There is a standard `include` directive, but it raises errors if the file is not found. This one only emits a warning.

`func`

The name of a Python function; dotted names may be used. The role text should not include trailing parentheses to enhance readability. The parentheses are stripped when searching for identifiers.

`data`

The name of a module-level variable or constant.

`const`

The name of a "defined" constant. This may be a C-language `#define` or a Python variable that is not intended to be changed.

`class`

A class name; a dotted name may be used.

`meth`

The name of a method of an object. The role text should include the type name and the method name. A dotted name may be used.

`attr`

The name of a data attribute of an object.

`exc`

The name of an exception. A dotted name may be used.

The name enclosed in this markup can include a module name and/or a class name. For example, ``:func:`filter` `` could refer to a function named `filter` in the current module, or the built-in function of that name. In contrast, ``:func:`foo.filter` `` clearly refers to the `filter` function in the `foo` module.

Normally, names in these roles are searched first without any further qualification, then with the current module name prepended, then with the current module and class name (if any) prepended. If you prefix the name with a dot, this order is reversed. For example, in the documentation of the `codecs` module, ``:func:`open` `` always refers to the built-in function, while ``:func:`.open` `` refers to `codecs.open()`.

A similar heuristic is used to determine whether the name is an attribute of the currently documented class.

The following roles create cross-references to C-language constructs if they are defined in the API documentation:

`c:data`

The name of a C-language variable.

`c:func`

The name of a C-language function. Should include trailing parentheses.

`c:macro`

The name of a "simple" C macro, as defined above.

`c:type`

The name of a C-language type.

`c:member`

The name of a C type member, as defined above.

The following roles do not refer to objects, but can create cross-references or internal links:

`envvar`

An environment variable. Index entries are generated.

`keyword`

The name of a Python keyword. Using this role will generate a link to the documentation of the keyword. `True`, `False` and `None` do not use this role, but simple code markup (` ``True`` `), given that they're fundamental to the language and should be known to any programmer.

`option`

A command-line option of Python. The leading hyphen(s) must be included. If a matching `cmdoption` directive exists, it is linked to. For options of other programs or scripts, use simple ` ``code`` ` markup.

`token`

The name of a grammar token (used in the reference manual to create links between production displays).

The following role creates a cross-reference to the term in the glossary:

`term`

Reference to a term in the glossary. The glossary is created using the `glossary` directive containing a definition list with terms and definitions. It does not have to be in the same file as the `term` markup, in fact, by default the Python docs have one global glossary in the `glossary.rst` file.

If you use a term that's not explained in a glossary, you'll get a warning during build.

The following roles don't do anything special except formatting the text in a different style:

`command`

The name of an OS-level command, such as `rm`.

`dfn`

Mark the defining instance of a term in the text. (No index entries are generated.)

`file`

The name of a file or directory. Within the contents, you can use curly braces to indicate a "variable" part, for example:

```
``spam`` is installed in :file:`/usr/lib/python2.{x}/site-packages` ...
```

In the built documentation, the `x` will be displayed differently to indicate that it is to be replaced by the Python minor version.

`guilabel`

Labels presented as part of an interactive user interface should be marked using `guilabel`. This includes labels from text-based interfaces such as those created using `curses` or other text-based libraries. Any label used in the interface should be marked with this role, including button labels, window titles, field names, menu and menu selection names, and even values in selection lists.

`kbd`

Mark a sequence of keystrokes. What form the key sequence takes may depend on platform- or application-specific conventions. When there are no relevant conventions, the names of modifier keys should be spelled out, to improve accessibility for new users and non-native speakers. For example, an *xemacs* key sequence may be marked like ``:kbd:`C-x C-f` ``, but without reference to a specific application or platform, the same sequence should be marked as ``:kbd:`Control-x Control-f` ``.

`mailheader`

The name of an RFC 822-style mail header. This markup does not imply that the header is being used in an email message, but can be used to refer to any header of the same "style." This is also used for headers defined by the various MIME specifications. The header name should be entered in the same way it would normally be found in practice, with the camel-casing conventions being preferred where there is more than one common usage. For example: `:mailheader:`Content-Type``.

`makevar`

The name of a **`make`** variable.

`manpage`

A reference to a Unix manual page including the section, e.g. `:manpage:`ls(1)``.

`menuselection`

Menu selections should be marked using the `menuselection` role. This is used to mark a complete sequence of menu selections, including selecting submenus and choosing a specific operation, or any subsequence of such a sequence. The names of individual selections should be separated by `-->`.

For example, to mark the selection "Start > Programs", use this markup:

```
:menuselection:`Start --> Programs`
```

When including a selection that includes some trailing indicator, such as the ellipsis some operating systems use to indicate that the command opens a dialog, the indicator should be omitted from the selection name.

`mimetype`

The name of a MIME type, or a component of a MIME type (the major or minor portion, taken alone).

`newsgroup`

The name of a Usenet newsgroup.

`program`

The name of an executable program. This may differ from the file name for the executable for some platforms. In particular, the `.exe` (or other) extension should be omitted for Windows programs.

`regexp`

A regular expression. Quotes should not be included.

`samp`

A piece of literal text, such as code. Within the contents, you can use curly braces to indicate a "variable" part, as in `:file:`.

If you don't need the "variable part" indication, use the standard ``` ``code`` ``` instead.

The following roles generate external links:

`pep`

A reference to a Python Enhancement Proposal. This generates appropriate index entries. The text "PEP *number*" is generated; in the HTML output, this text is a hyperlink to an online copy of the specified PEP. Such hyperlinks should not be a substitute for properly documenting the language in the manuals.

`rfc`

A reference to an Internet Request for Comments. This generates appropriate index entries. The text "RFC *number*" is generated; in the HTML output, this text is a hyperlink to an online copy of the specified RFC.

Note that there are no special roles for including hyperlinks as you can use the standard reST markup for that purpose.

Cross-linking markup

To support cross-referencing to arbitrary sections in the documentation, the standard reST labels are "abused" a bit: Every label must precede a section title; and every label name must be unique throughout the entire documentation source.

You can then reference to these sections using the `` :ref:`label-name` `` role.

Example:

```
.. _my-reference-label:

Section to cross-reference
--------------------------

This is the text of the section.

It refers to the section itself, see :ref:`my-reference-label`.
```

The `:ref:` invocation is replaced with the section title.

Alternatively, you can reference any label (not just section titles) if you provide the link text `` :ref:`link text <reference-label>` ``.

Paragraph-level markup

These directives create short paragraphs and can be used inside information units as well as normal text:

`note`

An especially important bit of information about an API that a user should be aware of when using whatever bit of API the note pertains to. The content of the directive should be written in complete sentences and include all appropriate punctuation.

Example:

```
.. note::

   This function is not suitable for sending spam e-mails.
```

`warning`

An important bit of information about an API that a user should be aware of when using whatever bit of API the warning pertains to. The content of the directive should be written in complete sentences and include all appropriate punctuation. In the interest of not scaring users away from pages filled with warnings, this directive should only be chosen over `note` for information regarding the possibility of crashes, data loss, or security implications.

`versionadded`

This directive documents the version of Python which added the described feature, or a part of it, to the library or C API. When this applies to an entire module, it should be placed at the top of the module section before any prose.

The first argument must be given and is the version in question. The second argument is optional and can be used to describe the details of the feature.

Example:

```
.. versionadded:: 3.5
```

versionchanged

Similar to versionadded, but describes when and what changed in the named feature in some way (new parameters, changed side effects, platform support, etc.). This one *must* have the second argument (explanation of the change).

Example:

```
.. versionchanged:: 3.1
   The *spam* parameter was added.
```

Note that there should be no blank line between the directive head and the explanation; this is to make these blocks visually continuous in the markup.

deprecated

Indicates the version from which the described feature is deprecated.

There is one required argument: the version from which the feature is deprecated.

Example:

```
.. deprecated:: 3.8
```

deprecated-removed

Like deprecated, but it also indicates in which version the feature is removed.

There are two required arguments: the version from which the feature is deprecated, and the version in which the feature is removed.

Example:

```
.. deprecated-removed:: 3.8 4.0
```

impl-detail

This directive is used to mark CPython-specific information. Use either with a block content or a single sentence as an argument, i.e. either

```
.. impl-detail::

   This describes some implementation detail.

   More explanation.
```

or

```
.. impl-detail:: This shortly mentions an implementation detail.
```

"**CPython implementation detail:**" is automatically prepended to the content.

seealso

Many sections include a list of references to module documentation or external documents. These lists are created using the seealso directive.

The seealso directive is typically placed in a section just before any sub-sections. For the HTML output, it is shown boxed off from the main flow of the text.

The content of the seealso directive should be a reST definition list. Example:

```
.. seealso::

   Module :mod:`zipfile`
      Documentation of the :mod:`zipfile` standard module.

   `GNU tar manual, Basic Tar Format <http://link>`_
      Documentation for tar archive files, including GNU tar extensions.
```

rubric

This directive creates a paragraph heading that is not used to create a table of contents node. It is currently used for the "Footnotes" caption.

centered

This directive creates a centered boldfaced paragraph. Use it as follows:

```
.. centered::

   Paragraph contents.
```

Table-of-contents markup

Since reST does not have facilities to interconnect several documents, or split documents into multiple output files, Sphinx uses a custom directive to add relations between the single files the documentation is made of, as well as tables of contents. The `toctree` directive is the central element.

toctree

This directive inserts a "TOC tree" at the current location, using the individual TOCs (including "sub-TOC trees") of the files given in the directive body. A numeric `maxdepth` option may be given to indicate the depth of the tree; by default, all levels are included.

Consider this example (taken from the library reference index):

```
.. toctree::
   :maxdepth: 2

   intro
   strings
   datatypes
   numeric
   (many more files listed here)
```

This accomplishes two things:

- Tables of contents from all those files are inserted, with a maximum depth of two, that means one nested heading. `toctree` directives in those files are also taken into account.
- Sphinx knows that the relative order of the files `intro`, `strings` and so forth, and it knows that they are children of the shown file, the library index. From this information it generates "next chapter", "previous chapter" and "parent chapter" links.

In the end, all files included in the build process must occur in one `toctree` directive; Sphinx will emit a warning if it finds a file that is not included, because that means that this file will not be reachable through standard navigation.

The special file `contents.rst` at the root of the source directory is the "root" of the TOC tree hierarchy; from it the "Contents" page is generated.

Index-generating markup

Sphinx automatically creates index entries from all information units (like functions, classes or attributes) like discussed before.

However, there is also an explicit directive available, to make the index more comprehensive and enable index entries in documents where information is not mainly contained in information units, such as the language reference.

The directive is `index` and contains one or more index entries. Each entry consists of a type and a value, separated by a colon.

For example:

```
.. index::
   single: execution; context
   module: __main__
   module: sys
   triple: module; search; path
```

This directive contains five entries, which will be converted to entries in the generated index which link to the exact location of the index statement (or, in case of offline media, the corresponding page number).

The possible entry types are:

single
: Creates a single index entry. Can be made a subentry by separating the subentry text with a semicolon (this notation is also used below to describe what entries are created).

pair
: `pair: loop; statement` is a shortcut that creates two index entries, namely `loop; statement` and `statement; loop`.

triple
: Likewise, `triple: module; search; path` is a shortcut that creates three index entries, which are `module; search path`, `search; path, module` and `path; module search`.

module, keyword, operator, object, exception, statement, builtin
: These all create two index entries. For example, `module: hashlib` creates the entries `module; hashlib` and `hashlib; module`. The builtin entry type is slightly different in that "built-in function" is used in place of "builtin" when creating the two entries.

For index directives containing only "single" entries, there is a shorthand notation:

```
.. index:: BNF, grammar, syntax, notation
```

This creates four index entries.

Grammar production displays

Special markup is available for displaying the productions of a formal grammar. The markup is simple and does not attempt to model all aspects of BNF (or any derived forms), but provides enough to allow context-free grammars to be displayed in a way that causes uses of a symbol to be rendered as hyperlinks to the definition of the symbol. There is this directive:

`productionlist`
: This directive is used to enclose a group of productions. Each production is given on a single line and consists of a name, separated by a colon from the following definition. If the definition spans multiple lines, each continuation line must begin with a colon placed at the same column as in the first line.

Blank lines are not allowed within `productionlist` directive arguments.

The definition can contain token names which are marked as interpreted text (e.g. `unaryneg ::= "-" `integer``) – this generates cross-references to the productions of these tokens.

Note that no further reST parsing is done in the production, so that you don't have to escape `*` or `|` characters.

The following is an example taken from the Python Reference Manual:

```
.. productionlist::
   try_stmt: try1_stmt | try2_stmt
   try1_stmt: "try" ":" `suite`
            : ("except" [`expression` ["," `target`]] ":" `suite`)+
            : ["else" ":" `suite`]
            : ["finally" ":" `suite`]
   try2_stmt: "try" ":" `suite`
            : "finally" ":" `suite`
```

Substitutions

The documentation system provides three substitutions that are defined by default. They are set in the build configuration file `conf.py`.

`|release|`

Replaced by the Python release the documentation refers to. This is the full version string including alpha/beta/release candidate tags, e.g. `2.5.2b3`.

`|version|`

Replaced by the Python version the documentation refers to. This consists only of the major and minor version parts, e.g. `2.5`, even for version 2.5.1.

`|today|`

Replaced by either today's date, or the date set in the build configuration file. Normally has the format `April 14, 2007`.

10.7.5 Building the documentation

The toolset used to build the docs is written in Python and is called Sphinx. Sphinx is maintained separately and is not included in this tree. Also needed are blurb, a tool to create `Misc/NEWS` on demand; and python-docs-theme, the Sphinx theme for the Python documentation.

To build the documentation, follow the instructions from one of the sections below. You can view the documentation after building the HTML by pointing a browser at the file `Doc/build/html/index.html`.

You are expected to have installed the latest stable version of Sphinx and blurb on your system or in a virtualenv (which can be created using `make venv`), so that the Makefile can find the `sphinx-build` command. You can also specify the location of `sphinx-build` with the `SPHINXBUILD` **`make`** variable.

Using make / make.bat

On Unix, run the following from the root of your *repository clone* to build the output as HTML:

```
cd Doc
make venv
make html
```

or alternatively `make -C Doc/ venv html`. `htmlview` can be used instead of `html` to conveniently open the docs in a browser once the build completes.

You can also use `make help` to see a list of targets supported by **`make`**. Note that `make check` is automatically run when you submit a *pull request*, so you should make sure that it runs without errors.

On Windows, a `make.bat` batchfile tries to emulate **`make`** as closely as possible, but the venv target is not implemented, so you will probably want to make sure you are working in a virtual environment before proceeding, otherwise all dependencies will be automatically installed on your system.

When ready, run the following from the root of your *repository clone* to build the output as HTML:

```
cd Doc
make html
```

You can also use `make help` to see a list of targets supported by `make.bat`.

See also `Doc/README.rst` for more information.

Using sphinx-build

Sometimes we directly want to execute the sphinx-build tool instead of through `make` (although the latter is still the preferred way). In this case, you can use the following command line from the `Doc` directory (make sure to install Sphinx, blurb and python-docs-theme packages from PyPI):

```
sphinx-build -b<builder> . build/<builder>
```

where <`builder`> is one of html, text, latex, or htmlhelp (for explanations see the make targets above).

10.7.6 Translating

Python documentation translations are governed by **PEP 545**. They are built by docsbuild-scripts and hosted on docs.python.org. There are several documentation translations already in production; others are works in progress.

Language	Contact	Links
Arabic (ar)	Abdur-Rahmaan Janhangeer	GitHub
Bengali as spoken in India (bn_IN)	Kushal Das	GitHub
French (fr)	Julien Palard (@JulienPalard)	GitHub
Hindi as spoken in India (hi_IN)		GitHub
Hungarian (hu)	Tamás Bajusz (@gbtami)	GitHub Mailing List
Indonesian (id)	Oon Arfiandwi	GitHub
Italian (it)		mail
Japanese (ja)	Kinebuchi Tomohiko (@cocoatomo)	GitHub Doc
Korean (ko)		GitHub Doc
Marathi (mr)	Sanket Garade	GitHub
Lithuanian (lt)		mail
Persian (fa)	Komeil Parseh (@mmdbalkhi)	GitHub
Polish (pl)		GitHub Translations Doc mail
Portuguese (pt)	Gustavo Toffo	
Portuguese as spoken in Brasil (pt-br)	Marco Rougeth	GitHub Wiki Telegram article
Russian (ru)		mail
Simplified Chinese (zh-cn)	Shengjing Zhu	Transifex GitHub Doc
Spanish (es)	Raúl Cumplido	GitHub
Traditional Chinese (zh-tw)	Matt Wang, Josix Wang	GitHub Doc
Turkish (tr)	Ege Akman (@egeakman)	GitHub

Starting a new translation

First subscribe to the translation mailing list, and introduce yourself and the translation you're starting. Translations fall under the aegis of the PSF Translation Workgroup

Then you can bootstrap your new translation by using our cookiecutter.

The important steps look like this:

- Create the GitHub repo (anywhere) with the right hierarchy (using the cookiecutter).
- Gather people to help you translate. You can't do it alone.
- You can use any tool to translate, as long as you can synchronize with git. Some use Transifex, and some use only GitHub. You can choose another way if you like; it's up to you.
- Ensure we update this page to reflect your work and progress, either via a PR or by asking on the translation mailing list.
- When `bugs.html`, `tutorial`, and `library/functions` are 100% completed, ask on the translation mailing list for your language to be added in the language picker on docs.python.org.

PEP 545 summary:

Here are the essential points of **PEP 545**:

- Each translation is assigned an appropriate lowercased language tag, with an optional region subtag, and joined with a dash, like `pt-br` or `fr`.
- Each translation is under CC0 and marked as such in the README (as in the cookiecutter).
- Translation files are hosted on `https://github.com/python/python-docs-{LANGUAGE_TAG}` (not mandatory to start a translation, but mandatory to land on `docs.python.org`).
- Translations having completed `tutorial/`, `library/stdtypes` and `library/functions` are hosted on `https://docs.python.org/{LANGUAGE_TAG}/{VERSION_TAG}/`.

How to get help

Discussions about translations occur on the translation mailing list, and there's a Libera.Chat IRC channel, `#python-doc`.

Translation FAQ

Which version of the Python documentation should be translated?

Consensus is to work on current stable. You can then propagate your translation from one branch to another using pomerge.

Are there some tools to help in managing the repo?

Here's what we're using:

- pomerge to propagate translations from one file to others.
- pospell to check for typos in `.po` files.
- powrap to rewrap the `.po` files before committing. This helps keep git diffs short.
- potodo to list what needs to be translated.

How is a coordinator elected?

There is no election; each translation has to sort this out. Here are some suggestions.

- Coordinator requests are to be public on the translation mailing list.
- If the given language has a native core dev, the core dev has their say on the choice.
- Anyone who wants to become coordinator for their native language and shows motivation by translating and building a community will be named coordinator.
- In case of concurrency between two persons, no one will sort this out for you. It is up to you two to organize a local election or whatever is needed to sort this out.
- If a coordinator becomes inactive or unreachable for a long period of time, someone else can ask for a takeover on the translation mailing list.

The entry for my translation is missing/not up to date on this page

Ask on the translation mailing list, or better, make a PR on the devguide.

I have a translation, but it's not in git. What should I do?

You can ask for help on the translation mailing list, and the team will help you create an appropriate repository. You can still use tools like transifex, if you like.

My git hierarchy does not match yours. Can I keep it?

No, inside the `github.com/python` organization we'll all have the exact same hierarchy so bots will be able to build all of our translations. So you may have to convert from one hierarchy to another. Ask for help on the translation mailing list if you're not sure on how to do it.

What hierarchy should I use in my GitHub repository?

As for every project, we have a *branch* per version. We store `.po` files in the root of the repository using the `gettext_compact=0` style.

10.8 Silence Warnings From the Test Suite

When running Python's test suite, no warnings should result when you run it under *strenuous testing conditions* (you can ignore the extra flags passed to `test` that cause randomness and parallel execution if you want). Unfortunately new warnings are added to Python on occasion which take some time to eliminate (e.g., `ResourceWarning`). Typically the easy warnings are dealt with quickly, but the more difficult ones that require some thought and work do not get fixed immediately.

If you decide to tackle a warning you have found, open an issue on the issue tracker (if one has not already been opened) and say you are going to try and tackle the issue, and then proceed to fix the issue.

10.9 Fixing "easy" Issues (and Beyond)

When you feel comfortable enough to want to help tackle issues by trying to create a patch to fix an issue, you can start by looking at the "easy" issues. These issues *should* be ones where it should take no longer than a day or weekend to fix. But because the "easy" classification is typically done at triage time it can turn out to be inaccurate, so do feel free to leave a comment if you think the classification no longer applies.

For the truly adventurous looking for a challenge, you can look for issues that are not considered easy and try to fix those. It must be warned, though, that it is quite possible that a bug that has been left open has been left into that state because of the difficulty compared to the benefit of the fix. It could also still be open because no consensus has been reached on how to fix the issue (although having a patch that proposes a fix can turn the tides of the discussion to help bring it to a close). Regardless of why the issue is open, you can also always provide useful comments if you do attempt a fix, successful or not.

10.10 Issue Tracking

10.10.1 Using the Issue Tracker

If you think you have found a bug in Python, you can report it to the issue tracker. The issue tracker is now hosted on GitHub, alongside the codebase and pull requests. Documentation bugs can also be reported there.

If you would like to file an issue about this devguide, please do so at the devguide repo.

Note: Python used to use a dedicated Roundup instance as its issue tracker. That old bug tracker was hosted under the domain `bugs.python.org` (sometimes called `bpo` for short). Currently a read-only version is still available on that domain for historical purposes. All `bpo` data has been migrated to the current issue tracker on GitHub.

If you're familiar with `bpo` and would like to learn more about GitHub issues, please read this page, and the *Triaging an Issue* page as they provide good introductory material. There is also a *GitHub issues for BPO users* document to answer some of the more popular questions.

Checking if a bug already exists

The first step before filing an issue report is to see whether the problem has already been reported. Checking if the problem is an existing issue will:

- help you see if the problem has already been resolved or has been fixed for the next release
- save time for you and the developers
- help you learn what needs to be done to fix it
- determine if additional information, such as how to replicate the issue, is needed

To see if an issue already exists, search the bug database using the search box above the list of bugs on the issues page. A form-based advanced search query builder is also available on GitHub to help creating the text query you need.

Reporting an issue

If the problem you're reporting is not already in the issue tracker, you can report it using the green "New issue" button on the right of the search box above the list of bugs. If you're not already signed in to GitHub, it will ask you to do so now.

First you need to select what kind of problem you want to report. The available choices are:

- **Bug report**: an existing feature isn't working as expected;
- **Documentation**: there is missing, invalid, or misleading documentation;
- **Enhancement**: suggest a new feature for Python;
- **Performance**: something should work faster;
- **Security**: there is a specific kind of weakness open to exploitation through the points of vulnerability;
- **Tests**: something is wrong with CPython's suite of regression tests;
- **Discuss**: you'd like to learn more about Python, discuss ideas for possible changes to future Python versions, track core development discussions, or join a specific special-interest group.

Depending on your choice, a dedicated form template will appear. In particular, you'll notice that the last button actually takes you to Discourse where many Python-related discussions take place.

The submission form has only two fields that you need to fill:

- in the **Title** field, enter a *very* short description of the problem; less than ten words is good;
- in the **Write** field, describe the problem in detail using hints from the template that was put in that field for you. Be sure to include what you expected to happen, what did happen, and how to replicate the problem. Be sure to include whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate). In particular, *what version of Python* you were using.

Understanding the issue's progress and status

There is a number of additional fields like **Assignees**, **Labels**, **Projects**, and **Milestone**. Those are filled by triagers and core developers, this is covered in the *Triaging an Issue* page. You don't need to worry about those when reporting issues as a Python user.

You will automatically receive an update each time an action is taken on the bug, unless you changed your GitHub notification settings.

10.10.2 Disagreement With a Resolution on the Issue Tracker

As humans, we will have differences of opinions from time to time. First and foremost, please be respectful that care, thought, and volunteer time went into the resolution.

With this in mind, take some time to consider any comments made in association with the resolution of the issue. On reflection, the resolution steps may seem more reasonable than you initially thought.

If you still feel the resolution is incorrect, then raise a thoughtful question on python-dev. Further argument and disrespectful discourse on python-dev after a consensus has been reached amongst the core developers is unlikely to win any converts.

As a reminder, issues closed by a core developer have already been carefully considered. Please do not reopen a closed issue. An issue can be closed with reason either as `complete` or `not planned`.

10.10.3 Helping Triage Issues

Once you know your way around how Python's source files are structured and you are comfortable working with patches, a great way to contribute is to help triage issues. Do realize, though, that experience working on Python is needed in order to effectively help triage.

Around the clock, new issues are being opened on the issue tracker and existing issues are being updated. Every issue needs to be triaged to make sure various things are in proper order. Even without special privileges you can help with this process.

Classifying Reports

For bugs, an issue needs to:

- clearly explain the bug so it can be reproduced
- include all relevant platform details
- state what version(s) of Python are affected by the bug.

These are things you can help with once you have experience developing for Python:

- try reproducing the bug: For instance, if a bug is not clearly explained enough for you to reproduce it then there is a good chance a core developer won't be able to either.
- see if the issue happens on a different Python version: It is always helpful to know if a bug not only affects the in-development version of Python, but whether it also affects other versions in maintenance mode.
- write a unit test: If the bug lacks a unit test that should end up in Python's test suite, having that written can be very helpful.

This is all helpful as it allows triagers (i.e., *people with the Developer role on the issue tracker*) to properly classify an issue so it can be handled by the right core developers in a timely fashion.

Reviewing Patches

If an issue has a pull request attached that has not been reviewed, you can help by making sure the patch:

- follows the style guides
- applies cleanly to an up-to-date clone
- is a good solution to the problem it is trying to solve
- includes proper tests
- includes proper documentation changes
- submitter is listed in `Misc/ACKS`, either already or the patch adds them

Doing all of this allows core developers and *triagers* to more quickly look for subtle issues that only people with extensive experience working on Python's code base will notice.

Finding an Issue You Can Help With

If you want to help triage issues, you might also want to search for issues in modules which you have a working knowledge. Search for the name of a module in the issue tracker or use the advanced search query builder to search for specific kinds of issues (e.g. the "Windows" label if you are a Windows developer, "Extension Modules" if you are familiar with C, etc.).

10.10.4 Gaining the "Triager" Role on the Issue Tracker

When you have consistently shown the ability to properly help triage issues without guidance, you may request that you be given the "Triager" role on the issue tracker. You can make the request to any person who already has the Triager role. If they decide you are ready to gain the extra privileges on the tracker they will then act as a mentor to you until you are ready to do things entirely on your own. There is no set rule as to how many issues you need to have helped with before or how long you have been participating. The key requirements are that you show the desire to help, you are able to work well with others (especially those already with the Triager role), and that have a firm grasp of how to do things on the issue tracker properly on your own.

Gaining the Triager role will allow you to set any value on any issue in the tracker, releasing you from the burden of having to ask others to set values on an issue for you in order to properly triage something. This will not only help speed up and simplify your work in helping out, but also help lessen the workload for everyone by gaining your help.

10.10.5 Sub-pages related to the Issue Tracker

GitHub Labels

We're using labels on GitHub to categorize issues and pull requests. Many labels are shared for both use cases, while some are dedicated only to one. Below is a possibly inexhaustive list, but it should get you going. For a full list, see here.

General purpose labels

type-behavior
Used for issues/PRs that address unintentional behavior, but do not pose significant security concerns. Generally, bugfixes will be attached to a specific issue where the unintended behavior was first reported.

type-documentation
Used for issues/PRs that exclusively involve changes to the documentation. Documentation includes `*.rst` files, docstrings, and code comments.

type-enhancement
Used for issues/PRs that provide additional functionality or capabilities beyond the existing specifications.

type-performance
Used for issues/PRs that provide performance optimizations.

type-security
Used for issues/PRs that involve critical security issues. Less severe security concerns can instead use the type-bugfix label.

type-tests
Used for issues/PRs that exclusively involve changes to the tests.

OS-Mac / OS-Windows
Used for issues/PRs involving changes which only have an effect upon a specific operating system.

spam
Used for issues/PRs that don't include enough eggs or bacon.

Labels specific to issues

Priority

release-blocker
The highest priority of an issue. If unaddressed, will cause the release manager to hold releasing a new version of Python.

deferred-blocker
A release blocker that was pushed one or more releases into the future. Possibly a temporary workaround was employed, or the version of Python the issue is affecting is still in alpha or beta stages of development.

Component

library
Used for issues involving Python modules in the `Lib/` dir.

docs
Used for issues involving documentation in the `Doc/` dir.

interpreter-core
Used for issues in interpreter core (`Objects/`, `Python/`, `Grammar/`, and `Parser/` dirs).

extension-modules
Used for issues involving C modules in the `Modules/` dir.

tests
Used for issues involving only Python's regression test suite, i.e. files in the `Lib/test/` dir.

Other

new
Denotes that the issue hasn't been looked at by triagers or core developers yet.

easy
Denotes that the issue is a good candidate for a newcomer to address.

Labels specific to PRs

DO-NOT-MERGE
Used on PRs to prevent miss-islington from being able to automatically merge the pull request. This label is appropriate when a PR has a non-trivial conflict with the branch it is being merged into.

expert-asyncio
Used for PRs which involve changes to the asyncio module or other asynchronous frameworks that utilize it.

invalid
Used manually for PRs that do not meet basic requirements and automatically added by bedevere when PR authors attempt to merge maintenace branches into the main branch. During events such as the October Hacktoberfest, this label will prevent the PR from counting toward the author's contributions.

needs backport to X.Y
Used for PRs which are appropriate to backport to branches prior to main. Generally, backports to the maintenance branches are primarily bugfixes and documentation clarifications. Backports to the security branches are

strictly reserved for PRs involving security fixes, such as crashes, privilege escalation, and DoS. The use of this label will cause miss-islington to attempt to automatically merge the PR into the branches specified.

skip issue
Used for PRs which involve trivial changes, such as typo fixes, comment changes, and section rephrases. The majority of PRs require an issue to be attached to, but if there are no code changes and the section being modified retains the same meaning, this label might be appropriate.

skip news
Similar to the skip issue label, this label is used for PRs which involve trivial changes, backports, or already have a relevant news entry in another PR. Any potentially impactful changes should have a corresponding news entry, but for trivial changes it's commonly at the discretion of the PR author if they wish to opt-out of making one.

sprint
Used for PRs authored during an in-person sprint, such as at PyCon, EuroPython, or other official Python events. The label is used to prioritize the review of those PRs during the sprint.

stale
Used for PRs that include changes which are no longer relevant, or when the author hasn't responded to feedback in a long period of time, or when the reviewer is unresponsive. This label helps core developers quickly identify PRs that are candidates for closure or require a ping to the author or reviewer.

awaiting review
Used for PRs that haven't been reviewed by anyone yet.

awaiting core review
Used when the PR is authored by a core developer or when a non-core developer has reviewed the PR, even if they requested changes. Note that reviewers could have been added manually by a triager or core developer, or included automatically through use of the CODEOWNERS file.

awaiting changes
A reviewer required changes to proceed with the PR.

awaiting change review
The PR author made requested changes, and they are waiting for review.

awaiting merge
The PR has been approved by a core developer and is ready to merge.

test-with-buildbots
Used on PRs to test the latest commit with the buildbot fleet. Generally for PRs with large code changes requiring more testing before merging. This may take multiple hours to complete. Triagers can also stop a stuck build using the web interface.

GitHub issues for BPO users

Here are some frequently asked questions about how to do things in GitHub issues that you used to be able to do on bpo.

Before you ask your own question, make sure you read *Issue Tracking* and *Triaging an Issue* (specifically including *GitHub Labels*) as those pages include a lot of introductory material.

How to format my comments nicely?

There is a wonderful beginner guide to writing and formatting on GitHub. Highly recommended.

One pro-tip we can sell you right here is that if you want to paste some longer log as a comment, attach a file instead (see how below). If you still insist on pasting it in your comment, do it like this:

```
<details>
<summary>This is the summary text, click me to expand</summary>

Here goes the long, long text.
It will be collapsed by default!
</details>
```

How to attach files to an issue?

Drag them into the comment field, wait until the file uploads, and GitHub will automatically put a link to your file in your comment text.

How to link to file paths in the repository when writing comments?

Use Markdown links. If you link to the default GitHub path, the file will link to the latest current version on the given branch.

You can get a permanent link to a given revision of a given file by pressing "y".

How to do advanced searches?

Use the GitHub search syntax or the interactive advanced search form that generates search queries for you.

Where is the "nosy list"?

Subscribe another person to the issue by tagging them in the comment with `@username`.

If you want to subscribe yourself to an issue, click the *Subscribe* button in the sidebar.

Similarly, if you were tagged by somebody else but decided this issue is not for you, you might click the *Unsubscribe* button in the sidebar.

There is no exact equivalent of the "nosy list" feature, so to preserve this information during the transfer, we list the previous members of this list in the first message on the migrated issue.

How to add issue dependencies?

Add a checkbox list like this in the issue description:

```
- [x] #739
- [ ] https://github.com/octo-org/octo-repo/issues/740
- [ ] Add delight to the experience when all tasks are complete :tada:
```

then those will become sub-tasks on the given issue. Moreover, GitHub will automatically mark a task as complete if the other referenced issue is closed. More details in the official GitHub documentation.

What on Earth is a "mannequin"?

For issues migrated to GitHub from bpo where the authors or commenters are not core developers, we opted not to link to their GitHub accounts directly. Users not in the python organization on GitHub might not like comments to appear under their name from an automated import. Others never linked GitHub on bpo in the first place so linking their account, if any, would be impossible.

In those cases a "mannequin" account is present to help follow the conversation that happened in the issue. In case the user did share their GitHub account name in their bpo profile, we use that. Otherwise, their classic bpo username is used instead.

Where did the "Resolution" field go?

Based on historical data we found it not being used very often.

Where did the "Low", "High", and "Critical" priorities go?

Based on historical data we found those not being used very often.

How to find a random issue?

This is not supported by GitHub.

Where are regression labels?

We rarely updated this information and it turned out not to be particularly useful outside of the change log.

See also:

Issues with Python and documentation

The Python issue tracker
Where to report issues about Python.

The New-bugs-announce mailing list
Where all the new issues created on the tracker are reported.

The Python-bugs-list mailing list
Where all the changes to issues are reported.

10.11 Triaging an Issue

This section of the devguide documents the issue tracker for users and developers.

Contributors with the Triager role on the issue tracker can triage issues directly without any assistance.

Additionally, this section provides an overview of the Python triage team.

10.11.1 Python triage team

The Python triage team is a group dedicated towards improving workflow efficiency through thoughtful review and triage of open issues and pull requests. This helps contributors receive timely feedback and enables core developers to focus on reviewed items which reduces their workload. The expectations of this role expand upon the "Triager" role on the issue tracker. The responsibilities listed below are primarily centered around the Python GitHub repositories. This extends beyond CPython, and, as needed, to other repos such as devguide and core-workflow.

Responsibilities include:

- **PR/issue management**
  - Reviewing PRs
  - Assisting contributors
  - Notifying appropriate core developers
- **Applying appropriate labels to PRs/Issues**
  - Skip news
  - Skip issue
  - Good first issue
  - Other categorizations

Although triagers have the power to close PRs, they should generally not do so without first consulting a core developer. By having triagers and core developers work together, the author receives a careful consideration of their PR. This encourages future contributions, regardless of whether their PR is accepted or closed.

Nonetheless, triagers should feel free to close a PR if they judge that the chance of the PR being merged would be exceedingly low, even if substantial revisions were made to the PR. This includes (but is not limited to) the following:

- PRs proposing solely cosmetic changes
- PRs proposing changes to deprecated modules
- **PRs that are no longer relevant. This includes:**
  - PRs proposing fixes for bugs that can no longer be reproduced
  - PRs proposing changes that have been rejected by Python core developers elsewhere (e.g. in an issue or a PEP rejection notice)

If a triager has any doubt about whether to close a PR, they should consult a core developer before taking any action.

Triagers can also make use of the `invalid` and `stale` labels to suggest that a PR may be suitable for closure. For more information, see the *GitHub PR labels* section.

Note that it is of paramount importance to treat every contributor to the Python project kindly and with respect. Regardless of whether they're entirely new or a veteran core developer, they're actively choosing to voluntarily donate their time towards the improvement of Python. As is the case with any member of the Python Software Foundation, always follow the PSF Code of Conduct.

10.11.2 Becoming a member of the Python triage team

Any Python core developers are welcome to invite a Python contributor to the Python triage team. Triagers will be responsible to handle not just issues, but also pull requests, and even managing backports. A Python triager has access to more repositories than just CPython.

Any existing active contributor to the Python repository on GitHub can transition into becoming a Python triager. They can request this to any core developer, and the core developer can pass the request to the Python organization admin on GitHub. The request can be made confidentially via a DM in Discourse, or publicly by opening an issue in the core-workflow repository.

For every new triager, it would be great to announce them in the python-committers mailing list and core-workflow category in Discourse. Example announcement.

GitHub Labels for PRs

An important component of triaging PRs for the CPython repo involves appropriately categorizing them through the usage of labels. For this purpose we're using *GitHub Labels*.

10.11.3 Applying labels for Issues

The major elements found in an issue report include:

- Classification (including *Title*) - Metadata that lets us categorize the issue. Apart from the *Title* field, we use some *type-*, *component-*, and *version-* specific labels.
- Process - These fields indicate the state of the issue and its progress toward resolution. The fields are *Status* (open/closed), *Assignees*, *Comment*, as well as *priority-* and *keyword-* specific labels.
- Messages
- History

Title

A brief description of the issue. Review whether the title is too generic or specifies an incorrect term or library.

(Optional) Add a prefix at the start of the title to indicate the module, e.g. IDLE, doc, or asyncio.

Type

Describes the type of issue. If an issue does not fit within any specific type, please do not set a type.

Type	Description
behavior	Unexpected behavior, result, or exception. Most bugs will have this type. This group also includes compile errors, and crashers.
enhancement	Issues that propose the addition of new functionality, such as new functions, classes, modules, or even new arguments for existing functions. Also used for improvements in the documentation, test suite and other refactorings. A good place to discuss enhancements prior to filing an issue is python-ideas mailing list.
performance	Situations where too much time is necessary to complete the task. For example, a common task now takes significantly longer to complete. This group also includes resource usage (e.g. too much memory needed) issues.
security	Issues that might have security implications. Report security vulnerabilities using the procedure found in the Reporting security issues in Python page on the python.org website.

Stage

A needed next action to advance the issue. The *stage* on GitHub issues is determined by presence of a linked PR and whether the issue is still open or closed. It is the PR that holds code review-related labels.

Components

The area or Python library affected by the issue. A single issue can apply multiple component labels.

One or more components may be selected for an issue:

Component	Description
Documentation	The documentation in Doc (source used to build HTML docs for https://docs.python.org/).
Extension Modules	C modules in Modules.
Interpreter Core	The interpreter core. The built-in objects in Objects, the Python, Grammar and Parser dirs.
Library (Lib)	Python modules in Lib.
Tests	The unittest framework in Lib/unittest The doctest framework Lib/doctest.py. The CPython tests in Lib/test. The test runner in Lib/test/regrtest.py. The test support utilities in Lib/test/support.

Versions

The known versions of Python that the issue affects and should be fixed for.

Thus if an issue for a new feature is assigned for e.g., Python 3.8 but is not applied before Python 3.8.0 is released, this label should be updated to say `python-3.9` as the version and drop `python-3.8`.

Priority

What is the severity and urgency?

Priority	Description
normal	The default value for most issues filed.
deferred blocker	The issue will not hold up the next release, *n*. It will be promoted to a *release blocker* for the following release, *n+1*.
release blocker	The issue **must** be fixed before *any* release is made, e.g., will block the next release even if it is an alpha release.

As a guideline, whether a bug is a *release blocker* for the current release schedule is decided by the release manager. Triagers may recommend this priority and should notify the release manager by tagging them in a comment using `@username`. If needed, consult the release schedule and the release's associated PEP for the release manager's name.

Keywords

Various informational flags about the issue. Multiple values are possible.

Keyword	Description
easy	Fixing the issue should not take longer than a day for someone new to contributing to Python to solve.

Nosy List

A list of people who may be interested in an issue.

This used to be a feature of the old issue tracker. On GitHub issues the same effect is achieved by tagging people in a comment using `@username`.

It is acceptable to tag someone to if you think the issue should be brought to their attention. Use the *Experts Index* to know who wants to be added to the nosy list for issues targeting specific areas.

If you want to subscribe yourself to an issue, click the *Subscribe* button in the sidebar. Similarly, if you were tagged by somebody else but decided this issue is not for you, you might click the *Unsubscribe* button in the sidebar.

Assignees

Who is expected to take the next step in resolving the issue.

It is acceptable to assign an issue to someone if the issue cannot move forward without their help, e.g., they need to make a technical decision to allow the issue to move forward. Also consult the *Experts Index* as certain stdlib modules should always be assigned to a specific person.

Note that in order to assign an issue to someone, that person **must** be a team member, likely a Triager or a core developer.

Dependencies

The issue requires the listed issue(s) to be resolved first before it can move forward. This is achieved using checkbox lists in the initial issue description comment. Long story short, if you add this:

```
- [x] #739
- [ ] https://github.com/octo-org/octo-repo/issues/740
- [ ] Add delight to the experience when all tasks are complete :tada:
```

then those will become sub-tasks on the given issue. Moreover, GitHub will automatically mark a task as complete if the other referenced issue is closed.

More details in the official GitHub documentation.

Superseder

The issue is a duplicate of the listed issue(s). To make GitHub mark an issue as duplicate, write "Duplicate of #xxxx" in a comment.

Status

Status	Description
open	Issue is not resolved.
closed	The issue has been resolved (somehow).

Linked pull requests

A link might be added manually using the cog icon next to this field. Most commonly though, if the PR includes "Fixes #xxx" in its description, the link will be added automatically.

10.11.4 Generating Special Links in a Comment

Using the following abbreviations in a comment will automatically generate a link to relevant web pages.

Comment abbreviation	Description
`#<number>`, `GH-<number>`	Links to the tracker issue or PR `<number>` (they share the same sequence of integers on GitHub).
`BPO-<number>`	Links to the old bug tracker at bugs.python.org.
a 10-, 11-, 12-, or 40-digit hex `<number>`	Indicates a Git changeset identifier and generates a link to changeset `<number>` on GitHub.

10.11.5 Checklist for Triaging

- Read the issue comment(s).
- **Review and set classification fields**
  - Title: should be concise with specifics which are helpful to someone scanning a list of issue titles. (Optional, if possible) Add a prefix at the start of the title to indicate the module, e.g. IDLE, doc, or async.
  - Type
  - Stage
  - Components: multiple items may be set
  - Versions: set if known, leave blank if unsure. Multiple items may be set.
- **Review and set process fields**
  - Status
  - Superseder
  - Assignees
  - Nosy List
  - Priority
  - Keywords
- (Optional) Leave a brief comment about the proposed next action needed. If there is a long message list, a summary can be very helpful.

10.12 Following Python's Development

Python's development is communicated through a myriad of ways, mostly through mailing lists, but also other forms.

10.12.1 Standards of behaviour in these communication channels

We try to foster environments of mutual respect, tolerance and encouragement, as described in the PSF's Diversity Statement. Abiding by the guidelines in this document and asking questions or posting suggestions in the appropriate channels are an excellent way to get started on the mutual respect part, greatly increasing the chances of receiving tolerance and encouragement in return.

10.12.2 Mailing Lists

python-dev is the primary mailing list for discussions about Python's development. The list is open to the public and is subscribed to by all core developers plus many people simply interested in following Python's development. Discussion is focused on issues related to Python's development, such as how to handle a specific issue, a PEP, etc.

- Ideas about new functionality should **not** start here and instead should be sent to python-ideas.
- Technical support questions should also not be asked here and instead should go to python-list or python-help.

Python-ideas is a mailing list open to the public to discuss ideas on changing Python. If a new idea does not start here (or python-list, discussed below), it will get redirected here.

Sometimes people post new ideas to python-list to gather community opinion before heading to python-ideas. The list is also sometimes known as comp.lang.python, the name of the newsgroup it mirrors (it is also known by the abbreviation c.l.py).

The python-committers mailing list is a private mailing list for core developers (the archives are publicly available). If something only affects core developers (e.g., the tree is frozen for commits, etc.), it is discussed here instead of python-dev to keep traffic down on the latter.

python-dev, python-committers, and python-ideas all use Mailman 3, and are hence accessible via the Mailman 3 web gateway.

Python-checkins sends out an email for every commit to Python's various repositories from https://github.com/python/cpython. All core developers subscribe to this list and are known to reply to these emails to make comments about various issues they catch in the commit. Replies get redirected to python-dev.

There are two mailing lists related to issues on the issue tracker. If you only want an email for when a new issue is open, subscribe to new-bugs-announce. If you would rather receive an email for all changes made to any issue, subscribe to python-bugs-list.

General Python questions should go to python-list or tutor or similar resources, such as StackOverflow or the `#python` IRC channel on Libera.Chat.

The core-workflow issue tracker is the place to discuss and work on improvements to the CPython core development workflow.

A complete list of Python mailing lists can be found at https://mail.python.org/mailman/listinfo. Most lists are also mirrored at GMANE and can be read and posted to in various ways, including via web browsers, NNTP newsreaders, and RSS feed readers.

10.12.3 Discourse (discuss.python.org web forum)

We have our own Discourse forum for both developers and users. This forum complements the python-dev, python-ideas, python-help, and python-list mailing lists.

This forum has different categories and most core development discussions take place in the open forum categories for PEPs and Core Development (these are the Discourse equivalents to the python-dev mailing list). All categories are open for users to read and post with the exception of the Committers category, where posting is restricted to the CPython core developers.

The Committers category is often used for announcements and notifications. It is also a common venue for the core developer promotion votes (this category is equivalent to the python-committers mailing list).

Tutorials for new users

To start a topic or participate in any discussions in the forum, sign up and create an account using an email address or GitHub account. You can do so by clicking the "Sign Up" button on the top right hand corner of the Discourse main page.

The Python Discourse Quick Start compiled by Carol Willing gives you a quick overview on how to kick off Python Discourse.

We recommend new users getting familiarised with the forum by going through Discobot tutorials. These tutorials can be activated by replying to a welcome message from "discourse Greetings!" received under Notifications and Messages in your user account.

- Click on your personal account found on the top right hand corner of the page.

- The dropdown menu will show four different icons: (Notifications), (Bookmarks), (Messages), and (Preferences).
- Select either Notifications or Messages.
- Open the "Greetings!" message sent by Discobot to start the tutorial.

Ensure that you read through the Python Code of Conduct. We are to be open, considerate and respectful to all users in the community. You can report messages that don't respect the CoC by clicking on the three dots under the message and then on the icon. You can also mention the @staff, @moderators, or @admins groups in a message.

Reading topics

Click a topic title and read down the list of replies in chronological order, following links or previewing replies and quotes as you go. Use your mouse to scroll the screen, or use the timeline scroll bar on the right which also shows you how far through the conversation you've read. On smaller screens, select the bottom progress bar to expand it.

Notifications

Following categories (Category notifications)

Notifications can be set for individual categories and topics. To change any of these defaults, you can either go to your user preferences, or visit the category page, and use the notification button above the topic list, on the top right hand corner of the category page beside the "+ New Topic" button.

Clicking on the Notification control will show a drop-down panel with 5 different options: Watching, Tracking, Watching First Post, Normal, and Muted. All categories are set by default in Normal mode where you will only be notified if someone mentions your @name or replies to you.

Following individual threads (Topic notifications)

To follow any individual topics or threads, you can adjust your notifications through the notification button found on the right of the topic at the end of the timeline. You can also do so at the bottom of each topic. Select "Watching" and you will be notified when there is any new updated reply from that particular thread.

Customising notifications on user preference

To get a bird's eye view of all your customised notifications, you can go to Preferences of your account. This allows you to make adjustments according to categories, users, and tags.

Enabling mailing list mode

In mailing list mode, you will receive one email per post, as happens with traditional mailing lists. This is desirable if you prefer to interact via email, without visiting the forum website. To activate the mailing list mode, go to the email preferences, check "Enable mailing list mode" and save changes.

10.12.4 Discord (private chat server)

For more real-time discussions, the core development team have a private Discord server available. Core developers, Steering Council members, triagers, and documentarians on the project are eligible to join the server. Joining the Discord server is entirely optional, as all essential communications occur on the mailing lists and Discourse forums.

For core developers, a long lived multiple use invitation link for this server can be found in the private core developer only section of the Discourse forum.

For triagers and documentarians joining the Discord server, a single use invitation link should be generated and sent to them directly.

When first joining the server, new users will only have access to the `#welcome` and `#rules-and-info` channels. To link their Discord ID with their project role, core developers may update their Steering Council voter record with their Discord ID before posting in the `#welcome` channel to request access to the rest of the server channels. Triagers, documentarians, and core developers that would prefer not to add their Discord ID to their Steering Council voter record may instead be vouched for by an existing member of the Discord server.

As a private, non-archived, forum, final decisions on design and development questions should not be made on Discord. Any conclusions from Discord discussions should be summarised and posted to the issue tracker, Discourse forum, or mailing list (the appropriate venue for sharing conclusions will depend on the specific discussion).

Note: existing Discord users may want to right click on their username in the automatic Discord welcome message and choose "Edit Server Profile" in order to set a specific Server Nickname

10.12.5 IRC

Some core developers still participate in the `#python-dev` IRC channel on `irc.libera.chat`. This is not a place to ask for help with Python, but to discuss issues related to Python's own development. See also the `#python-dev-notifs` channel for bots notifications.

10.12.6 Blogs

Several core developers are active bloggers and discuss Python's development that way. You can find their blogs (and various other developers who use Python) at https://planetpython.org/.

10.12.7 Setting Expectations for Open Source Participation

Burn-out is common in open source due to a misunderstanding of what users, contributors, and maintainers should expect from each other. Brett Cannon gave a talk about this topic that sets out to help everyone set reasonable expectations of each other in order to make open source pleasant for everyone involved.

10.12.8 Additional Repositories

Python Core Workflow hosts the codebase for tools such as cherry_picker and blurb.

Python Performance Benchmark project is intended to be an authoritative source of benchmarks for all Python implementations.

10.13 Porting Python to a new platform

The first step is to familiarize yourself with the development toolchain on the platform in question, notably the C compiler. Make sure you can compile and run a hello-world program using the target compiler.

Next, learn how to compile and run the Python interpreter on a platform to which it has already been ported; preferably Unix, but Windows will do, too. The build process for Python, in particular the `Makefile` in the source distribution, will give you a hint on which files to compile for Python. Not all source files are relevant: some are platform specific, others are only used in emergencies (e.g. `getopt.c`).

It is not recommended to start porting Python without at least medium-level understanding of your target platform; i.e. how it is generally used, how to write platform specific apps, etc. Also, some Python knowledge is required, or you will be unable to verify that your port is working correctly.

You will need a `pyconfig.h` file tailored for your platform. You can start with `pyconfig.h.in`, read the comments, and turn on definitions that apply to your platform. Also, you will need a `config.c` file, which lists the built-in modules you support. Again, starting with `Modules/config.c.in` is recommended.

Finally, you will run into some things that are not supported on your target platform. Forget about the `posix` module in the beginning. You can simply comment it out of the `config.c` file.

Keep working on it until you get a >>> prompt. You may have to disable the importing of `site.py` by passing the `-S` option. When you have a prompt, bang on it until it executes very simple Python statements.

At some point you will want to use the `os` module; this is the time to start thinking about what to do with the `posix` module. It is okay to simply comment out functions in the `posix` module that cause problems; the remaining ones will be quite useful.

Before you are done, it is highly recommended to run the Python regression test suite, as described in *Running & Writing Tests*.

10.14 How to Become a Core Developer

10.14.1 What it Takes

When you have consistently contributed patches which meet quality standards without requiring extensive rewrites prior to being committed, you may qualify for commit privileges and become a core developer of Python. You must also work well with other core developers (and people in general) as you become an ambassador for the Python project.

Typically a core developer will offer you the chance to gain commit privilege. The person making the offer will become your mentor and watch your commits for a while to make sure you understand the development process. If other core developers agree that you should gain commit privileges you are then extended an official offer. How core developers come to that agreement are outlined in **PEP 13**.

10.14.2 What it Means

As contributors to the CPython project, our shared responsibility is to collaborate constructively with other contributors, including core developers. This responsibility covers all forms of contribution, whether that's submitting patches to the implementation or documentation, reviewing other peoples' patches, triaging issues on the issue tracker, or discussing design and development ideas on the core mailing lists.

Core developers accept key additional responsibilities around the ongoing management of the project:

- core developers bear the additional responsibility of handling the consequences of accepting a change into the code base or documentation. That includes reverting or fixing it if it causes problems in the Buildbot fleet or someone spots a problem in post-commit review, as well as helping out the release manager in resolving any problems found during the pre-release testing cycle. While all contributors are free to help out with this part of the process, and it is most welcome when they do, the actual responsibility rests with the core developer that merged the change

- core developers also bear the primary responsibility for deciding when changes proposed on the issue tracker should be escalated to python-ideas or python-dev for wider discussion, as well as suggesting the use of the Python Enhancement Proposal process to manage the design and justification of complex changes, or changes with a potentially significant impact on end users

As a result of the additional responsibilities they accept, core developers gain the privilege of being able to approve proposed changes, as well as being able to reject them as inappropriate. Core developers are also able to request that even already merged changes be escalated to python-dev for further discussion, and potentially even reverted prior to release.

Becoming a core developer isn't a binary "all-or-nothing" status - CPython is a large project, and different core developers accept responsibility for making design and development decisions in different areas (as documented in the *Experts Index* and *Developer Log*).

10.14.3 Gaining Commit Privileges

The steps to gaining commit privileges are:

1. A core developer starts a poll at https://discuss.python.org/c/committers/5
   - Open for 7 days
   - Results shown upon close
2. The poll is announced on python-committers
3. Wait for the poll to close and see if the results confirm your membership as per the voting results required by PEP 13
4. The person who nominated you emails the steering council with your email address and a request that the council either accept or reject the proposed membership
5. Assuming the steering council does not object, a member of the council will email you asking for:
   - Account details as required by https://github.com/python/voters/
   - Your preferred email address to subscribe to python-committers with
   - A reminder about the Code of Conduct and to report issues to the PSF Conduct WG
6. Once you have provided the pertinent details, your various new privileges will be turned on
7. Your details will be added to https://github.com/python/voters/
8. They will update the devguide to publicly list your team membership at *Developer Log*
9. An announcement email by the steering council member handling your new membership will be sent to python-committers

Mailing Lists

You are expected to subscribe to python-committers, python-dev, python-checkins, and one of new-bugs-announce or python-bugs-list. See *Following Python's Development* for links to these mailing lists.

Sign a Contributor Agreement

Submitting a contributor form for Python licenses any code you contribute to the Python Software Foundation. While you retain the copyright, giving the PSF the ability to license your code means it can be put under the PSF license so it can be legally distributed with Python.

This is a very important step! Hopefully you have already submitted a contributor agreement if you have been submitting patches. But if you have not done this yet, it is best to do this ASAP, probably before you even do your first commit so as to not forget. Also do not forget to enter your GitHub username into your details on the issue tracker.

Pull Request merging

Once you have your commit privileges on GitHub you will be able to accept pull requests on GitHub. You should plan to continue to submit your own changes through pull requests as if you weren't a core developer to benefit from various things such as automatic integration testing, but you can accept your own pull requests if you feel comfortable doing so.

10.14.4 Responsibilities

As a core developer, there are certain things that are expected of you.

First and foremost, be a good person. This might sound melodramatic, but you are now a member of the Python project and thus represent the project and your fellow core developers whenever you discuss Python with anyone. We have a reputation for being a very nice group of people and we would like to keep it that way. Core developers responsibilities include following the PSF Code of Conduct.

Second, please be prompt in responding to questions. Many contributors to Python are volunteers so what little free time they can dedicate to Python should be spent being productive. If you have been asked to respond to an issue or answer a question and you put it off it ends up stalling other people's work. It is completely acceptable to say you are too busy, but you need to say that instead of leaving people waiting for an answer. This also applies to anything you do on the issue tracker.

Third, please list what areas you want to be considered an expert in the *Experts Index*. This allows triagers to direct issues to you which involve an area you are an expert in. But, as stated in the second point above, if you do not have the time to answer questions promptly then please remove yourself as needed from the file so that you will not be bothered in the future. Once again, we all understand how life gets in the way, so no one will be insulted if you remove yourself from the list.

Fourth, please consider whether or not you wish to add your name to the *Core Developer Motivations and Affiliations* list. Core contributor participation in the list helps the wider Python community to better appreciate the perspectives currently represented amongst the core development team, the Python Software Foundation to better assess the sustainability of current contributions to CPython core development, and also serves as a referral list for organisations seeking commercial Python support from the core development community.

And finally, enjoy yourself! Contributing to open source software should be fun (overall). If you find yourself no longer enjoying the work then either take a break or figure out what you need to do to make it enjoyable again.

10.15 Developer Log

This page lists the historical members of the Python development team. (The master list is kept in a private repository due to containing sensitive contact information.)

Name	GitHub username	Joined	Left	Notes
Erlend Egeberg Aasland	erlend-aasland	2022-05-05		
Jelle Zijlstra	JelleZijlstra	2022-02-15		
Dennis Sweeney	sweeneyde	2022-02-02		
Ken Jin	Fidget-Spinner	2021-08-26		
Ammar Askar	ammaraskar	2021-07-30		
Irit Katriel	iritkatriel	2021-05-10		
Batuhan Taskaya	isidentical	2020-11-08		
Brandt Bucher	brandtbucher	2020-09-14		
Lysandros Nikolaou	lysnikolaou	2020-06-29		
Kyle Stanley	aeros	2020-04-14		
Dong-hee Na	corona10	2020-04-08		
Karthikeyan Singaravelan	tirkarthi	2019-12-31		
Joannah Nanjekye	nanjekyejoannah	2019-09-23		
Abhilash Raj	maxking	2019-08-06		
Paul Ganssle	pganssle	2019-06-15		
Stéphane Wirtel	matrixise	2019-04-08		
Stefan Behnel	scoder	2019-04-08		
Cheryl Sabella	csabella	2019-02-19		
Lisa Roach	lisroach	2018-09-14		
Emily Morehouse	emilyemorehouse	2018-09-14		
Pablo Galindo	pablogsal	2018-06-06		
Mark Shannon	markshannon	2018-05-15		
Petr Viktorin	encukou	2018-04-16		
Nathaniel J. Smith	njsmith	2018-01-25		
Julien Palard	JulienPalard	2017-12-08		
Ivan Levkivskyi	ilevkivskyi	2017-12-06		
Carol Willing	willingc	2017-05-24		
Mariatta	Mariatta	2017-01-27		
Xiang Zhang	zhangyangyu	2016-11-21		
Inada Naoki	methane	2016-09-26		
Xavier de Gaye	xdegaye	2016-06-03	2018-01-25	Privileges relinquished on 2018-01-25
Davin Potts	applio	2016-03-06		
Martin Panter	vadmium	2015-08-10	2020-11-26	
Paul Moore	pfmoore	2015-03-15		
Robert Collins	rbtcollins	2014-10-16		To work on unittest
Berker Peksağ	berkerpeksag	2014-06-26		
Steve Dower	zooba	2014-05-10		
Kushal Das	kushaldas	2014-04-14		
Steven D'Aprano	stevendaprano	2014-02-08		For statistics module
Yury Selivanov	1st1	2014-01-23		
Zachary Ware	zware	2013-11-02		

continues on next page

Table 1 – continued from previous page

Name	GitHub username	Joined	Left	Notes
Donald Stufft	dstufft	2013-08-14		
Ethan Furman	ethanfurman	2013-05-11		
Serhiy Storchaka	serhiy-storchaka	2012-12-26		
Chris Jerdonek	cjerdonek	2012-09-24		
Eric Snow	ericsnowcurrently	2012-09-05		
Peter Moody		2012-05-20	2017-02-10	For ipaddress module; did not make GitHub transition
Hynek Schlawack	hynek	2012-05-14		
Richard Oudkerk		2012-04-29	2017-02-10	For multiprocessing module; did not make GitHub transition
Andrew Svetlov	asvetlov	2012-03-13		At PyCon sprint
Petri Lehtinen	akheron	2011-10-22	2020-11-12	
Meador Inge	meadori	2011-09-19	2020-11-26	
Jeremy Kloth	jkloth	2011-09-12		
Sandro Tosi	sandrotosi	2011-08-01		
Alex Gaynor	alex	2011-07-18		For PyPy compatibility (since expanded scope)
Charles-François Natali		2011-05-19	2017-02-10	Did not make GitHub transition
Nadeem Vawda		2011-04-10	2017-02-10	Did not make GitHub transition
Jason R. Coombs	jaraco	2011-03-14		For sprinting on distutils2
Ross Lagerwall		2011-03-13	2017-02-10	Did not make GitHub transition
Eli Bendersky	eliben	2011-01-11	2020-11-26	
Ned Deily	ned-deily	2011-01-09		
David Malcolm	davidmalcolm	2010-10-27	2020-11-12	relinquished privileges on 2020-11-12
Tal Einat	taleinat	2010-10-04		Initially for IDLE
Łukasz Langa	ambv	2010-09-08		
Daniel Stutzbach		2010-08-22	2017-02-10	Did not make GitHub transition
Éric Araujo	merwok	2010-08-10		
Brian Quinlan	brianquinlan	2010-07-26		For work related to PEP 3148
Alexander Belopolsky	abalkin	2010-05-25		
Tim Golden	tjguk	2010-04-21		
Giampaolo Rodolà	giampaolo	2010-04-17		
Jean-Paul Calderone		2010-04-06	2017-02-10	Did not make GitHub transition
Brian Curtin	briancurtin	2010-03-24		
Florent Xicluna		2010-02-25	2017-02-10	Did not make GitHub transition

continues on next page

Table 1 – continued from previous page

Name	GitHub username	Joined	Left	Notes
Dino Viehland	DinoV	2010-02-23		For IronPython compatibility
Larry Hastings	larryhastings	2010-02-22		
Victor Stinner	vstinner	2010-01-30		
Stefan Krah	skrah	2010-01-05	2020-10-07	For the decimal module
Doug Hellmann	dhellmann	2009-09-20	2020-11-11	For documentation; relinquished privileges on 2020-11-11
Frank Wierzbicki		2009-08-02	2017-02-10	For Jython compatibility; did not make GitHub transition
Ezio Melotti	ezio-melotti	2009-06-07		For documentation
Philip Jenvey	pjenvey	2009-05-07	2020-11-26	For Jython compatibility
Michael Foord	voidspace	2009-04-01		For IronPython compatibility
R. David Murray	bitdancer	2009-03-30		
Chris Withers	cjw296	2009-03-08		
Tarek Ziadé	tarekziade	2008-12-21	2017-02-10	For distutils module
Hirokazu Yamamoto		2008-08-12	2017-02-10	For Windows build; did not make GitHub transition
Armin Ronacher	mitsuhiko	2008-07-23	2020-11-26	For documentation toolset and ast module
Antoine Pitrou	pitrou	2008-07-16		
Senthil Kumaran	orsenthil	2008-06-16		For GSoC
Jesse Noller		2008-06-16	2017-02-10	For multiprocessing module; did not make GitHub transition
Jesús Cea	jcea	2008-05-13		For bsddb module
Guilherme Polo		2008-04-24	2017-02-10	Did not make GitHub transition
Jeroen Ruigrok van der Werven		2008-04-12	2017-02-10	For documentation; did not make GitHub transition
Benjamin Peterson	benjaminp	2008-03-25		For bug triage
David Wolever	wolever	2008-03-17	2020-11-21	For 2to3 module
Trent Nelson	tpn	2008-03-17	2020-11-26	
Mark Dickinson	mdickinson	2008-01-06		For maths-related work
Amaury Forgeot d'Arc	amauryfa	2007-11-09	2020-11-26	
Christian Heimes	tiran	2007-10-31		
Bill Janssen		2007-08-28	2017-02-10	For ssl module; did not make GitHub transition

continues on next page

Table 1 – continued from previous page

Name	GitHub username	Joined	Left	Notes
Jeffrey Yasskin		2007-08-09	2017-02-10	Did not make GitHub transition
Mark Summerfield		2007-08-01	2017-02-10	For documentation; did not make GitHub transition
Alexandre Vassalotti	avassalotti	2007-05-21	2020-11-12	For GSoC
Travis E. Oliphant		2007-04-17	2017-02-10	Did not make GitHub transition
Eric V. Smith	ericvsmith	2007-02-28		For PEP 3101 in a sandbox
Josiah Carlson	josiahcarlson	2007-01-06	2017-02-10	For asyncore and asynchat modules
Collin Winter		2007-01-05	2017-02-10	For PEP access; did not make GitHub transition
Richard Jones		2006-05-23	2017-02-10	For Need for Speed sprint; did not make GitHub transition
Kristján Valur Jónsson		2006-05-17	2017-02-10	For Need for Speed sprint; did not make GitHub transition
Jack Diederich	jackdied	2006-05-17	2020-11-26	For Need for Speed sprint
Steven Bethard		2006-04-27	2017-02-10	For PEP access and SourceForge maintenance; did not make GitHub transition
Gerhard Häring		2006-04-23	2017-02-10	Did not make the GitHub transition
George Yoshida		2006-04-17	2017-02-10	For tracker administration; did not make GitHub transition
Ronald Oussoren	ronaldoussoren	2006-03-03		For Mac-related work
Nick Coghlan	ncoghlan	2005-10-16		
Georg Brandl	birkenfeld	2005-05-28		
Terry Jan Reedy	terryjreedy	2005-04-07		
Bob Ippolito	etrepum	2005-03-02	2017-02-10	For Mac-related work
Peter Astrand		2004-10-21	2017-02-10	Did not make GitHub transition
Facundo Batista	facundobatista	2004-10-16		
Sean Reifschneider		2004-09-17	2017-02-10	Did not make GitHub transition
Johannes Gijsbers		2004-08-14	2005-07-27	Privileges relinquished on 2005-07-27
Matthias Klose	doko42	2004-08-04		

continues on next page

Table 1 – continued from previous page

Name	GitHub username	Joined	Left	Notes
PJ Eby	pjeby	2004-03-24	2020-11-26	
Vinay Sajip	vsajip	2004-02-20		
Hye-Shik Chang		2003-12-10	2017-02-10	Did not make GitHub transition
Armin Rigo		2003-10-24	2012-06-01	Privileges relinquished in 2012
Andrew McNamara		2003-06-09	2017-02-10	Did not make GitHub transition
Samuele Pedroni		2003-05-16	2017-02-10	Did not make GitHub transition
Alex Martelli	aleaxit	2003-04-22		
Brett Cannon	brettcannon	2003-04-18		
David Goodger		2003-01-02	2017-02-10	Did not make GitHub transition
Gustavo Niemeyer		2002-11-05	2017-02-10	Did not make GitHub transition
Tony Lownds		2002-09-22	2017-02-10	Did not make GitHub transition
Steve Holden		2002-06-14	2017-02-10	**Relinquished privileges on 2005-04-07,** but granted again for Need for Speed sprint; did not make GitHub transition
Christian Tismer	ctismer	2002-05-17		For Need for Speed sprint
Jason Tishler		2002-05-15	2017-02-10	Did not make GitHub transition
Walter Dörwald	doerwalter	2002-03-21		
Andrew MacIntyre		2002-02-17	2016-01-02	Privileges relinquished 2016-01-02
Gregory P. Smith	gpshead	2002-01-08		
Anthony Baxter		2001-12-21	2017-02-10	Did not make GitHub transition
Neal Norwitz		2001-12-19	2017-02-10	Did not make GitHub transition
Raymond Hettinger	rhettinger	2001-12-10		
Chui Tey		2001-10-31	2017-02-10	Did not make GitHub transition
Michael W. Hudson		2001-08-27	2017-02-10	Did not make GitHub transition
Finn Bock		2001-08-23	2005-04-13	Privileges relinquished on 2005-04-13

continues on next page

Table 1 – continued from previous page

Name	GitHub username	Joined	Left	Notes
Piers Lauder		2001-07-20	2017-02-10	Did not make GitHub transition
Kurt B. Kaiser	kbkaiser	2001-07-03		
Steven M. Gava		2001-06-25	2017-02-10	Did not make GitHub transition
Steve Purcell		2001-03-22	2017-02-10	Did not make GitHub transition
Jim Fulton		2000-10-06	2017-02-10	Did not make GitHub transition
Ka-Ping Yee		2000-10-03	2017-02-10	Did not make GitHub transition
Lars Gustäbel	gustaebel	2000-09-21	2020-11-26	For tarfile module
Neil Schemenauer	nascheme	2000-09-15		
Martin v. Löwis		2000-09-08	2017-02-10	Did not make GitHub transition
Thomas Heller	theller	2000-09-07	2020-11-18	
Moshe Zadka		2000-07-29	2005-04-08	Privileges relinquished on 2005-04-08
Thomas Wouters	Yhg1s	2000-07-14		
Peter Schneider-Kamp		2000-07-10	2017-02-10	Did not make GitHub transition
Paul Prescod		2000-07-01	2005-04-30	Privileges relinquished on 2005-04-30
Tim Peters	tim-one	2000-06-30		
Skip Montanaro	smontanaro	2000-06-30	2015-04-21	Privileges relinquished 2015-04-21
Fredrik Lundh		2000-06-29	2017-02-10	Did not make GitHub transition
Mark Hammond	mhammond	2000-06-09		
Marc-André Lemburg	malemburg	2000-06-07		
Trent Mick		2000-06-06	2017-02-10	Did not make GitHub transition
Eric S. Raymond		2000-06-02	2017-02-10	Did not make GitHub transition
Greg Stein		1999-11-07	2017-02-10	Did not make GitHub transition
Just van Rossum		1999-01-22	2017-02-10	Did not make GitHub transition
Greg Ward		1998-12-18	2017-02-10	Did not make GitHub transition
Andrew Kuchling	akuchling	1998-04-09		
Ken Manheimer		1998-03-03	2005-04-08	Privileges relinquished on 2005-04-08
Jeremy Hylton	jeremyhylton	1997-08-13	2020-11-26	
Roger E. Masse		1996-12-09	2017-02-10	Did not make GitHub transition

continues on next page

Table 1 – continued from previous page

Name	GitHub username	Joined	Left	Notes
Fred Drake	freddrake	1996-07-23		
Barry Warsaw	warsaw	1994-07-25		
Jack Jansen	jackjansen	1992-08-13		
Sjoerd Mullender	sjoerdmullender	1992-08-04	2020-11-14	
Guido van Rossum	gvanrossum	1989-12-25		

10.15.1 Procedure for Granting or Dropping Access

To be granted the ability to manage who is a committer, you must be a team maintainer of the Python core team on GitHub. Once you have that privilege you can add people to the team. They will be asked to accept the membership which they can do by visiting https://github.com/python and clicking on the appropriate button that will be displayed to them in the upper part of the page.

10.16 Accepting Pull Requests

This page is a step-by-step guide for core developers who need to assess, merge, and possibly backport a pull request on the main repository.

10.16.1 Assessing a pull request

Before you can accept a pull request, you need to make sure that it is ready to enter the public source tree. Ask yourself the following questions:

- **Are there ongoing discussions at the issue tracker?**
  Read the linked issue. If there are ongoing discussions, then we need to have a resolution there before we can merge the pull request.
- **Was the pull request first made against the appropriate branch?**
  The only branch that receives new features is `main`, the in-development branch. Pull requests should only target bug-fix branches if an issue appears in only that version and possibly older versions.
- **Are the changes acceptable?**
  If you want to share your work-in-progress code on a feature or bugfix, then you can open a `WIP`-prefixed pull request, publish patches on the issue tracker, or create a public fork of the repository.
- **Do the checks on the pull request show that the test suite passes?**
  Make sure that all of the status checks are passing.
- **Is the patch in a good state?**
  Check *Lifecycle of a Pull Request* and *Helping Triage Issues* to review what is expected of a patch.
- **Does the patch break backwards-compatibility without a strong reason?**
  Run the entire test suite to make sure that everything still passes. If there is a change to the semantics, then there needs to be a strong reason, because it will cause some peoples' code to break. If you are unsure if the breakage is worth it, then ask on python-dev.
- **Does documentation need to be updated?**
  If the pull request introduces backwards-incompatible changes (e.g. deprecating or removing a feature), then make sure that those changes are reflected in the documentation before you merge the pull request.
- **Were appropriate labels added to signify necessary backporting of the pull request?**
  If it is determined that a pull request needs to be backported into one or more of the maintenance branches,

then a core developer can apply the label `needs backport to X.Y` to the pull request. Once the backport pull request has been created, remove the `needs backport to X.Y` label from the original pull request. (Only core developers and members of the Python Triage Team can apply labels to GitHub pull requests).

- **Does the pull request pass a check indicating that the submitter has signed the CLA?**
  Make sure that the contributor has signed a Contributor Licensing Agreement (CLA), unless their change has no possible intellectual property associated with it (e.g. fixing a spelling mistake in documentation). The CPython CLA Bot checks whether the author has signed the CLA, and replies in the PR if they haven't. For further questions about the CLA process, write to contributors@python.org.
- **Were `What's New in Python` and `Misc/NEWS.d/next` updated?**
  If the change is particularly interesting for end users (e.g. new features, significant improvements, or backwards-incompatible changes), then an entry in the `What's New in Python` document (in `Doc/whatsnew/`) should be added as well. Changes that affect only documentation generally do not require a `NEWS` entry. (See the following section for more information.)

10.16.2 Updating NEWS and What's New in Python

Almost all changes made to the code base deserve an entry in `Misc/NEWS.d`. If the change is particularly interesting for end users (e.g. new features, significant improvements, or backwards-incompatible changes), then an entry in the `What's New in Python` document (in `Doc/whatsnew/`) should be added as well. Changes that affect documentation only generally do not require a `NEWS` entry.

There are two notable exceptions to this general principle, and they both relate to changes that:

- Already have a `NEWS` entry
- Have not yet been included in any formal release (including alpha and beta releases)

These are the two exceptions:

1. **If a change is reverted prior to release**, then the corresponding entry is simply removed. Otherwise, a new entry must be added noting that the change has been reverted (e.g. when a feature is released in an alpha and then cut prior to the first beta).
2. **If a change is a fix (or other adjustment) to an earlier unreleased change and the original `NEWS` entry remains valid**, then no additional entry is needed.

If a change needs an entry in `What's New in Python`, then it is very likely not suitable for including in a maintenance release.

`NEWS` entries go into the `Misc/NEWS.d` directory as individual files. The `NEWS` entry can be created by using blurb-it, or the blurb tool and its `blurb add` command.

If you are unable to use the tool, then you can create the `NEWS` entry file manually. The `Misc/NEWS.d` directory contains a sub-directory named `next`, which contains various sub-directories representing classifications for what was affected (e.g. `Misc/NEWS.d/next/Library` for changes relating to the standard library). The file name itself should be in the format `<datetime>.gh-issue-<issue-number>.<nonce>.rst`:

- `<datetime>` is today's date joined with a hyphen (`-`) to the current time, in the `YYYY-MM-DD-hh-mm-ss` format (e.g. `2017-05-27-16-46-23`).
- `<issue-number>` is the issue number the change is for (e.g. `12345` for `gh-issue-12345`).
- `<nonce>` is a unique string to guarantee that the file name is unique across branches (e.g. `Yl4gI2`). It is typically six characters long, but it can be any length of letters and numbers. Its uniqueness can be satisfied by typing random characters on your keyboard.

As a result, a file name can look something like `Misc/NEWS.d/next/Library/2017-05-27-16-46-23.gh-issue-12345.Yl4gI2.rst`.

The contents of a `NEWS` file should be valid reStructuredText. An 80 character column width should be used. There is no indentation or leading marker in the file (e.g. `-`). There is also no need to start the entry with the issue number since it is part of the file name. You can use *inline markups* too. Here is an example of a `NEWS` entry:

```
Fix warning message when :func:`os.chdir` fails inside
:func:`test.support.temp_cwd`. Patch by Chris Jerdonek.
```

The inline Sphinx roles like `:func:` can be used help readers find more information. You can build HTML and verify that the link target is appropriate by using *make html*.

While Sphinx roles can be beneficial to readers, they are not required. Inline ``` ``code blocks`` ``` can be used instead.

10.16.3 Working with Git

See also:

Git Bootcamp and Cheat Sheet

As a core developer, you have the ability to push changes to the official Python repositories, so you need to be careful with your workflow:

- **You should not push new branches to the main repository.** You can still use them in the fork that you use for the development of patches. You can also push these branches to a separate public repository for maintenance work before it is integrated into the main repository.
- **You should not commit directly into the `main` branch, or any of the maintenance branches.** You should commit against your own feature branch, and then create a pull request.
- **For a small change, you can make a quick edit through the GitHub web UI.** If you choose to use the web UI, be aware that GitHub will create a new branch in the main CPython repository rather than in your fork. Delete this newly created branch after it has been merged into the `main` branch or any of the maintenance branches. To keep the CPython repository tidy, remove the new branch within a few days.

Keep a fork of the main repository, since it will allow you to revert all local changes (even committed ones) if you're not happy with your local clone.

Seeing active branches

If you use `git branch`, then you will see a *list of branches*. The only branch that receives new features is `main`, the in-development branch. The other branches receive only bug fixes or security fixes.

Backporting changes to an older version

If it is determined that a pull request needs to be backported into one or more of the maintenance branches, then a core developer can apply the label `needs backport to X.Y` to the pull request.

After the pull request has been merged, miss-islington (bot) will first try to do the backport automatically. If miss-islington is unable to do it, then the pull request author or the core developer who merged it should look into backporting it themselves, using the backport generated by cherry_picker.py as a starting point.

You can get the commit hash from the original pull request, or you can use `git log` on the `main` branch. To display the 10 most recent commit hashes and their first line of the commit, use the following command:

```
git log -10 --oneline
```

You can prefix the backport pull request with the branch, and reference the pull request number from `main`. Here is an example:

```
[3.9] gh-12345: Fix the Spam Module (GH-NNNN)
```

Here "gh-12345" is the GitHub *issue* number, and "GH-NNNN" is the number of the original *pull request*. Note that cherry_picker.py adds the branch prefix automatically.

Once the backport pull request has been created, remove the `needs backport to X.Y` label from the original pull request. (Only core developers and members of the Python Triage Team can apply labels to GitHub pull requests).

Reverting a merged pull request

To revert a merged pull request, press the `Revert` button at the bottom of the pull request. That will bring up the page to create a new pull request where the commit can be reverted. It will also create a new branch on the main CPython repository. Delete the branch once the pull request has been merged.

Always include the reason for reverting the commit to help others understand why it was done. The reason should be included as part of the commit message. Here is an example:

```
Revert gh-NNNN: Fix Spam Module (GH-111)

Reverts python/cpython#111.
Reason: This commit broke the buildbot.
```

10.17 Development Cycle

The responsibilities of a core developer shift based on what kind of branch of Python a developer is working on and what stage the branch is in.

To clarify terminology, Python uses a `major.minor.micro` nomenclature for production-ready releases. So for Python 3.1.2 final, that is a *major version* of 3, a *minor version* of 1, and a *micro version* of 2.

- new *major versions* are exceptional; they only come when strongly incompatible changes are deemed necessary, and are planned very long in advance;
- new *minor versions* are feature releases; they get released annually, from the current *in-development* branch;
- new *micro versions* are bugfix releases; they get released roughly every 2 months; they are prepared in *maintenance* branches.

We also publish non-final versions which get an additional qualifier: *Alpha*, *Beta*, *release candidate*. These versions are aimed at testing by advanced users, not production use.

Each release of Python is tagged in the source repo with a tag of the form `vX.Y.ZTN`, where `X` is the major version, `Y` is the minor version, `Z` is the micro version, `T` is the release level (`a` for alpha releases, `b` for beta, `rc` release candidate, and *null* for final releases), and `N` is the release serial number. Some examples of release tags: `v3.7.0a1`, `v3.6.3`, `v2.7.14rc1`.

10.17.1 Branches

There is a branch for each *feature version*, whether released or not (e.g. 3.7, 3.8).

In-development (main) branch

The `main` branch is the branch for the next feature release; it is under active development for all kinds of changes: new features, semantic changes, performance improvements, bug fixes.

At some point during the life-cycle of a release, a new *maintenance branch* is created to host all bug fixing activity for further micro versions in a feature version (3.8.1, 3.8.2, etc.).

For versions 3.4 and before, this was conventionally done when the final release was cut (for example, 3.4.0 final).

Starting with the 3.5 release, we create the release maintenance branch (e.g. 3.5) at the time we enter beta (3.5.0 beta 1). This allows feature development for the release 3.n+1 to occur within the main branch alongside the beta and release candidate stabilization periods for release 3.n.

Maintenance branches

A branch for a previous feature release, currently being maintained for bug fixes, or for the next feature release in its *beta* or *release candidate* stages. There is usually either one or two maintenance branches at any given time for Python 3.x. After the final release of a new minor version (3.x.0), releases produced from a maintenance branch are called **bugfix** or **maintenance** releases; the terms are used interchangeably. These releases have a **micro version** number greater than zero.

The only changes allowed to occur in a maintenance branch without debate are bug fixes. Also, a general rule for maintenance branches is that compatibility must not be broken at any point between sibling micro releases (3.5.1, 3.5.2, etc.). For both rules, only rare exceptions are accepted and **must** be discussed first.

A new maintenance branch is normally created when the next feature release cycle reaches feature freeze, i.e. at its first beta pre-release. From that point on, changes intended for remaining pre-releases, the final release (3.x.0), and subsequent bugfix releases are merged to that maintenance branch.

Sometime following the final release (3.x.0), the maintenance branch for the previous minor version will go into *security mode*, usually after at least one more bugfix release at the discretion of the release manager. For example, the 3.4 maintenance branch was put into *security mode* after the 3.4.4 bugfix release which followed the release of 3.5.1.

Security branches

A branch less than 5 years old but no longer in bugfix mode is a security branch.

The only changes made to a security branch are those fixing issues exploitable by attackers such as crashes, privilege escalation and, optionally, other issues such as denial of service attacks. Any other changes are **not** considered a security risk and thus not backported to a security branch. You should also consider fixing hard-failing tests in open security branches since it is important to be able to run the tests successfully before releasing.

Commits to security branches are to be coordinated with the release manager for the corresponding feature version, as listed in the *Status of Python branches*. Merging of pull requests to security branches is restricted to release managers. Any release made from a security branch is source-only and done only when actual security patches have been applied to the branch. These releases have a **micro version** number greater than the last **bugfix** release.

End-of-life branches

The code base for a release cycle which has reached end-of-life status is frozen and no longer has a branch in the repo. The final state of the end-of-lifed branch is recorded as a tag with the same name as the former branch, e.g. `3.3` or `2.6`.

For reference, here are the Python versions that most recently reached their end-of-life:

Branch	Schedule	First release	End-of-life	Release manager
3.6	**PEP 494**	2016-12-23	2021-12-23	Ned Deily
3.5	**PEP 478**	2015-09-13	2020-09-30	Larry Hastings
3.4	**PEP 429**	2014-03-16	2019-03-18	Larry Hastings
3.3	**PEP 398**	2012-09-29	2017-09-29	Georg Brandl, Ned Deily (3.3.7+)
3.2	**PEP 392**	2011-02-20	2016-02-20	Georg Brandl
3.1	**PEP 375**	2009-06-27	2012-04-09	Benjamin Peterson
3.0	**PEP 361**	2008-12-03	2009-06-27	Barry Warsaw
2.7	**PEP 373**	2010-07-03	2020-01-01	Benjamin Peterson
2.6	**PEP 361**	2008-10-01	2013-10-29	Barry Warsaw

The latest release for each Python version can be found on the download page.

10.17.2 Stages

Based on what stage the *in-development* version of Python is in, the responsibilities of a core developer change in regards to commits to the VCS.

Pre-alpha

The branch is in this stage when no official release has been done since the latest final release. There are no special restrictions placed on commits, although the usual advice applies (getting patches reviewed, avoiding breaking the buildbots).

Alpha

Alpha releases typically serve as a reminder to core developers that they need to start getting in changes that change semantics or add something to Python as such things should not be added during a *Beta*. Otherwise no new restrictions are in place while in alpha.

Beta

After a first beta release is published, no new features are accepted. Only bug fixes and improvements to documentation and tests can now be committed. This is when core developers should concentrate on the task of fixing regressions and other new issues filed by users who have downloaded the alpha and beta releases.

Being in beta can be viewed much like being in *RC* but without the extra overhead of needing commit reviews.

Please see the note in the *In-development (main) branch* section above for new information about the creation of the 3.5 maintenance branch during beta.

Release Candidate (RC)

A branch preparing for an RC release can only have bugfixes applied that have been reviewed by other core developers. Generally, these issues must be severe enough (e.g. crashes) that they deserve fixing before the final release. All other issues should be deferred to the next development cycle, since stability is the strongest concern at this point.

While the goal is to have no code changes between a RC and a final release, there may be a need for final documentation or test fixes. Any such proposed changes should be discussed first with the release manager.

You **cannot** skip the peer review during an RC, no matter how small! Even if it is a simple copy-and-paste change, **everything** requires peer review from a core developer.

Final

When a final release is being cut, only the release manager (RM) can make changes to the branch. After the final release is published, the full *development cycle* starts again for the next minor version.

10.17.3 Repository Administration

The source code is currently hosted on GitHub in the Python organization.

Organization Repository Policy

Within the Python organization, repositories are expected to fall within these general categories:

1. The reference implementation of Python and related repositories (i.e. CPython)
2. Reference implementations of PEPs (e.g. mypy)
3. Tooling and support around CPython and the language (e.g. python.org repository)
4. PSF-related repositories (e.g. the Code of Conduct)
5. PSF Infrastructure repositories (e.g. the PSF Infrastructure Salt configurations)

For any repository which does not explicitly and clearly fall under one of these categories, permission should be sought from the Python steering council.

Organization Owner Policy

The GitHub Organization Owner role allows for full management of all aspects of the Python organization. Allowing for visibility and management of all aspects at all levels including organization membership, team membership, access control, and merge privileges on all repositories. For full details of the permission levels see GitHub's documentation on Organization permission levels. This role is paramount to the security of the Python Language, Community, and Infrastructure.

The Executive Director of the Python Software Foundation delegates authority on GitHub Organization Owner Status to Ee Durbin - Python Software Foundation Director of Infrastructure. Common reasons for this role are: Infrastructure Staff Membership, Python Software Foundation General Counsel, and Python Software Foundation Staff as fallback.

Inactive or unreachable members may be removed with or without notice. Members who no longer necessitate this level of access will be removed with notice.

Multi-Factor Authentication must be enabled by the user in order to remain an Owner of the Python Organization.

Current Owners

Name	Role	GitHub Username
Benjamin Peterson	Infrastructure Staff	benjaminp
Noah Kantrowitz	Infrastructure Staff	coderanger
Donald Stufft	Infrastructure Staff	dstufft
Ee Durbin	PSF Director of Infrastructure	ewdurbin
Van Lindberg	PSF General Counsel	VanL
Łukasz Langa	CPython Developer in Residence	ambv

Certain actions (blocking spam accounts, inviting new users, adjusting organization-level settings) can only be performed by owners of the Python organization on GitHub. The `@python/organization-owners` team can be mentioned to request assistance from an organization owner.

Repository Administrator Role Policy

The Administrator role on the repository allows for managing all aspects including collaborators, access control, integrations, webhooks, and branch protection. For full details of the permission levels see GitHub's documentation on repository permission levels. Common reasons for this role are: maintenance of Core Developer Workflow tooling, Release Managers for all *in-development*, *maintenance*, and *security mode* releases, and additional Python Core Developers as necessary for redundancy. Occasional temporary administrator access is acceptable as necessary for Core Developer workflow projects.

Inactive or unreachable members may be removed with or without notice. Members who no longer necessitate this level of access will be removed with notice.

Multi-Factor Authentication must be enabled by the user in order to remain an Administrator of the repository.

Current Administrators

Name	Role	GitHub Username
Pablo Galindo	Python 3.10 and 3.11 Release Manager, Maintainer of buildbot.python.org	pablogsal
Łukasz Langa	Python 3.8 and 3.9 Release Manager, PSF CPython Developer in Residence 2021-2022	ambv
Ned Deily	Python 3.6 and 3.7 Release Manager	ned-deily
Larry Hastings	Retired Release Manager (for Python 3.4 and 3.5)	larryhastings
Berker Peksag	Maintainer of bpo-linkify and cpython-emailer-webhook	berkerpeksag
Brett Cannon	Maintainer of bedevere and the-knights-who-say-ni	brettcannon
Ezio Melotti	Maintainer of bugs.python.org GitHub webhook integration	ezio-melotti
Mariatta Wijaya	Maintainer of blurb_it and miss-islington	Mariatta

Repository Release Manager Role Policy

Release Managers for *in-development*, *maintenance*, and *security mode* Python releases are granted Administrator privileges on the repository. Once a release branch has entered *end-of-life*, the Release Manager for that branch is removed as an Administrator and granted sole privileges (out side of repository administrators) to merge changes to that branch.

Multi-Factor Authentication must be enabled by the user in order to retain access as a Release Manager of the branch.

10.18 Continuous Integration

To assert that there are no regressions in the *development and maintenance branches*, Python has a set of dedicated machines (called *buildbots* or *build workers*) used for continuous integration. They span a number of hardware/operating system combinations. Furthermore, each machine hosts several *builders*, one per active branch: when a new change is pushed to this branch on the public GitHub repository, all corresponding builders will schedule a new build to be run as soon as possible.

The build steps run by the buildbots are the following:

- Check out the source tree for the changeset which triggered the build
- Compile Python
- Run the test suite using *strenuous settings*
- Clean up the build tree

It is your responsibility, as a core developer, to check the automatic build results after you push a change to the repository. It is therefore important that you get acquainted with the way these results are presented, and how various kinds of failures can be explained and diagnosed.

10.18.1 In case of trouble

Please read this page in full. If your questions aren't answered here and you need assistance with the buildbots, a good way to get help is to either:

- contact the `python-buildbots@python.org` mailing list where all buildbot worker owners are subscribed; or
- contact the release manager of the branch you have issues with.

10.18.2 Buildbot failures on Pull Requests

The `bedevere-bot` on GitHub will put a message on your merged Pull Request if building your commit on a stable buildbot worker fails. Take care to evaluate the failure, even if it looks unrelated at first glance.

Not all failures will generate a notification since not all builds are executed after each commit. In particular, reference leaks builds take several hours to complete so they are done periodically. This is why it's important for you to be able to check the results yourself, too.

10.18.3 Checking results of automatic builds

There are three ways of visualizing recent build results:

- The Web interface for each branch at https://www.python.org/dev/buildbot/, where the so-called "waterfall" view presents a vertical rundown of recent builds for each builder. When interested in one build, you'll have to click on it to know which changesets it corresponds to. Note that the buildbot web pages are often slow to load, be patient.
- The command-line `bbreport.py` client, which you can get from https://code.google.com/archive/p/bbreport. Installing it is trivial: just add the directory containing `bbreport.py` to your system path so that you can run it from any filesystem location. For example, if you want to display the latest build results on the development ("main") branch, type:

  ```
  bbreport.py -q 3.x
  ```

- The buildbot "console" interface at https://buildbot.python.org/all/ This works best on a wide, high resolution monitor. Clicking on the colored circles will allow you to open a new page containing whatever information about that particular build is of interest to you. You can also access builder information by clicking on the builder status bubbles in the top line.

If you like IRC, having an IRC client open to the #python-dev-notifs channel on irc.libera.chat is useful. Any time a builder changes state (last build passed and this one didn't, or vice versa), a message is posted to the channel. Keeping an eye on the channel after pushing a changeset is a simple way to get notified that there is something you should look in to.

Some buildbots are much faster than others. Over time, you will learn which ones produce the quickest results after a build, and which ones take the longest time.

Also, when several changesets are pushed in a quick succession in the same branch, it often happens that a single build is scheduled for all these changesets.

10.18.4 Stability

A subset of the buildbots are marked "stable". They are taken into account when making a new release. The rule is that all stable builders must be free of persistent failures when the release is cut. It is absolutely **vital** that core developers fix any issue they introduce on the stable buildbots, as soon as possible.

This does not mean that other builders' test results can be taken lightly, either. Some of them are known for having platform-specific issues that prevent some tests from succeeding (or even terminating at all), but introducing additional failures should generally not be an option.

10.18.5 Flags-dependent failures

Sometimes, while you have run the *whole test suite* before committing, you may witness unexpected failures on the buildbots. One source of such discrepancies is if different flags have been passed to the test runner or to Python itself. To reproduce, make sure you use the same flags as the buildbots: they can be found out simply by clicking the **stdio** link for the failing build's tests. For example:

```
./python.exe -Wd -E -bb  ./Lib/test/regrtest.py -uall -rwW
```

Note: Running `Lib/test/regrtest.py` is exactly equivalent to running `-m test`.

10.18.6 Ordering-dependent failures

Sometimes the failure is even subtler, as it relies on the order in which the tests are run. The buildbots *randomize* test order (by using the -r option to the test runner) to maximize the probability that potential interferences between library modules are exercised; the downside is that it can make for seemingly sporadic failures.

The --randseed option makes it easy to reproduce the exact randomization used in a given build. Again, open the stdio link for the failing test run, and check the beginning of the test output proper.

Let's assume, for the sake of example, that the output starts with:

```
./python -Wd -E -bb Lib/test/regrtest.py -uall -rwW
== CPython 3.3a0 (default:22ae2b002865, Mar 30 2011, 13:58:40) [GCC 4.4.5]
==   Linux-2.6.36-gentoo-r5-x86_64-AMD_Athlon-tm-_64_X2_Dual_Core_Processor_4400+-with-
↪gentoo-1.12.14 little-endian
==   /home/buildbot/buildarea/3.x.ochtman-gentoo-amd64/build/build/test_python_29628
Testing with flags: sys.flags(debug=0, inspect=0, interactive=0, optimize=0, dont_write_
↪bytecode=0, no_user_site=0, no_site=0, ignore_environment=1, verbose=0, bytes_
↪warning=2, quiet=0)
Using random seed 2613169
[  1/353] test_augassign
[  2/353] test_functools
```

You can reproduce the exact same order using:

```
./python -Wd -E -bb -m test -uall -rwW --randseed 2613169
```

It will run the following sequence (trimmed for brevity):

```
[  1/353] test_augassign
[  2/353] test_functools
[  3/353] test_bool
[  4/353] test_contains
[  5/353] test_compileall
[  6/353] test_unicode
```

If this is enough to reproduce the failure on your setup, you can then bisect the test sequence to look for the specific interference causing the failure. Copy and paste the test sequence in a text file, then use the --fromfile (or -f) option of the test runner to run the exact sequence recorded in that text file:

```
./python -Wd -E -bb -m test -uall -rwW --fromfile mytestsequence.txt
```

In the example sequence above, if test_unicode had failed, you would first test the following sequence:

```
[  1/353] test_augassign
[  2/353] test_functools
[  3/353] test_bool
[  6/353] test_unicode
```

And, if it succeeds, the following one instead (which, hopefully, shall fail):

```
[  4/353] test_contains
[  5/353] test_compileall
[  6/353] test_unicode
```

Then, recursively, narrow down the search until you get a single pair of tests which triggers the failure. It is very rare that such an interference involves more than **two** tests. If this is the case, we can only wish you good luck!

Note: You cannot use the `-j` option (for parallel testing) when diagnosing ordering-dependent failures. Using `-j` isolates each test in a pristine subprocess and, therefore, prevents you from reproducing any interference between tests.

10.18.7 Transient failures

While we try to make the test suite as reliable as possible, some tests do not reach a perfect level of reproducibility. Some of them will sometimes display spurious failures, depending on various conditions. Here are common offenders:

- Network-related tests, such as `test_poplib`, `test_urllibnet`, etc. Their failures can stem from adverse network conditions, or imperfect thread synchronization in the test code, which often has to run a server in a separate thread.
- Tests dealing with delicate issues such as inter-thread or inter-process synchronization, or Unix signals: `test_multiprocessing`, `test_threading`, `test_subprocess`, `test_threadsignals`.

When you think a failure might be transient, it is recommended you confirm by waiting for the next build. Still, even if the failure does turn out sporadic and unpredictable, the issue should be reported on the bug tracker; even better if it can be diagnosed and suppressed by fixing the test's implementation, or by making its parameters - such as a timeout - more robust.

10.18.8 Custom builders

When working on a platform-specific issue, you may want to test your changes on the buildbot fleet rather than just on GitHub Actions and Azure Pipelines. To do so, you can make use of the custom builders. These builders track the `buildbot-custom` short-lived branch of the `python/cpython` repository, which is only accessible to core developers.

To start a build on the custom builders, push the commit you want to test to the `buildbot-custom` branch:

```
$ git push upstream <local_branch_name>:buildbot-custom
```

You may run into conflicts if another developer is currently using the custom builders or forgot to delete the branch when they finished. In that case, make sure the other developer is finished and either delete the branch or force-push (add the `-f` option) over it.

When you have gotten the results of your tests, delete the branch:

```
$ git push upstream :buildbot-custom     # or use the GitHub UI
```

If you are interested in the results of a specific test file only, we recommend you change (temporarily, of course) the contents of the `buildbottest` clause in `Makefile.pre.in`; or, for Windows builders, the `Tools/buildbot/test.bat` script.

See also:

Running a buildbot worker

10.19 Adding to the Stdlib

While the stdlib contains a great amount of useful code, sometimes you want more than is provided. This document is meant to explain how you can get either a new addition to a pre-existing module in the stdlib or add an entirely new module.

Changes to pre-existing code is not covered as that is considered a bugfix and thus is treated as a bug that should be filed on the issue tracker.

10.19.1 Adding to a pre-existing module

If you have found that a function, method, or class is useful and you believe it would be useful to the general Python community, there are some steps to go through in order to see it added to the stdlib.

First is you should gauge the usefulness of the code. Typically this is done by sharing the code publicly. You have a couple of options for this. One is to post it online at the Python Cookbook. Based on feedback or reviews of the recipe you can see if others find the functionality as useful as you do. A search of the issue tracker for previous suggestions related to the proposed addition may turn up a rejected issue that explains why the suggestion will not be accepted. Another is to do a blog post about the code and see what kind of responses you receive. Posting to python-list (see *Following Python's Development* for where to find the list and other mailing lists) to discuss your code also works. Finally, asking on a specific SIG (special interest group) from mail.python.org or python-ideas is also acceptable. This is not a required step but it is suggested.

If you have found general acceptance and usefulness for your code from people, you can open an issue on the issue tracker with the code attached as a *pull request*. If possible, also submit a *contributor agreement*.

If a core developer decides that your code would be useful to the general Python community, they will then commit your code. If your code is not picked up by a core developer and committed then please do not take this personally. Through your public sharing of your code in order to gauge community support for it you at least can know that others will come across it who may find it useful.

10.19.2 Adding a new module

It must be stated upfront that getting a new module into the stdlib is very difficult. Adding any significant amount of code to the stdlib increases the burden placed upon core developers. It also means that the module somewhat becomes "sanctioned" by the core developers as a good way to do something, typically leading to the rest of the Python community to using the new module over other available solutions. All of this means that additions to the stdlib are not taken lightly.

Acceptable Types of Modules

Typically two types of modules get added to the stdlib. One type is a module which implements something that is difficult to get right. A good example of this is the `multiprocessing` package. Working out the various OS issues, working through concurrency issues, etc. are all very difficult to get right.

The second type of module is one that implements something that people re-implement constantly. The `itertools` module is a good example of this type as its constituent parts are not necessarily complex, but are used regularly in a wide range of programs and can be a little tricky to get right. Modules that parse widely used data formats also fall under this type of module that the stdlib consists of.

While a new stdlib module does not need to appeal to all users of Python, it should be something that a large portion of the community will find useful. This makes sure that the developer burden placed upon core developers is worth it.

Requirements

In order for a module to even be considered for inclusion into the stdlib, a couple of requirements must be met.

The most basic is that the code must meet *standard patch requirements*. For code that has been developed outside the stdlib typically this means making sure the coding style guides are followed and that the proper tests have been written.

The module needs to have been out in the community for at least a year. Because of Python's conservative nature when it comes to backwards-compatibility, when a module is added to the stdlib its API becomes frozen. This means that a module should only enter the stdlib when it is mature and gone through its "growing pains".

The module needs to be considered best-of-breed. When something is included in the stdlib it tends to be chosen first for products over other third-party solutions. By virtue of having been available to the public for at least a year, a module needs to have established itself as (one of) the top choices by the community for solving the problem the module is intended for.

The development of the module must move into Python's infrastructure (i.e., the module is no longer directly maintained outside of Python). This prevents a divergence between the code that is included in the stdlib and that which is released outside the stdlib (typically done to provide the module to older versions of Python). It also removes the burden of forcing core developers to have to redirect bug reports or patches to an external issue tracker and VCS.

Someone involved with the development of the module must promise to help maintain the module in the stdlib for two years. This not only helps out other core developers by alleviating workload from bug reports that arrive from the first Python release containing the module, but also helps to make sure that the overall design of the module continues to be uniform.

Proposal Process

If the module you want to propose adding to the stdlib meets the proper requirements, you may propose its inclusion. To start, you should email python-list or python-ideas to make sure the community in general would support the inclusion of the module (see *Following Python's Development*).

If the feedback from the community is positive overall, you will need to write a PEP (Python enhancement proposal) for the module's inclusion. It should outline what the module's overall goal is, why it should be included in the stdlib, and specify the API of the module. See the PEP index for PEPs that have been accepted before that proposed a module for inclusion.

Once your PEP is written, send it to python-ideas for basic vetting. Be prepared for extensive feedback and lots of discussion (not all of it positive). This will help make the PEP be of good quality and properly formatted.

When you have listened to, responded, and integrated as appropriate the feedback from python-ideas into your PEP, you may send it to python-dev. You will once again receive a large amount of feedback and discussion. A PEP dictator will be assigned who makes the final call on whether the PEP will be accepted or not. If the PEP dictator agrees to accept your PEP (which typically means that the core developers end up agreeing in general to accepting your PEP) then the module will be added to the stdlib once the creators of the module sign *contributor agreements*.

10.20 Changing the Python Language

On occasion people come up with an idea on how to change or improve Python as a programming language. This document is meant to explain exactly what changes have a reasonable chance of being considered and what the process is to propose changes to the language.

10.20.1 What Qualifies

First and foremost, it must be understood that changes to the Python programming language are difficult to make. When the language changes, **every** Python programmer already in existence and all Python programmers to come will end up eventually learning about the change you want to propose. Books will need updating, code will be changed, and a new way to do things will need to be learned. Changes to the Python programming language are never taken lightly.

Because of the seriousness that language changes carry, any change must be beneficial to a large proportion of Python users. If the change only benefits a small percentage of Python developers then the change will not be made. A good way to see if your idea would work for a large portion of the Python community is to ask on *python-list or python-ideas*. You can also go through Python's stdlib and find examples of code which would benefit from your proposed change (which helps communicate the usefulness of your change to others). For further guidance, see *Suggesting new features and language changes*.

Your proposed change also needs to be *Pythonic*. While only the Steering Council can truly classify something as Pythonic, you can read the **Zen of Python** for guidance.

10.20.2 PEP Process

Once you are certain you have a language change proposal which will appeal to the general Python community, you can begin the process of officially proposing the change. This process is the Python Enhancement Proposal (PEP) process. **PEP 1** describes it in detail.

You will first need a PEP that you will present to python-ideas. You may be a little hazy on the technical details as various core developers can help with that, but do realize that if you do not present your idea to python-ideas or python-list ahead of time you may find out it is technically not possible. Expect extensive comments on the PEP, some of which will be negative.

Once your PEP has been modified to be of proper quality and to take into account comments made on python-ideas, it may proceed to python-dev. There it will be assigned a PEP dictator and another general discussion will occur. Once again, you will need to modify your PEP to incorporate the large amount of comments you will receive.

The PEP dictator decides if your PEP is accepted (typically based on whether most core developers support the PEP). If that occurs then your proposed language change will be introduced in the next release of Python. Otherwise your PEP will be recorded as rejected along with an explanation as to why so that others do not propose the same language change in the future.

10.20.3 Suggesting new features and language changes

The python-ideas mailing list is specifically intended for discussion of new features and language changes. Please don't be disappointed if your idea isn't met with universal approval: as the long list of Rejected and Withdrawn PEPs in the PEP Index attests, and as befits a reasonably mature programming language, getting significant changes into Python isn't a simple task.

If the idea is reasonable, someone will suggest posting it as a feature request on the issue tracker, or, for larger changes, writing it up as a **draft PEP**.

Sometimes core developers will differ in opinion, or merely be collectively unconvinced. When there isn't an obvious victor then the Status Quo Wins a Stalemate as outlined in the linked post.

For some examples on language changes that were accepted please read Justifying Python Language Changes.

10.21 Experts Index

This document has tables that list Python Modules, Tools, Platforms and Interest Areas and GitHub names for each item that indicate a maintainer or an expert in the field. This list is intended to be used by issue submitters, issue triage people, and other issue participants to find people to @mention or add as reviewers to issues and pull requests. People on this list may be asked to render final judgement on a feature or bug. If no active maintainer is listed for a given module, then questionable changes should go to python-dev, while any other issues can and should be decided by any committer.

Developers can choose to follow labels, so if a label that they are following is added to an issue or pull request, they will be notified automatically. The `CODEOWNERS` file is also used to indicate maintainers that will be automatically added as reviewers to pull requests.

Unless a name is followed by a '*', you should never assign an issue to that person. Names followed by a '*' may be assigned issues involving the module or topic.

Names followed by a '^' indicate old bugs.python.org usernames, for people that did not transition to GitHub.

The Platform and Interest Area tables list broader fields in which various people have expertise. These people can also be contacted for help, opinions, and decisions when issues involve their areas.

If a listed maintainer does not respond to requests for comment for an extended period (three weeks or more), they should be marked as inactive in this list by placing the word 'inactive' in parenthesis behind their tracker id. They are of course free to remove that inactive mark at any time.

Committers should update these tables as their areas of expertise widen. New topics may be added to the Interest Area table at will.

The existence of this list is not meant to indicate that these people *must* be contacted for decisions; it is, rather, a resource to be used by non-committers to find responsible parties, and by committers who do not feel qualified to make a decision in a particular context.

10.21.1 Stdlib

Module	Maintainers
__future__	
__main__	gvanrossum, ncoghlan
_dummy_thread	brettcannon
_thread	
_testbuffer	
abc	
aifc	bitdancer
argparse	
array	
ast	benjaminp, pablogsal, isidentical
asynchat	josiahcarlson, giampaolo*, stutzbach^
asyncio	1st1, asvetlov
asyncore	josiahcarlson, giampaolo*, stutzbach^
atexit	
audioop	serhiy-storchaka
base64	
bdb	
binascii	

continues on next page

Table 2 – continued from previous page

Module	Maintainers
binhex	
bisect	rhettinger*
builtins	
bz2	
calendar	
cgi	ethanfurman*
cgitb	ethanfurman*
chunk	
cmath	mdickinson
cmd	
code	
codecs	malemburg, doerwalter
codeop	
collections	rhettinger*
collections.abc	rhettinger*, stutzbach^
colorsys	
compileall	
concurrent.futures	pitrou, brianquinlan, gpshead*
configparser	ambv*
contextlib	ncoghlan, 1st1
contextvars	
copy	avassalotti
copyreg	avassalotti
cProfile	
crypt	jafo^*
csv	smontanaro (inactive)
ctypes	theller (inactive), abalkin, amauryfa, meadori
curses	Yhg1s
dataclasses	ericvsmith*
datetime	abalkin, pganssle
dbm	
decimal	facundobatista, rhettinger, mdickinson
difflib	tim-one (inactive)
dis	1st1
distutils	merwok, dstufft
doctest	tim-one (inactive)
dummy_threading	brettcannon
email	warsaw, bitdancer*, maxking
encodings	malemburg
ensurepip	ncoghlan, dstufft, pradyunsg
enum	eliben*, warsaw, ethanfurman*
errno	Yhg1s
faulthandler	vstinner, gpshead
fcntl	Yhg1s
filecmp	
fileinput	
fnmatch	
formatter	
fractions	mdickinson

continues on next page

Table 2 – continued from previous page

Module	Maintainers
ftplib	giampaolo*
functools	rhettinger*
gc	pitrou, pablogsal
getopt	
getpass	
gettext	
glob	
grp	
gzip	
hashlib	tiran, gpshead*
heapq	rhettinger*, stutzbach^
hmac	tiran, gpshead*
html	ezio-melotti*
http	
idlelib	kbkaiser (inactive), terryjreedy*, serwy (inactive), taleinat
imaplib	
imghdr	
imp	
importlib	brettcannon
inspect	1st1
io	benjaminp, stutzbach^
ipaddress	pmoody^
itertools	rhettinger*
json	etrepum (inactive), ezio-melotti, rhettinger
keyword	
lib2to3	benjaminp
libmpdec	
linecache	
locale	malemburg
logging	vsajip
lzma	
mailbox	
mailcap	
marshal	
math	mdickinson, rhettinger, stutzbach^
mimetypes	
mmap	Yhg1s
modulefinder	theller (inactive), jvr^
msilib	
msvcrt	
multiprocessing	applio*, pitrou, jnoller^ (inactive), sbt^ (inactive), gpshead*
netrc	
nis	
nntplib	
numbers	
operator	
optparse	mitsuhiko
os	
os.path	serhiy-storchaka

continues on next page

Table 2 – continued from previous page

Module	Maintainers
ossaudiodev	
parser	benjaminp, pablogsal
pathlib	
pdb	
pickle	avassalotti
pickletools	avassalotti
pipes	
pkgutil	
platform	malemburg
plistlib	
poplib	
posix	larryhastings, gpshead
pprint	freddrake
profile	
pstats	
pty	Yhg1s*
pwd	
py_compile	
pyclbr	isidentical
pydoc	
queue	rhettinger*
quopri	
random	rhettinger, mdickinson
re	ezio-melotti, serhiy-storchaka
readline	Yhg1s
reprlib	
resource	Yhg1s
rlcompleter	
runpy	ncoghlan
sched	
secrets	
select	
selectors	neologix^, giampaolo
shelve	
shlex	
shutil	tarekziade, giampaolo
signal	gpshead
site	
smtpd	giampaolo
smtplib	
sndhdr	
socket	gpshead
socketserver	
spwd	
sqlite3	ghaering^
ssl	jackjansen, tiran, dstufft, alex
stat	tiran
statistics	stevendaprano, rhettinger
string	

continues on next page

Table 2 – continued from previous page

Module	Maintainers
stringprep	
struct	mdickinson, meadori
subprocess	astrand^ (inactive), giampaolo, gpshead*
sunau	
symbol	
symtable	benjaminp
sys	
sysconfig	tarekziade
syslog	jafo^*
tabnanny	tim-one (inactive)
tarfile	gustaebel
telnetlib	
tempfile	
termios	Yhg1s
test	ezio-melotti
textwrap	
threading	pitrou, gpshead
time	abalkin, pganssle
timeit	
tkinter	gpolo^, serhiy-storchaka
token	
tokenize	meadori
trace	abalkin
traceback	iritkatriel
tracemalloc	vstinner
tty	Yhg1s*
turtle	gregorlingl^, willingc
types	1st1
typing	gvanrossum, Fidget-Spinner, JelleZijlstra*
unicodedata	malemburg, ezio-melotti
unittest	voidspace*, ezio-melotti, rbtcollins, gpshead
unittest.mock	voidspace*
urllib	orsenthil
uu	
uuid	
venv	vsajip
warnings	
wave	
weakref	freddrake
webbrowser	
winreg	stutzbach^
winsound	
wsgiref	pjenvey
xdrlib	
xml.dom	
xml.dom.minidom	
xml.dom.pulldom	
xml.etree	eliben*, scoder
xml.parsers.expat	

continues on next page

Table 2 – continued from previous page

Module	Maintainers
xml.sax	
xml.sax.handler	
xml.sax.saxutils	
xml.sax.xmlreader	
xmlrpc	
zipapp	pfmoore
zipfile	alanmcintyre^, serhiy-storchaka, Yhg1s, gpshead
zipimport	Yhg1s*
zlib	Yhg1s, gpshead*

10.21.2 Tools

Tool	Maintainers
Argument Clinic	larryhastings
PEG Generator	gvanrossum, pablogsal, lysnikolaou

10.21.3 Platforms

Platform	Maintainers
AIX	David.Edelsohn^
Cygwin	jlt63^, stutzbach^
FreeBSD	
HP-UX	
Linux	
Mac OS X	ronaldoussoren, ned-deily
NetBSD1	
OS2/EMX	aimacintyre^
Solaris/OpenIndiana	jcea
Windows	tjguk, zware, zooba, pfmoore
JVM/Java	frank.wierzbicki^

10.21.4 Miscellaneous

Interest Area	Maintainers
algorithms	rhettinger*
argument clinic	larryhastings
ast/compiler	benjaminp, brettcannon, 1st1, pablogsal, markshannon, isidentical, brandtbucher
autoconf/makefiles	Yhg1s*
bsd	
issue tracker	ezio-melotti
buildbots	zware, pablogsal
bytecode	benjaminp, 1st1, markshannon, brandtbucher
context managers	ncoghlan
core workflow	Mariatta, ezio-melotti

continues on ne

Table 3 – continued from previous page

Interest Area	Maintainers
coverity scan	tiran, brettcannon, Yhg1s
cryptography	gpshead, dstufft
data formats	mdickinson
database	malemburg
devguide	merwok, ezio-melotti, willingc, Mariatta
documentation	ezio-melotti, merwok, JulienPalard, willingc
emoji	Mariatta
extension modules	encukou, ncoghlan
filesystem	giampaolo
f-strings	ericvsmith*
GUI	
i18n	malemburg, merwok
import machinery	brettcannon, ncoghlan, ericsnowcurrently
io	benjaminp, stutzbach^, gpshead
locale	malemburg
mathematics	mdickinson, malemburg, stutzbach^, rhettinger
memory management	tim-one, malemburg, Yhg1s
memoryview	
networking	giampaolo, gpshead
object model	benjaminp, Yhg1s
packaging	tarekziade, malemburg, alexis^, merwok, dstufft, pfmoore
pattern matching	brandtbucher*
peg parser	gvanrossum, pablogsal, lysnikolaou
performance	brettcannon, vstinner, serhiy-storchaka, 1st1, rhettinger, markshannon, brandtbucher
pip	ncoghlan, dstufft, pfmoore, Marcus.Smith^, pradyunsg
py3 transition	benjaminp
release management	tarekziade, malemburg, benjaminp, warsaw, gvanrossum, anthonybaxter^, merwok, ned-deily, birkenfeld, Juli
str.format	ericvsmith*
testing	voidspace, ezio-melotti
test coverage	
threads	gpshead
time and dates	malemburg, abalkin, pganssle
unicode	malemburg, ezio-melotti, benjaminp
version control	merwok, ezio-melotti

10.21.5 Documentation Translations

For a list of translators, see *this table about translations*.

10.22 gdb Support

If you experience low-level problems such as crashes or deadlocks (e.g. when tinkering with parts of CPython which are written in C), it can be convenient to use a low-level debugger such as gdb in order to diagnose and fix the issue. By default, however, gdb (or any of its front-ends) doesn't know about high-level information specific to the CPython interpreter, such as which Python function is currently executing, or what type or value has a given Python object represented by a standard `PyObject *` pointer. We hereafter present two ways to overcome this limitation.

10.22.1 gdb 7 and later

In gdb 7, support for extending gdb with Python was added. When CPython is built you will notice a `python-gdb.py` file in the root directory of your checkout. Read the module docstring for details on how to use the file to enhance gdb for easier debugging of a CPython process.

To activate support, you must add the directory containing `python-gdb.py` to GDB's "auto-load-safe-path". Put this in your `~/.gdbinit` file:

```
add-auto-load-safe-path /path/to/checkout
```

You can also add multiple paths, separated by `:`.

This is what a backtrace looks like (truncated) when this extension is enabled:

```
#0  0x000000000041a6b1 in PyObject_Malloc (nbytes=Cannot access memory at address␣
↪0x7fffff7fefe8
) at Objects/obmalloc.c:748
#1  0x000000000041b7c0 in _PyObject_DebugMallocApi (id=111 'o', nbytes=24) at Objects/
↪obmalloc.c:1445
#2  0x000000000041b717 in _PyObject_DebugMalloc (nbytes=24) at Objects/obmalloc.c:1412
#3  0x000000000044060a in _PyUnicode_New (length=11) at Objects/unicodeobject.c:346
#4  0x00000000004466aa in PyUnicodeUCS2_DecodeUTF8Stateful (s=0x5c2b8d "__lltrace__",␣
↪size=11, errors=0x0, consumed=
    0x0) at Objects/unicodeobject.c:2531
#5  0x0000000000446647 in PyUnicodeUCS2_DecodeUTF8 (s=0x5c2b8d "__lltrace__", size=11,␣
↪errors=0x0)
    at Objects/unicodeobject.c:2495
#6  0x0000000000440d1b in PyUnicodeUCS2_FromStringAndSize (u=0x5c2b8d "__lltrace__",␣
↪size=11)
    at Objects/unicodeobject.c:551
#7  0x0000000000440d94 in PyUnicodeUCS2_FromString (u=0x5c2b8d "__lltrace__") at Objects/
↪unicodeobject.c:569
#8  0x0000000000584abd in PyDict_GetItemString (v=
    {'Yuck': <type at remote 0xad4730>, '__builtins__': <module at remote 0x7ffff7fd5ee8>
↪, '__file__': 'Lib/test/crashers/nasty_eq_vs_dict.py', '__package__': None, 'y':
↪<Yuck(i=0) at remote 0xaacd80>, 'dict': {0: 0, 1: 1, 2: 2, 3: 3}, '__cached__': None,
↪'__name__': '__main__', 'z': <Yuck(i=0) at remote 0xaace60>, '__doc__': None}, key=
    0x5c2b8d "__lltrace__") at Objects/dictobject.c:2171
```

(Notice how the dictionary argument to `PyDict_GetItemString` is displayed as its `repr()`, rather than an opaque `PyObject *` pointer.)

The extension works by supplying a custom printing routine for values of type `PyObject *`. If you need to access lower-level details of an object, then cast the value to a pointer of the appropriate type. For example:

```
(gdb) p globals
$1 = {'__builtins__': <module at remote 0x7ffff7fb1868>, '__name__':
'__main__', 'ctypes': <module at remote 0x7ffff7f14360>, '__doc__': None,
'__package__': None}

(gdb) p *(PyDictObject*)globals
$2 = {ob_refcnt = 3, ob_type = 0x3dbdf85820, ma_fill = 5, ma_used = 5,
ma_mask = 7, ma_table = 0x63d0f8, ma_lookup = 0x3dbdc7ea70
<lookdict_string>, ma_smalltable = {{me_hash = 7065186196740147912,
```

(continues on next page)

(continued from previous page)

```
me_key = '__builtins__', me_value = <module at remote 0x7ffff7fb1868>},
{me_hash = -368181376027291943, me_key = '__name__',
me_value ='__main__'}, {me_hash = 0, me_key = 0x0, me_value = 0x0},
{me_hash = 0, me_key = 0x0, me_value = 0x0},
{me_hash = -9177857982131165996, me_key = 'ctypes',
me_value = <module at remote 0x7ffff7f14360>},
{me_hash = -8518757509529533123, me_key = '__doc__', me_value = None},
{me_hash = 0, me_key = 0x0, me_value = 0x0}, {
  me_hash = 6614918939584953775, me_key = '__package__', me_value = None}}}
```

The pretty-printers try to closely match the `repr()` implementation of the underlying implementation of Python, and thus vary somewhat between Python 2 and Python 3.

An area that can be confusing is that the custom printer for some types look a lot like gdb's built-in printer for standard types. For example, the pretty-printer for a Python 3 `int` gives a `repr()` that is not distinguishable from a printing of a regular machine-level integer:

```
(gdb) p some_machine_integer
$3 = 42

(gdb) p some_python_integer
$4 = 42

(gdb) p *(PyLongObject*)some_python_integer
$5 = {ob_base = {ob_base = {ob_refcnt = 8, ob_type = 0x3dad39f5e0}, ob_size = 1},
ob_digit = {42}}
```

A similar confusion can arise with the `str` type, where the output looks a lot like gdb's built-in printer for `char *`:

```
(gdb) p ptr_to_python_str
$6 = '__builtins__'
```

The pretty-printer for `str` instances defaults to using single-quotes (as does Python's `repr` for strings) whereas the standard printer for `char *` values uses double-quotes and contains a hexadecimal address:

```
(gdb) p ptr_to_char_star
$7 = 0x6d72c0 "hello world"
```

Here's how to see the implementation details of a `str` instance (for Python 3, where a `str` is a `PyUnicodeObject *`):

```
(gdb) p *(PyUnicodeObject*)$6
$8 = {ob_base = {ob_refcnt = 33, ob_type = 0x3dad3a95a0}, length = 12,
str = 0x7ffff2128500, hash = 7065186196740147912, state = 1, defenc = 0x0}
```

As well as adding pretty-printing support for `PyObject *`, the extension adds a number of commands to gdb:

`py-list`

List the Python source code (if any) for the current frame in the selected thread. The current line is marked with a ">":

```
(gdb) py-list
 901        if options.profile:
 902            options.profile = False
 903            profile_me()
```

(continues on next page)

(continued from previous page)

```
 904            return
 905
>906        u = UI()
 907        if not u.quit:
 908            try:
 909                gtk.main()
 910            except KeyboardInterrupt:
 911                # properly quit on a keyboard interrupt...
```

Use `py-list START` to list at a different line number within the python source, and `py-list START,END` to list a specific range of lines within the python source.

py-up and py-down

The `py-up` and `py-down` commands are analogous to gdb's regular `up` and `down` commands, but try to move at the level of CPython frames, rather than C frames.

gdb is not always able to read the relevant frame information, depending on the optimization level with which CPython was compiled. Internally, the commands look for C frames that are executing `PyEval_EvalFrameEx` (which implements the core bytecode interpreter loop within CPython) and look up the value of the related `PyFrameObject *`.

They emit the frame number (at the C level) within the thread.

For example:

```
(gdb) py-up
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-packages/
gnome_sudoku/main.py, line 906, in start_game ()
    u = UI()
(gdb) py-up
#40 Frame 0x948e82c, for file /usr/lib/python2.6/site-packages/
gnome_sudoku/gnome_sudoku.py, line 22, in start_game(main=<module at remote␣
↪0xb771b7f4>)
    main.start_game()
(gdb) py-up
Unable to find an older python frame
```

so we're at the top of the python stack. Going back down:

```
(gdb) py-down
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-packages/gnome_sudoku/main.py,
↪ line 906, in start_game ()
    u = UI()
(gdb) py-down
#34 (unable to read python frame information)
(gdb) py-down
#23 (unable to read python frame information)
(gdb) py-down
#19 (unable to read python frame information)
(gdb) py-down
#14 Frame 0x99262ac, for file /usr/lib/python2.6/site-packages/gnome_sudoku/game_
↪selector.py, line 201, in run_swallowed_dialog (self=<NewOrSavedGameSelector(new_
↪game_model=<gtk.ListStore at remote 0x98fab44>, puzzle=None, saved_games=[{'gsd.
↪auto_fills': 0, 'tracking': {}, 'trackers': {}, 'notes': [], 'saved_at':␣
↪1270084485, 'game': '7 8 0 0 0 0 0 5 6 0 0 9 0 8 0 1 0 0 0 4 6 0 0 0 0 7 0 6 5 0␣
↪0 0 4 7 9 2 0 0 0 9 0 1 0 0 0 3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0 4 0 6 0␣
↪0 2 1 0 0 0 0 0 4 5\n7 8 0 0 0 0 0 5 6 0 0 9 0 8 0 1 0 0 0 4 6 0 0 0 0 7 0 6 5 1␣
↪8 3 4 7 9 2 0 0 0 9 0 1 0 0 0 3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0 4 0 6 0␣
↪0 2 1 0 0 0 0 0 4 5', 'gsd.impossible_hints': 0, 'timer.__absolute_start_time__':
↪<float at remote 0x984b474>, 'gsd.hints': 0, 'timer.active_time': <float at␣
↪remote 0x984b494>, 'timer.total_time': <float at remote 0x984b464>}], dialog=<gtk.
Dialog at remote 0x98faaa4>, saved_game_model=<gtk.ListStore at remote 0x98fad24>,
```

(continues on next page)

(continued from previous page)

```
            swallower.run_dialog(self.dialog)
(gdb) py-down
#11 Frame 0x9aead74, for file /usr/lib/python2.6/site-packages/gnome_sudoku/dialog_
↪swallower.py, line 48, in run_dialog (self=<SwappableArea(running=<gtk.Dialog at␣
↪remote 0x98faaa4>, main_page=0) at remote 0x98fa6e4>, d=<gtk.Dialog at remote␣
↪0x98faaa4>)
            gtk.main()
(gdb) py-down
#8 (unable to read python frame information)
(gdb) py-down
Unable to find a newer python frame
```

and we're at the bottom of the python stack.

py-bt

The py-bt command attempts to display a Python-level backtrace of the current thread.

For example:

```
(gdb) py-bt
#8 (unable to read python frame information)
#11 Frame 0x9aead74, for file /usr/lib/python2.6/site-packages/gnome_sudoku/dialog_
↪swallower.py, line 48, in run_dialog (self=<SwappableArea(running=<gtk.Dialog at␣
↪remote 0x98faaa4>, main_page=0) at remote 0x98fa6e4>, d=<gtk.Dialog at remote␣
↪0x98faaa4>)
            gtk.main()
#14 Frame 0x99262ac, for file /usr/lib/python2.6/site-packages/gnome_sudoku/game_
↪selector.py, line 201, in run_swallowed_dialog (self=<NewOrSavedGameSelector(new_
↪game_model=<gtk.ListStore at remote 0x98fab44>, puzzle=None, saved_games=[{'gsd.
↪auto_fills': 0, 'tracking': {}, 'trackers': {}, 'notes': [], 'saved_at':␣
↪1270084485, 'game': '7 8 0 0 0 0 0 5 6 0 0 9 0 8 0 1 0 0 0 4 6 0 0 0 0 7 0 6 5 0␣
↪0 0 4 7 9 2 0 0 0 9 0 1 0 0 0 3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0 4 0 6 0␣
↪0 2 1 0 0 0 0 0 4 5\n7 8 0 0 0 0 0 5 6 0 0 9 0 8 0 1 0 0 0 4 6 0 0 0 0 7 0 6 5 1␣
↪8 3 4 7 9 2 0 0 0 9 0 1 0 0 0 3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0 4 0 6 0␣
↪0 2 1 0 0 0 0 0 4 5', 'gsd.impossible_hints': 0, 'timer.__absolute_start_time__':
↪<float at remote 0x984b474>, 'gsd.hints': 0, 'timer.active_time': <float at␣
↪remote 0x984b494>, 'timer.total_time': <float at remote 0x984b464>}], dialog=<gtk.
↪Dialog at remote 0x98faaa4>, saved_game_model=<gtk.ListStore at remote 0x98fad24>,
↪ sudoku_maker=<SudokuMaker(terminated=False, played=[], batch_siz...(truncated)
            swallower.run_dialog(self.dialog)
#19 (unable to read python frame information)
#23 (unable to read python frame information)
#34 (unable to read python frame information)
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-packages/gnome_sudoku/main.py,
↪ line 906, in start_game ()
    u = UI()
#40 Frame 0x948e82c, for file /usr/lib/python2.6/site-packages/gnome_sudoku/gnome_
↪sudoku.py, line 22, in start_game (main=<module at remote 0xb771b7f4>)
    main.start_game()
```

The frame numbers correspond to those displayed by gdb's standard backtrace command.

py-print

The py-print command looks up a Python name and tries to print it. It looks in locals within the current thread, then globals, then finally builtins:

```
(gdb) py-print self
local 'self' = <SwappableArea(running=<gtk.Dialog at remote 0x98faaa4>,
main_page=0) at remote 0x98fa6e4>
(gdb) py-print __name__
global '__name__' = 'gnome_sudoku.dialog_swallower'
(gdb) py-print len
builtin 'len' = <built-in function len>
(gdb) py-print scarlet_pimpernel
'scarlet_pimpernel' not found
```

`py-locals`

The `py-locals` command looks up all Python locals within the current Python frame in the selected thread, and prints their representations:

```
(gdb) py-locals
self = <SwappableArea(running=<gtk.Dialog at remote 0x98faaa4>,
main_page=0) at remote 0x98fa6e4>
d = <gtk.Dialog at remote 0x98faaa4>
```

You can of course use other gdb commands. For example, the `frame` command takes you directly to a particular frame within the selected thread. We can use it to go a specific frame shown by `py-bt` like this:

```
(gdb) py-bt
(output snipped)
#68 Frame 0xaa4560, for file Lib/test/regrtest.py, line 1548, in <module> ()
        main()
(gdb) frame 68
#68 0x00000000004cd1e6 in PyEval_EvalFrameEx (f=Frame 0xaa4560, for file Lib/test/
↪regrtest.py, line 1548, in <module> (), throwflag=0) at Python/ceval.c:2665
2665                            x = call_function(&sp, oparg);
(gdb) py-list
1543        # Run the tests in a context manager that temporary changes the CWD to a
1544        # temporary and writable directory. If it's not possible to create or
1545        # change the CWD, the original CWD will be used. The original CWD is
1546        # available from test_support.SAVEDCWD.
1547        with test_support.temp_cwd(TESTCWD, quiet=True):
>1548            main()
```

The `info threads` command will give you a list of the threads within the process, and you can use the `thread` command to select a different one:

```
(gdb) info threads
  105 Thread 0x7fffefa18710 (LWP 10260)  sem_wait () at ../nptl/sysdeps/unix/sysv/linux/
↪x86_64/sem_wait.S:86
  104 Thread 0x7fffdf5fe710 (LWP 10259)  sem_wait () at ../nptl/sysdeps/unix/sysv/linux/
↪x86_64/sem_wait.S:86
* 1 Thread 0x7ffff7fe2700 (LWP 10145)  0x00000038e46d73e3 in select () at ../sysdeps/
↪unix/syscall-template.S:82
```

You can use `thread apply all COMMAND` or (`t a a COMMAND` for short) to run a command on all threads. You can use this with `py-bt` to see what every thread is doing at the Python level:

```
(gdb) t a a py-bt

Thread 105 (Thread 0x7fffefa18710 (LWP 10260)):
#5 Frame 0x7fffd00019d0, for file /home/david/coding/python-svn/Lib/threading.py, line␣
↪155, in _acquire_restore (self=<_RLock(_Verbose__verbose=False, _RLock__
↪owner=140737354016512, _RLock__block=<thread.lock at remote 0x858770>, _RLock__
↪count=1) at remote 0xd7ff40>, count_owner=(1, 140737213728528), count=1,␣
↪owner=140737213728528)
        self.__block.acquire()
#8 Frame 0x7fffac001640, for file /home/david/coding/python-svn/Lib/threading.py, line␣
↪269, in wait (self=<_Condition(_Condition__lock=<_RLock(_Verbose__verbose=False, _
↪RLock__owner=140737354016512, _RLock__block=<thread.lock at remote 0x858770>, _RLock__
↪count=1) at remote 0xd7ff40>, acquire=<instancemethod at remote 0xd80260>, _is_owned=
↪<instancemethod at remote 0xd80160>, _release_save=<instancemethod at remote 0xd803e0>,
↪ release=<instancemethod at remote 0xd802e0>, _acquire_restore=<instancemethod at␣
↪remote 0xd7ee60>, _Verbose__verbose=False, _Condition__waiters=[]) at remote 0xd7fd10>,
↪ timeout=None, waiter=<thread.lock at remote 0x858a90>, saved_state=(1,␣
↪140737213728528))
            self._acquire_restore(saved_state)
#12 Frame 0x7fffb8001a10, for file /home/david/coding/python-svn/Lib/test/lock_tests.py,␣
↪line 348, in f ()
            cond.wait()
#16 Frame 0x7fffb8001c40, for file /home/david/coding/python-svn/Lib/test/lock_tests.py,␣
↪line 37, in task (tid=140737213728528)
                f()

Thread 104 (Thread 0x7fffdf5fe710 (LWP 10259)):
#5 Frame 0x7fffe4001580, for file /home/david/coding/python-svn/Lib/threading.py, line␣
↪155, in _acquire_restore (self=<_RLock(_Verbose__verbose=False, _RLock__
↪owner=140737354016512, _RLock__block=<thread.lock at remote 0x858770>, _RLock__
↪count=1) at remote 0xd7ff40>, count_owner=(1, 140736940992272), count=1,␣
↪owner=140736940992272)
        self.__block.acquire()
#8 Frame 0x7fffc8002090, for file /home/david/coding/python-svn/Lib/threading.py, line␣
↪269, in wait (self=<_Condition(_Condition__lock=<_RLock(_Verbose__verbose=False, _
↪RLock__owner=140737354016512, _RLock__block=<thread.lock at remote 0x858770>, _RLock__
↪count=1) at remote 0xd7ff40>, acquire=<instancemethod at remote 0xd80260>, _is_owned=
↪<instancemethod at remote 0xd80160>, _release_save=<instancemethod at remote 0xd803e0>,
↪ release=<instancemethod at remote 0xd802e0>, _acquire_restore=<instancemethod at␣
↪remote 0xd7ee60>, _Verbose__verbose=False, _Condition__waiters=[]) at remote 0xd7fd10>,
↪ timeout=None, waiter=<thread.lock at remote 0x858860>, saved_state=(1,␣
↪140736940992272))
            self._acquire_restore(saved_state)
#12 Frame 0x7fffac001c90, for file /home/david/coding/python-svn/Lib/test/lock_tests.py,␣
↪line 348, in f ()
            cond.wait()
#16 Frame 0x7fffac0011c0, for file /home/david/coding/python-svn/Lib/test/lock_tests.py,␣
↪line 37, in task (tid=140736940992272)
                f()

Thread 1 (Thread 0x7ffff7fe2700 (LWP 10145)):
#5 Frame 0xcb5380, for file /home/david/coding/python-svn/Lib/test/lock_tests.py, line␣
↪16, in _wait ()
```

(continues on next page)

(continued from previous page)

```
    time.sleep(0.01)
#8 Frame 0x7fffd00024a0, for file /home/david/coding/python-svn/Lib/test/lock_tests.py,
↪line 378, in _check_notify (self=<ConditionTests(_testMethodName='test_notify', _
↪resultForDoCleanups=<TestResult(_original_stdout=<cStringIO.StringO at remote 0xc191e0>
↪, skipped=[], _mirrorOutput=False, testsRun=39, buffer=False, _original_stderr=<file
↪at remote 0x7ffff7fc6340>, _stdout_buffer=<cStringIO.StringO at remote 0xc9c7f8>, _
↪stderr_buffer=<cStringIO.StringO at remote 0xc9c790>, _moduleSetUpFailed=False,
↪expectedFailures=[], errors=[], _previousTestClass=<type at remote 0x928310>,
↪unexpectedSuccesses=[], failures=[], shouldStop=False, failfast=False) at remote
↪0xc185a0>, _threads=(0,), _cleanups=[], _type_equality_funcs={<type at remote 0x7eba00>
↪: <instancemethod at remote 0xd750e0>, <type at remote 0x7e7820>: <instancemethod at
↪remote 0xd75160>, <type at remote 0x7e30e0>: <instancemethod at remote 0xd75060>,
↪<type at remote 0x7e7d20>: <instancemethod at remote 0xd751e0>, <type at remote
↪0x7f19e0...(truncated)
        _wait()
```

Note: This is only available for Python 2.7, 3.2 and higher.

10.22.2 gdb 6 and earlier

The file at `Misc/gdbinit` contains a gdb configuration file which provides extra commands when working with a CPython process. To register these commands permanently, either copy the commands to your personal gdb configuration file or symlink `~/.gdbinit` to `Misc/gdbinit`. To use these commands from a single gdb session without registering them, type `source Misc/gdbinit` from your gdb session.

10.22.3 Updating auto-load-safe-path to allow test_gdb to run

`test_gdb` attempts to automatically load additional Python specific hooks into gdb in order to test them. Unfortunately, the command line options it uses to do this aren't always supported correctly.

If `test_gdb` is being skipped with an "auto-loading has been declined" message, then it is necessary to identify any Python build directories as auto-load safe. One way to achieve this is to add a line like the following to `~/.gdbinit` (edit the specific list of paths as appropriate):

```
add-auto-load-safe-path ~/devel/py3k:~/devel/py32:~/devel/py27
```

10.23 Exploring CPython's Internals

This is a quick guide for people who are interested in learning more about CPython's internals. It provides a summary of the source code structure and contains references to resources providing a more in-depth view.

10.23.1 CPython Source Code Layout

This guide gives an overview of CPython's code structure. It serves as a summary of file locations for modules and builtins.

For Python modules, the typical layout is:

- `Lib/<module>.py`
- `Modules/_<module>.c` (if there's also a C accelerator module)
- `Lib/test/test_<module>.py`
- `Doc/library/<module>.rst`

For extension-only modules, the typical layout is:

- `Modules/<module>module.c`
- `Lib/test/test_<module>.py`
- `Doc/library/<module>.rst`

For builtin types, the typical layout is:

- `Objects/<builtin>object.c`
- `Lib/test/test_<builtin>.py`
- `Doc/library/stdtypes.rst`

For builtin functions, the typical layout is:

- `Python/bltinmodule.c`
- `Lib/test/test_builtin.py`
- `Doc/library/functions.rst`

Some exceptions:

- builtin type `int` is at `Objects/longobject.c`
- builtin type `str` is at `Objects/unicodeobject.c`
- builtin module `sys` is at `Python/sysmodule.c`
- builtin module `marshal` is at `Python/marshal.c`
- Windows-only module `winreg` is at `PC/winreg.c`

10.23.2 Additional References

For over 20 years the CPython code base has been changing and evolving. Here's a sample of resources about the architecture of CPython aimed at building your understanding of both the 2.x and 3.x versions of CPython:

Table 4: **Current references**

Title	Brief	Author	Version
A guide from parser to objects, observed using GDB	Code walk from Parser, AST, Sym Table and Objects	Louie Lu	3.7.a0
Green Tree Snakes	The missing Python AST docs	Thomas Kluyver	3.6
Yet another guided tour of CPython	A guide for how CPython REPL works	Guido van Rossum	3.5
Python Asynchronous I/O Walkthrough	How CPython async I/O, generator and coroutine works	Philip Guo	3.5
Coding Patterns for Python Extensions	Reliable patterns of coding Python Extensions in C	Paul Ross	3.4
Your Guide to the CPython Source Code	Your Guide to the CPython Source Code	Anthony Shaw	3.8

Table 5: **Historical references**

Title	Brief	Author	Version
Python's Innards Series	ceval, objects, pystate and miscellaneous topics	Yaniv Aknin	3.1
Eli Bendersky's Python Internals	Objects, Symbol tables and miscellaneous topics	Eli Bendersky	3.x
A guide from parser to objects, observed using Eclipse	Code walk from Parser, AST, Sym Table and Objects	Prashanth Raghu	2.7.12
CPython internals: A ten-hour codewalk through the Python interpreter source code	Code walk from source code to generators	Philip Guo	2.7.8

10.24 Changing CPython's Grammar

10.24.1 Abstract

There's more to changing Python's grammar than editing `Grammar/python.gram`. Here's a checklist.

Note: These instructions are for Python 3.9 and beyond. Earlier versions use a different parser technology. You probably shouldn't try to change the grammar of earlier Python versions, but if you really want to, use GitHub to track down the earlier version of this file in the devguide.

For more information on how to use the new parser, check the *section on how to use CPython's parser*.

10.24.2 Checklist

Note: sometimes things mysteriously don't work. Before giving up, try `make clean`.

- `Grammar/python.gram`: The grammar, with actions that build AST nodes. After changing it, run `make regen-pegen` (or `build.bat --regen` on Windows), to regenerate `Parser/parser.c`. (This runs Python's parser generator, `Tools/peg_generator`).
- `Grammar/Tokens` is a place for adding new token types. After changing it, run `make regen-token` to regenerate `Include/token.h`, `Parser/token.c`, `Lib/token.py` and `Doc/library/token-list.inc`. If you change both `python.gram` and `Tokens`, run `make regen-token` before `make regen-pegen`. On Windows, `build.bat --regen` will regenerate both at the same time.
- `Parser/Python.asdl` may need changes to match the grammar. Then run `make regen-ast` to regenerate `Include/Python-ast.h` and `Python/Python-ast.c`.
- `Parser/tokenizer.c` contains the tokenization code. This is where you would add a new type of comment or string literal, for example.
- `Python/ast.c` will need changes to validate AST objects involved with the grammar change.
- `Python/ast_unparse.c` will need changes to unparse AST objects involved with the grammar change ("unparsing" is used to turn annotations into strings per **PEP 563**).
- The *Design of CPython's Compiler* has its own page.
- `_Unparser` in the `Lib/ast.py` file may need changes to accommodate any modifications in the AST nodes.
- `Doc/library/ast.rst` may need to be updated to reflect changes to AST nodes.
- Add some usage of your new syntax to `test_grammar.py`.
- Certain changes may require tweaks to the library module `pyclbr`.
- `Lib/tokenize.py` needs changes to match changes to the tokenizer.
- Documentation must be written! Specifically, one or more of the pages in `Doc/reference/` will need to be updated.

10.25 Guide to CPython's Parser

Author
Pablo Galindo Salgado

10.25.1 Abstract

The Parser in CPython is currently a PEG (Parser Expression Grammar) parser. The first version of the parser used to be an LL(1) based parser that was one of the oldest parts of CPython implemented before it was replaced by **PEP 617**. In particular, both the current parser and the old LL(1) parser are the output of a parser generator. This means that the way the parser is written is by feeding a description of the Grammar of the Python language to a special program (the parser generator) which outputs the parser. The way the Python language is changed is therefore by modifying the grammar file and developers rarely need to interact with the parser generator itself other than use it to generate the parser.

10.25.2 How PEG Parsers Work

A PEG (Parsing Expression Grammar) grammar (like the current one) differs from a context-free grammar in that the way it is written more closely reflects how the parser will operate when parsing it. The fundamental technical difference is that the choice operator is ordered. This means that when writing:

```
rule: A | B | C
```

a context-free-grammar parser (like an LL(1) parser) will generate constructions that given an input string will *deduce* which alternative (A, B or C) must be expanded, while a PEG parser will check if the first alternative succeeds and only if it fails, will it continue with the second or the third one in the order in which they are written. This makes the choice operator not commutative.

Unlike LL(1) parsers, PEG-based parsers cannot be ambiguous: if a string parses, it has exactly one valid parse tree. This means that a PEG-based parser cannot suffer from the ambiguity problems that can arise with LL(1) parsers and with context-free grammars in general.

PEG parsers are usually constructed as a recursive descent parser in which every rule in the grammar corresponds to a function in the program implementing the parser and the parsing expression (the "expansion" or "definition" of the rule) represents the "code" in said function. Each parsing function conceptually takes an input string as its argument, and yields one of the following results:

- A "success" result. This result indicates that the expression can be parsed by that rule and the function may optionally move forward or consume one or more characters of the input string supplied to it.
- A "failure" result, in which case no input is consumed.

Notice that "failure" results do not imply that the program is incorrect, nor do they necessarily mean that the parsing has failed. Since the choice operator is ordered, a failure very often merely indicates "try the following option". A direct implementation of a PEG parser as a recursive descent parser will present exponential time performance in the worst case, because PEG parsers have infinite lookahead (this means that they can consider an arbitrary number of tokens before deciding for a rule). Usually, PEG parsers avoid this exponential time complexity with a technique called "packrat parsing"[1] which not only loads the entire program in memory before parsing it but also allows the parser to backtrack arbitrarily. This is made efficient by memoizing the rules already matched for each position. The cost of the memoization cache is that the parser will naturally use more memory than a simple LL(1) parser, which normally are table-based.

Key ideas

Important: Don't try to reason about a PEG grammar in the same way you would to with an EBNF or context free grammar. PEG is optimized to describe **how** input strings will be parsed, while context-free grammars are optimized to generate strings of the language they describe (in EBNF, to know if a given string is in the language, you need to do work to find out as it is not immediately obvious from the grammar).

- Alternatives are ordered (`A | B` is not the same as `B | A`).
- If a rule returns a failure, it doesn't mean that the parsing has failed, it just means "try something else".
- By default PEG parsers run in exponential time, which can be optimized to linear by using memoization.
- If parsing fails completely (no rule succeeds in parsing all the input text), the PEG parser doesn't have a concept of "where the `SyntaxError` is".

[1] Ford, Bryan https://pdos.csail.mit.edu/~baford/packrat/thesis/

Consequences or the ordered choice operator

Although PEG may look like EBNF, its meaning is quite different. The fact that in PEG parsers alternatives are ordered (which is at the core of how PEG parsers work) has deep consequences, other than removing ambiguity.

If a rule has two alternatives and the first of them succeeds, the second one is **not** attempted even if the caller rule fails to parse the rest of the input. Thus the parser is said to be "eager". To illustrate this, consider the following two rules (in these examples, a token is an individual character):

```
first_rule:  ( 'a' | 'aa' ) 'a'
second_rule: ('aa' | 'a'  ) 'a'
```

In a regular EBNF grammar, both rules specify the language `{aa, aaa}` but in PEG, one of these two rules accepts the string `aaa` but not the string `aa`. The other does the opposite – it accepts the string `aa` but not the string `aaa`. The rule `('a'|'aa')'a'` does not accept `aaa` because `'a'|'aa'` consumes the first `a`, letting the final `a` in the rule consume the second, and leaving out the third `a`. As the rule has succeeded, no attempt is ever made to go back and let `'a'|'aa'` try the second alternative. The expression `('aa'|'a')'a'` does not accept `aa` because `'aa'|'a'` accepts all of `aa`, leaving nothing for the final `a`. Again, the second alternative of `'aa'|'a'` is not tried.

Caution: The effects of ordered choice, such as the ones illustrated above, may be hidden by many levels of rules.

For this reason, writing rules where an alternative is contained in the next one is in almost all cases a mistake, for example:

```
my_rule:
    | 'if' expression 'then' block
    | 'if' expression 'then' block 'else' block
```

In this example, the second alternative will never be tried because the first one will succeed first (even if the input string has an `'else' block` that follows). To correctly write this rule you can simply alter the order:

```
my_rule:
    | 'if' expression 'then' block 'else' block
    | 'if' expression 'then' block
```

In this case, if the input string doesn't have an `'else' block`, the first alternative will fail and the second will be attempted without said part.

10.25.3 Syntax

The grammar consists of a sequence of rules of the form:

```
rule_name: expression
```

Optionally, a type can be included right after the rule name, which specifies the return type of the C or Python function corresponding to the rule:

```
rule_name[return_type]: expression
```

If the return type is omitted, then a `void *` is returned in C and an `Any` in Python.

Grammar Expressions

`# comment`

Python-style comments.

`e1 e2`

Match `e1`, then match `e2`.

```
rule_name: first_rule second_rule
```

`e1 | e2`

Match `e1` or `e2`.

The first alternative can also appear on the line after the rule name for formatting purposes. In that case, a | must be used before the first alternative, like so:

```
rule_name[return_type]:
    | first_alt
    | second_alt
```

`( e )`

Match `e`.

```
rule_name: (e)
```

A slightly more complex and useful example includes using the grouping operator together with the repeat operators:

```
rule_name: (e1 e2)*
```

`[ e ] or e?`

Optionally match `e`.

```
rule_name: [e]
```

A more useful example includes defining that a trailing comma is optional:

```
rule_name: e (',' e)* [',']
```

e*

Match zero or more occurrences of e.

```
rule_name: (e1 e2)*
```

e+

Match one or more occurrences of e.

```
rule_name: (e1 e2)+
```

s.e+

Match one or more occurrences of e, separated by s. The generated parse tree does not include the separator. This is otherwise identical to (e (s e)*).

```
rule_name: ','.e+
```

&e

Succeed if e can be parsed, without consuming any input.

!e

Fail if e can be parsed, without consuming any input.

An example taken from the Python grammar specifies that a primary consists of an atom, which is not followed by a . or a (or a [:

```
primary: atom !'.' !'(' !'['
```

~

Commit to the current alternative, even if it fails to parse (this is called the "cut").

```
rule_name: '(' ~ some_rule ')' | some_alt
```

In this example, if a left parenthesis is parsed, then the other alternative won't be considered, even if some_rule or) fail to be parsed.

Left recursion

PEG parsers normally do not support left recursion but CPython's parser generator implements a technique similar to the one described in Medeiros et al.[2] but using the memoization cache instead of static variables. This approach is closer to the one described in Warth et al.[3]. This allows us to write not only simple left-recursive rules but also more complicated rules that involve indirect left-recursion like:

```
rule1: rule2 | 'a'
rule2: rule3 | 'b'
rule3: rule1 | 'c'
```

and "hidden left-recursion" like:

```
rule: 'optional'? rule '@' some_other_rule
```

Variables in the Grammar

A sub-expression can be named by preceding it with an identifier and an = sign. The name can then be used in the action (see below), like this:

```
rule_name[return_type]: '(' a=some_other_rule ')' { a }
```

Grammar actions

To avoid the intermediate steps that obscure the relationship between the grammar and the AST generation the PEG parser allows directly generating AST nodes for a rule via grammar actions. Grammar actions are language-specific expressions that are evaluated when a grammar rule is successfully parsed. These expressions can be written in Python or C depending on the desired output of the parser generator. This means that if one would want to generate a parser in Python and another in C, two grammar files should be written, each one with a different set of actions, keeping everything else apart from said actions identical in both files. As an example of a grammar with Python actions, the piece of the parser generator that parses grammar files is bootstrapped from a meta-grammar file with Python actions that generate the grammar tree as a result of the parsing.

In the specific case of the PEG grammar for Python, having actions allows directly describing how the AST is composed in the grammar itself, making it more clear and maintainable. This AST generation process is supported by the use of some helper functions that factor out common AST object manipulations and some other required operations that are not directly related to the grammar.

To indicate these actions each alternative can be followed by the action code inside curly-braces, which specifies the return value of the alternative:

```
rule_name[return_type]:
    | first_alt1 first_alt2 { first_alt1 }
    | second_alt1 second_alt2 { second_alt1 }
```

If the action is omitted, a default action is generated:

- If there's a single name in the rule, it gets returned.
- If there is more than one name in the rule, a collection with all parsed expressions gets returned (the type of the collection will be different in C and Python).

[2] Medeiros et al. https://arxiv.org/pdf/1207.0443.pdf
[3] Warth et al. http://web.cs.ucla.edu/~todd/research/pepm08.pdf

This default behaviour is primarily made for very simple situations and for debugging purposes.

Warning: It's important that the actions don't mutate any AST nodes that are passed into them via variables referring to other rules. The reason for mutation being not allowed is that the AST nodes are cached by memoization and could potentially be reused in a different context, where the mutation would be invalid. If an action needs to change an AST node, it should instead make a new copy of the node and change that.

The full meta-grammar for the grammars supported by the PEG generator is:

```
start[Grammar]: grammar ENDMARKER { grammar }

grammar[Grammar]:
    | metas rules { Grammar(rules, metas) }
    | rules { Grammar(rules, []) }

metas[MetaList]:
    | meta metas { [meta] + metas }
    | meta { [meta] }

meta[MetaTuple]:
    | "@" NAME NEWLINE { (name.string, None) }
    | "@" a=NAME b=NAME NEWLINE { (a.string, b.string) }
    | "@" NAME STRING NEWLINE { (name.string, literal_eval(string.string)) }

rules[RuleList]:
    | rule rules { [rule] + rules }
    | rule { [rule] }

rule[Rule]:
    | rulename ":" alts NEWLINE INDENT more_alts DEDENT {
          Rule(rulename[0], rulename[1], Rhs(alts.alts + more_alts.alts)) }
    | rulename ":" NEWLINE INDENT more_alts DEDENT { Rule(rulename[0], rulename[1], more_
↪alts) }
    | rulename ":" alts NEWLINE { Rule(rulename[0], rulename[1], alts) }

rulename[RuleName]:
    | NAME '[' type=NAME '*' ']' {(name.string, type.string+"*")}
    | NAME '[' type=NAME ']' {(name.string, type.string)}
    | NAME {(name.string, None)}

alts[Rhs]:
    | alt "|" alts { Rhs([alt] + alts.alts)}
    | alt { Rhs([alt]) }

more_alts[Rhs]:
    | "|" alts NEWLINE more_alts { Rhs(alts.alts + more_alts.alts) }
    | "|" alts NEWLINE { Rhs(alts.alts) }

alt[Alt]:
    | items '$' action { Alt(items + [NamedItem(None, NameLeaf('ENDMARKER'))],␣
↪action=action) }
    | items '$' { Alt(items + [NamedItem(None, NameLeaf('ENDMARKER'))], action=None) }
```

(continues on next page)

(continued from previous page)

```
    | items action { Alt(items, action=action) }
    | items { Alt(items, action=None) }

items[NamedItemList]:
    | named_item items { [named_item] + items }
    | named_item { [named_item] }

named_item[NamedItem]:
    | NAME '=' ~ item {NamedItem(name.string, item)}
    | item {NamedItem(None, item)}
    | it=lookahead {NamedItem(None, it)}

lookahead[LookaheadOrCut]:
    | '&' ~ atom {PositiveLookahead(atom)}
    | '!' ~ atom {NegativeLookahead(atom)}
    | '~' {Cut()}

item[Item]:
    | '[' ~ alts ']' {Opt(alts)}
    |  atom '?' {Opt(atom)}
    |  atom '*' {Repeat0(atom)}
    |  atom '+' {Repeat1(atom)}
    |  sep=atom '.' node=atom '+' {Gather(sep, node)}
    |  atom {atom}

atom[Plain]:
    | '(' ~ alts ')' {Group(alts)}
    | NAME {NameLeaf(name.string) }
    | STRING {StringLeaf(string.string)}

# Mini-grammar for the actions

action[str]: "{" ~ target_atoms "}" { target_atoms }

target_atoms[str]:
    | target_atom target_atoms { target_atom + " " + target_atoms }
    | target_atom { target_atom }

target_atom[str]:
    | "{" ~ target_atoms "}" { "{" + target_atoms + "}" }
    | NAME { name.string }
    | NUMBER { number.string }
    | STRING { string.string }
    | "?" { "?" }
    | ":" { ":" }
```

As an illustrative example this simple grammar file allows directly generating a full parser that can parse simple arithmetic expressions and that returns a valid C-based Python AST:

```
start[mod_ty]: a=expr_stmt* ENDMARKER { _PyAST_Module(a, NULL, p->arena) }
expr_stmt[stmt_ty]: a=expr NEWLINE { _PyAST_Expr(a, EXTRA) }

```

(continues on next page)

(continued from previous page)

```
expr[expr_ty]:
    | l=expr '+' r=term { _PyAST_BinOp(l, Add, r, EXTRA) }
    | l=expr '-' r=term { _PyAST_BinOp(l, Sub, r, EXTRA) }
    | term

term[expr_ty]:
    | l=term '*' r=factor { _PyAST_BinOp(l, Mult, r, EXTRA) }
    | l=term '/' r=factor { _PyAST_BinOp(l, Div, r, EXTRA) }
    | factor

factor[expr_ty]:
    | '(' e=expr ')' { e }
    | atom

atom[expr_ty]:
    | NAME
    | NUMBER
```

Here `EXTRA` is a macro that expands to `start_lineno, start_col_offset, end_lineno, end_col_offset, p->arena`, those being variables automatically injected by the parser; `p` points to an object that holds on to all state for the parser.

A similar grammar written to target Python AST objects:

```
start[ast.Module]: a=expr_stmt* ENDMARKER { ast.Module(body=a or [] }
expr_stmt: a=expr NEWLINE { ast.Expr(value=a, EXTRA) }

expr:
    | l=expr '+' r=term { ast.BinOp(left=l, op=ast.Add(), right=r, EXTRA) }
    | l=expr '-' r=term { ast.BinOp(left=l, op=ast.Sub(), right=r, EXTRA) }
    | term

term:
    | l=term '*' r=factor { ast.BinOp(left=l, op=ast.Mult(), right=r, EXTRA) }
    | l=term '/' r=factor { ast.BinOp(left=l, op=ast.Div(), right=r, EXTRA) }
    | factor

factor:
    | '(' e=expr ')' { e }
    | atom

atom:
    | NAME
    | NUMBER
```

10.25.4 Pegen

Pegen is the parser generator used in CPython to produce the final PEG parser used by the interpreter. It is the program that can be used to read the python grammar located in `Grammar/Python.gram` and produce the final C parser. It contains the following pieces:

- A parser generator that can read a grammar file and produce a PEG parser written in Python or C that can parse said grammar. The generator is located at `Tools/peg_generator/pegen`.
- A PEG meta-grammar that automatically generates a Python parser that is used for the parser generator itself (this means that there are no manually-written parsers). The meta-grammar is located at `Tools/peg_generator/pegen/metagrammar.gram`.
- A generated parser (using the parser generator) that can directly produce C and Python AST objects.

The source code for Pegen lives at `Tools/peg_generator/pegen` but normally all typical commands to interact with the parser generator are executed from the main makefile.

How to regenerate the parser

Once you have made the changes to the grammar files, to regenerate the `C` parser (the one used by the interpreter) just execute:

```
make regen-pegen
```

using the `Makefile` in the main directory. If you are on Windows you can use the Visual Studio project files to regenerate the parser or to execute:

```
./PCbuild/build.bat --regen
```

The generated parser file is located at `Parser/parser.c`.

How to regenerate the meta-parser

The meta-grammar (the grammar that describes the grammar for the grammar files themselves) is located at `Tools/peg_generator/pegen/metagrammar.gram`. Although it is very unlikely that you will ever need to modify it, if you make any modifications to this file (in order to implement new Pegen features) you will need to regenerate the meta-parser (the parser that parses the grammar files). To do so just execute:

```
make regen-pegen-metaparser
```

If you are on Windows you can use the Visual Studio project files to regenerate the parser or to execute:

```
./PCbuild/build.bat --regen
```

Grammatical elements and rules

Pegen has some special grammatical elements and rules:

- Strings with single quotes (') (e.g. `'class'`) denote KEYWORDS.
- Strings with double quotes (") (e.g. `"match"`) denote SOFT KEYWORDS.
- Upper case names (e.g. `NAME`) denote tokens in the `Grammar/Tokens` file.
- Rule names starting with `invalid_` are used for specialized syntax errors.
  - These rules are NOT used in the first pass of the parser.
  - Only if the first pass fails to parse, a second pass including the invalid rules will be executed.
  - If the parser fails in the second phase with a generic syntax error, the location of the generic failure of the first pass will be used (this avoids reporting incorrect locations due to the invalid rules).
  - The order of the alternatives involving invalid rules matter (like any rule in PEG).

Tokenization

It is common among PEG parser frameworks that the parser does both the parsing and the tokenization, but this does not happen in Pegen. The reason is that the Python language needs a custom tokenizer to handle things like indentation boundaries, some special keywords like `ASYNC` and `AWAIT` (for compatibility purposes), backtracking errors (such as unclosed parenthesis), dealing with encoding, interactive mode and much more. Some of these reasons are also there for historical purposes, and some others are useful even today.

The list of tokens (all uppercase names in the grammar) that you can use can be found in the `Grammar/Tokens` file. If you change this file to add new tokens, make sure to regenerate the files by executing:

```
make regen-token
```

If you are on Windows you can use the Visual Studio project files to regenerate the tokens or to execute:

```
./PCbuild/build.bat --regen
```

How tokens are generated and the rules governing this is completely up to the tokenizer (`Parser/tokenizer.c`) and the parser just receives tokens from it.

Memoization

As described previously, to avoid exponential time complexity in the parser, memoization is used.

The C parser used by Python is highly optimized and memoization can be expensive both in memory and time. Although the memory cost is obvious (the parser needs memory for storing previous results in the cache) the execution time cost comes for continuously checking if the given rule has a cache hit or not. In many situations, just parsing it again can be faster. Pegen **disables memoization by default** except for rules with the special marker `memo` after the rule name (and type, if present):

```
rule_name[typr] (memo):
    ...
```

By selectively turning on memoization for a handful of rules, the parser becomes faster and uses less memory.

Note: Left-recursive rules always use memoization, since the implementation of left-recursion depends on it.

To know if a new rule needs memoization or not, benchmarking is required (comparing execution times and memory usage of some considerably big files with and without memoization). There is a very simple instrumentation API available in the generated C parse code that allows to measure how much each rule uses memoization (check the `Parser/pegen.c` file for more information) but it needs to be manually activated.

Automatic variables

To make writing actions easier, Pegen injects some automatic variables in the namespace available when writing actions. In the C parser, some of these automatic variable names are:

- `p`: The parser structure.
- `EXTRA`: This is a macro that expands to `(_start_lineno, _start_col_offset, _end_lineno, _end_col_offset, p->arena)`, which is normally used to create AST nodes as almost all constructors need these attributes to be provided. All of the location variables are taken from the location information of the current token.

Hard and Soft keywords

Note: In the grammar files, keywords are defined using **single quotes** (e.g. *'class'*) while soft keywords are defined using **double quotes** (e.g. *"match"*).

There are two kinds of keywords allowed in pegen grammars: *hard* and *soft* keywords. The difference between hard and soft keywords is that hard keywords are always reserved words, even in positions where they make no sense (e.g. `x = class + 1`), while soft keywords only get a special meaning in context. Trying to use a hard keyword as a variable will always fail:

```
>>> class = 3
File "<stdin>", line 1
    class = 3
        ^
SyntaxError: invalid syntax
>>> foo(class=3)
File "<stdin>", line 1
    foo(class=3)
        ^^^^^
SyntaxError: invalid syntax
```

While soft keywords don't have this limitation if used in a context other the one where they are defined as keywords:

```
>>> match = 45
>>> foo(match="Yeah!")
```

The `match` and `case` keywords are soft keywords, so that they are recognized as keywords at the beginning of a match statement or case block respectively, but are allowed to be used in other places as variable or argument names.

You can get a list of all keywords defined in the grammar from Python:

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
```

(continues on next page)

(continued from previous page)

```
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

as well as soft keywords:

```
>>> import keyword
>>> keyword.softkwlist
['_', 'case', 'match']
```

Caution: Soft keywords can be a bit challenging to manage as they can be accepted in places you don't intend to, given how the order alternatives behave in PEG parsers (see *consequences of ordered choice section* for some background on this). In general, try to define them in places where there is not a lot of alternatives.

Error handling

When a pegen-generated parser detects that an exception is raised, it will **automatically stop parsing**, no matter what the current state of the parser is and it will unwind the stack and report the exception. This means that if a *rule action* raises an exception all parsing will stop at that exact point. This is done to allow to correctly propagate any exception set by calling Python C-API functions. This also includes `SyntaxError` exceptions and this is the main mechanism the parser uses to report custom syntax error messages.

Note: Tokenizer errors are normally reported by raising exceptions but some special tokenizer errors such as unclosed parenthesis will be reported only after the parser finishes without returning anything.

How Syntax errors are reported

As described previously in the *how PEG parsers work section*, PEG parsers don't have a defined concept of where errors happened in the grammar, because a rule failure doesn't imply a parsing failure like in context free grammars. This means that some heuristic has to be used to report generic errors unless something is explicitly declared as an error in the grammar.

To report generic syntax errors, pegen uses a common heuristic in PEG parsers: the location of *generic* syntax errors is reported in the furthest token that was attempted to be matched but failed. This is only done if parsing has failed (the parser returns `NULL` in C or `None` in Python) but no exception has been raised.

Caution: Positive and negative lookaheads will try to match a token so they will affect the location of generic syntax errors. Use them carefully at boundaries between rules.

As the Python grammar was primordially written as an LL(1) grammar, this heuristic has an extremely high success rate, but some PEG features can have small effects, such as *positive lookaheads* and *negative lookaheads*.

To generate more precise syntax errors, custom rules are used. This is a common practice also in context free grammars: the parser will try to accept some construct that is known to be incorrect just to report a specific syntax error for that construct. In pegen grammars, these rules start with the `invalid_` prefix. This is because trying to match these rules normally has a performance impact on parsing (and can also affect the 'correct' grammar itself in some tricky cases, depending on the ordering of the rules) so the generated parser acts in two phases:

1. The first phase will try to parse the input stream without taking into account rules that start with the `invalid_` prefix. If the parsing succeeds it will return the generated AST and the second phase will not be attempted.
2. If the first phase failed, a second parsing attempt is done including the rules that start with an `invalid_` prefix. By design this attempt **cannot succeed** and is only executed to give to the invalid rules a chance to detect specific situations where custom, more precise, syntax errors can be raised. This also allows to trade a bit of performance for precision reporting errors: given that we know that the input text is invalid, there is no need to be fast because the interpreter is going to stop anyway.

Important: When defining invalid rules:

- Make sure all custom invalid rules raise `SyntaxError` exceptions (or a subclass of it).
- Make sure **all** invalid rules start with the `invalid_` prefix to not impact performance of parsing correct Python code.
- Make sure the parser doesn't behave differently for regular rules when you introduce invalid rules (see the *how PEG parsers work section* for more information).

You can find a collection of macros to raise specialized syntax errors in the `Parser/pegen.h` header file. These macros allow also to report ranges for the custom errors that will be highlighted in the tracebacks that will be displayed when the error is reported.

Tip: A good way to test if an invalid rule will be triggered when you expect is to test if introducing a syntax error **after** valid code triggers the rule or not. For example:

```
<valid python code> $ 42
```

Should trigger the syntax error in the `$` character. If your rule is not correctly defined this won't happen. For example, if you try to define a rule to match Python 2 style `print` statements to make a better error message and you define it as:

```
invalid_print: "print" expression
```

This will **seem** to work because the parser will correctly parse `print(something)` because it is valid code and the second phase will never execute but if you try to parse `print(something) $ 3` the first pass of the parser will fail (because of the `$`) and in the second phase, the rule will match the `print(something)` as `print` followed by the variable `something` between parentheses and the error will be reported there instead of the `$` character.

Generating AST objects

The output of the C parser used by CPython that is generated by the `Grammar/Python.gram` grammar file is a Python AST object (using C structures). This means that the actions in the grammar file generate AST objects when they succeed. Constructing these objects can be quite cumbersome (see the *AST compiler section* for more information on how these objects are constructed and how they are used by the compiler) so special helper functions are used. These functions are declared in the `Parser/pegen.h` header file and defined in the `Parser/action_helpers.c` file. These functions allow you to join AST sequences, get specific elements from them or to do extra processing on the generated tree.

Caution: Actions must **never** be used to accept or reject rules. It may be tempting in some situations to write a very generic rule and then check the generated AST to decide if is valid or not but this will render the official grammar partially incorrect (because actions are not included) and will make it more difficult for other Python implementations to adapt the grammar to their own needs.

As a general rule, if an action spawns multiple lines or requires something more complicated than a single expression of C code, is normally better to create a custom helper in `Parser/action_helpers.c` and expose it in the `Parser/pegen.h` header file so it can be used from the grammar.

If the parsing succeeds, the parser **must** return a **valid** AST object.

10.25.5 Testing

There are three files that contain tests for the grammar and the parser:

- *Lib/test/test_grammar.py*.
- *Lib/test/test_syntax.py*.
- *Lib/test/test_exceptions.py*.

Check the contents of these files to know which is the best place to place new tests depending on the nature of the new feature you are adding.

Tests for the parser generator itself can be found in the `Lib/test/test_peg_generator` directory.

10.25.6 Debugging generated parsers

Making experiments

As the generated C parser is the one used by Python, this means that if something goes wrong when adding some new rules to the grammar you cannot correctly compile and execute Python anymore. This makes it a bit challenging to debug when something goes wrong, especially when making experiments.

For this reason it is a good idea to experiment first by generating a Python parser. To do this, you can go to the `Tools/peg_generator/` directory on the CPython repository and manually call the parser generator by executing:

```
$ python -m pegen python <PATH TO YOUR GRAMMAR FILE>
```

This will generate a file called `parse.py` in the same directory that you can use to parse some input:

```
$ python parse.py file_with_source_code_to_test.py
```

As the generated `parse.py` file is just Python code, you can modify it and add breakpoints to debug or better understand some complex situations.

Verbose mode

When Python is compiled in debug mode (by adding `--with-pydebug` when running the configure step in Linux or by adding `-d` when calling the `PCbuild/build.bat` script in Windows), it is possible to activate a **very** verbose mode in the generated parser. This is very useful to debug the generated parser and to understand how it works, but it can be a bit hard to understand at first.

Note: When activating verbose mode in the Python parser, it is better to not use interactive mode as it can be much harder to understand, because interactive mode involves some special steps compared to regular parsing.

To activate verbose mode you can add the `-d` flag when executing Python:

```
$ python -d file_to_test.py
```

This will print **a lot** of output to `stderr` so is probably better to dump it to a file for further analysis. The output consists of trace lines with the following structure:

<indentation> ('>'|'-'|'+'|'!') <rule_name>[<token_location>]: <alternative> ...

Every line is indented by a different amount (`<indentation>`) depending on how deep the call stack is. The next character marks the type of the trace:

- `>` indicates that a rule is going to be attempted to be parsed.
- `-` indicates that a rule has failed to be parsed.
- `+` indicates that a rule has been parsed correctly.
- `!` indicates that an exception or an error has been detected and the parser is unwinding.

The <token_location> part indicates the current index in the token array, the <rule_name> part indicates what rule is being parsed and the <alternative> part indicates what alternative within that rule is being attempted.

10.25.7 References

10.26 Design of CPython's Compiler

10.26.1 Abstract

In CPython, the compilation from source code to bytecode involves several steps:

1. Tokenize the source code (`Parser/tokenizer.c`)
2. Parse the stream of tokens into an Abstract Syntax Tree (`Parser/parser.c`)
3. Transform AST into a Control Flow Graph (`Python/compile.c`)
4. Emit bytecode based on the Control Flow Graph (`Python/compile.c`)

The purpose of this document is to outline how these steps of the process work.

This document does not touch on how parsing works beyond what is needed to explain what is needed for compilation. It is also not exhaustive in terms of the how the entire system works. You will most likely need to read some source to have an exact understanding of all details.

10.26.2 Parsing

As of Python 3.9, Python's parser is a PEG parser of a somewhat unusual design (since its input is a stream of tokens rather than a stream of characters as is more common with PEG parsers).

The grammar file for Python can be found in `Grammar/python.gram`. The definitions for literal tokens (such as `:`, numbers, etc.) can be found in `Grammar/Tokens`. Various C files, including `Parser/parser.c` are generated from these (see *Changing CPython's Grammar*).

10.26.3 Abstract Syntax Trees (AST)

Green Tree Snakes

See also Green Tree Snakes - the missing Python AST docs by Thomas Kluyver.

The abstract syntax tree (AST) is a high-level representation of the program structure without the necessity of containing the source code; it can be thought of as an abstract representation of the source code. The specification of the AST nodes is specified using the Zephyr Abstract Syntax Definition Language (ASDL) [Wang97].

The definition of the AST nodes for Python is found in the file `Parser/Python.asdl`.

Each AST node (representing statements, expressions, and several specialized types, like list comprehensions and exception handlers) is defined by the ASDL. Most definitions in the AST correspond to a particular source construct, such as an 'if' statement or an attribute lookup. The definition is independent of its realization in any particular programming language.

The following fragment of the Python ASDL construct demonstrates the approach and syntax:

```
module Python
{
    stmt = FunctionDef(identifier name, arguments args, stmt* body,
                        expr* decorators)
           | Return(expr? value) | Yield(expr? value)
           attributes (int lineno)
}
```

The preceding example describes two different kinds of statements and an expression: function definitions, return statements, and yield expressions. All three kinds are considered of type `stmt` as shown by `|` separating the various kinds. They all take arguments of various kinds and amounts.

Modifiers on the argument type specify the number of values needed; `?` means it is optional, `*` means 0 or more, while no modifier means only one value for the argument and it is required. `FunctionDef`, for instance, takes an `identifier` for the *name*, `arguments` for *args*, zero or more `stmt` arguments for *body*, and zero or more `expr` arguments for *decorators*.

Do notice that something like 'arguments', which is a node type, is represented as a single AST node and not as a sequence of nodes as with stmt as one might expect.

All three kinds also have an 'attributes' argument; this is shown by the fact that 'attributes' lacks a '|' before it.

The statement definitions above generate the following C structure type:

```
typedef struct _stmt *stmt_ty;

struct _stmt {
      enum { FunctionDef_kind=1, Return_kind=2, Yield_kind=3 } kind;
      union {
              struct {
                      identifier name;
                      arguments_ty args;
                      asdl_seq *body;
              } FunctionDef;

              struct {
```

(continues on next page)

(continued from previous page)

```
                        expr_ty value;
                } Return;

                struct {
                        expr_ty value;
                } Yield;
        } v;
        int lineno;
}
```

Also generated are a series of constructor functions that allocate (in this case) a `stmt_ty` struct with the appropriate initialization. The `kind` field specifies which component of the union is initialized. The `FunctionDef()` constructor function sets 'kind' to `FunctionDef_kind` and initializes the *name*, *args*, *body*, and *attributes* fields.

10.26.4 Memory Management

Before discussing the actual implementation of the compiler, a discussion of how memory is handled is in order. To make memory management simple, an arena is used. This means that a memory is pooled in a single location for easy allocation and removal. What this gives us is the removal of explicit memory deallocation. Because memory allocation for all needed memory in the compiler registers that memory with the arena, a single call to free the arena is all that is needed to completely free all memory used by the compiler.

In general, unless you are working on the critical core of the compiler, memory management can be completely ignored. But if you are working at either the very beginning of the compiler or the end, you need to care about how the arena works. All code relating to the arena is in either `Include/Internal/pycore_pyarena.h` or `Python/pyarena.c`.

`PyArena_New()` will create a new arena. The returned `PyArena` structure will store pointers to all memory given to it. This does the bookkeeping of what memory needs to be freed when the compiler is finished with the memory it used. That freeing is done with `PyArena_Free()`. This only needs to be called in strategic areas where the compiler exits.

As stated above, in general you should not have to worry about memory management when working on the compiler. The technical details have been designed to be hidden from you for most cases.

The only exception comes about when managing a PyObject. Since the rest of Python uses reference counting, there is extra support added to the arena to cleanup each PyObject that was allocated. These cases are very rare. However, if you've allocated a PyObject, you must tell the arena about it by calling `PyArena_AddPyObject()`.

10.26.5 Source Code to AST

The AST is generated from source code using the function `_PyParser_ASTFromString()` or `_PyParser_ASTFromFile()` (from `Parser/peg_api.c`) depending on the input type.

After some checks, a helper function in `Parser/parser.c` begins applying production rules on the source code it receives; converting source code to tokens and matching these tokens recursively to their corresponding rule. The rule's corresponding rule function is called on every match. These rule functions follow the format `xx_rule`. Where *xx* is the grammar rule that the function handles and is automatically derived from `Grammar/python.gram` by `Tools/peg_generator/pegen/c_generator.py`.

Each rule function in turn creates an AST node as it goes along. It does this by allocating all the new nodes it needs, calling the proper AST node creation functions for any required supporting functions and connecting them as needed. This continues until all nonterminal symbols are replaced with terminals. If an error occurs, the rule functions backtrack and try another rule function. If there are no more rules, an error is set and the parsing ends.

The AST node creation helper functions have the name `_PyAST_xx` where *xx* is the AST node that the function creates. These are defined by the ASDL grammar and contained in `Python/Python-ast.c` (which is generated by `Parser/asdl_c.py` from `Parser/Python.asdl`). This all leads to a sequence of AST nodes stored in `asdl_seq` structs.

To demonstrate everything explained so far, here's the rule function responsible for a simple named import statement such as `import sys`. Note that error-checking and debugging code has been omitted. Removed parts are represented by `...`. Furthermore, some comments have been added for explanation. These comments may not be present in the actual code.

```
// This is the production rule (from python.gram) the rule function
// corresponds to:
// import_name: 'import' dotted_as_names
static stmt_ty
import_name_rule(Parser *p)
{
    ...
    stmt_ty _res = NULL;
    { // 'import' dotted_as_names
        ...
        Token * _keyword;
        asdl_alias_seq* a;
        // The tokenizing steps.
        if (
            (_keyword = _PyPegen_expect_token(p, 513))  // token='import'
            &&
            (a = dotted_as_names_rule(p))  // dotted_as_names
        )
        {
            ...
            // Generate an AST for the import statement.
            _res = _PyAST_Import ( a , ...);
            ...
            goto done;
        }
        ...
    }
    _res = NULL;
  done:
    ...
    return _res;
}
```

To improve backtracking performance, some rules (chosen by applying a `(memo)` flag in the grammar file) are memoized. Each rule function checks if a memoized version exists and returns that if so, else it continues in the manner stated in the previous paragraphs.

There are macros for creating and using `asdl_xx_seq *` types, where *xx* is a type of the ASDL sequence. Three main types are defined manually – `generic`, `identifier` and `int`. These types are found in `Python/asdl.c` and its corresponding header file `Include/Internal/pycore_asdl.h`. Functions and macros for creating `asdl_xx_seq *` types are as follows:

`_Py_asdl_generic_seq_new(Py_ssize_t, PyArena *)`
    Allocate memory for an `asdl_generic_seq` of the specified length

`_Py_asdl_identifier_seq_new(Py_ssize_t, PyArena *)`
    Allocate memory for an `asdl_identifier_seq` of the specified length

`_Py_asdl_int_seq_new(Py_ssize_t, PyArena *)`
Allocate memory for an `asdl_int_seq` of the specified length

In addition to the three types mentioned above, some ASDL sequence types are automatically generated by `Parser/asdl_c.py` and found in `Include/Internal/pycore_ast.h`. Macros for using both manually defined and automatically generated ASDL sequence types are as follows:

`asdl_seq_GET(asdl_xx_seq *, int)`
Get item held at a specific position in an `asdl_xx_seq`

`asdl_seq_SET(asdl_xx_seq *, int, stmt_ty)`
Set a specific index in an `asdl_xx_seq` to the specified value

Untyped counterparts exist for some of the typed macros. These are useful when a function needs to manipulate a generic ASDL sequence:

`asdl_seq_GET_UNTYPED(asdl_seq *, int)`
Get item held at a specific position in an `asdl_seq`

`asdl_seq_SET_UNTYPED(asdl_seq *, int, stmt_ty)`
Set a specific index in an `asdl_seq` to the specified value

`asdl_seq_LEN(asdl_seq *)`
Return the length of an `asdl_seq` or `asdl_xx_seq`

Note that typed macros and functions are recommended over their untyped counterparts. Typed macros carry out checks in debug mode and aid debugging errors caused by incorrectly casting from `void *`.

If you are working with statements, you must also worry about keeping track of what line number generated the statement. Currently the line number is passed as the last parameter to each `stmt_ty` function.

Changed in version 3.9: The new PEG parser generates an AST directly without creating a parse tree. `Python/ast.c` is now only used to validate the AST for debugging purposes.

See also:

PEP 617 (PEP 617 – New PEG parser for CPython)

10.26.6 Control Flow Graphs

A *control flow graph* (often referenced by its acronym, CFG) is a directed graph that models the flow of a program. A node of a CFG is not an individual bytecode instruction, but instead represents a sequence of bytecode instructions that always execute sequentially. Each node is called a *basic block* and must always execute from start to finish, with a single entry point at the beginning and a single exit point at the end. If some bytecode instruction *a* needs to jump to some other bytecode instruction *b*, then *a* must occur at the end of its basic block, and *b* must occur at the start of its basic block.

As an example, consider the following code snippet:

```
if x < 10:
    f1()
    f2()
else:
    g()
end()
```

The `x < 10` guard is represented by its own basic block that compares `x` with `10` and then ends in a conditional jump based on the result of the comparison. This conditional jump allows the block to point to both the body of the `if` and the body of the `else`. The `if` basic block contains the `f1()` and `f2()` calls and points to the `end()` basic block. The `else` basic block contains the `g()` call and similarly points to the `end()` block.

Note that more complex code in the guard, the `if` body, or the `else` body may be represented by multiple basic blocks. For instance, short-circuiting boolean logic in a guard like `if x or y:` will produce one basic block that tests the truth value of `x` and then points both (1) to the start of the `if` body and (2) to a different basic block that tests the truth value of y.

CFGs are usually one step away from final code output. Code is directly generated from the basic blocks (with jump targets adjusted based on the output order) by doing a post-order depth-first search on the CFG following the edges.

10.26.7 AST to CFG to Bytecode

With the AST created, the next step is to create the CFG. The first step is to convert the AST to Python bytecode without having jump targets resolved to specific offsets (this is calculated when the CFG goes to final bytecode). Essentially, this transforms the AST into Python bytecode with control flow represented by the edges of the CFG.

Conversion is done in two passes. The first creates the namespace (variables can be classified as local, free/cell for closures, or global). With that done, the second pass essentially flattens the CFG into a list and calculates jump offsets for final output of bytecode.

The conversion process is initiated by a call to the function `_PyAST_Compile()` in `Python/compile.c`. This function does both the conversion of the AST to a CFG and outputting final bytecode from the CFG. The AST to CFG step is handled mostly by two functions called by `_PyAST_Compile()`; `_PySymtable_Build()` and `compiler_mod()`. The former is in `Python/symtable.c` while the latter is in `Python/compile.c`.

`_PySymtable_Build()` begins by entering the starting code block for the AST (passed-in) and then calling the proper `symtable_visit_xx` function (with *xx* being the AST node type). Next, the AST tree is walked with the various code blocks that delineate the reach of a local variable as blocks are entered and exited using `symtable_enter_block()` and `symtable_exit_block()`, respectively.

Once the symbol table is created, it is time for CFG creation, whose code is in `Python/compile.c`. This is handled by several functions that break the task down by various AST node types. The functions are all named `compiler_visit_xx` where *xx* is the name of the node type (such as `stmt`, `expr`, etc.). Each function receives a `struct compiler *` and `xx_ty` where *xx* is the AST node type. Typically these functions consist of a large 'switch' statement, branching based on the kind of node type passed to it. Simple things are handled inline in the 'switch' statement with more complex transformations farmed out to other functions named `compiler_xx` with *xx* being a descriptive name of what is being handled.

When transforming an arbitrary AST node, use the `VISIT()` macro. The appropriate `compiler_visit_xx` function is called, based on the value passed in for <node type> (so `VISIT(c, expr, node)` calls `compiler_visit_expr(c, node)`). The `VISIT_SEQ()` macro is very similar, but is called on AST node sequences (those values that were created as arguments to a node that used the '*' modifier). There is also `VISIT_SLICE()` just for handling slices.

Emission of bytecode is handled by the following macros:

`ADDOP(struct compiler *, int)`
add a specified opcode

`ADDOP_NOLINE(struct compiler *, int)`
like `ADDOP` without a line number; used for artificial opcodes without no corresponding token in the source code

`ADDOP_IN_SCOPE(struct compiler *, int)`
like `ADDOP`, but also exits current scope; used for adding return value opcodes in lambdas and closures

`ADDOP_I(struct compiler *, int, Py_ssize_t)`
add an opcode that takes an integer argument

`ADDOP_O(struct compiler *, int, PyObject *, TYPE)`
add an opcode with the proper argument based on the position of the specified PyObject in PyObject sequence object, but with no handling of mangled names; used for when you need to do named lookups of objects such as

globals, consts, or parameters where name mangling is not possible and the scope of the name is known; *TYPE* is the name of PyObject sequence (`names` or `varnames`)

`ADDOP_N(struct compiler *, int, PyObject *, TYPE)`
just like `ADDOP_O`, but steals a reference to PyObject

`ADDOP_NAME(struct compiler *, int, PyObject *, TYPE)`
just like `ADDOP_O`, but name mangling is also handled; used for attribute loading or importing based on name

`ADDOP_LOAD_CONST(struct compiler *, PyObject *)`
add the `LOAD_CONST` opcode with the proper argument based on the position of the specified PyObject in the consts table.

`ADDOP_LOAD_CONST_NEW(struct compiler *, PyObject *)`
just like `ADDOP_LOAD_CONST_NEW`, but steals a reference to PyObject

`ADDOP_JUMP(struct compiler *, int, basicblock *)`
create a jump to a basic block

`ADDOP_JUMP_NOLINE(struct compiler *, int, basicblock *)`
like `ADDOP_JUMP` without a line number; used for artificial jumps without no corresponding token in the source code.

`ADDOP_JUMP_COMPARE(struct compiler *, cmpop_ty)`
depending on the second argument, add an `ADDOP_I` with either an `IS_OP`, `CONTAINS_OP`, or `COMPARE_OP` opcode.

Several helper functions that will emit bytecode and are named `compiler_xx()` where *xx* is what the function helps with (`list`, `boolop`, etc.). A rather useful one is `compiler_nameop()`. This function looks up the scope of a variable and, based on the expression context, emits the proper opcode to load, store, or delete the variable.

As for handling the line number on which a statement is defined, this is handled by `compiler_visit_stmt()` and thus is not a worry.

In addition to emitting bytecode based on the AST node, handling the creation of basic blocks must be done. Below are the macros and functions used for managing basic blocks:

`NEXT_BLOCK(struct compiler *)`
create an implicit jump from the current block to the new block

`compiler_new_block(struct compiler *)`
create a block but don't use it (used for generating jumps)

`compiler_use_next_block(struct compiler *, basicblock *block)`
set a previously created block as a current block

Once the CFG is created, it must be flattened and then final emission of bytecode occurs. Flattening is handled using a post-order depth-first search. Once flattened, jump offsets are backpatched based on the flattening and then a `PyCodeObject` is created. All of this is handled by calling `assemble()`.

10.26.8 Introducing New Bytecode

Sometimes a new feature requires a new opcode. But adding new bytecode is not as simple as just suddenly introducing new bytecode in the AST -> bytecode step of the compiler. Several pieces of code throughout Python depend on having correct information about what bytecode exists.

First, you must choose a name and a unique identifier number. The official list of bytecode can be found in `Lib/opcode.py`. If the opcode is to take an argument, it must be given a unique number greater than that assigned to `HAVE_ARGUMENT` (as found in `Lib/opcode.py`).

Once the name/number pair has been chosen and entered in `Lib/opcode.py`, you must also enter it into `Doc/library/dis.rst`, and regenerate `Include/opcode.h` and `Python/opcode_targets.h` by running `make regen-opcode regen-opcode-targets`.

With a new bytecode you must also change what is called the magic number for .pyc files. The variable `MAGIC_NUMBER` in `Lib/importlib/_bootstrap_external.py` contains the number. Changing this number will lead to all .pyc files with the old `MAGIC_NUMBER` to be recompiled by the interpreter on import. Whenever `MAGIC_NUMBER` is changed, the ranges in the `magic_values` array in `PC/launcher.c` must also be updated. Changes to `Lib/importlib/_bootstrap_external.py` will take effect only after running `make regen-importlib`. Running this command before adding the new bytecode target to `Python/ceval.c` will result in an error. You should only run `make regen-importlib` after the new bytecode target has been added.

Note: On Windows, running the `./build.bat` script will automatically regenerate the required files without requiring additional arguments.

Finally, you need to introduce the use of the new bytecode. Altering `Python/compile.c` and `Python/ceval.c` will be the primary places to change. You must add the case for a new opcode into the 'switch' statement in the `stack_effect()` function in `Python/compile.c`. If the new opcode has a jump target, you will need to update macros and 'switch' statements in `Python/peephole.c`. If it affects a control flow or the block stack, you may have to update the `frame_setlineno()` function in `Objects/frameobject.c`. `Lib/dis.py` may need an update if the new opcode interprets its argument in a special way (like `FORMAT_VALUE` or `MAKE_FUNCTION`).

If you make a change here that can affect the output of bytecode that is already in existence and you do not change the magic number constantly, make sure to delete your old .py(c|o) files! Even though you will end up changing the magic number if you change the bytecode, while you are debugging your work you will be changing the bytecode output without constantly bumping up the magic number. This means you end up with stale .pyc files that will not be recreated. Running `find . -name '*.py[co]' -exec rm -f '{}'` + should delete all .pyc files you have, forcing new ones to be created and thus allow you test out your new bytecode properly. Run `make regen-importlib` for updating the bytecode of frozen importlib files. You have to run `make` again after this for recompiling generated C files.

10.26.9 Code Objects

The result of `PyAST_CompileObject()` is a `PyCodeObject` which is defined in `Include/code.h`. And with that you now have executable Python bytecode!

The code objects (byte code) are executed in `Python/ceval.c`. This file will also need a new case statement for the new opcode in the big switch statement in `_PyEval_EvalFrameDefault()`.

10.26.10 Important Files

- Parser/

  Python.asdl
  ASDL syntax file

  asdl.py
  Parser for ASDL definition files. Reads in an ASDL description and parses it into an AST that describes it.

  asdl_c.py
  "Generate C code from an ASDL description." Generates `Python/Python-ast.c` and `Include/Internal/pycore_ast.h`.

parser.c
The new PEG parser introduced in Python 3.9. Generated by `Tools/peg_generator/pegen/c_generator.py` from the grammar `Grammar/python.gram`. Creates the AST from source code. Rule functions for their corresponding production rules are found here.

peg_api.c
Contains high-level functions which are used by the interpreter to create an AST from source code .

pegen.c
Contains helper functions which are used by functions in `Parser/parser.c` to construct the AST. Also contains helper functions which help raise better error messages when parsing source code.

pegen.h
Header file for the corresponding `Parser/pegen.c`. Also contains definitions of the `Parser` and `Token` structs.

- Python/

Python-ast.c
Creates C structs corresponding to the ASDL types. Also contains code for marshalling AST nodes (core ASDL types have marshalling code in `asdl.c`). "File automatically generated by `Parser/asdl_c.py`". This file must be committed separately after every grammar change is committed since the `__version__` value is set to the latest grammar change revision number.

asdl.c
Contains code to handle the ASDL sequence type. Also has code to handle marshalling the core ASDL types, such as number and identifier. Used by `Python-ast.c` for marshalling AST nodes.

ast.c
Used for validating the AST.

ast_opt.c
Optimizes the AST.

ast_unparse.c
Converts the AST expression node back into a string (for string annotations).

ceval.c
Executes byte code (aka, eval loop).

compile.c
Emits bytecode based on the AST.

symtable.c
Generates a symbol table from AST.

peephole.c
Optimizes the bytecode.

pyarena.c
Implementation of the arena memory manager.

wordcode_helpers.h
Helpers for generating bytecode.

opcode_targets.h
One of the files that must be modified if `Lib/opcode.py` is.

- Include/

code.h
Header file for `Objects/codeobject.c`; contains definition of `PyCodeObject`.

opcode.h
One of the files that must be modified if `Lib/opcode.py` is.

- Internal/

  pycore_ast.h
  Contains the actual definitions of the C structs as generated by `Python/Python-ast.c`. "Automatically generated by `Parser/asdl_c.py`".

  pycore_asdl.h
  Header for the corresponding `Python/ast.c`

  pycore_ast.h
  Declares `_PyAST_Validate()` external (from `Python/ast.c`).

  pycore_symtable.h
  Header for `Python/symtable.c`. `struct symtable` and `PySTEntryObject` are defined here.

  pycore_parser.h
  Header for the corresponding `Parser/peg_api.c`.

  pycore_pyarena.h
  Header file for the corresponding `Python/pyarena.c`.

- Objects/

  codeobject.c
  Contains PyCodeObject-related code (originally in `Python/compile.c`).

  frameobject.c
  Contains the `frame_setlineno()` function which should determine whether it is allowed to make a jump between two points in a bytecode.

- Lib/

  opcode.py
  Master list of bytecode; if this file is modified you must modify several other files accordingly (see "*Introducing New Bytecode*")

  importlib/_bootstrap_external.py
  Home of the magic number (named `MAGIC_NUMBER`) for bytecode versioning.

10.26.11 Known Compiler-related Experiments

This section lists known experiments involving the compiler (including bytecode).

Skip Montanaro presented a paper at a Python workshop on a peephole optimizer[1].

Michael Hudson has a non-active SourceForge project named Bytecodehacks[2] that provides functionality for playing with bytecode directly.

An opcode to combine the functionality of `LOAD_ATTR/CALL_FUNCTION` was created named `CALL_ATTR`[3]. Currently only works for classic classes and for new-style classes rough benchmarking showed an actual slowdown thanks to having to support both classic and new-style classes.

[1] Skip Montanaro's Peephole Optimizer Paper (https://legacy.python.org/workshops/1998-11/proceedings/papers/montanaro/montanaro.html)
[2] Bytecodehacks Project (http://bytecodehacks.sourceforge.net/bch-docs/bch/index.html)
[3] CALL_ATTR opcode (https://bugs.python.org/issue709744)

10.26.12 References

10.27 Design of CPython's Garbage Collector

Author
Pablo Galindo Salgado

10.27.1 Abstract

The main garbage collection algorithm used by CPython is reference counting. The basic idea is that CPython counts how many different places there are that have a reference to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When an object's reference count becomes zero, the object is deallocated. If it contains references to other objects, their reference counts are decremented. Those other objects may be deallocated in turn, if this decrement makes their reference count become zero, and so on. The reference count field can be examined using the `sys.getrefcount` function (notice that the value returned by this function is always 1 more as the function also has a reference to the object when called):

```
>>> x = object()
>>> sys.getrefcount(x)
2
>>> y = x
>>> sys.getrefcount(x)
3
>>> del y
>>> sys.getrefcount(x)
2
```

The main problem with the reference counting scheme is that it does not handle reference cycles. For instance, consider this code:

```
>>> container = []
>>> container.append(container)
>>> sys.getrefcount(container)
3
>>> del container
```

In this example, `container` holds a reference to itself, so even when we remove our reference to it (the variable "container") the reference count never falls to 0 because it still has its own internal reference. Therefore it would never be cleaned just by simple reference counting. For this reason some additional machinery is needed to clean these reference cycles between objects once they become unreachable. This is the cyclic garbage collector, usually called just Garbage Collector (GC), even though reference counting is also a form of garbage collection.

10.27.2 Memory layout and object structure

Normally the C structure supporting a regular Python object looks as follows:

```
object -----> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ \
              |                    ob_refcnt                  | |
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ | PyObject_HEAD
              |                    *ob_type                   | |
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ /
              |                      ...                      |
```

In order to support the garbage collector, the memory layout of objects is altered to accommodate extra information **before** the normal layout:

```
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ \
              |                    *_gc_next                  | |
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ | PyGC_Head
              |                    *_gc_prev                  | |
object -----> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ /
              |                    ob_refcnt                  | \
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ | PyObject_HEAD
              |                    *ob_type                   | |
              +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ /
              |                      ...                      |
```

In this way the object can be treated as a normal python object and when the extra information associated to the GC is needed the previous fields can be accessed by a simple type cast from the original object: `((PyGC_Head *)(the_object)-1)`.

As is explained later in the *Optimization: reusing fields to save memory* section, these two extra fields are normally used to keep doubly linked lists of all the objects tracked by the garbage collector (these lists are the GC generations, more on that in the *Optimization: generations* section), but they are also reused to fulfill other purposes when the full doubly linked list structure is not needed as a memory optimization.

Doubly linked lists are used because they efficiently support most frequently required operations. In general, the collection of all objects tracked by GC are partitioned into disjoint sets, each in its own doubly linked list. Between collections, objects are partitioned into "generations", reflecting how often they've survived collection attempts. During collections, the generation(s) being collected are further partitioned into, e.g., sets of reachable and unreachable objects. Doubly linked lists support moving an object from one partition to another, adding a new object, removing an object entirely (objects tracked by GC are most often reclaimed by the refcounting system when GC isn't running at all!), and merging partitions, all with a small constant number of pointer updates. With care, they also support iterating over a partition while objects are being added to - and removed from - it, which is frequently required while GC is running.

Specific APIs are offered to allocate, deallocate, initialize, track, and untrack objects with GC support. These APIs can be found in the Garbage Collector C API documentation.

Apart from this object structure, the type object for objects supporting garbage collection must include the `Py_TPFLAGS_HAVE_GC` in its `tp_flags` slot and provide an implementation of the `tp_traverse` handler. Unless it can be proven that the objects cannot form reference cycles with only objects of its type or unless the type is immutable, a `tp_clear` implementation must also be provided.

10.27.3 Identifying reference cycles

The algorithm that CPython uses to detect those reference cycles is implemented in the `gc` module. The garbage collector **only focuses** on cleaning container objects (i.e. objects that can contain a reference to one or more objects). These can be arrays, dictionaries, lists, custom class instances, classes in extension modules, etc. One could think that cycles are uncommon but the truth is that many internal references needed by the interpreter create cycles everywhere. Some notable examples:

- Exceptions contain traceback objects that contain a list of frames that contain the exception itself.
- Module-level functions reference the module's dict (which is needed to resolve globals), which in turn contains entries for the module-level functions.
- Instances have references to their class which itself references its module, and the module contains references to everything that is inside (and maybe other modules) and this can lead back to the original instance.
- When representing data structures like graphs, it is very typical for them to have internal links to themselves.

To correctly dispose of these objects once they become unreachable, they need to be identified first. Inside the function that identifies cycles, two doubly linked lists are maintained: one list contains all objects to be scanned, and the other will contain all objects "tentatively" unreachable.

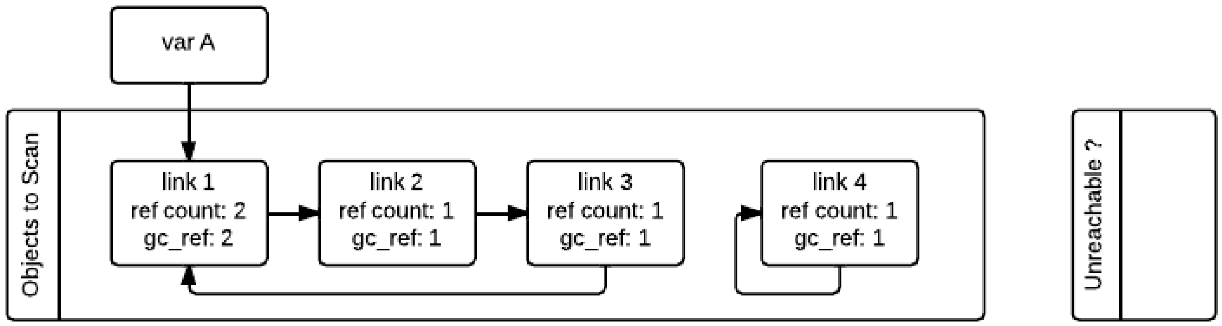
To understand how the algorithm works, let's take the case of a circular linked list which has one link referenced by a variable `A`, and one self-referencing object which is completely unreachable:

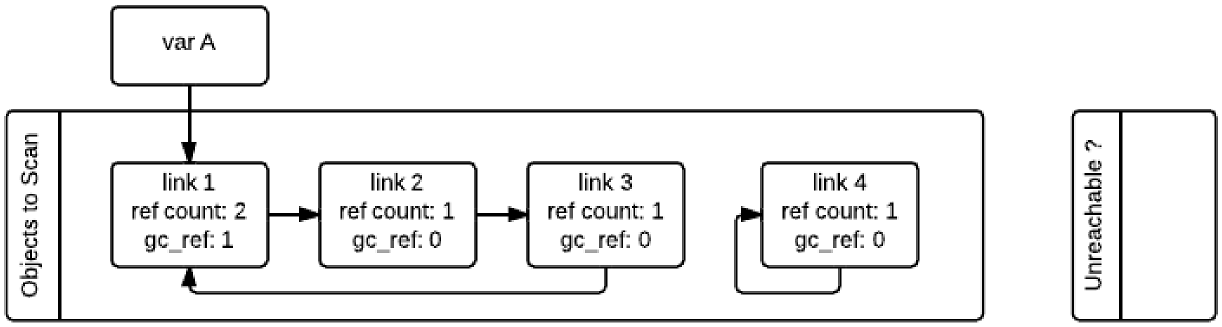
```
>>> import gc

>>> class Link:
...    def __init__(self, next_link=None):
...        self.next_link = next_link

>>> link_3 = Link()
>>> link_2 = Link(link_3)
>>> link_1 = Link(link_2)
>>> link_3.next_link = link_1
>>> A = link_1
>>> del link_1, link_2, link_3

>>> link_4 = Link()
>>> link_4.next_link = link_4
>>> del link_4

# Collect the unreachable Link object (and its .__dict__ dict).
>>> gc.collect()
2
```

When the GC starts, it has all the container objects it wants to scan on the first linked list. The objective is to move all the unreachable objects. Since most objects turn out to be reachable, it is much more efficient to move the unreachable as this involves fewer pointer updates.

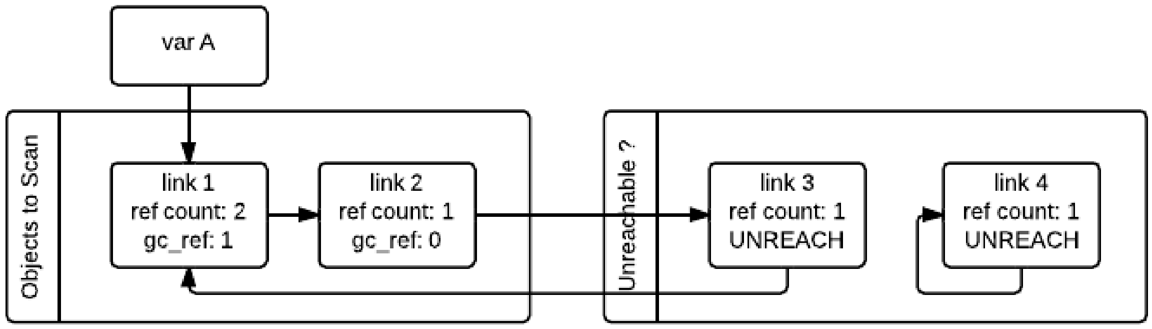
Every object that supports garbage collection will have an extra reference count field initialized to the reference count (`gc_ref` in the figures) of that object when the algorithm starts. This is because the algorithm needs to modify the reference count to do the computations and in this way the interpreter will not modify the real reference count field.

The GC then iterates over all containers in the first list and decrements by one the `gc_ref` field of any other object that container is referencing. Doing this makes use of the `tp_traverse` slot in the container class (implemented using the C API or inherited by a superclass) to know what objects are referenced by each container. After all the objects have been scanned, only the objects that have references from outside the "objects to scan" list will have `gc_refs > 0`.



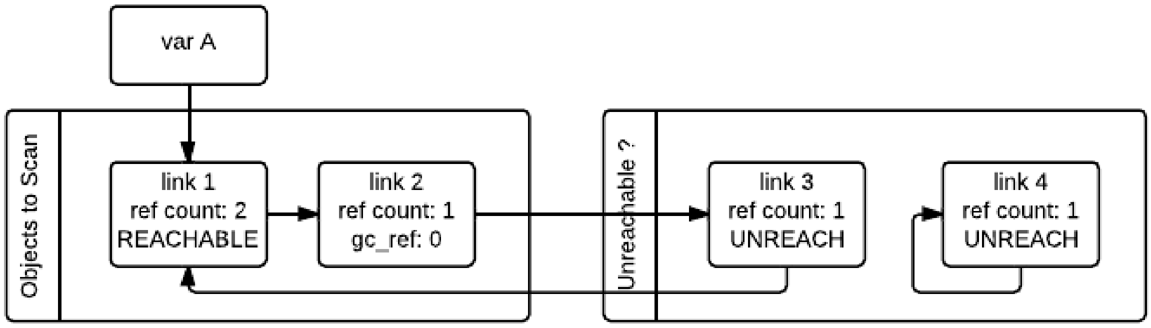


Notice that having `gc_refs == 0` does not imply that the object is unreachable. This is because another object that is reachable from the outside (`gc_refs > 0`) can still have references to it. For instance, the `link_2` object in our example ended having `gc_refs == 0` but is referenced still by the `link_1` object that is reachable from the outside. To obtain the set of objects that are really unreachable, the garbage collector re-scans the container objects using the `tp_traverse` slot; this time with a different traverse function that marks objects with `gc_refs == 0` as "tentatively unreachable" and then moves them to the tentatively unreachable list. The following image depicts the state of the lists in a moment when the GC processed the `link_3` and `link_4` objects but has not processed `link_1` and `link_2` yet.

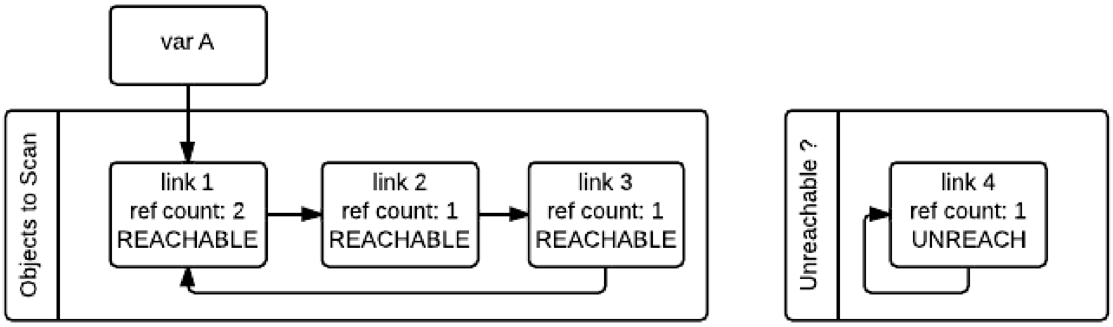


Then the GC scans the next `link_1` object. Because it has `gc_refs == 1`, the gc does not do anything special because it knows it has to be reachable (and is already in what will become the reachable list):

When the GC encounters an object which is reachable (`gc_refs > 0`), it traverses its references using the `tp_traverse` slot to find all the objects that are reachable from it, moving them to the end of the list of reachable objects (where they started originally) and setting its `gc_refs` field to 1. This is what happens to `link_2` and `link_3` below as they are reachable from `link_1`. From the state in the previous image and after examining the objects referred to by `link_1` the GC knows that `link_3` is reachable after all, so it is moved back to the original list and its `gc_refs` field is set to 1 so that if the GC visits it again, it will know that it's reachable. To avoid visiting an object twice, the



GC marks all objects that have already been visited once (by unsetting the `PREV_MASK_COLLECTING` flag) so that if an object that has already been processed is referenced by some other object, the GC does not process it twice.



Notice that an object that was marked as "tentatively unreachable" and was later moved back to the reachable list will be visited again by the garbage collector as now all the references that that object has need to be processed as well. This process is really a breadth first search over the object graph. Once all the objects are scanned, the GC knows that all container objects in the tentatively unreachable list are really unreachable and can thus be garbage collected.

Pragmatically, it's important to note that no recursion is required by any of this, and neither does it in any other way require additional memory proportional to the number of objects, number of pointers, or the lengths of pointer chains. Apart from `O(1)` storage for internal C needs, the objects themselves contain all the storage the GC algorithms require.

Why moving unreachable objects is better

It sounds logical to move the unreachable objects under the premise that most objects are usually reachable, until you think about it: the reason it pays isn't actually obvious.

Suppose we create objects A, B, C in that order. They appear in the young generation in the same order. If B points to A, and C to B, and C is reachable from outside, then the adjusted refcounts after the first step of the algorithm runs will be 0, 0, and 1 respectively because the only reachable object from the outside is C.

When the next step of the algorithm finds A, A is moved to the unreachable list. The same for B when it's first encountered. Then C is traversed, B is moved *back* to the reachable list. B is eventually traversed, and then A is moved back to the reachable list.

So instead of not moving at all, the reachable objects B and A are each moved twice. Why is this a win? A straightforward algorithm to move the reachable objects instead would move A, B, and C once each. The key is that this dance leaves the objects in order C, B, A - it's reversed from the original order. On all *subsequent* scans, none of them will move. Since most objects aren't in cycles, this can save an unbounded number of moves across an unbounded number of later collections. The only time the cost can be higher is the first time the chain is scanned.

10.27.4 Destroying unreachable objects

Once the GC knows the list of unreachable objects, a very delicate process starts with the objective of completely destroying these objects. Roughly, the process follows these steps in order:

1. Handle and clean weak references (if any). If an object that is in the unreachable set is going to be destroyed and has weak references with callbacks, these callbacks need to be honored. This process is **very** delicate as any error can cause objects that will be in an inconsistent state to be resurrected or reached by some Python functions invoked from the callbacks. In addition, weak references that also are part of the unreachable set (the object and its weak reference are in cycles that are unreachable) need to be cleaned immediately, without executing the callback. Otherwise it will be triggered later, when the `tp_clear` slot is called, causing havoc. Ignoring the weak reference's callback is fine because both the object and the weakref are going away, so it's legitimate to say the weak reference is going away first.
2. If an object has legacy finalizers (`tp_del` slot) move them to the `gc.garbage` list.
3. Call the finalizers (`tp_finalize` slot) and mark the objects as already finalized to avoid calling them twice if they resurrect or if other finalizers have removed the object first.
4. Deal with resurrected objects. If some objects have been resurrected, the GC finds the new subset of objects that are still unreachable by running the cycle detection algorithm again and continues with them.
5. Call the `tp_clear` slot of every object so all internal links are broken and the reference counts fall to 0, triggering the destruction of all unreachable objects.

10.27.5 Optimization: generations

In order to limit the time each garbage collection takes, the GC uses a popular optimization: generations. The main idea behind this concept is the assumption that most objects have a very short lifespan and can thus be collected shortly after their creation. This has proven to be very close to the reality of many Python programs as many temporary objects are created and destroyed very fast. The older an object is the less likely it is that it will become unreachable.

To take advantage of this fact, all container objects are segregated into three spaces/generations. Every new object starts in the first generation (generation 0). The previous algorithm is executed only over the objects of a particular generation and if an object survives a collection of its generation it will be moved to the next one (generation 1), where it will be surveyed for collection less often. If the same object survives another GC round in this new generation (generation 1) it will be moved to the last generation (generation 2) where it will be surveyed the least often.

Generations are collected when the number of objects that they contain reaches some predefined threshold, which is unique for each generation and is lower the older the generations are. These thresholds can be examined using the `gc.get_threshold` function:

```
>>> import gc
>>> gc.get_threshold()
(700, 10, 10)
```

The content of these generations can be examined using the `gc.get_objects(generation=NUM)` function and collections can be triggered specifically in a generation by calling `gc.collect(generation=NUM)`.

```
>>> import gc
>>> class MyObj:
...     pass
...

# Move everything to the last generation so it's easier to inspect
# the younger generations.
```

(continues on next page)

(continued from previous page)

```
>>> gc.collect()
0

# Create a reference cycle.

>>> x = MyObj()
>>> x.self = x

# Initially the object is in the youngest generation.

>>> gc.get_objects(generation=0)
[..., <__main__.MyObj object at 0x7fbcc12a3400>, ...]

# After a collection of the youngest generation the object
# moves to the next generation.

>>> gc.collect(generation=0)
0
>>> gc.get_objects(generation=0)
[]
>>> gc.get_objects(generation=1)
[..., <__main__.MyObj object at 0x7fbcc12a3400>, ...]
```

Collecting the oldest generation

In addition to the various configurable thresholds, the GC only triggers a full collection of the oldest generation if the ratio `long_lived_pending / long_lived_total` is above a given value (hardwired to 25%). The reason is that, while "non-full" collections (i.e., collections of the young and middle generations) will always examine roughly the same number of objects (determined by the aforementioned thresholds) the cost of a full collection is proportional to the total number of long-lived objects, which is virtually unbounded. Indeed, it has been remarked that doing a full collection every <constant number> of object creations entails a dramatic performance degradation in workloads which consist of creating and storing lots of long-lived objects (e.g. building a large list of GC-tracked objects would show quadratic performance, instead of linear as expected). Using the above ratio, instead, yields amortized linear performance in the total number of objects (the effect of which can be summarized thusly: "each full garbage collection is more and more costly as the number of objects grows, but we do fewer and fewer of them").

10.27.6 Optimization: reusing fields to save memory

In order to save memory, the two linked list pointers in every object with GC support are reused for several purposes. This is a common optimization known as "fat pointers" or "tagged pointers": pointers that carry additional data, "folded" into the pointer, meaning stored inline in the data representing the address, taking advantage of certain properties of memory addressing. This is possible as most architectures align certain types of data to the size of the data, often a word or multiple thereof. This discrepancy leaves a few of the least significant bits of the pointer unused, which can be used for tags or to keep other information – most often as a bit field (each bit a separate tag) – as long as code that uses the pointer masks out these bits before accessing memory. E.g., on a 32-bit architecture (for both addresses and word size), a word is 32 bits = 4 bytes, so word-aligned addresses are always a multiple of 4, hence end in `00`, leaving the last 2 bits available; while on a 64-bit architecture, a word is 64 bits = 8 bytes, so word-aligned addresses end in `000`, leaving the last 3 bits available.

The CPython GC makes use of two fat pointers that correspond to the extra fields of PyGC_Head discussed in the *Memory layout and object structure* section:

> **Warning:** Because the presence of extra information, "tagged" or "fat" pointers cannot be dereferenced directly and the extra information must be stripped off before obtaining the real memory address. Special care needs to be taken with functions that directly manipulate the linked lists, as these functions normally assume the pointers inside the lists are in a consistent state.

- The _gc_prev field is normally used as the "previous" pointer to maintain the doubly linked list but its lowest two bits are used to keep the flags PREV_MASK_COLLECTING and _PyGC_PREV_MASK_FINALIZED. Between collections, the only flag that can be present is _PyGC_PREV_MASK_FINALIZED that indicates if an object has been already finalized. During collections _gc_prev is temporarily used for storing a copy of the reference count (gc_refs), in addition to two flags, and the GC linked list becomes a singly linked list until _gc_prev is restored.
- The _gc_next field is used as the "next" pointer to maintain the doubly linked list but during collection its lowest bit is used to keep the NEXT_MASK_UNREACHABLE flag that indicates if an object is tentatively unreachable during the cycle detection algorithm. This is a drawback to using only doubly linked lists to implement partitions: while most needed operations are constant-time, there is no efficient way to determine which partition an object is currently in. Instead, when that's needed, ad hoc tricks (like the NEXT_MASK_UNREACHABLE flag) are employed.

10.27.7 Optimization: delay tracking containers

Certain types of containers cannot participate in a reference cycle, and so do not need to be tracked by the garbage collector. Untracking these objects reduces the cost of garbage collection. However, determining which objects may be untracked is not free, and the costs must be weighed against the benefits for garbage collection. There are two possible strategies for when to untrack a container:

1. When the container is created.
2. When the container is examined by the garbage collector.

As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects…) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances. Some examples of native types that benefit from delayed tracking:

- Tuples containing only immutable objects (integers, strings etc, and recursively, tuples of immutable objects) do not need to be tracked. The interpreter creates a large number of tuples, many of which will not survive until garbage collection. It is therefore not worthwhile to untrack eligible tuples at creation time. Instead, all tuples except the empty tuple are tracked when created. During garbage collection it is determined whether any surviving tuples can be untracked. A tuple can be untracked if all of its contents are already not tracked. Tuples are examined for untracking in all garbage collection cycles. It may take more than one cycle to untrack a tuple.
- Dictionaries containing only immutable objects also do not need to be tracked. Dictionaries are untracked when created. If a tracked item is inserted into a dictionary (either as a key or value), the dictionary becomes tracked. During a full garbage collection (all generations), the collector will untrack any dictionaries whose contents are not tracked.

The garbage collector module provides the Python function is_tracked(obj), which returns the current tracking status of the object. Subsequent garbage collections may change the tracking status of the object.

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
```

(continues on next page)

(continued from previous page)

```
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

10.28 Updating standard library extension modules

In this section, we could explain how to write a CPython extension with the C language, but the topic can take a complete book.

For this reason, we prefer to give you some links where you can read a very good documentation.

Read the following references:

- https://docs.python.org/dev/c-api/
- https://docs.python.org/dev/extending/
- **PEP 399**
- https://pythonextensionpatterns.readthedocs.io/en/latest/

10.29 Changing Python's C API

The C API is divided into three sections:

1. The internal, private API, available with `Py_BUILD_CORE` defined. Ideally declared in `Include/internal/`. Any API named with a leading underscore is also considered private.
2. The public C API, available when `Python.h` is included normally. Ideally declared in `Include/cpython/`.
3. The Limited API, available with `Py_LIMITED_API` defined. Ideally declared directly under `Include/`.

Each section has higher stability & maintenance requirements, and you will need to think about more issues when you add or change definitions in it.

The compatibility guarantees for public C API are explained in the user documentation, `Doc/c-api/stable.rst` (C API Stability).

10.29.1 The internal API

Internal API is defined in `Include/internal/` and is only available for building CPython itself, as indicated by a macro like `Py_BUILD_CORE`.

While internal API can be changed at any time, it's still good to keep it stable: other API or other CPython developers may depend on it.

With PyAPI_FUNC or PyAPI_DATA

Functions or structures in `Include/internal/` defined with `PyAPI_FUNC` or `PyAPI_DATA` are internal functions which are exposed only for specific use cases like debuggers and profilers.

With the extern keyword

Functions in `Include/internal/` defined with the `extern` keyword *must not and can not* be used outside the CPython code base. Only built-in stdlib extensions (built with the `Py_BUILD_CORE_BUILTIN` macro defined) can use such functions.

When in doubt, new internal C functions should be defined in `Include/internal` using the `extern` keyword.

Private names

Any API named with a leading underscore is also considered internal. There are two main use cases for using such names rather than putting the definition in `Include/internal/` (or directly in a `.c` file):

- Internal helpers for other public API; users should not use these directly;
- "Provisional" API, included in a Python release to test real-world usage of new API. Such names should be renamed when stabilized; preferably with a macro aliasing the old name to the new one. See **"Finalizing the API" in PEP 590** for an example.

10.29.2 Public C API

CPython's public C API is available when `Python.h` is included normally (that is, without defining macros to select the other variants).

It should be defined in `Include/cpython/` (unless part of the Limited API, see below).

Guidelines for expanding/changing the public API:

- Make sure the new API follows reference counting conventions. (Following them makes the API easier to reason about, and easier use in other Python implementations.)
  - Functions *must not* steal references
  - Functions *must not* return borrowed references
  - Functions returning references *must* return a strong reference
- Make sure the ownership rules and lifetimes of all applicable struct fields, arguments and return values are well defined.

10.29.3 Limited API

The Limited API is a subset of the C API designed to guarantee ABI stability across Python 3 versions. Defining the macro `Py_LIMITED_API` will limit the exposed API to this subset.

No changes that break the Stable ABI are allowed.

The Limited API should be defined in `Include/`, excluding the `cpython` and `internal` subdirectories.

Guidelines for changing the Limited API, and removing items from it

While the *Stable ABI* must not be broken, the existing Limited API can be changed, and items can be removed from it, if:

- the Backwards Compatibility Policy (**PEP 387**) is followed, and
- the Stable ABI is not broken – that is, extensions compiled with Limited API of older versions of Python continue to work on newer versions of Python.

This is tricky to do and requires careful thought. Some examples:

- Functions, structs etc. accessed by macros in *any version* of the Limited API are part of the Stable ABI, even if they are named with an underscore. They must not be removed and their signature must not change. (Their implementation may change, though.)
- Structs members cannot be rearranged if they were part of any version of the Limited API.
- If the Limited API allows users to allocate a struct directly, its size must not change.
- Exported symbols (functions and data) must continue to be available as exported symbols. Specifically, a function can only be converted to a `static inline` function (or macro) if Python also continues to provide the actual function. For an example, see the `Py_NewRef` macro and redefinition in 3.10.

It is possible to remove items marked as part of the Stable ABI, but only if there was no way to use them in any past version of the Limited API.

Guidelines for adding to the Limited API

- Guidelines for the general *Public C API* apply.
- New Limited API should only be defined if `Py_LIMITED_API` is set to the version the API was added in or higher. (See below for the proper `#if` guard.)
- All parameter types, return values, struct members, etc. need to be part of the Limited API.
  - Functions that deal with `FILE*` (or other types with ABI portability issues) should not be added.
- Think twice when defining macros.
  - Macros should not expose implementation details
  - Functions must be exported as actual functions, not (only) as functions-like macros.
  - If possible, avoid macros. This makes the Limited API more usable in languages that don't use the C preprocessor.
- Please start a public discussion before expanding the Limited API
- The Limited API and must follow standard C, not just features of currently supported platforms. The exact C dialect is described in **PEP 7**.
  - Documentation examples (and more generally: the intended use of the API) should also follow standard C.
  - In particular, do not cast a function pointer to `void*` (a data pointer) or vice versa.
- Think about ease of use for the user.
  - In C, ease of use itself is not very important; what is useful is reducing boilerplate code needed to use the API. Bugs like to hide in boiler plates.
  - If a function will be often called with specific value for an argument, consider making it default (used when `NULL` is passed in).
  - The Limited API needs to be well documented.

- Think about future extensions
  - If it's possible that future Python versions will need to add a new field to your struct, make sure it can be done.
  - Make as few assumptions as possible about implementation details that might change in future CPython versions or differ across C API implementations. The most important CPython-specific implementation details involve:
    - The GIL
    - *Garbage collection*
    - Memory layout of PyObject, lists/tuples and other structures

If following these guidelines would hurt performance, add a fast function (or macro) to the non-limited API and a stable equivalent to the Limited API.

If anything is unclear, or you have a good reason to break the guidelines, consider discussing the change at the capi-sig mailing list.

Adding a new definition to the Limited API

- Add the declaration to a header file directly under `Include/`, into a block guarded with the following:

  ```
  #if !defined(Py_LIMITED_API) || Py_LIMITED_API+0 >= 0x03yy0000
  ```

  with the `yy` corresponding to the target CPython version, e.g. `0x030A0000` for Python 3.10.
- Append an entry to the Stable ABI manifest, `Misc/stable_abi.toml`
- Regenerate the autogenerated files using `make regen-limited-abi`. On platforms without `make`, run this command directly:

  ```
  ./python ./Tools/scripts/stable_abi.py --generate-all ./Misc/stable_abi.toml
  ```

- Build Python and check the using `make check-limited-abi`. On platforms without `make`, run this command directly:

  ```
  ./python ./Tools/scripts/stable_abi.py --all ./Misc/stable_abi.toml
  ```

10.30 Coverity Scan

Coverity Scan is a free service for static code analysis of Open Source projects. It is based on Coverity's commercial product and is able to analyze C, C++ and Java code.

Coverity's static code analysis doesn't run the code. Instead of that it uses abstract interpretation to gain information about the code's control flow and data flow. It's able to follow all possible code paths that a program may take. For example the analyzer understands that `malloc()` returns a memory that must be freed with `free()` later. It follows all branches and function calls to see if all possible combinations free the memory. The analyzer is able to detect all sorts of issues like resource leaks (memory, file descriptors), NULL dereferencing, use after free, unchecked return values, dead code, buffer overflows, integer overflows, uninitialized variables, and many more.

10.30.1 Access to analysis reports

The results are available on the Coverity Scan website. In order to access the results you have to create an account yourself. Then go to *Projects using Scan* and add yourself to the Python project. New members must be approved by an admin (see *Contact*).

Access is restricted to Python core developers only. Other individuals may be given access at our own discretion, too. Every now and then Coverity detects a critical issue in Python's code – new analyzers may even find new bugs in mature code. We don't want to disclose issues prematurely.

10.30.2 Building and uploading analysis

The process is automated. A script checks out the code, runs `cov-build` and uploads the latest analysis to Coverity. Since Coverity has limited the maximum number of builds per week Python is analyzed every second day. The build runs on a dedicated virtual machine on PSF's infrastructure at OSU Open Source Labs. The process is maintained by Christian Heimes (see *Contact*). At present only the tip is analyzed with the 64bit Linux tools.

10.30.3 Known limitations

Some aspects of Python's C code are not yet understood by Coverity.

False positives

`Py_BuildValue("N", PyObject*)`
: Coverity doesn't understand that `N` format char passes the object along without touching its reference count. On this ground the analyzer detects a resource leak. CID 719685

`PyLong_FromLong()` for negative values
: Coverity claims that `PyLong_FromLong()` and other `PyLong_From*()` functions cannot handle a negative value because the value might be used as an array index in `get_small_int()`. CID 486783

`PyLong_FromLong()` for n in [-5 ... +255]
: For integers in the range of Python's small int cache the `PyLong_From*()` function can never fail and never returns NULL. CID 1058291

`PyArg_ParseTupleAndKeywords(args, kwargs, "s#", &data, &length)`
: Some functions use the format char combination such as `s#`, `u#` or `z#` to get data and length of a character array. Coverity doesn't recognize the relation between data and length. Sometimes it detects a buffer overflow if data is written to a fixed size buffer although `length <= sizeof(buffer)`. CID 486613

`path_converter()` dereferencing after null check
: The `path_converter()` function in `posixmodule.c` makes sure that either `path_t.narrow` or `path_t.wide` is filled unless `path_t.nullable` is explicitly enabled. CID 719648

10.30.4 Modeling

Modeling is explained in the *Coverity Help Center* which is available in the help menu of Coverity Connect. coverity_model.c contains a copy of Python's modeling file for Coverity. Please keep the copy in sync with the model file in *Analysis Settings* of Coverity Scan.

10.30.5 Workflow

False positive and intentional issues

If the problem is listed under *Known limitations* then please set the classification to either "False positive" or "Intentional", the action to "Ignore", owner to your own account and add a comment why the issue is considered false positive or intentional.

If you think it's a new false positive or intentional then please contact an admin. The first step should be an updated to Python's *Modeling* file.

Positive issues

You should always create an issue unless it's really a trivial case. Please add the full url to the ticket under *Ext. Reference* and add the CID (Coverity ID) to both the ticket and the checkin message. It makes it much easier to understand the relation between tickets, fixes and Coverity issues.

10.30.6 Contact

Please include both Brett and Christian in any mail regarding Coverity. Mails to Coverity should go through Brett or Christian, too.

Christian Heimes <christian (at) python (dot) org>
admin, maintainer of build machine, intermediary between Python and Coverity

Brett Cannon <brett (at) python (dot) org>
co-admin

Dakshesh Vyas <scan-admin@coverity.com>
Technical Manager - Coverity Scan

See also:

Coverity Scan FAQ

10.31 Dynamic Analysis with Clang

This document describes how to use Clang to perform analysis on Python and its libraries. In addition to performing the analysis, the document will cover downloading, building and installing the latest Clang/LLVM combination (which is currently 3.4).

This document does not cover interpreting the findings. For a discussion of interpreting results, see Marshall Clow's Testing libc++ with -fsanitize=undefined. The blog posting is a detailed examinations of issues uncovered by Clang in `libc++`.

10.31.1 What is Clang?

Clang is the C, C++ and Objective C front-end for the LLVM compiler. The front-end provides access to LLVM's optimizer and code generator. The sanitizers - or checkers - are hooks into the code generation phase to instrument compiled code so suspicious behavior is flagged.

10.31.2 What are Sanitizers?

Clang sanitizers are runtime checkers used to identify suspicious and undefined behavior. The checking occurs at runtime with actual runtime parameters so false positives are kept to a minimum.

There are a number of sanitizers available, but two that should be used on a regular basis are the Address Sanitizer (or ASan) and the Undefined Behavior Sanitizer (or UBSan). ASan is invoked with the compiler option `-fsanitize=address`, and UBSan is invoked with `-fsanitize=undefined`. The flags are passed through `CFLAGS` and `CXXFLAGS`, and sometimes through `CC` and `CXX` (in addition to the compiler).

A complete list of sanitizers can be found at Controlling Code Generation.

Note: Because sanitizers operate at runtime on real program parameters, its important to provide a complete set of positive and negative self tests.

Clang and its sanitizers have strengths (and weaknesses). Its just one tool in the war chest to uncovering bugs and improving code quality. Clang should be used to compliment other methods, including Code Reviews, Valgrind, Coverity, etc.

10.31.3 Clang/LLVM Setup

This portion of the document covers downloading, building and installing Clang and LLVM. There are three components to download and build. They are the LLVM compiler, the compiler front end and the compiler runtime library.

In preparation you should create a scratch directory. Also ensure you are using Python 2 and not Python 3. Python 3 will cause the build to fail.

Download, Build and Install

Perform the following to download, build and install the Clang/LLVM 3.4.

```
# Download
wget https://llvm.org/releases/3.4/llvm-3.4.src.tar.gz
wget https://llvm.org/releases/3.4/clang-3.4.src.tar.gz
wget https://llvm.org/releases/3.4/compiler-rt-3.4.src.tar.gz

# LLVM
tar xvf llvm-3.4.src.tar.gz
cd llvm-3.4/tools

# Clang Front End
tar xvf ../../clang-3.4.src.tar.gz
mv clang-3.4 clang

# Compiler RT
```

(continues on next page)

(continued from previous page)

```
cd ../projects
tar xvf ../../compiler-rt-3.4.src.tar.gz
mv compiler-rt-3.4/ compiler-rt

# Build
cd ..
./configure --enable-optimized --prefix=/usr/local
make -j4
sudo make install
```

Note: If you receive an error `'LibraryDependencies.inc' file not found`, then ensure you are utilizing Python 2 and not Python 3. If you encounter the error after switching to Python 2, then delete everything and start over.

After `make install` executes, the compilers will be installed in `/usr/local/bin` and the various libraries will be installed in `/usr/local/lib/clang/3.4/lib/linux/`:

```
$ ls /usr/local/lib/clang/3.4/lib/linux/
libclang_rt.asan-x86_64.a   libclang_rt.profile-x86_64.a
libclang_rt.dfsan-x86_64.a  libclang_rt.san-x86_64.a
libclang_rt.full-x86_64.a   libclang_rt.tsan-x86_64.a
libclang_rt.lsan-x86_64.a   libclang_rt.ubsan_cxx-x86_64.a
libclang_rt.msan-x86_64.a   libclang_rt.ubsan-x86_64.a
```

On Mac OS X, the libraries are installed in `/usr/local/lib/clang/3.3/lib/darwin/`:

```
$ ls /usr/local/lib/clang/3.3/lib/darwin/
libclang_rt.10.4.a                    libclang_rt.ios.a
libclang_rt.asan_osx.a                libclang_rt.osx.a
libclang_rt.asan_osx_dynamic.dylib    libclang_rt.profile_ios.a
libclang_rt.cc_kext.a                 libclang_rt.profile_osx.a
libclang_rt.cc_kext_ios5.a            libclang_rt.ubsan_osx.a
libclang_rt.eprintf.a
```

Note: You should never have to add the libraries to a project. Clang will handle it for you. If you find you cannot pass the `-fsanitize=XXX` flag through `make`'s implicit variables (`CFLAGS`, `CXXFLAGS`, `CC`, `CXXFLAGS`, `LDFLAGS`) during `configure`, then you should modify the makefile after configuring to ensure the flag is passed through the compiler.

The installer does not install all the components needed on occasion. For example, you might want to run a `scan-build` or examine the results with `scan-view`. You can copy the components by hand with:

```
sudo mkdir /usr/local/bin/scan-build
sudo cp -r llvm-3.4/tools/clang/tools/scan-build /usr/local/bin
sudo mkdir /usr/local/bin/scan-view
sudo cp -r llvm-3.4/tools/clang/tools/scan-view /usr/local/bin
```

Note: Because the installer does not install all the components needed on occasion, you should not delete the scratch directory until you are sure things work as expected. If a library is missing, then you should search for it in the

Clang/LLVM build directory.

10.31.4 Python Build Setup

This portion of the document covers invoking Clang and LLVM with the options required so the sanitizers analyze Python with under its test suite. Two checkers are used - ASan and UBSan.

Because the sanitizers are runtime checkers, its best to have as many positive and negative self tests as possible. You can never have enough self tests.

The general idea is to compile and link with the sanitizer flags. At link time, Clang will include the needed runtime libraries. However, you can't use `CFLAGS` and `CXXFLAGS` to pass the options through the compiler to the linker because the makefile rules for `BUILDPYTHON`, `_testembed` and `_freeze_importlib` don't use the implicit variables.

As a workaround to the absence of flags to the linker, you can pass the sanitizer options by way of the compilers - `CC` and `CXX`. Passing the flags though the compiler is used below, but passing them through `LDFLAGS` is also reported to work.

Building Python

To begin, export the variables of interest with the desired sanitizers. Its OK to specify both sanitizers:

```
# ASan
export CC="/usr/local/bin/clang -fsanitize=address"
export CXX="/usr/local/bin/clang++ -fsanitize=address -fno-sanitize=vptr"
```

Or:

```
# UBSan
export CC="/usr/local/bin/clang -fsanitize=undefined"
export CXX="/usr/local/bin/clang++ -fsanitize=undefined -fno-sanitize=vptr"
```

The `-fno-sanitize=vptr` removes vtable checks that are part of UBSan from C++ projects due to noise. Its not needed with Python, but you will likely need it for other C++ projects.

After exporting `CC` and `CXX`, `configure` as normal:

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for --enable-universalsdk... no
checking for --with-universal-archs... 32-bit
checking MACHDEP... linux
checking for --without-gcc... no
checking for gcc... /usr/local/bin/clang -fsanitize=undefined
checking whether the C compiler works... yes
...
```

Next is a standard `make` (formatting added for clarity):

```
$ make
/usr/local/bin/clang -fsanitize=undefined -c -Wno-unused-result
    -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -I.
```

(continues on next page)

(continued from previous page)

```
    -IInclude -I./Include -DPy_BUILD_CORE -o Modules/python.o
    ./Modules/python.c
/usr/local/bin/clang -fsanitize=undefined -c -Wno-unused-result
    -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -I.
    -IInclude -I./Include -DPy_BUILD_CORE -o Parser/acceler.o
    Parser/acceler.c
...
```

Finally is `make test` (formatting added for clarity):

```
Objects/longobject.c:39:42: runtime error: index -1 out of bounds
    for type 'PyLongObject [262]'
Objects/tupleobject.c:188:13: runtime error: member access within
    misaligned address 0x2b76be018078 for type 'PyGC_Head' (aka
    'union _gc_head'), which requires 16 byte alignment
    0x2b76be018078: note: pointer points here
    00 00 00 00  40 53 5a b6 76 2b 00 00  60 52 5a b6 ...
                 ^
...
```

If you are using the address sanitizer, its important to pipe the output through `asan_symbolize.py` to get a good trace. For example, from Issue 20953 during compile (formatting added for clarity):

```
$ make test 2>&1 | asan_symbolize.py
...

/usr/local/bin/clang -fsanitize=address -Xlinker -export-dynamic
    -o python Modules/python.o libpython3.3m.a -ldl -lutil
    /usr/local/ssl/lib/libssl.a /usr/local/ssl/lib/libcrypto.a -lm
./python -E -S -m sysconfig --generate-posix-vars
=================================================================
==24064==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x619000004020 at pc 0x4ed4b2 bp 0x7fff80fff010 sp 0x7fff80fff008
READ of size 4 at 0x619000004020 thread T0
  #0 0x4ed4b1 in PyObject_Free Python-3.3.5/./Objects/obmalloc.c:987
  #1 0x7a2141 in code_dealloc Python-3.3.5/./Objects/codeobject.c:359
  #2 0x620c00 in PyImport_ImportFrozenModuleObject
      Python-3.3.5/./Python/import.c:1098
  #3 0x620d5c in PyImport_ImportFrozenModule
      Python-3.3.5/./Python/import.c:1114
  #4 0x63fd07 in import_init Python-3.3.5/./Python/pythonrun.c:206
  #5 0x63f636 in _Py_InitializeEx_Private
      Python-3.3.5/./Python/pythonrun.c:369
  #6 0x681d77 in Py_Main Python-3.3.5/./Modules/main.c:648
  #7 0x4e6894 in main Python-3.3.5/././Modules/python.c:62
  #8 0x2abf9a525eac in __libc_start_main
      /home/aurel32/eglibc/eglibc-2.13/csu/libc-start.c:244
  #9 0x4e664c in _start (Python-3.3.5/./python+0x4e664c)

AddressSanitizer can not describe address in more detail (wild
memory access suspected).
SUMMARY: AddressSanitizer: heap-buffer-overflow
```

(continues on next page)

(continued from previous page)

```
  Python-3.3.5/./Objects/obmalloc.c:987 PyObject_Free
Shadow bytes around the buggy address:
  0x0c327fff87b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff87c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff87d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff87e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff87f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c327fff8800: fa fa fa fa[fa]fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff8810: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff8820: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff8830: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff8840: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c327fff8850: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:     fa
  Heap right redzone:    fb
  Freed heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
  ASan internal:         fe
==24064==ABORTING
make: *** [pybuilddir.txt] Error 1
```

Note: `asan_symbolize.py` is supposed to be installed during `make install`. If its not installed, then look in the Clang/LLVM build directory for it and copy it to `/usr/local/bin`.

Blacklisting (Ignoring) Findings

Clang allows you to alter the behavior of sanitizer tools for certain source-level by providing a special blacklist file at compile-time. The blacklist is needed because it reports every instance of an issue, even if the issue is reported 10's of thousands of time in un-managed library code.

You specify the blacklist with `-fsanitize-blacklist=XXX`. For example:

```
-fsanitize-blacklist=my_blacklist.txt
```

`my_blacklist.txt` would then contain entries such as the following. The entry will ignore a bug in `libc++`'s `ios` formatting functions:

```
fun:_Ios_Fmtflags
```

As an example with Python 3.4.0, `audioop.c` will produce a number of findings:

```
./Modules/audioop.c:422:11: runtime error: left shift of negative value -1
./Modules/audioop.c:446:19: runtime error: left shift of negative value -1
./Modules/audioop.c:476:19: runtime error: left shift of negative value -1
./Modules/audioop.c:504:16: runtime error: left shift of negative value -1
./Modules/audioop.c:533:22: runtime error: left shift of negative value -128
./Modules/audioop.c:775:19: runtime error: left shift of negative value -70
./Modules/audioop.c:831:19: runtime error: left shift of negative value -70
./Modules/audioop.c:881:19: runtime error: left shift of negative value -1
./Modules/audioop.c:920:22: runtime error: left shift of negative value -70
./Modules/audioop.c:967:23: runtime error: left shift of negative value -70
./Modules/audioop.c:968:23: runtime error: left shift of negative value -70
...
```

One of the function of interest is `audioop_getsample_impl` (flagged at line 422), and the blacklist entry would include:

```
fun:audioop_getsample_imp
```

Or, you could ignore the entire file with:

```
src:Modules/audioop.c
```

Unfortunately, you won't know what to blacklist until you run the sanitizer.

The documentation is available at Sanitizer special case list.

10.32 Running a buildbot worker

Python's *Continuous Integration* system was discussed earlier. We sometimes refer to the collection of *build workers* as our "buildbot fleet". The machines that comprise the fleet are voluntarily contributed resources. Many are run by individual volunteers out of their own pockets and time, while others are supported by corporations. Even the corporate sponsored buildbots, however, tend to exist because some individual championed them, made them a reality, and is committed to maintaining them.

Anyone can contribute a buildbot to the fleet. This chapter describes how to go about setting up a buildbot worker, getting it added, and some hints about buildbot maintenance.

Anyone running a buildbot that is part of the fleet should subscribe to the python-buildbots mailing list. This mailing list is also the place to contact if you want to contribute a buildbot but have questions.

As for what kind of buildbot to run. . . take a look at our current fleet. Pretty much anything that isn't on that list would be interesting: different Linux/UNIX distributions, different versions of the various OSes, other OSes if you or someone are prepared to make the test suite actually pass on that new OS. Even if you only want to run an OS that's already on our list there may be utility in setting it up; we also need to build and test python under various alternate build configurations. Post to the mailing list and talk about what you'd like to contribute.

10.32.1 Preparing for buildbot worker setup

Since the goal is to build Python from source, the system will need to have everything required to do normal python development: a compiler, a linker, and (except on windows) the "development" headers for any of the optional modules (zlib, OpenSSL, etc) supported by the platform. Follow the steps outlined in *Getting Started* for the target platform, all the way through to having a working compiled python.

In order to set up the buildbot software, you will need to obtain an identifier and password for your worker so it can join the fleet. Email python-buildbots@python.org to discuss adding your worker and to obtain the needed workername and password. You can do some of the steps that follow before having the credentials, but it is easiest to have them before the "buildbot worker" step below.

10.32.2 Setting up the buildbot worker

Conventional always-on machines

You need a recent version of the buildbot software, and you will probably want a separate 'buildbot' user to run the buildbot software. You may also want to set the buildbot up using a virtual environment, depending on how you manage your system. We won't cover how to that here; it doesn't differ from setting up a virtual environment for any other software, but you'll need to modify the sequence of steps below as appropriate if you choose that path.

For Linux:

- If your package manager provides the buildbot worker software, that is probably the best way to install it; it may create the buildbot user for you, in which case you can skip that step. Otherwise, do `pip install buildbot-worker`.
- Create a `buildbot` user (using, eg: `useradd`) if necessary.
- Log in as the buildbot user.

For Mac:

- Create a buildbot user using the OS/X control panel user admin. It should be a "standard" user.
- Log in as the buildbot user.
- Install the buildbot worker[1] by running `pip install buildbot-worker`.

For Windows:

- Create a buildbot user as a "standard" user.
- Install the latest version of Python from python.org.
- Open a Command Prompt.
- Execute `python -m pip install pypiwin32 buildbot-worker` (note that `python.exe` is not added to `PATH` by default, making the `python` command accessible is left as an exercise for the user).

In a terminal window for the buildbot user, issue the following commands (you can put the `buildarea` wherever you want to):

```
mkdir buildarea
buildbot-worker create-worker buildarea buildbot-api.python.org:9020 workername␣
↪workerpasswd
```

[1] If the buildbot is going to do Framework builds, it is better to use the Apple-shipped Python so as to avoid any chance of the buildbot picking up components from the installed python.org python.

(Note that on Windows, the `buildbot-worker` command will be in the `Scripts` directory of your Python installation.)

Once this initial worker setup completes, you should edit the files `buildarea/info/admin` and `buildarea/info/host` to provide your contact info and information on the host configuration, respectively. This information will be presented in the buildbot web pages that display information about the builders running on your worker.

You will also want to make sure that the worker is started when the machine reboots:

For Linux:

- Add the following line to `/etc/crontab`:

  ```
  @reboot buildbot-worker restart /path/to/buildarea
  ```

  Note that we use `restart` rather than `start` in case a crash has left a `twistd.pid` file behind.

For OSX:

- Create a bin directory for your buildbot user:

  ```
  mkdir bin
  ```

- Place the following script, named `run_worker.sh`, into that directory:

  ```
  #!/bin/bash
  export PATH=/usr/local/bin:/Library/Frameworks/Python.framework/Versions/2.
  ↪7/bin:$PATH
  export LC_CTYPE=en_US.utf-8
  cd /Users/buildbot/buildarea
  twistd --nodaemon --python=buildbot.tac --logfile=buildbot.log --
  ↪prefix=worker
  ```

  If you use pip with Apple's system python, add '/System' to the front of the path to the Python bin directory.

- Place a file with the following contents into `/Library/LaunchDaemons`:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
        "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
  <plist version="1.0">
  <dict>
        <key>Label</key>
        <string>net.buildbot.worker</string>
        <key>UserName</key>
        <string>buildbot</string>
        <key>WorkingDirectory</key>
        <string>/Users/buildbot/buildarea</string>
        <key>ProgramArguments</key>
        <array>
                <string>/Users/buildbot/bin/run_worker.sh</string>
        </array>
        <key>StandardOutPath</key>
        <string>twistd.log</string>
        <key>StandardErrorPath</key>
        <string>twistd.log</string>
        <key>KeepAlive</key>
  ```

(continues on next page)

(continued from previous page)

```
        <true/>
        <key>SessionCreate</key>
        <true/>
</dict>
</plist>
```

The recommended name for the file is `net.buildbot.worker`.

For Windows:

- Add a Scheduled Task to run `buildbot-worker start buildarea` as the buildbot user "when the computer starts up". It is best to provide absolute paths to the `buildbot-worker` command and the `buildarea` directory. It is also recommended to set the task to run in the directory that contains the `buildarea` directory.
- Alternatively (note: don't do both!), set up the worker service as described in the buildbot documentation.

To start the worker running for your initial testing, you can do:

```
buildbot-worker start buildarea
```

Then you can either wait for someone to make a commit, or you can pick a builder associated with your worker from the list of builders and force a build.

In any case you should initially monitor builds on your builders to make sure the tests are passing and to resolve any platform issues that may be revealed by tests that fail. Unfortunately we do not currently have a way to notify you only of failures on your builders, so doing periodic spot checks is also a good idea.

Latent workers

We also support running latent workers on the AWS EC2 service. To set up such a worker:

- Start an instance of your chosen base AMI and set it up as a conventional worker.
- After the instance is fully set up as a conventional worker (including worker name and password, and admin and host information), create an AMI from the instance and stop the instance.
- Contact the buildmaster administrator who gave you your worker name and password and give them the following information:
  - Instance size (such as `m4.large`)
  - Full region specification (such as `us-west-2`)
  - AMI ID (such as `ami-1234beef`)
  - An Access Key ID and Access Key. It is recommended to set up a separate IAM user with full access to EC2 and provide the access key information for that user rather than for your main account.

The buildmaster cannot guarantee that it will always shut down your instance(s), so it is recommended to periodically check and make sure there are no "zombie" instances running on your account, created by the buildbot master. Also, if you notice that your worker seems to have been down for an unexpectedly long time, please ping the python-buildbots list to request that the master be restarted.

Latent workers should also be updated periodically to include operating system or other software updates, but when to do such maintenance is largely up to you as the worker owner. There are a couple different options for doing such updates:

- Start an instance from your existing AMI, do updates on that instance, and save a new AMI from the updated instance. Note that (especially for Windows workers) you should do at least one restart of the instance after doing updates to be sure that any post-reboot update work is done before creating the new AMI.

- Create an entirely new setup from a newer base AMI using your existing worker name and password.

Whichever way you choose to update your AMI, you'll need to provide the buildmaster administrators with the new AMI ID.

10.32.3 Buildbot worker operation

Most of the time, running a worker is a "set and forget" operation, depending on the level of involvement you want to have in resolving bugs revealed by your builders. There are, however, times when it is helpful or even necessary for you to get involved. As noted above, you should be subscribed to `python-buildbots@python.org` so that you will be made aware of any fleet-wide issues.

Necessary tasks include, obviously, keeping the buildbot running. Currently the system for notifying buildbot owners when their workers go offline is not working; this is something we hope to resolve. So currently it is helpful if you periodically check the status of your worker. We will also contact you via your contact address in `buildarea/info/admin` when we notice there is a problem that has not been resolved for some period of time and you have not responded to a posting on the python-buildbots list about it.

We currently do not have a minimum version requirement for the worker software. However, this is something we will probably establish as we tune the fleet, so another task will be to occasionally upgrade the buildbot worker software. Coordination for this will be done via `python-buildbots@python.org`.

The most interesting extra involvement is when your worker reveals a unique or almost-unique problem: a test that is failing on your system but not on other systems. In this case you should be prepared to offer debugging help to the people working on the bug: running tests by hand on the worker machine or, if possible, providing ssh access to a committer to run experiments to try to resolve the issue.

10.32.4 Required Ports

The worker operates as a *client* to the *buildmaster*. This means that all network connections are *outbound*. This is true also for the network tests in the test suite. Most consumer firewalls will allow any outbound traffic, so normally you do not need to worry about what ports the buildbot uses. However, corporate firewalls are sometimes more restrictive, so here is a table listing all of the outbound ports used by the buildbot and the python test suite (this list may not be complete as new tests may have been added since this table was last vetted):

Port	Host	Description
20, 21	ftp.debian.org	test_urllib2net
53	your DNS server	test_socket, and others implicitly
80	python.org example.com	(several tests)
119	news.gmane.org	test_nntplib
443	(various)	test_ssl
465	smtp.gmail.com	test_smtpnet
587	smtp.gmail.com	test_smtpnet
9020	python.org	connection to buildmaster

Many tests will also create local TCP sockets and connect to them, usually using either `localhost` or `127.0.0.1`.

10.32.5 Required Resources

Based on the last time we did a survey on buildbot requirements, the recommended resource allocations for a python buildbot are at least:

- 2 CPUs
- 512 MB RAM
- 30 GB free disk space

The bigmem tests won't run in this configuration, since they require substantially more memory, but these resources should be sufficient to ensure that Python compiles correctly on the platform and can run the rest of the test suite.

10.32.6 Security Considerations

We only allow builds to be triggered against commits to the CPython repository on GitHub. This means that the code your buildbot will run will have been vetted by a committer. However, mistakes and bugs happen, as could a compromise, so keep this in mind when siting your buildbot on your network and establishing the security around it. Treat the buildbot like you would any resource that is public facing and might get hacked (use a VM and/or jail/chroot/solaris zone, put it in a DMZ, etc). While the buildbot does not have any ports open for inbound traffic (and is not public facing in that sense), committer mistakes do happen, and security flaws are discovered in both released and unreleased code, so treating the buildbot as if it were fully public facing is a good policy.

Code runs differently as privileged and unprivileged users. We would love to have builders running as privileged accounts, but security considerations do make that difficult, as access to root can provide access to surprising resources (such as spoofed IP packets, changes in MAC addresses, etc) even on a VM setup. But if you are confident in your setup, we'd love to have a buildbot that runs python as root.

Note that the above is a summary of a discussion on python-dev about buildbot security that includes examples of the tests for which privilege matters. There was no final consensus, but the information is useful as a point of reference.

10.33 Core Developer Motivations and Affiliations

CPython core developers participate in the core development process for a variety of reasons. Being accepted as a core developer indicates that an individual is interested in acquiring those responsibilities, has the ability to collaborate effectively with existing core developers, and has had the time available to demonstrate both that interest and that ability.

This page allows core developers that choose to do so to provide more information to the rest of the Python community regarding their personal situation (such as their general location and professional affiliations), as well as any personal motivations that they consider particularly relevant.

Core developers that wish to provide this additional information add a new entry to the *Published entries* section below. Guidelines relating to content and layout are included as comments in the source code for this page.

Core developers that are available for training, consulting, contract, or full-time work, or are seeking crowdfunding support for their community contributions, may also choose to provide that information here (including linking out to commercial sites with the relevant details).

For more information on the origins and purpose of this page, see *Goals of this page*.

10.33.1 Published entries

The following core developers have chosen to provide additional details regarding their professional affiliations and (optionally) other reasons for participating in the CPython core development process:

Brett Cannon (Canada)

- Personal site: snarky.ca
- Microsoft (Software Developer)
- Python Software Foundation (Fellow)

Nick Coghlan (Australia)

- Personal site: Curious Efficiency
- Extended bio
- Tritium (Software Developer)
- Python Software Foundation (Fellow, Packaging Working Group)

Nick began using Python as a testing and prototyping language while working for Boeing Defence Australia, and continues to use it for that purpose today.

As a core developer, he is primarily interested in helping to ensure Python's continued suitability for educational, testing and data analysis use cases, as well as in encouraging good architectural practices when assembling Python applications and test harnesses from open source components.

Steve Dower (United States/Australia)

- Microsoft (Software Developer)
- Personal site: stevedower.id.au
- Speaking: stevedower.id.au/speaking
- Work blog: devblogs.microsoft.com/python/
- Email address: steve.dower@python.org

Steve started with Python while automating a test harness for medical devices, and now works for Microsoft on anything that makes Python more accessible to developers on any platform.

As a core developer, his focus is on maintaining the already excellent Windows support and improving Python's ability to be embedded in other applications.

Christian Heimes (Germany)

- Red Hat (Software Developer, Security Engineering / Identity Management)
- Python Software Foundation (Fellow)

Mariatta (Canada)

- Personal site: mariatta.ca
- Works as a Software Engineer in Vancouver, helps organize Vancouver PyLadies meetup on the side, and sometimes speaks at conferences.
- Email address: mariatta@python.org
- Sponsor Mariatta on GitHub
- Patreon

Support Mariatta by becoming a sponsor, sending her a happiness packet, or paypal.

R. David Murray (United States)

- Personal site: bitdance.com
- Available for Python and Internet Services Consulting and Python contract programming

David has been involved in the Internet since the days when the old IBM BITNET and the ARPANet got cross connected, and in Python programming since he first discovered it around the days of Python 1.4. After transitioning from being Director of Operations for dialup Internet providers (when that business started declining) to being a full time independent consultant, David started contributing directly to CPython development. He became a committer in 2009. He subsequently took over primary maintenance of the email package from Barry Warsaw, and contributed the unicode oriented API. David is also active in mentoring new contributors and, when time is available, working on the infrastructure that supports CPython development, specifically the Roundup-based bug tracker and the buildbot system.

David currently does both proprietary and open source development work, primarily in Python, through the company in which he is a partner, Murray & Walker, Inc. He has done contract work focused specifically on CPython development both through the PSF (the kickstart of the email unicode API development) and directly funded by interested corporations (additional development work on email funded by QNX, and work on CPython ICC support funded by Intel). He would like to spend more of his (and his company's) time on open source work, and so is actively seeking additional such contract opportunities.

Antoine Pitrou (France)

- LinkedIn: https://www.linkedin.com/in/pitrou/ (Senior Software Engineer)
- Voltron Data
- Python Software Foundation (Fellow)
- Email address: antoine@python.org

Antoine started working with Python in 2005 in order to implement a decentralized virtual world protocol. He started contributing to CPython in 2007 and became a core developer in 2008. His motivations have been driven both by the abstract desire to make Python better for the whole world, and by the concrete roadblocks he was hitting in professional settings. Topics of choice have included interpreter optimizations, garbage collection, network programming, system programming and concurrent programming (such as maintaining `multiprocessing`).

As a professional, Antoine has been first specializing in network programming, and more lately in open source data science infrastructure. He is currently working full time on Apache Arrow as a technical leader for Voltron Data.

Victor Stinner (France)

- Personal website
- Red Hat (Senior Software Engineer)

Victor is paid by Red Hat to maintain Python upstream and downstream (RHEL, CentOS, Fedora & Software collections). See Victor's contributions to Python.

Kushal Das (India)

- Personal website
- Freedom of the Press Foundation (Staff)
- Python Software Foundation (Fellow)

Barry Warsaw (United States)

- LinkedIn: (Senior Staff Software Engineer - Python Foundation team)
- Personal site: barry.warsaw.us
- Blog: We Fear Change
- Email address: barry@python.org
- Python Software Foundation (Fellow)

Barry has been working in, with, and on Python since 1994. He attended the first Python workshop at NBS (now NIST) in Gaithersburg, MD in 1994, where he met Guido and several other early Python adopters. Barry subsequently worked with Guido for 8 years while at CNRI. From 2007 until 2017, Barry worked for Canonical, corporate sponsor of Ubuntu Linux, primarily on the Python ecosystem, and is both an Ubuntu and a Debian uploading developer. Barry has served as Python's postmaster, webmaster, release manager, Language Summit co-chair, Jython project leader, GNU Mailman project leader, and probably lots of other things he shouldn't admit to.

Eric Snow (United States)

- Microsoft (Software Developer)
- Python Software Foundation (Fellow)

Dino Viehland (United States)

- Meta (Software Engineer)
- Email address: dinoviehland@gmail.com

Dino started working with Python in 2005 by working on IronPython, an implementation of Python running on .NET. He was one of the primary developers on the project for 6 years. After that he started the Python Tools for Visual Studio project focusing on providing advanced code completion and debugging features for Python. Today he works on Cinder improving Python performance for Instagram.

Carol Willing (United States)

- Noteable: https://noteable.io/about-us/ (Technical Evangelist)
- Personal site: Willing Consulting
- Extended bio
- Project Jupyter (Steering Council, Core Team for JupyterHub/Binder)
- Python Software Foundation (Fellow)

Carol is focused on Python's usage in education and scientific research. She is interested in organizational development, operational workflows, and sustainability of open source projects.

10.33.2 Goals of this page

The issue metrics automatically collected by the CPython issue tracker strongly suggest that the current core development process is bottlenecked on core developer time - this is most clearly indicated in the first metrics graph, which shows both the number of open issues and the number of patches awaiting review growing steadily over time, despite CPython being one of the most active open source projects in the world. This bottleneck then impacts not only resolving open issues and applying submitted patches, but also the process of identifying, nominating and mentoring new core developers.

The core commit statistics monitored by sites like OpenHub provide a good record as to *who* is currently handling the bulk of the review and maintenance work, but don't provide any indication as to the factors currently influencing people's ability to spend time on reviewing proposed changes, or mentoring new contributors.

This page aims to provide at least some of that missing data by encouraging core developers to highlight professional affiliations in the following two cases (even if not currently paid for time spent participating in the core development process):

- developers working for vendors that distribute a commercially supported Python runtime
- developers working for Sponsor Members of the Python Software Foundation

These are cases where documenting our affiliations helps to improve the overall transparency of the core development process, as well as making it easier for staff at these organisations to locate colleagues that can help them to participate in and contribute effectively to supporting the core development process.

Core developers working for organisations with a vested interest in the sustainability of the CPython core development process are also encouraged to seek opportunities to spend work time on mentoring potential new core developers, whether through the general core mentorship program, through mentoring colleagues, or through more targeted efforts like Outreachy's paid internships and Google's Summer of Code.

Core developers that are available for consulting or contract work on behalf of the Python Software Foundation or other organisations are also encouraged to provide that information here, as this will help the PSF to better facilitate funding of core development work by organisations that don't directly employ any core developers themselves.

Finally, some core developers seeking to increase the time they have available to contribute to CPython may wish to pursue crowdfunding efforts that allow their contributions to be funded directly by the community, rather than relying on institutional sponsors allowing them to spend some or all of their work time contributing to CPython development.

10.33.3 Limitations on scope

- Specific technical areas of interest for core developers should be captured in the *Experts Index*.
- This specific listing is limited to CPython core developers (since it's focused on the specific constraint that is core developer time), but it would be possible to create a more expansive listing on the Python wiki that also covers issue triagers, and folks seeking to become core developers.
- Changes to the software and documentation maintained by core developers, together with related design discussions, all take place in public venues, and hence are inherently subject to full public review. Accordingly, core developers are NOT required to publish their motivations and affiliations if they do not choose to do so. This helps to ensure that core contribution processes remain open to anyone that is in a position to sign the Contributor Licensing Agreement, the details of which are filed privately with the Python Software Foundation, rather than publicly.

10.34 Git Bootcamp and Cheat Sheet

Note: This section provides instructions on common tasks in CPython's workflow. It's designed to assist new contributors who have some familiarity with git and GitHub.

If you are new to git and GitHub, please become comfortable with these instructions before submitting a pull request. As there are several ways to accomplish these tasks using git and GitHub, this section reflects one method suitable for new contributors. Experienced contributors may desire a different approach.

In this section, we will go over some commonly used Git commands that are relevant to CPython's workflow.

Note: Setting up git aliases for common tasks can be useful to you. You can get more information about that in git documentation

10.34.1 Forking CPython GitHub Repository

You will only need to do this once.

1. Go to https://github.com/python/cpython.
2. Press `Fork` on the top right.
3. When asked where to fork the repository, choose to fork it to your username.
4. Your forked CPython repository will be created at https://github.com/<username>/cpython.

10.34.2 Cloning a Forked CPython Repository

You will only need to do this once. From your command line:

```
git clone git@github.com:<username>/cpython.git
```

It is also recommended to configure an `upstream` remote repository:

```
cd cpython
git remote add upstream git@github.com:python/cpython.git
```

You can also use SSH-based or HTTPS-based URLs.

10.34.3 Listing the Remote Repositories

To list the remote repositories that are configured, along with their URLs:

```
git remote -v
```

You should have two remote repositories: `origin` pointing to your forked CPython repository, and `upstream` pointing to the official CPython repository:

```
origin  git@github.com:<username>/cpython.git (fetch)
origin  git@github.com:<username>/cpython.git (push)
upstream        git@github.com:python/cpython.git (fetch)
upstream        git@github.com:python/cpython.git (push)
```

10.34.4 Setting Up Your Name and Email Address

```
git config --global user.name "Your Name"
git config --global user.email your.email@example.com
```

The `--global` flag sets these parameters globally while the `--local` flag sets them only for the current project.

10.34.5 Enabling `autocrlf` on Windows

The `autocrlf` option will fix automatically any Windows-specific line endings. This should be enabled on Windows, since the public repository has a hook which will reject all changesets having the wrong line endings:

```
git config --global core.autocrlf input
```

10.34.6 Creating and Switching Branches

Important: Never commit directly to the `main` branch.

Create a new branch and switch to it:

```
# creates a new branch off main and switch to it
git checkout -b <branch-name> main
```

This is equivalent to:

```
# create a new branch from main, without checking it out
git branch <branch-name> main
# check out the branch
git checkout <branch-name>
```

To find the branch you are currently on:

```
git branch
```

The current branch will have an asterisk next to the branch name. Note, this will only list all of your local branches.

To list all the branches, including the remote branches:

```
git branch -a
```

To switch to a different branch:

```
git checkout <another-branch-name>
```

Other releases are just branches in the repository. For example, to work on the 2.7 release from the `upstream` remote:

```
git checkout -b 2.7 upstream/2.7
```

10.34.7 Deleting Branches

To delete a **local** branch that you no longer need:

```
git checkout main
git branch -D <branch-name>
```

To delete a **remote** branch:

```
git push origin -d <branch-name>
```

You may specify more than one branch for deletion.

10.34.8 Renaming Branch

The CPython repository's default branch was renamed from `master` to `main` after the Python 3.10b1 release.

If you have a fork on GitHub (as described in *Forking CPython GitHub Repository*) that was created before the rename, you should visit the GitHub page for your fork to rename the branch there. You only have to do this once. GitHub should provide you with a dialog for this. If it doesn't (or the dialog was already dismissed), you can rename the branch in your fork manually by following these GitHub instructions

After renaming the branch in your fork, you need to update any local clones as well. This only has to be done once per clone:

```
git branch -m master main
git fetch origin
git branch -u origin/main main
git remote set-head origin -a
```

(GitHub also provides these instructions after you rename the branch.)

If you do not have a fork on GitHub, but rather a direct clone of the main repo created before the branch rename, you still have to update your local clones. This still only has to be done once per clone. In that case, you can rename your local branch as follows:

```
git branch -m master main
git fetch upstream
git branch -u upstream/main main
```

10.34.9 Staging and Committing Files

1. To show the current changes:

   ```
   git status
   ```

2. To stage the files to be included in your commit:

   ```
   git add <filename1> <filename2>
   ```

3. To commit the files that have been staged (done in step 2):

   ```
   git commit -m "This is the commit message."
   ```

10.34.10 Reverting Changes

To revert changes to a file that has not been committed yet:

```
git checkout <filename>
```

If the change has been committed, and now you want to reset it to whatever the origin is at:

```
git reset --hard HEAD
```

10.34.11 Stashing Changes

To stash away changes that are not ready to be committed yet:

```
git stash
```

To re-apply the last stashed change:

```
git stash pop
```

10.34.12 Committing Changes

Add the files you want to commit:

```
git add <filename>
```

Commit the files:

```
git commit -m "<message>"
```

10.34.13 Comparing Changes

View all non-commited changes:

```
git diff
```

Compare to the `main` branch:

```
git diff main
```

Exclude generated files from diff using an `attr` pathspec (note the single quotes):

```
git diff main ':(attr:!generated)'
```

Exclude generated files from diff by default:

```
git config diff.generated.binary true
```

The `generated` attribute is defined in `.gitattributes`, found in the repository root.

10.34.14 Pushing Changes

Once your changes are ready for a review or a pull request, you will need to push them to the remote repository.

```
git checkout <branch-name>
git push origin <branch-name>
```

10.34.15 Creating a Pull Request

1. Go to https://github.com/python/cpython.
2. Press the `New pull request` button.
3. Click the `compare across forks` link.
4. Select the base repository: `python/cpython` and base branch: `main`.
5. Select the head repository: `<username>/cpython` and head branch: the branch containing your changes.
6. Press the `Create pull request` button.

10.34.16 Updating your CPython Fork

Scenario:

- You forked the CPython repository some time ago.
- Time passes.
- There have been new commits made in the upstream CPython repository.
- Your forked CPython repository is no longer up to date.
- You now want to update your forked CPython repository to be the same as the upstream CPython repository.

Please do not try to solve this by creating a pull request from `python:main` to `<username>:main` as the authors of the patches will get notified unnecessarily.

Solution:

```
git checkout main
git pull upstream main
git push origin main
```

Note: For the above commands to work, please follow the instructions found in the *Get the source code* section

Another scenario:

- You created `some-branch` some time ago.
- Time passes.
- You made some commits to `some-branch`.
- Meanwhile, there are recent changes from the upstream CPython repository.
- You want to incorporate the recent changes from the upstream CPython repository into `some-branch`.

Solution:

```
git checkout some-branch
git fetch upstream
git merge upstream/main
git push origin some-branch
```

You may see error messages like "CONFLICT" and "Automatic merge failed;" when you run `git merge upstream/main`.

When it happens, you need to resolve conflict. See these articles about resolving conflicts:

- About merge conflicts
- Resolving a merge conflict using the command line

10.34.17 Applying a Patch to Git

Scenario:

- A patch exists but there is no pull request for it.

Solution:

1. Download the patch locally.
2. Apply the patch:

   ```
   git apply /path/to/patch.diff
   ```

   If there are errors, update to a revision from when the patch was created and then try the `git apply` again:

   ```
   git checkout $(git rev-list -n 1 --before="yyyy-mm-dd hh:mm:ss" main)
   git apply /path/to/patch.diff
   ```

If the patch still won't apply, then a patch tool will not be able to apply the patch and it will need to be re-implemented manually.

3. If the apply was successful, create a new branch and switch to it.
4. Stage and commit the changes.
5. If the patch was applied to an old revision, it needs to be updated and merge conflicts need to be resolved:

```
git rebase main
git mergetool
```

   For very old changes, `git merge --no-ff` may be easier than a rebase, with regards to resolving conflicts.

6. Push the changes and open a pull request.

10.34.18 Downloading Other's Patches

Scenario:

- A contributor made a pull request to CPython.
- Before merging it, you want to be able to test their changes locally.

If you've got GitHub CLI or hub installed, you can simply do:

```
$ gh pr checkout <pr_number>   # GitHub CLI
$ hub pr checkout <pr_number>  # hub
```

Both of these tools will configure a remote URL for the branch, so you can `git push` if the pull request author checked "Allow edits from maintainers" when creating the pull request.

If you don't have GitHub CLI or hub installed, you can set up a git alias. On Unix and macOS:

```
$ git config --global alias.pr '!sh -c "git fetch upstream pull/${1}/head:pr_${1} && git␣
↪checkout pr_${1}" -'
```

On Windows, reverse the single (`'`) and double (`"`) quotes:

```
git config --global alias.pr "!sh -c 'git fetch upstream pull/${1}/head:pr_${1} && git␣
↪checkout pr_${1}' -"
```

The alias only needs to be done once. After the alias is set up, you can get a local copy of a pull request as follows:

```
git pr <pr_number>
```

10.34.19 Accepting and Merging a Pull Request

Pull requests can be accepted and merged by a Python Core Developer.

1. At the bottom of the pull request page, click the `Squash and merge` button.
2. Replace the reference to GitHub pull request `#NNNN` with `GH-NNNN`. If the title is too long, the pull request number can be added to the message body.
3. Adjust and clean up the commit message.

   Example of good commit message:

```
gh-12345: Improve the spam module (GH-777)

* Add method A to the spam module
* Update the documentation of the spam module
```

Example of bad commit message:

```
gh-12345: Improve the spam module (#777)

* Improve the spam module
* merge from main
* adjust code based on review comment
* rebased
```

Note: How to Write a Git Commit Message is a nice article describing how to write a good commit message.

4. Press the `Confirm squash and merge` button.

10.34.20 Backporting Merged Changes

A pull request may need to be backported into one of the maintenance branches after it has been accepted and merged into `main`. It is usually indicated by the label `needs backport to X.Y` on the pull request itself.

Use the utility script cherry_picker.py from the core-workflow repository to backport the commit.

The commit hash for backporting is the squashed commit that was merged to the `main` branch. On the merged pull request, scroll to the bottom of the page. Find the event that says something like:

```
<core_developer> merged commit <commit_sha1> into python:main <sometime> ago.
```

By following the link to `<commit_sha1>`, you will get the full commit hash.

Alternatively, the commit hash can also be obtained by the following git commands:

```
git fetch upstream
git rev-parse ":/gh-12345"
```

The above commands will print out the hash of the commit containing `"gh-12345"` as part of the commit message.

When formatting the commit message for a backport commit: leave the original one as is and delete the number of the backport pull request.

Example of good backport commit message:

```
gh-12345: Improve the spam module (GH-777)

* Add method A to the spam module
* Update the documentation of the spam module

(cherry picked from commit 62adc55)
```

Example of bad backport commit message:

```
gh-12345: Improve the spam module (GH-777) (#888)

* Add method A to the spam module
* Update the documentation of the spam module
```

10.34.21 Editing a Pull Request Prior to Merging

When a pull request submitter has enabled the Allow edits from maintainers option, Python Core Developers may decide to make any remaining edits needed prior to merging themselves, rather than asking the submitter to do them. This can be particularly appropriate when the remaining changes are bookkeeping items like updating `Misc/ACKS`.

To edit an open pull request that targets `main`:

1. In the pull request page, under the description, there is some information about the contributor's forked CPython repository and branch name that will be useful later:

   ```
   <contributor> wants to merge 1 commit into python:main from <contributor>:<branch_
   ↪name>
   ```

2. Fetch the pull request, using the *git pr* alias:

   ```
   git pr <pr_number>
   ```

   This will checkout the contributor's branch at `<pr_number>`.

3. Make and commit your changes on the branch. For example, merge in changes made to `main` since the PR was submitted (any merge commits will be removed by the later `Squash and Merge` when accepting the change):

   ```
   git fetch upstream
   git merge upstream/main
   git add <filename>
   git commit -m "<message>"
   ```

4. Push the changes back to the contributor's PR branch:

   ```
   git push git@github.com:<contributor>/cpython <pr_number>:<branch_name>
   ```

5. Optionally, *delete the PR branch*.

10.35 Appendix: Topics

10.35.1 Basics for contributors

Note: Recommended reading

- *Getting Started*
- *Lifecycle of a Pull Request*

- *Where to Get Help*
- *Following Python's Development*

10.35.2 Core developers

Note: Recommended reading

- *Getting Started*
- *Lifecycle of a Pull Request*
- *Accepting Pull Requests*

- *How to Become a Core Developer*
- *Developer Log*
- *Core Developer Motivations and Affiliations*
- *Experts Index*

10.35.3 Development workflow for contributors

- *Development Cycle*
- *Lifecycle of a Pull Request*
- *Fixing "easy" Issues (and Beyond)*

10.35.4 Documenting Python and style guide

- *Helping with Documentation*
- *Documenting Python*

10.35.5 Issue tracking and triaging

- *Issue Tracking*
- *Triaging an Issue*
- *GitHub Labels*
- *GitHub issues for BPO users*

10.35.6 Language development in depth

- *Exploring CPython's Internals*
- *Changing CPython's Grammar*
- *Design of CPython's Compiler*
- *Design of CPython's Garbage Collector*
- *Adding to the Stdlib*
- *Changing the Python Language*
- *Porting Python to a new platform*

10.35.7 Testing and continuous integration

BIBLIOGRAPHY

[Wang97] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Chris S. Serra. The Zephyr Abstract Syntax Description Language. In Proceedings of the Conference on Domain-Specific Languages, pp. 213–227, 1997.

INDEX