
Python Frequently Asked Questions

3.7.2

Guido van Rossum
and the Python development team

12, 2019

Python Software Foundation
Email: docs@python.org

Contents

1	Python	1
2		9
3		39
4	FAQ	53
5	Extending/Embedding FAQ	63
6	Python on Windows FAQ	71
7	Graphic User Interface FAQ	77
8	“ Python ”	81
A		83
B		93
C	History and License	95
D		113
		115

1.1

1.1.1 Python

Python Python Unix Mac Python C C ++
tutorial-index Python Windows 2000 Python

1.1.2 Python

The Python Software Foundation is an independent non-profit organization that holds the copyright on Python versions 2.1 and newer. The PSF's mission is to advance open source technology related to the Python programming language and to publicize the use of Python. The PSF's home page is at <https://www.python.org/psf/>.

Donations to the PSF are tax-exempt in the US. If you use Python and find it helpful, please contribute via [the PSF donation page](#).

1.1.3 Are there copyright restrictions on the use of Python?

You can do anything you want with the source, as long as you leave the copyrights in and display those copyrights in any documentation about Python that you produce. If you honor the copyright rules, it's OK to use Python for commercial use, to sell copies of Python in source or binary form (modified or unmodified), or to sell products that incorporate Python in some form. We would still like to know about all commercial use of Python, of course.

See [the PSF license page](#) to find further explanations and a link to the full text of the license.

The Python logo is trademarked, and in certain cases permission is required to use it. Consult [the Trademark Usage Policy](#) for more information.

1.1.4 Why was Python created in the first place?

Here's a *very* brief summary of what started it all, written by Guido van Rossum:

I had extensive experience with implementing an interpreted language in the ABC group at CWI, and from working with this group I had learned a lot about language design. This is the origin of many Python features, including the use of indentation for statement grouping and the inclusion of very-high-level data types (although the details are all different in Python).

I had a number of gripes about the ABC language, but also liked many of its features. It was impossible to extend the ABC language (or its implementation) to remedy my complaints – in fact its lack of extensibility was one of its biggest problems. I had some experience with using Modula-2+ and talked with the designers of Modula-3 and read the Modula-3 report. Modula-3 is the origin of the syntax and semantics used for exceptions, and some other Python features.

I was working in the Amoeba distributed operating system group at CWI. We needed a better way to do system administration than by writing either C programs or Bourne shell scripts, since Amoeba had its own system call interface which wasn't easily accessible from the Bourne shell. My experience with error handling in Amoeba made me acutely aware of the importance of exceptions as a programming language feature.

It occurred to me that a scripting language with a syntax like ABC but with access to the Amoeba system calls would fill the need. I realized that it would be foolish to write an Amoeba-specific language, so I decided that I needed a language that was generally extensible.

During the 1989 Christmas holidays, I had a lot of time on my hand, so I decided to give it a try. During the next year, while still mostly working on it in my own time, Python was used in the Amoeba project with increasing success, and the feedback from colleagues made me add many early improvements.

In February 1991, after just over a year of development, I decided to post to USENET. The rest is in the `Misc/HISTORY` file.

1.1.5 What is Python good for?

Python is a high-level general-purpose programming language that can be applied to many different classes of problems.

The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for library-index to get an idea of what's available. A wide variety of third-party extensions are also available. Consult [the Python Package Index](#) to find packages of interest to you.

1.1.6 How does the Python version numbering scheme work?

Python versions are numbered A.B.C or A.B. A is the major version number – it is only incremented for really major changes in the language. B is the minor version number, incremented for less earth-shattering changes. C is the micro-level – it is incremented for each bugfix release. See [PEP 6](#) for more information about bugfix releases.

Not all releases are bugfix releases. In the run-up to a new major release, a series of development releases are made, denoted as alpha, beta, or release candidate. Alphas are early releases in which interfaces aren't yet finalized; it's not unexpected to see an interface change between two alpha releases. Betas are more stable,

preserving existing interfaces but possibly adding new modules, and release candidates are frozen, making no changes except as needed to fix critical bugs.

Alpha, beta and release candidate versions have an additional suffix. The suffix for an alpha version is "aN" for some small number N, the suffix for a beta version is "bN" for some small number N, and the suffix for a release candidate version is "cN" for some small number N. In other words, all versions labeled 2.0aN precede the versions labeled 2.0bN, which precede versions labeled 2.0cN, and *those* precede 2.0.

You may also find version numbers with a "+" suffix, e.g. "2.2+". These are unreleased versions, built directly from the CPython development repository. In practice, after a final minor release is made, the version is incremented to the next minor version, which becomes the "a0" version, e.g. "2.4a0".

See also the documentation for `sys.version`, `sys.hexversion`, and `sys.version_info`.

1.1.7 How do I obtain a copy of the Python source?

The latest Python source distribution is always available from python.org, at <https://www.python.org/downloads/>. The latest development sources can be obtained at <https://github.com/python/cpython/>.

The source distribution is a gzipped tar file containing the complete C source, Sphinx-formatted documentation, Python library modules, example programs, and several useful pieces of freely distributable software. The source will compile and run out of the box on most UNIX platforms.

Consult the [Getting Started](#) section of the [Python Developer's Guide](#) for more information on getting the source code and compiling it.

1.1.8 How do I get documentation on Python?

The standard documentation for the current stable version of Python is available at <https://docs.python.org/3/>. PDF, plain text, and downloadable HTML versions are also available at <https://docs.python.org/3/download.html>.

The documentation is written in reStructuredText and processed by the [Sphinx documentation tool](#). The reStructuredText source for the documentation is part of the Python source distribution.

1.1.9 I've never programmed before. Is there a Python tutorial?

There are numerous tutorials and books available. The standard documentation includes [tutorial-index](#).

Consult the [Beginner's Guide](#) to find information for beginning Python programmers, including lists of tutorials.

1.1.10 Is there a newsgroup or mailing list devoted to Python?

There is a newsgroup, [comp.lang.python](#), and a mailing list, [python-list](#). The newsgroup and mailing list are gatewayed into each other – if you can read news it's unnecessary to subscribe to the mailing list. [comp.lang.python](#) is high-traffic, receiving hundreds of postings every day, and Usenet readers are often more able to cope with this volume.

Announcements of new software releases and events can be found in [comp.lang.python.announce](#), a low-traffic moderated list that receives about five postings per day. It's available as the [python-announce mailing list](#).

More info about other mailing lists and newsgroups can be found at <https://www.python.org/community/lists/>.

1.1.11 How do I get a beta test version of Python?

Alpha and beta releases are available from <https://www.python.org/downloads/>. All releases are announced on the comp.lang.python and comp.lang.python.announce newsgroups and on the Python home page at <https://www.python.org/>; an RSS feed of news is available.

You can also access the development version of Python through Git. See [The Python Developer's Guide](#) for details.

1.1.12 How do I submit bug reports and patches for Python?

To report a bug or submit a patch, please use the Roundup installation at <https://bugs.python.org/>.

You must have a Roundup account to report bugs; this makes it possible for us to contact you if we have follow-up questions. It will also enable Roundup to send you updates as we act on your bug. If you had previously used SourceForge to report bugs to Python, you can obtain your Roundup password through Roundup's [password reset procedure](#).

For more information on how Python is developed, consult [the Python Developer's Guide](#).

1.1.13 Are there any published articles about Python that I can reference?

It's probably best to cite your favorite book about Python.

The very first article about Python was written in 1991 and is now quite outdated.

Guido van Rossum and Jelke de Boer, "Interactively Testing Remote Servers Using the Python Programming Language", CWI Quarterly, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283–303.

1.1.14 Are there any books on Python?

Yes, there are many, and more are being published. See the python.org wiki at <https://wiki.python.org/moin/PythonBooks> for a list.

You can also search online bookstores for "Python" and filter out the Monty Python references; or perhaps search for "Python" and "language".

1.1.15 Where in the world is www.python.org located?

The Python project's infrastructure is located all over the world and is managed by the Python Infrastructure Team. Details [here](#).

1.1.16 Why is it called Python?

When he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus", a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

1.1.17 Do I have to like "Monty Python's Flying Circus"?

No, but it helps. :)

1.2 Python in the real world

1.2.1 How stable is Python?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. Currently there are usually around 18 months between major releases.

The developers issue "bugfix" releases of older versions, so the stability of existing releases gradually improves. Bugfix releases, indicated by a third component of the version number (e.g. 3.5.3, 3.6.2), are managed for stability; only fixes for known problems are included in a bugfix release, and it's guaranteed that interfaces will remain the same throughout a series of bugfix releases.

The latest stable releases can always be found on the [Python download page](#). There are two production-ready version of Python: 2.x and 3.x, but the recommended one at this times is Python 3.x. Although Python 2.x is still widely used, [it will not be maintained after January 1, 2020](#). Python 2.x was known for having more third-party libraries available, however, by the time of this writing, most of the widely used libraries support Python 3.x, and some are even dropping the Python 2.x support.

1.2.2 How many people are using Python?

There are probably tens of thousands of users, though it's difficult to obtain an exact count.

Python is available for free download, so there are no sales figures, and it's available from many different sites and packaged with many Linux distributions, so download statistics don't tell the whole story either.

The `comp.lang.python` newsgroup is very active, but not all Python users post to the group or even read it.

1.2.3 Have any significant projects been done in Python?

See <https://www.python.org/about/success> for a list of projects that use Python. Consulting the proceedings for [past Python conferences](#) will reveal contributions from many different companies and organizations.

High-profile Python projects include [the Mailman mailing list manager](#) and [the Zope application server](#). Several Linux distributions, most notably [Red Hat](#), have written part or all of their installer and system administration software in Python. Companies that use Python internally include Google, Yahoo, and Lucasfilm Ltd.

1.2.4 What new developments are expected for Python in the future?

See <https://www.python.org/dev/peps/> for the Python Enhancement Proposals (PEPs). PEPs are design documents describing a suggested new feature for Python, providing a concise technical specification and a rationale. Look for a PEP titled "Python X.Y Release Schedule", where X.Y is a version that hasn't been publicly released yet.

New development is discussed on [the python-dev mailing list](#).

1.2.5 Is it reasonable to propose incompatible changes to Python?

In general, no. There are already millions of lines of Python code around the world, so any change in the language that invalidates more than a very small fraction of existing programs has to be frowned upon. Even if you can provide a conversion program, there's still the problem of updating all documentation; many books have been written about Python, and we don't want to invalidate them all at a single stroke.

Providing a gradual upgrade path is necessary if a feature has to be changed. [PEP 5](#) describes the procedure followed for introducing backward-incompatible changes while minimizing disruption for users.

1.2.6 Is Python a good language for beginning programmers?

It is still common to start students with a procedural and statically typed language such as Pascal, C, or a subset of C++ or Java. Students may be better served by learning Python as their first language. Python has a very simple and consistent syntax and a large standard library and, most importantly, using Python in a beginning programming course lets students concentrate on important programming skills such as problem decomposition and data type design. With Python, students can be quickly introduced to basic concepts such as loops and procedures. They can probably even work with user-defined objects in their very first course.

For a student who has never programmed before, using a statically typed language seems unnatural. It presents additional complexity that the student must master and slows the pace of the course. The students are trying to learn to think like a computer, decompose problems, design consistent interfaces, and encapsulate data. While learning to use a statically typed language is important in the long term, it is not necessarily the best topic to address in the students' first programming course.

Many other aspects of Python make it a good first language. Like Java, Python has a large standard library so that students can be assigned programming projects very early in the course that *do* something. Assignments aren't restricted to the standard four-function calculator and check balancing programs. By using the standard library, students can gain the satisfaction of working on realistic applications as they learn the fundamentals of programming. Using the standard library also teaches students about code reuse. Third-party modules such as PyGame are also helpful in extending the students' reach.

Python's interactive interpreter enables students to test language features while they're programming. They can keep a window with the interpreter running while they enter their program's source in another window. If they can't remember the methods for a list, they can do something like this:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
```

()

()

```
>>> L  
[1]
```

With the interpreter, documentation is never far from the student as they are programming.

There are also good IDEs for Python. IDLE is a cross-platform IDE for Python that is written in Python using Tkinter. PythonWin is a Windows-specific IDE. Emacs users will be happy to know that there is a very good Python mode for Emacs. All of these programming environments provide syntax highlighting, auto-indenting, and access to the interactive interpreter while coding. Consult [the Python wiki](#) for a full list of Python editing environments.

If you want to discuss Python's use in education, you may be interested in joining [the edu-sig mailing list](#).

2.1

2.1.1 Python

pdb	Python	Python	pdb
Python	IDLE	Tools/scripts/idle	

PythonWin is a Python IDE that includes a GUI debugger based on pdb. The Pythonwin debugger colors breakpoints and has quite a few cool features such as debugging non-Pythonwin programs. Pythonwin is available as part of the [Python for Windows Extensions](https://www.activestate.com/activepython/) project and as a part of the ActivePython distribution (see <https://www.activestate.com/activepython/>).

[Boa Constructor](#) is an IDE and GUI builder that uses wxWidgets. It offers visual frame creation and manipulation, an object inspector, many views on the source like object browsers, inheritance hierarchies, doc string generated html documentation, an advanced debugger, integrated help, and Zope support.

[Eric](#) is an IDE built on PyQt and the Scintilla editing component.

Pydb is a version of the standard Python debugger pdb, modified for use with DDD (Data Display Debugger), a popular graphical debugger front end. Pydb can be found at <http://bashdb.sourceforge.net/pydb/> and DDD can be found at <https://www.gnu.org/software/ddd>.

There are a number of commercial Python IDEs that include graphical debuggers. They include:

- Wing IDE (<https://wingware.com/>)
- Komodo IDE (<https://komodoide.com/>)
- PyCharm (<https://www.jetbrains.com/pycharm/>)

2.1.2 Is there a tool to help find bugs or perform static analysis?

PyChecker is a static analysis tool that finds bugs in Python source code and warns about code complexity and style. You can get PyChecker from <http://pychecker.sourceforge.net/>.

Pylint is another tool that checks if a module satisfies a coding standard, and also makes it possible to write plug-ins to add a custom feature. In addition to the bug checking that PyChecker performs, Pylint offers some additional features such as checking line length, whether variable names are well-formed according to your coding standard, whether declared interfaces are fully implemented, and more. <https://docs.pylint.org/> provides a full list of Pylint's features.

Static type checkers such as [Mypy](#), [Pyre](#), and [Pytype](#) can check type hints in Python source code.

2.1.3 How can I create a stand-alone binary from a Python script?

You don't need the ability to compile Python to C code if all you want is a stand-alone program that users can download and run without having to install the Python distribution first. There are a number of tools that determine the set of modules required by a program and bind these modules together with a Python binary to produce a single executable.

One is to use the freeze tool, which is included in the Python source tree as `Tools/freeze`. It converts Python byte code to C arrays; a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

It works by scanning your source recursively for import statements (in both forms) and looking for the modules in the standard Python path as well as in the source directory (for built-in modules). It then turns the bytecode for modules written in Python into C code (array initializers that can be turned into code objects using the marshal module) and creates a custom-made config file that only contains those built-in modules which are actually used in the program. It then compiles the generated C code and links it with the rest of the Python interpreter to form a self-contained binary which acts exactly like your script.

Obviously, freeze requires a C compiler. There are several other utilities which don't. One is Thomas Heller's py2exe (Windows only) at

<http://www.py2exe.org/>

Another tool is Anthony Tuininga's `cx_Freeze`.

2.1.4 Are there coding standards or a style guide for Python programs?

Yes. The coding style required for standard library modules is documented as [PEP 8](#).

2.2 Core Language

2.2.1 Why am I getting an UnboundLocalError when the variable has a value?

It can be a surprise to get the UnboundLocalError in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

This code:

```
>>> x = 10
>>> def bar():
...     print(x)
>>> bar()
10
```

works, but this code:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

results in an `UnboundLocalError`:

```
>>> foo()
Traceback (most recent call last):
...
UnboundLocalError: local variable 'x' referenced before assignment
```

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in `foo` assigns a new value to `x`, the compiler recognizes it as a local variable. Consequently when the earlier `print(x)` attempts to print the uninitialized local variable and an error results.

In the example above you can access the outer scope variable by declaring it global:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

This explicit declaration is required in order to remind you that (unlike the superficially analogous situation with class and instance variables) you are actually modifying the value of the variable in the outer scope:

```
>>> print(x)
11
```

You can do a similar thing in a nested scope using the `nonlocal` keyword:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
>>> foo()
10
11
```

2.2.2 What are the rules for local and global variables in Python?

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring `global` for assigned variables provides a bar against unintended side-effects. On the other hand, if `global` was required for all global references, you'd be using `global` all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the `global` declaration for identifying side-effects.

2.2.3 Why do lambdas defined in a loop with different values all return the same result?

Assume you use a for loop to define a few different lambdas (or even plain functions), e.g.:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

This gives you a list that contains 5 lambdas that calculate `x**2`. You might expect that, when called, they would return, respectively, 0, 1, 4, 9, and 16. However, when you actually try you will see that they all return 16:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

This happens because `x` is not local to the lambdas, but is defined in the outer scope, and it is accessed when the lambda is called — not when it is defined. At the end of the loop, the value of `x` is 4, so all the functions now return `4**2`, i.e. 16. You can also verify this by changing the value of `x` and see how the results of the lambdas change:

```
>>> x = 8
>>> squares[2]()
64
```

In order to avoid this, you need to save the values in variables local to the lambdas, so that they don't rely on the value of the global `x`:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Here, `n=x` creates a new variable `n` local to the lambda and computed when the lambda is defined so that it has the same value that `x` had at that point in the loop. This means that the value of `n` will be 0 in the first lambda, 1 in the second, 2 in the third, and so on. Therefore each lambda will now return the correct result:

```
>>> squares[2]()
4
>>> squares[4]()
16
```


Note that this behaviour is not peculiar to lambdas, but applies to regular functions too.

2.2.4 How do I share global variables across modules?

The canonical way to share information across modules within a single program is to create a special module (often called `config` or `cfg`). Just import the `config` module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere. For example:

`config.py`:

```
x = 0    # Default value of the 'x' configuration setting
```

`mod.py`:

```
import config
config.x = 1
```

`main.py`:

```
import config
import mod
print(config.x)
```

Note that using a module is also the basis for implementing the Singleton design pattern, for the same reason.

2.2.5 What are the "best practices" for using import in a module?

In general, don't use `from modulename import *`. Doing so clutters the importer's namespace, and makes it much harder for linters to detect undefined names.

Import modules at the top of a file. Doing so makes it clear what other modules your code requires and avoids questions of whether the module name is in scope. Using one import per line makes it easy to add and delete module imports, but using multiple imports per line uses less screen space.

It's good practice if you import modules in the following order:

1. standard library modules – e.g. `sys`, `os`, `getopt`, `re`
2. third-party library modules (anything installed in Python's site-packages directory) – e.g. `mx.DateTime`, `ZODB`, `PIL.Image`, etc.
3. locally-developed modules

It is sometimes necessary to move imports to a function or class to avoid problems with circular imports. Gordon McMillan says:

Circular imports are fine where both modules use the "import <module>" form of import. They fail when the 2nd module wants to grab a name out of the first ("from module import name") and the import is at the top level. That's because names in the 1st are not yet available, because the first module is busy importing the 2nd.

In this case, if the second module is only used in one function, then the import can easily be moved into that function. By the time the import is called, the first module will have finished initializing, and the second module can do its import.

It may also be necessary to move imports out of the top level of code if some of the modules are platform-specific. In that case, it may not even be possible to import all of the modules at the top of the file. In this case, importing the correct modules in the corresponding platform-specific code is a good option.

Only move imports into a local scope, such as inside a function definition, if it's necessary to solve a problem such as avoiding a circular import or are trying to reduce the initialization time of a module. This technique is especially helpful if many of the imports are unnecessary depending on how the program executes. You may also want to move imports into a function if the modules are only ever used in that function. Note that loading a module the first time may be expensive because of the one time initialization of the module, but loading a module multiple times is virtually free, costing only a couple of dictionary lookups. Even if the module name has gone out of scope, the module is probably available in `sys.modules`.

2.2.6 Why are default values shared between objects?

This type of bug commonly bites neophyte programmers. Consider this function:

```
def foo(mydict={}): # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

The first time you call this function, `mydict` contains a single item. The second time, `mydict` contains two items because when `foo()` begins executing, `mydict` starts out with an item already in it.

It is often expected that a function call creates new objects for default values. This is not what happens. Default values are created exactly once, when the function is defined. If that object is changed, like the dictionary in this example, subsequent calls to the function will refer to this changed object.

By definition, immutable objects such as numbers, strings, tuples, and `None`, are safe from change. Changes to mutable objects such as dictionaries, lists, and class instances can lead to confusion.

Because of this feature, it is good programming practice to not use mutable objects as default values. Instead, use `None` as the default value and inside the function, check if the parameter is `None` and create a new list/dictionary/whatever if it is. For example, don't write:

```
def foo(mydict={}):
    ...
```

but:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {} # create a new dict for local namespace
```

This feature can be useful. When you have a function that's time-consuming to compute, a common technique is to cache the parameters and the resulting value of each call to the function, and return the cached value if the same value is requested again. This is called "memoizing", and can be implemented like this:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
```

()

()

```
_cache[(arg1, arg2)] = result      # Store result in the cache
return result
```

You could use a global variable containing a dictionary instead of the default value; it's a matter of taste.

2.2.7 How can I pass optional or keyword parameters from one function to another?

Collect the arguments using the `*` and `**` specifiers in the function's parameter list; this gives you the positional arguments as a tuple and the keyword arguments as a dictionary. You can then pass these arguments when calling another function by using `*` and `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

2.2.8 What is the difference between arguments and parameters?

Parameters are defined by the names that appear in a function definition, whereas *arguments* are the values actually passed to a function when calling it. Parameters define what types of arguments a function can accept. For example, given the function definition:

```
def func(foo, bar=None, **kwargs):
    pass
```

`foo`, `bar` and `kwargs` are parameters of `func`. However, when calling `func`, for example:

```
func(42, bar=314, extra=somevar)
```

the values 42, 314, and `somevar` are arguments.

2.2.9 Why did changing list 'y' also change list 'x'?

If you wrote code like:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

you might be wondering why appending an element to `y` changed `x` too.

There are two factors that produce this result:

- 1) Variables are simply names that refer to objects. Doing `y = x` doesn't create a copy of the list – it creates a new variable `y` that refers to the same object `x` refers to. This means that there is only one object (the list), and both `x` and `y` refer to it.

2) Lists are *mutable*, which means that you can change their content.

After the call to `append()`, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

If we instead assign an immutable object to `x`:

```
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

we can see that in this case `x` and `y` are not equal anymore. This is because integers are *immutable*, and when we do `x = x + 1` we are not mutating the int 5 by incrementing its value; instead, we are creating a new object (the int 6) and assigning it to `x` (that is, changing which object `x` refers to). After this assignment we have two objects (the ints 6 and 5) and two variables that refer to them (`x` now refers to 6 but `y` still refers to 5).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and `sorted(y)`) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily diagnosed error.

However, there is one class of operations where the same operation sometimes has different behaviors with different types: the augmented assignment operators. For example, `+=` mutates lists but not tuples or ints (`a_list += [1, 2, 3]` is equivalent to `a_list.extend([1, 2, 3])` and mutates `a_list`, whereas `some_tuple += (1, 2, 3)` and `some_int += 1` create new objects).

In other words:

- If we have a mutable object (`list`, `dict`, `set`, etc.), we can use some specific operations to mutate it and all the variables that refer to it will see the change.
- If we have an immutable object (`str`, `int`, `tuple`, etc.), all the variables that refer to it will always see the same value, but operations that transform that value into a new value always return a new object.

If you want to know if two variables refer to the same object or not, you can use the `is` operator, or the built-in function `id()`.

2.2.10 How do I write a function with output parameters (call by reference)?

Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there's no alias between an argument name in the caller and callee, and so no call-by-reference per se. You can achieve the desired effect in a number of ways.

1) By returning a tuple of the results:

```
def func2(a, b):
    a = 'new-value'          # a and b are local names
    b = b + 1                # assigned to new objects
    return a, b              # return new values
```

()

()

```
x, y = 'old-value', 99
x, y = func2(x, y)
print(x, y)                # output: new-value 100
```

This is almost always the clearest solution.

- 2) By using global variables. This isn't thread-safe, and is not recommended.
- 3) By passing a mutable (changeable in-place) object:

```
def func1(a):
    a[0] = 'new-value'      # 'a' references a mutable list
    a[1] = a[1] + 1         # changes a shared object

args = ['old-value', 99]
func1(args)
print(args[0], args[1])    # output: new-value 100
```

- 4) By passing in a dictionary that gets mutated:

```
def func3(args):
    args['a'] = 'new-value'  # args is a mutable dictionary
    args['b'] = args['b'] + 1 # change it in-place

args = {'a': 'old-value', 'b': 99}
func3(args)
print(args['a'], args['b'])
```

- 5) Or bundle up values in a class instance:

```
class callByRef:
    def __init__(self, **args):
        for (key, value) in args.items():
            setattr(self, key, value)

def func4(args):
    args.a = 'new-value'      # args is a mutable callByRef
    args.b = args.b + 1       # change object in-place

args = callByRef(a='old-value', b=99)
func4(args)
print(args.a, args.b)
```

There's almost never a good reason to get this complicated.

Your best choice is to return a tuple containing the multiple results.

2.2.11 How do you make a higher order function in Python?

You have two choices: you can use nested scopes or you can use callable objects. For example, suppose you wanted to define `linear(a,b)` which returns a function `f(x)` that computes the value `a*x+b`. Using nested scopes:

```
def linear(a, b):  
    def result(x):  
        return a * x + b  
    return result
```

Or using a callable object:

```
class linear:  
  
    def __init__(self, a, b):  
        self.a, self.b = a, b  
  
    def __call__(self, x):  
        return self.a * x + self.b
```

In both cases,

```
taxes = linear(0.3, 2)
```

gives a callable object where `taxes(10e6) == 0.3 * 10e6 + 2`.

The callable object approach has the disadvantage that it is a bit slower and results in slightly longer code. However, note that a collection of callables can share their signature via inheritance:

```
class exponential(linear):  
    # __init__ inherited  
    def __call__(self, x):  
        return self.a * (x ** self.b)
```

Object can encapsulate state for several methods:

```
class counter:  
  
    value = 0  
  
    def set(self, x):  
        self.value = x  
  
    def up(self):  
        self.value = self.value + 1  
  
    def down(self):  
        self.value = self.value - 1  
  
count = counter()  
inc, dec, reset = count.up, count.down, count.set
```

Here `inc()`, `dec()` and `reset()` act like functions which share the same counting variable.

2.2.12 How do I copy an object in Python?

In general, try `copy.copy()` or `copy.deepcopy()` for the general case. Not all objects can be copied, but most can.

Some objects can be copied more easily. Dictionaries have a `copy()` method:

```
newdict = olddict.copy()
```

Sequences can be copied by slicing:

```
new_l = l[:]
```

2.2.13 How can I find the methods or attributes of an object?

For an instance `x` of a user-defined class, `dir(x)` returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

2.2.14 How can my code discover the name of an object?

Generally speaking, it can't, because objects don't really have names. Essentially, assignment always binds a name to a value; The same is true of `def` and `class` statements, but in that case the value is a callable. Consider the following code:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two names and invoked through the name `B` the created instance is still reported as an instance of class `A`. However, it is impossible to say whether the instance's name is `a` or `b`, since both names are bound to the same value.

Generally speaking it should not be necessary for your code to "know the names" of particular values. Unless you are deliberately writing introspective programs, this is usually an indication that a change of approach might be beneficial.

In `comp.lang.python`, Fredrik Lundh once gave an excellent analogy in answer to this question:

The same way as you get the name of that cat you found on your porch: the cat (object) itself cannot tell you its name, and it doesn't really care – so the only way to find out what it's called is to ask all your neighbours (namespaces) if it's their cat (object)...

....and don't be surprised if you'll find that it's known by many names, or no name at all!

2.2.15 What's up with the comma operator's precedence?

Comma is not an operator in Python. Consider this session:

```
>>> "a" in "b", "a"
(False, 'a')
```

Since the comma is not an operator, but a separator between expressions the above is evaluated as if you had entered:

```
("a" in "b"), "a"
```

not:

```
"a" in ("b", "a")
```

The same is true of the various assignment operators (`=`, `+=` etc). They are not truly operators but syntactic delimiters in assignment statements.

2.2.16 Is there an equivalent of C's "?:" ternary operator?

Yes, there is. The syntax is as follows:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

Before this syntax was introduced in Python 2.5, a common idiom was to use logical operators:

```
[expression] and [on_true] or [on_false]
```

However, this idiom is unsafe, as it can give wrong results when `on_true` has a false boolean value. Therefore, it is always better to use the `... if ... else ...` form.

2.2.17 Is it possible to write obfuscated one-liners in Python?

Yes. Usually this is done by nesting `lambda` within `lambda`. See the following three examples, due to Ulf Bartelt:

```
from functools import reduce

# Primes < 1000
print(list(filter(None, map(lambda y: y*reduce(lambda x, y: x*y!=0,
map(lambda x, y: y% x, range(2, int(pow(y, 0.5)+1))), 1), range(2, 1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x, f: lambda x, f: (f(x-1, f)+f(x-2, f)) if x>1 else 1:
f(x, f), range(10))))

# Mandelbrot set
print((lambda Ru, Ro, Iu, Io, IM, Sx, Sy: reduce(lambda x, y: x+y, map(lambda y,
Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, Sy=Sy, L=lambda yc, Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, i=IM,
Sx=Sx, Sy=Sy: reduce(lambda x, y: x+y, map(lambda x, xc=Ru, yc=yc, Ru=Ru, Ro=Ro,
i=i, Sx=Sx, F=lambda xc, yc, x, y, k, f: lambda xc, yc, x, y, k, f: (k<=0) or (x*x+y*y
>=4.0) or 1+f(xc, yc, x*x-y*y+xc, 2.0*x*y+yc, k-1, f): f(xc, yc, x, y, k, f): chr(
64+F(Ru+x*(Ro-Ru)/Sx, yc, 0, 0, i)), range(Sx))), L(Iu+y*(Io-Iu)/Sy), range(Sy
)))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#      \___ ___/ \___ ___/ / / /___ lines on screen
#          V          V / /_____ columns on screen
#          /          / /_____ maximum of "iterations"
```

()

()

```
#      /      /_____ range on y axis
#      /_____ range on x axis
```

Don't try this at home, kids!

2.2.18 What does the slash(/) in the parameter list of a function mean?

A slash in the argument list of a function denotes that the parameters prior to it are positional-only. Positional-only parameters are the ones without an externally-usable name. Upon calling a function that accepts positional-only parameters, arguments are mapped to parameters based solely on their position. For example, `pow()` is a function that accepts positional-only parameters. Its documentation looks like this:

```
>>> help(pow)
Help on built-in function pow in module builtins:

pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three arguments)

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

The slash at the end of the parameter list means that all three parameters are positional-only. Thus, calling `pow()` with keyword arguments would lead to an error:

```
>>> pow(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pow() takes no keyword arguments
```

Note that as of this writing this is only documentation and no valid syntax in Python, although there is [PEP 570](#), which proposes a syntax for position-only parameters in Python.

2.3 Numbers and strings

2.3.1 How do I specify hexadecimal and octal integers?

To specify an octal digit, precede the octal value with a zero, and then a lower or uppercase "o". For example, to set the variable "a" to the octal value "10" (8 in decimal), type:

```
>>> a = 0o10
>>> a
8
```

Hexadecimal is just as easy. Simply precede the hexadecimal number with a zero, and then a lower or uppercase "x". Hexadecimal digits can be specified in lower or uppercase. For example, in the Python interpreter:

```
>>> a = 0xa5
>>> a
165
```

()

()

```
>>> b = 0XB2
>>> b
178
```

2.3.2 Why does `-22 // 10` return `-3`?

It's primarily driven by the desire that `i % j` have the same sign as `j`. If you want that, and also want:

```
i == (i // j) * j + (i % j)
```

then integer division has to return the floor. C also requires that identity to hold, and then compilers that truncate `i // j` need to make `i % j` have the same sign as `i`.

There are few real use cases for `i % j` when `j` is negative. When `j` is positive, there are many, and in virtually all of them it's more useful for `i % j` to be ≥ 0 . If the clock says 10 now, what did it say 200 hours ago? `-190 % 12 == 2` is useful; `-190 % 12 == -10` is a bug waiting to bite.

2.3.3 How do I convert a string to a number?

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to floating-point, e.g. `float('144') == 144.0`.

By default, these interpret the number as decimal, so that `int('0144') == 144` and `int('0x144')` raises `ValueError`. `int(string, base)` takes the base to convert from as a second optional argument, so `int('0x144', 16) == 324`. If the base is specified as 0, the number is interpreted using Python's rules: a leading '0o' indicates octal, and '0x' indicates a hex number.

Do not use the built-in function `eval()` if all you need is to convert strings to numbers. `eval()` will be significantly slower and it presents a security risk: someone could pass you a Python expression that might have unwanted side effects. For example, someone could pass `__import__('os').system("rm -rf $HOME")` which would erase your home directory.

`eval()` also has the effect of interpreting numbers as Python expressions, so that e.g. `eval('09')` gives a syntax error because Python does not allow leading '0' in a decimal number (except '0').

2.3.4 How do I convert a number to a string?

To convert, e.g., the number 144 to the string '144', use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the f-strings and formatstrings sections, e.g. `"{:04d}".format(144)` yields '0144' and `"{:.3f}".format(1.0/3.0)` yields '0.333'.

2.3.5 How do I modify a string in place?

You can't, because strings are immutable. In most situations, you should simply construct a new string from the various parts you want to assemble it from. However, if you need an object with the ability to modify in-place unicode data, try using an `io.StringIO` object or the `array` module:

```

>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('u', s)
>>> print(a)
array('u', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'

```

2.3.6 How do I use strings to call functions/methods?

There are various techniques.

- The best is to use a dictionary that maps strings to functions. The primary advantage of this technique is that the strings do not need to match the names of the functions. This is also the primary technique used to emulate a case construct:

```

def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function

```

- Use the built-in function `getattr()`:

```

import foo
getattr(foo, 'bar')()

```

Note that `getattr()` works on any object, including classes, class instances, modules, and so on.

This is used in several places in the standard library, like this:

```

class Foo:
    def do_foo(self):
        ...

```

()

```
def do_bar(self):  
    ...  
  
f = getattr(foo_instance, 'do_' + opname)  
f()
```

- Use `locals()` or `eval()` to resolve the function name:

```
def myFunc():  
    print("hello")  
  
fname = "myFunc"  
  
f = locals()[fname]  
f()  
  
f = eval(fname)  
f()
```

Note: Using `eval()` is slow and dangerous. If you don't have absolute control over the contents of the string, someone could pass a string that resulted in an arbitrary function being executed.

2.3.7 Is there an equivalent to Perl's `chomp()` for removing trailing newlines from strings?

You can use `S.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `S` without removing other trailing whitespace. If the string `S` represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed:

```
>>> lines = ("line 1 \r\n"  
...         "\r\n"  
...         "\r\n")  
>>> lines.rstrip("\n\r")  
'line 1 '
```

Since this is typically only desired when reading text one line at a time, using `S.rstrip()` this way works well.

2.3.8 Is there a `scanf()` or `sscanf()` equivalent?

Not as such.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional "sep" parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C's `sscanf()` and better suited for the task.

2.3.9 What does 'UnicodeDecodeError' or 'UnicodeEncodeError' error mean?

See the [unicode-howto](#).

2.4 Performance

2.4.1 My program is too slow. How do I speed it up?

That's a tough one, in general. First, here are a list of things to remember before diving further:

- Performance characteristics vary across Python implementations. This FAQ focusses on *CPython*.
- Behaviour can vary across operating systems, especially when talking about I/O or multi-threading.
- You should always find the hot spots in your program *before* attempting to optimize any code (see the `profile` module).
- Writing benchmark scripts will allow you to iterate quickly when searching for improvements (see the `timeit` module).
- It is highly recommended to have good code coverage (through unit testing or any other technique) before potentially introducing regressions hidden in sophisticated optimizations.

That being said, there are many tricks to speed up Python code. Here are some general principles which go a long way towards reaching acceptable performance levels:

- Making your algorithms faster (or changing to faster ones) can yield much larger benefits than trying to sprinkle micro-optimization tricks all over your code.
- Use the right data structures. Study documentation for the builtin-types and the `collections` module.
- When the standard library provides a primitive for doing something, it is likely (although not guaranteed) to be faster than any alternative you may come up with. This is doubly true for primitives written in C, such as builtins and some extension types. For example, be sure to use either the `list.sort()` built-in method or the related `sorted()` function to do sorting (and see the [sortinghowto](#) for examples of moderately advanced usage).
- Abstractions tend to create indirections and force the interpreter to work more. If the levels of indirection outweigh the amount of useful work done, your program will be slower. You should avoid excessive abstraction, especially under the form of tiny functions or methods (which are also often detrimental to readability).

If you have reached the limit of what pure Python can allow, there are tools to take you further away. For example, [Cython](#) can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms. Cython can take advantage of compilation (and optional type annotations) to make your code significantly faster than when interpreted. If you are confident in your C programming skills, you can also write a C extension module yourself.

:

The wiki page devoted to [performance tips](#).

2.4.2 What is the most efficient way to concatenate many strings together?

`str` and `bytes` objects are immutable, therefore concatenating many strings together is inefficient as each concatenation creates a new object. In the general case, the total runtime cost is quadratic in the total string length.

To accumulate many `str` objects, the recommended idiom is to place them into a list and call `str.join()` at the end:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(another reasonably efficient idiom is to use `io.StringIO`)

To accumulate many `bytes` objects, the recommended idiom is to extend a `bytearray` object using in-place concatenation (the `+=` operator):

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

2.5 Sequences (Tuples/Lists)

2.5.1 How do I convert between tuples and lists?

The type constructor `tuple(seq)` converts any sequence (actually, any iterable) into a tuple with the same items in the same order.

For example, `tuple([1, 2, 3])` yields `(1, 2, 3)` and `tuple('abc')` yields `('a', 'b', 'c')`. If the argument is a tuple, it does not make a copy but returns the same object, so it is cheap to call `tuple()` when you aren't sure that an object is already a tuple.

The type constructor `list(seq)` converts any sequence or iterable into a list with the same items in the same order. For example, `list((1, 2, 3))` yields `[1, 2, 3]` and `list('abc')` yields `['a', 'b', 'c']`. If the argument is a list, it makes a copy just like `seq[:]` would.

2.5.2 What's a negative index?

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `S[:-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.

2.5.3 How do I iterate over a sequence in reverse order?

Use the `reversed()` built-in function, which is new in Python 2.4:

```
for x in reversed(sequence):
    ... # do something with x ...
```

This won't touch your original sequence, but build a new copy with reversed order to iterate over.

With Python 2.3, you can use an extended slice syntax:

```
for x in sequence[::-1]:
    ... # do something with x ...
```

2.5.4 How do you remove duplicates from a list?

See the Python Cookbook for a long discussion of many ways to do this:

<https://code.activestate.com/recipes/52560/>

If you don't mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

If all elements of the list may be used as set keys (i.e. they are all *hashable*) this is often faster

```
mylist = list(set(mylist))
```

This converts the list into a set, thereby removing duplicates, and then back into a list.

2.5.5 How do you make an array in Python?

Use a list:

```
["this", 1, "is", "an", "array"]
```

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that the Numeric extensions and others define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate cons cells using tuples:

```
lisp_list = ("like", ("this", ("example", None) ) )
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of lisp car is `lisp_list[0]` and the analogue of cdr is `lisp_list[1]`. Only do this if you're sure you really need to, because it's usually a lot slower than using Python lists.

2.5.6 How do I create a multidimensional list?

You probably tried to make a multidimensional array like this:

```
>>> A = [[None] * 2] * 3
```

This looks correct if you print it:

```
>>> A
[[None, None], [None, None], [None, None]]
```

But when you assign a value, it shows up in multiple places:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

The reason is that replicating a list with `*` doesn't create copies, it only creates references to the existing objects. The `*3` creates a list containing 3 references to the same list of length two. Changes to one row will show in all rows, which is almost certainly not what you want.

The suggested approach is to create a list of the desired length first and then fill in each element with a newly created list:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

This generates a list containing 3 different lists of length two. You can also use a list comprehension:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Or, you can use an extension that provides a matrix datatype; `NumPy` is the best known.

2.5.7 How do I apply a method to a sequence of objects?

Use a list comprehension:

```
result = [obj.method() for obj in mylist]
```

2.5.8 Why does `a_tuple[i] += ['item']` raise an exception when the addition works?

This is because of a combination of the fact that augmented assignment operators are *assignment* operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we'll use a `list` and `+=` as our exemplar.

If you wrote:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```


The reason for the exception should be immediately clear: 1 is added to the object `a_tuple[0]` points to (1), producing the result object, 2, but when we attempt to assign the result of the computation, 2, to element 0 of the tuple, we get an error because we can't change what an element of a tuple points to.

Under the covers, what this augmented assignment statement is doing is approximately this:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

It is the assignment part of the operation that produces the error, since a tuple is immutable.

When you write something like:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, `__iadd__` is equivalent to calling `extend` on the list and returning the list. That's why we say that for lists, `+=` is a "shorthand" for `list.extend`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

This is equivalent to:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

The object pointed to by `a_list` has been mutated, and the pointer to the mutated object is assigned back to `a_list`. The end result of the assignment is a no-op, since it is a pointer to the same object that `a_list` was previously pointing to, but the assignment still happens.

Thus, in our tuple example what is happening is equivalent to:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

The `__iadd__` succeeds, and thus the list is extended, but even though `result` points to the same object that `a_tuple[0]` already points to, that final assignment still results in an error, because tuples are immutable.

2.5.9 I want to do a complicated sort: can you do a Schwartzian Transform in Python?

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its "sort value". In Python, use the `key` argument for the `list.sort()` method:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

2.5.10 How can I sort one list by values from another list?

Merge them into an iterator of tuples, sort the resulting list, and then pick out the element you want.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[('I'm', 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

An alternative for the last step is:

```
>>> result = []
>>> for p in pairs: result.append(p[1])
```

If you find this more legible, you might prefer to use this instead of the final list comprehension. However, it is almost twice as slow for long lists. Why? First, the `append()` operation has to reallocate memory, and while it uses some tricks to avoid doing that each time, it still has to do it occasionally, and that costs quite a bit. Second, the expression "result.append" requires an extra attribute lookup, and third, there's a speed reduction from having to make all those function calls.

2.6 Objects

2.6.1 What is a class?

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic `Mailbox` class that provides basic accessor methods for a mailbox, and subclasses such as `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` that handle various specific mailbox formats.

2.6.2 What is a method?

A method is a function on some object `x` that you normally call as `x.name(arguments...)`. Methods are defined as functions inside the class definition:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

2.6.3 What is self?

Self is merely a conventional name for the first argument of a method. A method defined as `meth(self, a, b, c)` should be called as `x.meth(a, b, c)` for some instance `x` of the class in which the definition occurs; the called method will think it is called as `meth(x, a, b, c)`.

See also *Why must 'self' be used explicitly in method definitions and calls?*.

2.6.4 How do I check if an object is an instance of a given class or of a subclass of it?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, float, complex))`.

Note that most programs do not use `isinstance()` on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object's class and doing a different thing based on what class it is. For example, if you have a function that does something:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ... # code to search a mailbox
    elif isinstance(obj, Document):
        ... # code to search a document
    elif ...
```

A better approach is to define a `search()` method on all the classes and just call it:

```
class Mailbox:
    def search(self):
        ... # code to search a mailbox

class Document:
    def search(self):
        ... # code to search a document

obj.search()
```

2.6.5 What is delegation?

Delegation is an object oriented technique (also called a design pattern). Let's say you have an object `x` and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of `x`.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self._outfile.write()` method. All other methods are delegated to the underlying `self._outfile` object. The delegation is accomplished via the `__getattr__` method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a `__setattr__()` method too, and it must do so carefully. The basic implementation of `__setattr__()` is roughly equivalent to the following:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Most `__setattr__()` implementations must modify `self.__dict__` to store local state for `self` without causing an infinite recursion.

2.6.6 How do I call a method defined in a base class from a derived class that overrides it?

Use the built-in `super()` function:

```
class Derived(Base):
    def meth(self):
        super(Derived, self).meth()
```

For version prior to 3.0, you may be using classic classes: For a class definition such as `class Derived(Base): ...` you can call method `meth()` defined in `Base` (or one of `Base`'s base classes) as `Base.meth(self, arguments...)`. Here, `Base.meth` is an unbound method, so you need to provide the `self` argument.

2.6.7 How can I organize my code to make it easier to change the base class?

You could define an alias for the base class, assign the real base class to it before your class definition, and use the alias throughout your class. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

```
BaseAlias = <real base class>
```

()

()

```
class Derived(BaseAlias):
    def meth(self):
        BaseAlias.meth(self)
    ...
```

2.6.8 How do I create static class data and static class methods?

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value to the attribute, you have to explicitly use the class name in the assignment:

```
class C:
    count = 0    # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count    # or return self.count
```

`c.count` also refers to `C.count` for any `c` such that `isinstance(c, C)` holds, unless overridden by `c` itself or by some class on the base-class search path from `c.__class__` back to `C`.

Caution: within a method of `C`, an assignment like `self.count = 42` creates a new and unrelated instance named "count" in `self`'s own dict. Rebinding of a class-static data name must always specify the class whether inside a method or not:

```
C.count = 314
```

Static methods are possible:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
    ...
```

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:

```
def getcount():
    return C.count
```

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

2.6.9 How can I overload constructors (or methods) in Python?

This answer actually applies to all methods, but the question usually comes up first in the context of constructors.

In C++ you'd write

```
class C {  
    C() { cout << "No arguments\n"; }  
    C(int i) { cout << "Argument is " << i << "\n"; }  
}
```

In Python you have to write a single constructor that catches all cases using default arguments. For example:

```
class C:  
    def __init__(self, i=None):  
        if i is None:  
            print("No arguments")  
        else:  
            print("Argument is", i)
```

This is not entirely equivalent, but close enough in practice.

You could also try a variable-length argument list, e.g.

```
def __init__(self, *args):  
    ...
```

The same approach works for all method definitions.

2.6.10 I try to use `__spam` and I get an error about `__SomeClassName__spam`.

Variable names with double leading underscores are "mangled" to provide a simple but effective way to define class private variables. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `__classname__spam`, where `classname` is the current class name with any leading underscores stripped.

This doesn't guarantee privacy: an outside user can still deliberately access the `"__classname__spam"` attribute, and private values are visible in the object's `__dict__`. Many Python programmers never bother to use private variable names at all.

2.6.11 My class defines `__del__` but it is not called when I delete the object.

There are several possible reasons for this.

The `del` statement does not necessarily call `__del__()` – it simply decrements the object's reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

Despite the cycle collector, it's still a good idea to define an explicit `close()` method on objects to be called whenever you're done with them. The `close()` method can then remove attributes that refer to subobjects. Don't call `__del__()` directly – `__del__()` should call `close()` and `close()` should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the `weakref` module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need them!).

Finally, if your `__del__()` method raises an exception, a warning message is printed to `sys.stderr`.

2.6.12 How do I get a list of all instances of a given class?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class's constructor to keep track of all instances by keeping a list of weak references to each instance.

2.6.13 Why does the result of `id()` appear to be not unique?

The `id()` builtin returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object's memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```
>>> id(1000) # doctest: +SKIP
13901272
>>> id(2000) # doctest: +SKIP
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately after execution of the `id()` call. To be sure that objects whose id you want to examine are still alive, create another reference to the object:

```
>>> a = 1000; b = 2000
>>> id(a) # doctest: +SKIP
13901272
>>> id(b) # doctest: +SKIP
13891296
```

2.7

2.7.1 How do I create a .pyc file?

When a module is imported for the first time (or when the source file has changed since the current compiled file was created) a `.pyc` file containing the compiled code should be created in a `__pycache__` subdirectory of the directory containing the `.py` file. The `.pyc` file will have a filename that starts with the same name as the `.py` file, and ends with `.pyc`, with a middle component that depends on the particular `python` binary that created it. (See [PEP 3147](#) for details.)

One reason that a `.pyc` file may not be created is a permissions problem with the directory containing the source file, meaning that the `__pycache__` subdirectory cannot be created. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server.

Unless the `PYTHONDONTWRITEBYTECODE` environment variable is set, creation of a `.pyc` file is automatic if you're importing a module and Python has the ability (permissions, free space, etc...) to create a `__pycache__` subdirectory and write the compiled module to that subdirectory.

Running Python on a top level script is not considered an import and no `.pyc` will be created. For example, if you have a top-level module `foo.py` that imports another module `xyz.py`, when you run `foo` (by typing `python foo.py` as a shell command), a `.pyc` will be created for `xyz` because `xyz` is imported, but no `.pyc` file will be created for `foo` since `foo.py` isn't being imported.

If you need to create a `.pyc` file for `foo` – that is, to create a `.pyc` file for a module that is not imported – you can, using the `py_compile` and `compileall` modules.

The `py_compile` module can manually compile any module. One way is to use the `compile()` function in that module interactively:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

This will write the `.pyc` to a `__pycache__` subdirectory in the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

You can also automatically compile all files in a directory or directories using the `compileall` module. You can do it from the shell prompt by running `compileall.py` and providing the path of a directory containing Python files to compile:

```
python -m compileall .
```

2.7.2 How do I find the current module name?

A module can find out its own module name by looking at the predefined global variable `__name__`. If this has the value `'__main__'`, the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking `__name__`:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

2.7.3 How can I have modules that mutually import each other?

Suppose you have the following modules:

`foo.py`:

```
from bar import bar_var
foo_var = 1
```

`bar.py`:

```
from foo import foo_var
bar_var = 2
```

The problem is that the interpreter will perform the following steps:

- main imports foo

- Empty globals for `foo` are created
- `foo` is compiled and starts executing
- `foo` imports `bar`
- Empty globals for `bar` are created
- `bar` is compiled and starts executing
- `bar` imports `foo` (which is a no-op since there already is a module named `foo`)
- `bar.foo_var = foo.foo_var`

The last step fails, because Python isn't done with interpreting `foo` yet and the global symbol dictionary for `foo` is still empty.

The same thing happens when you use `import foo`, and then try to access `foo.foo_var` in global code.

There are (at least) three possible workarounds for this problem.

Guido van Rossum recommends avoiding all uses of `from <module> import ...`, and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as `<module>.<name>`.

Jim Roskind suggests performing steps in the following order in each module:

- exports (globals, functions, and classes that don't need imported base classes)
- `import` statements
- active code (including globals that are initialized from imported values).

van Rossum doesn't like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs recommends restructuring your code so that the recursive import is not necessary in the first place.

These solutions are not mutually exclusive.

2.7.4 `__import__('x.y.z')` returns `<module 'x'>`; how do I get `z`?

Consider using the convenience function `import_module()` from `importlib` instead:

```
z = importlib.import_module('x.y.z')
```

2.7.5 When I edit an imported module and reimport it, the changes don't show up. Why does this happen?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force re-reading of a changed module, do this:

```
import importlib
import modname
importlib.reload(modname)
```

Warning: this technique is not 100% fool-proof. In particular, modules containing statements like

```
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will *not* be updated to use the new class definition. This can result in the following paradoxical behaviour:

```
>>> import importlib
>>> import cls
>>> c = cls.C()           # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)  # isinstance is false!?!
False
```

The nature of the problem is made clear if you print out the "identity" of the class objects:

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

3.1 Python

Guido van Rossum Python

/ C :

```
if (x <= y)
    x++;
    y--;
z++;
```

x++ C x > y y

/ Python C

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.

3.2 Why am I getting strange results with simple arithmetic operations?

See the next question.

3.3 Why are floating-point calculations so inaccurate?

Users are often surprised by results like this:

```
>>> 1.2 - 1.0
0.19999999999999996
```

and think it is a bug in Python. It's not. This has little to do with Python, and much more to do with how the underlying platform handles floating-point numbers.

The `float` type in CPython uses a C `double` for storage. A `float` object's value is stored in binary floating-point with a fixed precision (typically 53 bits) and Python uses C operations, which in turn rely on the hardware implementation in the processor, to perform floating-point operations. This means that as far as floating-point operations are concerned, Python behaves like many popular languages including C and Java.

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating-point. For example, after:

```
>>> x = 1.2
```

the value stored for `x` is a (very good) approximation to the decimal value 1.2, but is not exactly equal to it. On a typical machine, the actual stored value is:

```
1.0011001100110011001100110011001100110011001100110011001100110011 (binary)
```

which is exactly:

```
1.199999999999999555910790149937383830547332763671875 (decimal)
```

The typical precision of 53 bits provides Python floats with 15–16 decimal digits of accuracy.

For a fuller explanation, please see the floating point arithmetic chapter in the Python tutorial.

3.4 Why are Python strings immutable?

There are several advantages.

One is performance: knowing that a string is immutable means we can allocate space for it at creation time, and the storage requirements are fixed and unchanging. This is also one of the reasons for the distinction between tuples and lists.

Another advantage is that strings in Python are considered as "elemental" as numbers. No amount of activity will change the value 8 to anything else, and in Python, no amount of activity will change the string "eight" to anything else.

3.5 Why must 'self' be used explicitly in method definitions and calls?

The idea was borrowed from Modula-3. It turns out to be very useful, for a variety of reasons.

First, it's more obvious that you are using a method or instance attribute instead of a local variable. Reading `self.x` or `self.meth()` makes it absolutely clear that an instance variable or method is used even if you don't know the class definition by heart. In C++, you can sort of tell by the lack of a local variable declaration (assuming globals are rare or easily recognizable) – but in Python, there are no local variable declarations, so you'd have to look up the class definition to be sure. Some C++ and Java coding standards call for instance attributes to have an `m_` prefix, so this explicitness is still useful in those languages, too.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a

derived class, you have to use the `::` operator – in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

Finally, for instance variables it solves a syntactic problem with assignment: since local variables in Python are (by definition!) those variables to which a value is assigned in a function body (and that aren't explicitly declared global), there has to be some way to tell the interpreter that an assignment was meant to assign to an instance variable instead of to a local variable, and it should preferably be syntactic (for efficiency reasons). C++ does this through declarations, but Python doesn't have declarations and it would be a pity having to introduce them just for this purpose. Using the explicit `self.var` solves this nicely. Similarly, for using instance variables, having to write `self.var` means that references to unqualified names inside a method don't have to search the instance's directories. To put it another way, local variables and instance variables live in two different namespaces, and you need to tell Python which namespace to use.

3.6 Why can't I use an assignment in an expression?

Many people used to C or Perl complain that they want to use this C idiom:

```
while (line = readline(f)) {
    // do something with line
}
```

where in Python you're forced to write this:

```
while True:
    line = f.readline()
    if not line:
        break
    ... # do something with line
```

The reason for not allowing assignment in Python expressions is a common, hard-to-find bug in those other languages, caused by this construct:

```
if (x = 0) {
    // error handling
}
else {
    // code that only works for nonzero x
}
```

The error is a simple typo: `x = 0`, which assigns 0 to the variable `x`, was written while the comparison `x == 0` is certainly what was intended.

Many alternatives have been proposed. Most are hacks that save some typing but use arbitrary or cryptic syntax or keywords, and fail the simple criterion for language change proposals: it should intuitively suggest the proper meaning to a human reader who has not yet been introduced to the construct.

An interesting phenomenon is that most experienced Python programmers recognize the `while True` idiom and don't seem to be missing the assignment in expression construct much; it's only newcomers who express a strong desire to add this to the language.

There's an alternative way of spelling this that seems attractive but is generally less robust than the "while True" solution:

```
line = f.readline()
while line:
    ... # do something with line...
    line = f.readline()
```

The problem with this is that if you change your mind about exactly how you get the next line (e.g. you want to change it into `sys.stdin.readline()`) you have to remember to change two places in your program – the second occurrence is hidden at the bottom of the loop.

The best approach is to use iterators, making it possible to loop through objects using the `for` statement. For example, *file objects* support the iterator protocol, so you can write simply:

```
for line in f:
    ... # do something with line...
```

3.7 Why does Python use methods for some functionality (e.g. `list.index()`) but functions for other (e.g. `len(list)`)?

As Guido said:

- (a) For some operations, prefix notation just reads better than postfix – prefix (and infix!) operations have a long tradition in mathematics which likes notations where the visuals help the mathematician thinking about a problem. Compare the ease with which we rewrite a formula like $x*(a+b)$ into $x*a + x*b$ to the clumsiness of doing the same thing using a raw OO notation.
- (b) When I read code that says `len(x)` I *know* that it is asking for the length of something. This tells me two things: the result is an integer, and the argument is some kind of container. To the contrary, when I read `x.len()`, I have to already know that `x` is some kind of container implementing an interface or inheriting from a class that has a standard `len()`. Witness the confusion we occasionally have when a class that is not implementing a mapping has a `get()` or `keys()` method, or something that isn't a file has a `write()` method.

—<https://mail.python.org/pipermail/python-3000/2006-November/004643.html>

3.8 Why is `join()` a string method instead of a list or tuple method?

Strings became much more like other standard types starting in Python 1.6, when methods were added which give the same functionality that has always been available using the functions of the string module. Most of these new methods have been widely accepted, but the one which appears to make some programmers feel uncomfortable is:

```
", ".join(['1', '2', '4', '8', '16'])
```

which gives the result:

```
"1, 2, 4, 8, 16"
```

There are two common arguments against this usage.

The first runs along the lines of: "It looks really ugly using a method of a string literal (string constant)", to which the answer is that it might, but a string literal is just a fixed value. If the methods are to be allowed on names bound to strings there is no logical reason to make them unavailable on literals.

The second objection is typically cast as: "I am really telling a sequence to join its members together with a string constant". Sadly, you aren't. For some reason there seems to be much less difficulty with having `split()` as a string method, since in that case it is easy to see that

```
"1, 2, 4, 8, 16".split(", ")
```

is an instruction to a string literal to return the substrings delimited by the given separator (or, by default, arbitrary runs of white space).

`join()` is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself. Similar methods exist for bytes and bytearray objects.

3.9 How fast are exceptions?

A try/except block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

This only made sense when you expected the dict to have the key almost all the time. If that wasn't the case, you coded it like this:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

For this specific case, you could also use `value = dict.setdefault(key, getvalue(key))`, but only if the `getvalue()` call is cheap enough because it is evaluated in all cases.

3.10 Why isn't there a switch or case statement in Python?

You can do this easily enough with a sequence of `if... elif... elif... else`. There have been some proposals for switch statement syntax, but there is no consensus (yet) on whether and how to do range tests. See [PEP 275](#) for complete details and the current status.

For cases where you need to choose from a very large number of possibilities, you can create a dictionary mapping case values to functions to call. For example:

```
def function_1(...):
    ...

functions = {'a': function_1,
            'b': function_2,
            'c': self.method_1, ...}
```

()

()

```
func = functions[value]
func()
```

For calling methods on objects, you can simplify yet further by using the `getattr()` built-in to retrieve methods with a particular name:

```
def visit_a(self, ...):
    ...
    ...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
    method()
```

It's suggested that you use a prefix for the method names, such as `visit_` in this example. Without such a prefix, if values are coming from an untrusted source, an attacker would be able to call any method on your object.

3.11 Can't you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?

Answer 1: Unfortunately, the interpreter pushes at least one C stack frame for each Python stack frame. Also, extensions can call back into Python at almost random moments. Therefore, a complete threads implementation requires thread support for C.

Answer 2: Fortunately, there is [Stackless Python](#), which has a completely redesigned interpreter loop that avoids the C stack.

3.12 Why can't lambda expressions contain statements?

Python lambda expressions cannot contain statements because Python's syntactic framework can't handle statements nested inside expressions. However, in Python, this is not a serious problem. Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you're too lazy to define a function.

Functions are already first class objects in Python, and can be declared in a local scope. Therefore the only advantage of using a lambda instead of a locally-defined function is that you don't need to invent a name for the function – but that's just a local variable to which the function object (which is exactly the same type of object that a lambda expression yields) is assigned!

3.13 Can Python be compiled to machine code, C or some other language?

[Cython](#) compiles a modified version of Python with optional annotations into C extensions. [Nuitka](#) is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language. For compiling to Java you can consider [VOC](#).

3.14 How does Python manage memory?

The details of Python memory management depend on the implementation. The standard implementation of Python, *CPython*, uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. The `gc` module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Other implementations (such as *Jython* or *PyPy*), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

In some Python implementations, the following code (which is fine in CPython) will probably run out of file descriptors:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Indeed, using CPython's reference counting and destructor scheme, each new assignment to `f` closes the previous file. With a traditional GC, however, those file objects will only get collected (and closed) at varying and possibly long intervals.

If you want to write code that will work with any Python implementation, you should explicitly close the file or use the `with` statement; this will work regardless of memory management scheme:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

3.15 Why doesn't CPython use a more traditional garbage collection scheme?

For one thing, this is not a C standard feature and hence it's not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for *most* common platforms, not for all of them, and although it is mostly transparent, it isn't completely transparent; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard `malloc()` and `free()` with versions provided by the GC library, an application embedding Python may want to have its *own* substitute for `malloc()` and `free()`, and may not want Python's. Right now, CPython works with anything that implements `malloc()` and `free()` properly.

3.16 Why isn't all memory freed when CPython exits?

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation use the `atexit` module to run a function that will force those deletions.

3.17 Why are there separate tuple and list data types?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal records or C structs; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Tuples are immutable, meaning that once a tuple has been created, you can't replace any of its elements with a new value. Lists are mutable, meaning that you can always change a list's elements. Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys.

3.18 How are lists implemented in CPython?

CPython's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list `a[i]` an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

3.19 How are dictionaries implemented in CPython?

CPython's dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key and a per-process seed; for example, "Python" could hash to -539294296 while "python", a string that differs by a single bit, could hash to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time – $O(1)$, in Big-O notation – to retrieve a key.

3.20 Why must dictionary keys be immutable?

The hash table implementation of dictionaries uses a hash value calculated from the key value to find the key. If the key were a mutable object, its value could change, and thus its hash could also change. But since whoever changes the key object can't tell that it was being used as a dictionary key, it can't move the entry around in the dictionary. Then, when you try to look up the same object in the dictionary it won't be found because its hash value is different. If you tried to look up the old value it wouldn't be found either, because the value of the object found in that hash bin would be different.

If you want a dictionary indexed with a list, simply convert the list to a tuple first; the function `tuple(L)` creates a tuple with the same entries as the list `L`. Tuples are immutable and can therefore be used as dictionary keys.

Some unacceptable solutions that have been proposed:

- Hash lists by their address (object ID). This doesn't work because if you construct a new list with the same value it won't be found; e.g.:

```
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

would raise a `KeyError` exception because the id of the `[1, 2]` used in the second line differs from that in the first line. In other words, dictionary keys should be compared using `==`, not using `is`.

- Make a copy when using a list as a key. This doesn't work because the list, being a mutable object, could contain a reference to itself, and then the copying code would run into an infinite loop.
- Allow lists as keys but tell the user not to modify them. This would allow a class of hard-to-track bugs in programs when you forgot or modified a list by accident. It also invalidates an important invariant of dictionaries: every value in `d.keys()` is usable as a key of the dictionary.
- Mark lists as read-only once they are used as a dictionary key. The problem is that it's not just the top-level object that could change its value; you could use a tuple containing a list as a key. Entering anything as a key into a dictionary would require marking all objects reachable from there as read-only – and again, self-referential objects could cause an infinite loop.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Note that the hash computation is complicated by the possibility that some members of the list may be unhashable and also by the possibility of arithmetic overflow.

Furthermore it must always be the case that if `o1 == o2` (ie `o1.__eq__(o2)` is `True`) then `hash(o1) == hash(o2)` (ie, `o1.__hash__() == o2.__hash__()`), regardless of whether the object is in a dictionary or not. If you fail to meet these restrictions dictionaries and other hash based structures will misbehave.

In the case of `ListWrapper`, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the

consequences of not meeting them correctly. Consider yourself warned.

3.21 Why doesn't `list.sort()` return the sorted list?

In situations where performance matters, making a copy of the list just to sort it would be wasteful. Therefore, `list.sort()` sorts the list in place. In order to remind you of that fact, it does not return the sorted list. This way, you won't be fooled into accidentally overwriting a list when you need a sorted copy but also need to keep the unsorted version around.

If you want to return a new list, use the built-in `sorted()` function instead. This function creates a new list from a provided iterable, sorts it and returns it. For example, here's how to iterate over the keys of a dictionary in sorted order:

```
for key in sorted(mydict):  
    ... # do whatever with mydict[key]...
```

3.22 How do you specify and enforce an interface spec in Python?

An interface specification for a module as provided by languages such as C++ and Java describes the prototypes for the methods and functions of the module. Many feel that compile-time enforcement of interface specifications helps in the construction of large programs.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections.abc` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components. There is also a tool, PyChecker, which can be used to find problems due to subclassing.

A good test suite for a module can both provide a regression test and serve as a module interface specification and a set of examples. Many Python modules can be run as a script to provide a simple "self test." Even modules which use complex external interfaces can often be tested in isolation using trivial "stub" emulations of the external interface. The `doctest` and `unittest` modules or third-party test frameworks can be used to construct exhaustive test suites that exercise every line of code in a module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `append()` method is expected to add new elements to the end of some internal list; an interface specification cannot test that your `append()` implementation will actually do this correctly, but it's trivial to check this property in a test suite.

Writing test suites is very helpful, and you might want to design your code with an eye to making it easily tested. One increasingly popular technique, test-directed development, calls for writing parts of the test suite first, before you write any of the actual code. Of course Python allows you to be sloppy and not write test cases at all.

3.23 Why is there no `goto`?

You can use exceptions to provide a "structured `goto`" that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the "go" or "goto" constructs of C, Fortran, and other languages. For example:

```

class label(Exception): pass # declare a label

try:
    ...
    if condition: raise label() # goto label
    ...
except label: # where to goto
    pass
...

```

This doesn't allow you to jump into the middle of a loop, but that's usually considered an abuse of goto anyway. Use sparingly.

3.24 Why can't raw strings (r-strings) end with a backslash?

More precisely, they can't end with an odd number of backslashes: the unpaired backslash at the end escapes the closing quote character, leaving an unterminated string.

Raw strings were designed to ease creating input for processors (chiefly regular expression engines) that want to do their own backslash escape processing. Such processors consider an unmatched trailing backslash to be an error anyway, so raw strings disallow that. In return, they allow you to pass on the string quote character by escaping it with a backslash. These rules work well when r-strings are used for their intended purpose.

If you're trying to build Windows pathnames, note that all Windows system calls accept forward slashes too:

```
f = open("/mydir/file.txt") # works fine!
```

If you're trying to build a pathname for a DOS command, try e.g. one of

```

dir = r"\\this\\is\\my\\dos\\dir" "\\\"
dir = r"\\this\\is\\my\\dos\\dir\" "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"

```

3.25 Why doesn't Python have a "with" statement for attribute assignments?

Python has a 'with' statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some language have a construct that looks like this:

```

with obj:
    a = 1 # equivalent to obj.a = 1
    total = total + 1 # obj.total = obj.total + 1

```

In Python, such a construct would be ambiguous.

Other languages, such as Object Pascal, Delphi, and C++, use static types, so it's possible to know, in an unambiguous way, what member is being assigned to. This is the main point of static typing – the compiler *always* knows the scope of every variable at compile time.

Python uses dynamic types. It is impossible to know in advance which attribute will be referenced at runtime. Member attributes may be added or removed from objects on the fly. This makes it impossible

to know, from a simple reading, what attribute is being referenced: a local one, a global one, or a member attribute?

For instance, take the following incomplete snippet:

```
def foo(a):  
    with a:  
        print(x)
```

The snippet assumes that "a" must have a member attribute called "x". However, there is nothing in Python that tells the interpreter this. What should happen if "a" is, let us say, an integer? If there is a global variable named "x", will it be used inside the with block? As you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of "with" and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of:

```
function(args).mydict[index][index].a = 21  
function(args).mydict[index][index].b = 42  
function(args).mydict[index][index].c = 63
```

write this:

```
ref = function(args).mydict[index][index]  
ref.a = 21  
ref.b = 42  
ref.c = 63
```

This also has the side-effect of increasing execution speed because name bindings are resolved at run-time in Python, and the second version only needs to perform the resolution once.

3.26 Why are colons required for the if/while/def/class statements?

The colon is required primarily to enhance readability (one of the results of the experimental ABC language). Consider this:

```
if a == b  
    print(a)
```

versus

```
if a == b:  
    print(a)
```

Notice how the second one is slightly easier to read. Notice further how a colon sets off the example in this FAQ answer; it's a standard usage in English.

Another minor reason is that the colon makes it easier for editors with syntax highlighting; they can look for colons to decide when indentation needs to be increased instead of having to do a more elaborate parsing of the program text.

3.27 Why does Python allow commas at the end of lists and tuples?

Python lets you add a trailing comma at the end of lists, tuples, and dictionaries:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7], # last trailing comma is optional but good style
}
```

There are several reasons to allow this.

When you have a literal value for a list, tuple, or dictionary spread across multiple lines, it's easier to add more elements because you don't have to remember to add a comma to the previous line. The lines can also be reordered without creating a syntax error.

Accidentally omitting the comma can lead to errors that are hard to diagnose. For example:

```
x = [
    "fee",
    "fie"
    "foo",
    "fum"
]
```

This list looks like it has four elements, but it actually contains three: "fee", "fiefoo" and "fum". Always adding the comma avoids this source of error.

Allowing the trailing comma may also make programmatic code generation easier.

4.1

4.1.1 XXX

[Python Package Index](#) [Google](#) “Python”

4.1.2 `math.py` `socket.py` `regex.py`

C C++ `mathmodule.c` C Python

Python

- 1) Python `.py`)
- 2) C `.dll .pyd .so .sl`
- 3) C

```
import sys
print(sys.builtin_module_names)
```

4.1.3 Unix Python

`#!` Python

`chmod +x scriptfile` `chmod 755 scriptfile`

```
#!/usr/local/bin/python
```

Python
 Python env Python PATH Unix

```
#!/usr/bin/env python
```

CGI CGI PATH
 /usr/bin/env env hack Alex Rezinsky

```
#!/bin/sh
"""
exec python $0 ${1+"$@"}
"""
```

__doc__

```
__doc__ = "...Whatever..."
```

4.1.4 Python curses/termcap

Unix Python Modules curses Windows — Windows curses
 curses curses ncurses SYSV curses BSD curses
 Windows consolelib .

4.1.5 Python C onexit()

atexit C onexit()

4.1.6

```
handler(signum, frame)
```

```
def handler(signum, frame):
    ...
```

4.2

4.2.1 Python

Python doctest docstring docstring
 unittest Java Smalltalk

“ ”

```
if __name__ == "__main__":
    main_logic()
```

bug Python “ ”

```
if __name__ == "__main__":
    self_test()
```

Python “ ”

4.2.2 docstring

pydoc Python docstring HTML epydoc docstring API Sphinx docstring

4.2.3

Unix curses curses

4.3

4.3.1

threading _thread threading _thread
Aahz threading <http://www.pythoncraft.com/OSCON2001/>

4.3.2

sleep

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10) # <-----! 
```

sleep

```
def thread_task(name, n):
    time.sleep(0.001) # <-----!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

time.sleep()

queue

queue

append

queue

queue

4.3.3

concurrent.futures

ThreadPoolExecutor

queue

Queue

.put(obj)

.get()

```
import threading, queue, time

# The worker thread gets jobs off the queue. When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.currentThread(), end=' ')
            print('queue empty')
            break
        else:
            print('Worker', threading.currentThread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)
```

()

()

```
# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

Queue

4.3.4

Python VM	<i>global interpreter lock</i>	GIL	Python	<code>sys.setswitchinterval()</code>
Python		C		
PVM		int list dict	" "	
L L1 L2	D D1 D2	x y	i j	int

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

0 `__del__()`

4.3.5

global interpreter lock GIL Python Python GIL Python CPU
 Python 1.5 Greg Stein “free threadings” GIL Adam Olsen [python-safethread](#)
 GIL 30%
 Python * * * * concurrent.futures ProcessPoolExecutor
 multiprocessing API
 C C C GIL zlib hashlib
 GIL

4.4

4.4.1

.....
 os.remove(filename) os.unlink(filename) os.unlink() Unix
 os.rmdir() os.mkdir() os.makedirs(path) path os.removedirs(path)
 shutil.rmtree()
 os.rename(old_path, new_path)
 f = open(filename, "rb+") f.truncate(offset) offset os.open() os.
 ftruncate(fd, offset) fd
 shutil copyfile() copytree() rmtree()

4.4.2

shutil copyfile() MacOS 9 resource fork Finder info

4.4.3

struct Python
 2 4

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

'>' 'h' " " 2 'l' " " 4
 array

: open() "rb" "r" f.read() str bytes

4.4.4 `os.popen()` `os.read()`

`os.read()` — `os.popen()` `open()` `os.popen()` *p* *n*
`p.read(n)`

4.4.5 RS232

Win32 POSIX Linux BSD Jython

<http://pyserial.sourceforge.net>

Unix Mitch Chapman

<https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com>

4.4.6 `sys.stdout` `stdin` `stderr`

Python C

Python `open()` `f.close()` Python C `f`

`stdin` `stdout` `stderr` C Python `sys.stdout.close()` Python * *

C I/O `os.close()`

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

0 1 2

4.5 / Internet

4.5.1 Python WWW

internet netdata Python web

Paul Boddie <https://wiki.python.org/moin/WebProgramming>

Cameron Laird Python web http://phaseit.net/claird/comp.lang.python/web_python

4.5.2 CGI `METHOD=POST`

POST

`urllib.request`

```
#!/usr/local/bin/python

import urllib.request

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"
```

()

()

```
# connect and send the server a path
req = urllib.request.urlopen('http://www.some-server.out-there'
                             '/cgi-bin/some-cgi-script', data=qs)
with req:
    msg, hdrs = req.read(), req.info()
```

POST urllib.parse.urlencode() name=Guy Steele, Jr.

```
>>> import urllib.parse
>>> urllib.parse.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

:

urllib-howto

4.5.3 HTML

Web wiki

4.5.4 Python

smtplib

SMTP

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Unix sendmail sendmail /usr/lib/sendmail /usr/sbin/sendmail sendmail

```
import os

SENDMAIL = "/usr/sbin/sendmail" # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
```

()

()

```
p.write("Subject: test\n")
p.write("\n") # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

4.5.5 socket connect()

select	socket	I/O							
TCP	socket	connect()	.error	errno.EINPROGRESS					
connect_ex()		connect_ex()	— 0	errno.EISCONN	select				
socket									
: asyncore		Twisted							

4.6

4.6.1 Python

Python DBM GDBM sqlite3
wiki

4.6.2 Python

pickle shelve pickle (g)dbm Python

4.7

4.7.1 Python

```
random
import random
random.random()
```

[0, 1)

- randrange(a, b) [a, b)

- `uniform(a, b)` `[a, b)`
- `normalvariate(mean, sdev)`

- `choice(S)`
- `shuffle(L)`

Random

5.1 Can I create my own functions in C?

Yes, you can create built-in modules containing functions, variables, exceptions and even new types in C. This is explained in the document [extending-index](#).

Most intermediate or advanced Python books will also cover this topic.

5.2 Can I create my own functions in C++?

Yes, using the C compatibility features found in C++. Place `extern "C" { ... }` around the Python include files and put `extern "C"` before each function that is going to be called by the Python interpreter. Global or static C++ objects with constructors are probably not a good idea.

5.3 Writing C is hard; are there any alternatives?

There are a number of alternatives to writing your own C extensions, depending on what you're trying to do.

[Cython](#) and its relative [Pyrex](#) are compilers that accept a slightly modified form of Python and generate the corresponding C code. Cython and Pyrex make it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as [SWIG](#). [SIP](#), [CXX Boost](#), or [Weave](#) are also alternatives for wrapping C++ libraries.

5.4 How can I execute arbitrary Python statements from C?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns 0 for success and -1 when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`; see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`.

5.5 How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.

5.6 How do I extract C values from a Python object?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C's `strlen()` should not be used.

To test the type of an object, first make sure it isn't `NULL`, and then use `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called 'abstract' interface – read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc. as well as many other useful protocols such as numbers (`PyNumber_Index()` et al.) and mappings in the `PyMapping` APIs.

5.7 How do I use `Py_BuildValue()` to create a tuple of arbitrary length?

You can't. Use `PyTuple_Pack()` instead.

5.8 How do I call an object's method from C?

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

This works for any object that has methods – whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`'ing the return value.

To call, e.g., a file object's "seek" method with arguments 10, 0 (assuming the file object pointer is "f"):

```

res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
    ... an exception occurred ...
}
else {
    Py_DECREF(res);
}

```

Note that since `PyObject_CallObject()` *always* wants a tuple for the argument list, to call a function without arguments, pass `()` for the format, and to call a function with one argument, surround the argument in parentheses, e.g. `"(i)"`.

5.9 How do I catch the output from `PyErr_Print()` (or anything that prints to `stdout/stderr`)?

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call `print_error`, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

The easiest way to do this is to use the `io.StringIO` class:

```

>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!

```

A custom object to do the same would look like this:

```

>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!

```

5.10 How do I access a module written in Python from C?

You can get a pointer to the module object as follows:

```
module = PyImport_ImportModule("<modulename>");
```

If the module hasn't been imported yet (i.e. it is not yet present in `sys.modules`), this initializes the module; otherwise it simply returns the value of `sys.modules["<modulename>"]`. Note that it doesn't enter the module into any namespace – it only ensures it has been initialized and is stored in `sys.modules`.

You can then access the module's attributes (i.e. any name defined in the module) as follows:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

Calling `PyObject_SetAttrString()` to assign to variables in the module also works.

5.11 How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the "Extending and Embedding" document. Realize that for the Python run-time system, there isn't a whole lot of difference between C and C++ – so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

For C++ libraries, see *Writing C is hard; are there any alternatives?*.

5.12 I added a module using the Setup file and the make fails; why?

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn't seem worth the effort.)

5.13 How do I debug an extension?

When using GDB with dynamically loaded extensions, you can't set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

Then, when you run GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 I want to compile a Python module on my Linux system, but some files are missing. Why?

Most packaged versions of Python don't include the `/usr/lib/python2.x/config/` directory, which contains various files required for compiling Python extensions.

For Red Hat, install the `python-devel` RPM to get the necessary files.

For Debian, run `apt-get install python-dev`.

5.15 How do I tell "incomplete input" from "invalid input"?

Sometimes you want to emulate the Python interactive interpreter's behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an "if" statement or you didn't close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

In Python you can use the `codeop` module, which approximates the parser's behavior sufficiently. IDLE uses this, for example.

The easiest way to do it in C is to call `PyRun_InteractiveLoop()` (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the `PyOS_ReadlineFunctionPointer()` to point at your custom input function. See `Modules/readline.c` and `Parser/myreadline.c` for more hints.

However sometimes you have to run the embedded Python interpreter in the same thread as your rest application and you can't allow the `PyRun_InteractiveLoop()` to stop while waiting for user input. The one solution then is to call `PyParser_ParseString()` and test for `e.error` equal to `E_EOF`, which means the input is incomplete. Here's a sample code fragment, untested, inspired by code from Alex Farber:

```
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>

int testcomplete(char *code)
/* code should end in \n */
/* return -1 for error, 0 for incomplete, 1 for complete */
{
    node *n;
    perrdetail e;

    n = PyParser_ParseString(code, &PyParser_Grammar,
                             Py_file_input, &e);

    if (n == NULL) {
        if (e.error == E_EOF)
            return 0;
        return -1;
    }

    PyNode_Free(n);
}
```

()

()

```

    return 1;
}

```

Another solution is trying to compile the received string with `Py_CompileString()`. If it compiles without errors, try to execute the returned code object by calling `PyEval_EvalCode()`. Otherwise save the input for later. If the compilation fails, find out if it's an error or just more input is required - by extracting the message string from the exception tuple and comparing it to the string "unexpected EOF while parsing". Here is a complete example using the GNU readline library (you may want to ignore **SIGINT** while calling `readline()`):

```

#include <stdio.h>
#include <readline.h>

#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>

int main (int argc, char* argv[])
{
    int i, j, done = 0;                                /* lengths of line, code */
    char ps1[] = ">>> ";
    char ps2[] = "... ";
    char *prompt = ps1;
    char *msg, *line, *code = NULL;
    PyObject *src, *glb, *loc;
    PyObject *exc, *val, *trb, *obj, *dum;

    Py_Initialize ();
    loc = PyDict_New ();
    glb = PyDict_New ();
    PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

    while (!done)
    {
        line = readline (prompt);

        if (NULL == line)                                /* Ctrl-D pressed */
        {
            done = 1;
        }
        else
        {
            i = strlen (line);

            if (i > 0)
                add_history (line);                        /* save non-empty lines */

            if (NULL == code)                            /* nothing in code yet */
                j = 0;
            else
                j = strlen (code);

```

()

()

```

code = realloc (code, i + j + 2);
if (NULL == code)                                /* out of memory */
    exit (1);

if (0 == j)                                       /* code was empty, so */
    code[0] = '\0';                             /* keep strncat happy */

strncat (code, line, i);                         /* append line to code */
code[i + j] = '\n';                             /* append '\n' to code */
code[i + j + 1] = '\0';

src = Py_CompileString (code, "<stdin>", Py_single_input);

if (NULL != src)                                /* compiled just fine - */
{
    if (ps1 == prompt ||                         /* ">>> " or */
        '\n' == code[i + j - 1])                /* "... " and double '\n' */
    {                                             /* so execute it */
        dum = PyEval_EvalCode (src, glb, loc);
        Py_XDECREF (dum);
        Py_XDECREF (src);
        free (code);
        code = NULL;
        if (PyErr_Occurred ())
            PyErr_Print ();
        prompt = ps1;
    }
}
/* syntax error or E_EOF? */
else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
{
    PyErr_Fetch (&exc, &val, &trb);            /* clears exception! */

    if (PyArg_ParseTuple (val, "s0", &msg, &obj) &&
        !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
    {
        Py_XDECREF (exc);
        Py_XDECREF (val);
        Py_XDECREF (trb);
        prompt = ps2;
    }
    else                                       /* some other syntax error */
    {
        PyErr_Restore (exc, val, trb);
        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
    }
}
else                                       /* some non-syntax error */
{

```

()

```
PyErr_Print ();
free (code);
code = NULL;
prompt = ps1;
}

free (line);
}
}

Py_XDECREF(glb);
Py_XDECREF(loc);
Py_Finalize();
exit(0);
}
```

5.16 How do I find undefined g++ symbols `__builtin_new` or `__pure_virtual`?

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change LINKCC in the Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).

5.17 Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

Yes, you can inherit from built-in classes such as `int`, `list`, `dict`, etc.

The Boost Python Library (BPL, <http://www.boost.org/libs/python/doc/index.html>) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).

6.1 How do I run a Python program under Windows?

This is not necessarily a straightforward question. If you are already familiar with running programs from the Windows command line then everything will seem obvious; otherwise, you might need a little more guidance.

Unless you use some sort of integrated development environment, you will end up *typing* Windows commands into what is variously referred to as a "DOS window" or "Command prompt window". Usually you can create such a window from your search bar by searching for `cmd`. You should be able to recognize when you have started such a window because you will see a Windows "command prompt", which usually looks like this:

```
C:\>
```

The letter may be different, and there might be other things after it, so you might just as easily see something like:

```
D:\YourName\Projects\Python>
```

depending on how your computer has been set up and what else you have recently done with it. Once you have started such a window, you are well on the way to running Python programs.

You need to realize that your Python scripts have to be processed by another program called the Python *interpreter*. The interpreter reads your script, compiles it into bytecodes, and then executes the bytecodes to run your program. So, how do you arrange for the interpreter to handle your Python?

First, you need to make sure that your command window recognises the word "py" as an instruction to start the interpreter. If you have opened a command window, you should try entering the command `py` and hitting return:

```
C:\Users\YourName> py
```

You should then see something like:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You have started the interpreter in "interactive mode". That means you can enter Python statements or expressions interactively and have them executed or evaluated while you wait. This is one of Python's strongest features. Check it by entering a few expressions of your choice and seeing the results:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Many people use the interactive mode as a convenient yet highly programmable calculator. When you want to end your interactive Python session, call the `exit()` function or hold the **Ctrl** key down while you enter a **Z**, then hit the "Enter" key to get back to your Windows command prompt.

You may also find that you have a Start-menu entry such as *Start → Programs → Python 3.x → Python (command line)* that results in you seeing the `>>>` prompt in a new window. If so, the window will disappear after you call the `exit()` function or enter the **Ctrl-Z** character; Windows is running a single "python" command in the window, and closes it when you terminate the interpreter.

Now that we know the `py` command is recognized, you can give your Python script to it. You'll have to give either an absolute or a relative path to the Python script. Let's say your Python script is located in your desktop and is named `hello.py`, and your command prompt is nicely opened in your home directory so you're seeing something similar to:

```
C:\Users\YourName>
```

So now you'll ask the `py` command to give your script to Python by typing `py` followed by your script path:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

6.2 How do I make Python scripts executable?

On Windows, the standard Python installer already associates the `.py` extension with a file type (Python.File) and gives that file type an open command that runs the interpreter (`D:\Program Files\Python\python.exe "%1" %*`). This is enough to make scripts executable from the command prompt as `'foo.py'`. If you'd rather be able to execute the script by simple typing `'foo'` with no extension you need to add `.py` to the `PATHEXT` environment variable.

6.3 Why does Python sometimes take so long to start?

Usually Python starts very quickly on Windows, but occasionally there are bug reports that Python suddenly begins to take a long time to start up. This is made even more puzzling because Python will work fine on other Windows systems which appear to be configured identically.

The problem may be caused by a misconfiguration of virus checking software on the problem machine. Some virus scanners have been known to introduce startup overhead of two orders of magnitude when the scanner is configured to monitor all reads from the filesystem. Try checking the configuration of virus scanning

software on your systems to ensure that they are indeed configured identically. McAfee, when configured to scan all file system read activity, is a particular offender.

6.4 How do I make an executable from a Python script?

See [cx_Freeze](#) for a distutils extension that allows you to create console and GUI executables from Python code. [py2exe](#), the most popular extension for building Python 2.x-based executables, does not yet support Python 3 but a version that does is in development.

6.5 Is a *.pyd file the same as a DLL?

Yes, .pyd files are dll's, but there are a few differences. If you have a DLL named `foo.pyd`, then it must have a function `PyInit_foo()`. You can then write Python "import foo", and Python will search for `foo.pyd` (as well as `foo.py`, `foo.pyc`) and if it finds it, will attempt to call `PyInit_foo()` to initialize it. You do not link your .exe with `foo.lib`, as that would cause Windows to require the DLL to be present.

Note that the search path for `foo.pyd` is `PYTHONPATH`, not the same as the path that Windows uses to search for `foo.dll`. Also, `foo.pyd` need not be present to run your program, whereas if you linked your program with a `dll`, the `dll` is required. Of course, `foo.pyd` is required if you want to say `import foo`. In a `DLL`, linkage is declared in the source code with `__declspec(dllexport)`. In a .pyd, linkage is defined in a list of available functions.

6.6 How can I embed Python into a Windows application?

Embedding the Python interpreter in a Windows app can be summarized as follows:

1. Do `__not__` build Python into your .exe file directly. On Windows, Python must be a DLL to handle importing modules that are themselves DLL's. (This is the first key undocumented fact.) Instead, link to `pythonNN.dll`; it is typically installed in `C:\Windows\System`. `NN` is the Python version, a number such as "33" for Python 3.3.

You can link to Python in two different ways. Load-time linking means linking against `pythonNN.lib`, while run-time linking means linking against `pythonNN.dll`. (General note: `pythonNN.lib` is the so-called "import lib" corresponding to `pythonNN.dll`. It merely defines symbols for the linker.)

Run-time linking greatly simplifies link options; everything happens at run time. Your code must load `pythonNN.dll` using the Windows `LoadLibraryEx()` routine. The code must also use access routines and data in `pythonNN.dll` (that is, Python's C API's) using pointers obtained by the Windows `GetProcAddress()` routine. Macros can make using these pointers transparent to any C code that calls routines in Python's C API.

Borland note: convert `pythonNN.lib` to OMF format using `Coff2Omf.exe` first.

2. If you use SWIG, it is easy to create a Python "extension module" that will make the app's data and methods available to Python. SWIG will handle just about all the grungy details for you. The result is C code that you link *into* your .exe file (!) You do `__not__` have to create a DLL file, and this also simplifies linking.
3. SWIG will create an init function (a C function) whose name depends on the name of the extension module. For example, if the name of the module is `leo`, the init function will be called `initleo()`. If you use SWIG shadow classes, as you should, the init function will be called `initleoc()`. This initializes a mostly hidden helper class used by the shadow class.

The reason you can link the C code in step 2 into your .exe file is that calling the initialization function is equivalent to importing the module into Python! (This is the second key undocumented fact.)

4. In short, you can use the following code to initialize the Python interpreter with your extension module.

```
#include "python.h"
...
Py_Initialize(); // Initialize Python.
initmyAppc(); // Initialize (import) the helper class.
PyRun_SimpleString("import myApp"); // Import the shadow class.
```

5. There are two problems with Python's C API which will become apparent if you use a compiler other than MSVC, the compiler used to build pythonNN.dll.

Problem 1: The so-called "Very High Level" functions that take FILE * arguments will not work in a multi-compiler environment because each compiler's notion of a struct FILE will be different. From an implementation standpoint these are very `_low_` level functions.

Problem 2: SWIG generates the following code when generating wrappers to void functions:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Alas, `Py_None` is a macro that expands to a reference to a complex data structure called `_Py_NoneStruct` inside `pythonNN.dll`. Again, this code will fail in a multi-compiler environment. Replace such code by:

```
return Py_BuildValue("");
```

It may be possible to use SWIG's `%typemap` command to make the change automatically, though I have not been able to get this to work (I'm a complete SWIG newbie).

6. Using a Python shell script to put up a Python interpreter window from inside your Windows app is not a good idea; the resulting window will be independent of your app's windowing system. Rather, you (or the `wxPythonWindow` class) should create a "native" interpreter window. It is easy to connect that window to the Python interpreter. You can redirect Python's i/o to `_any_` object that supports `read` and `write`, so all you need is a Python object (defined in your extension module) that contains `read()` and `write()` methods.

6.7 How do I keep editors from inserting tabs into my Python source?

The FAQ does not recommend using tabs, and the Python style guide, [PEP 8](#), recommends 4 spaces for distributed Python code; this is also the Emacs python-mode default.

Under any editor, mixing tabs and spaces is a bad idea. MSVC is no different in this respect, and is easily configured to use spaces: Take *Tools* → *Options* → *Tabs*, and for file type "Default" set "Tab size" and "Indent size" to 4, and select the "Insert spaces" radio button.

Python raises `IndentationError` or `TabError` if mixed tabs and spaces are causing problems in leading whitespace. You may also run the `tabnanny` module to check a directory tree in batch mode.

6.8 How do I check for a keypress without blocking?

Use the `msvcrt` module. This is a standard Windows-specific extension module. It defines a function `kbhit()` which checks whether a keyboard hit is present, and `getch()` which gets one character without echoing it.

7.1 General GUI Questions

7.2 What platform-independent GUI toolkits exist for Python?

Depending on what platform(s) you are aiming at, there are several. Some of them haven't been ported to Python 3 yet. At least *Tkinter* and *Qt* are known to be Python 3-compatible.

7.2.1 Tkinter

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called *tkinter*. This is probably the easiest to install (since it comes included with most [binary distributions](#) of Python) and use. For more info about Tk, including pointers to the source, see the [Tcl/Tk home page](#). Tcl/Tk is fully portable to the Mac OS X, Windows, and Unix platforms.

7.2.2 wxWidgets

wxWidgets (<https://www.wxwidgets.org>) is a free, portable GUI class library written in C++ that provides a native look and feel on a number of platforms, with Windows, Mac OS X, GTK, X11, all listed as current stable targets. Language bindings are available for a number of languages including Python, Perl, Ruby, etc.

wxPython is the Python binding for *wxwidgets*. While it often lags slightly behind the official *wxWidgets* releases, it also offers a number of features via pure Python extensions that are not available in other language bindings. There is an active *wxPython* user and developer community.

Both *wxWidgets* and *wxPython* are free, open source, software with permissive licences that allow their use in commercial products as well as in freeware or shareware.

7.2.3 Qt

There are bindings available for the Qt toolkit (using either [PyQt](#) or [PySide](#)) and for KDE ([PyKDE4](#)). PyQt is currently more mature than PySide, but you must buy a PyQt license from [Riverbank Computing](#) if you want to write proprietary applications. PySide is free for all applications.

Qt 4.5 upwards is licensed under the LGPL license; also, commercial licenses are available from [The Qt Company](#).

7.2.4 Gtk+

The [GObject introspection bindings](#) for Python allow you to write GTK+ 3 applications. There is also a [Python GTK+ 3 Tutorial](#).

The older PyGtk bindings for the [Gtk+ 2 toolkit](#) have been implemented by James Henstridge; see [<http://www.pygtk.org>](http://www.pygtk.org).

7.2.5 Kivy

Kivy is a cross-platform GUI library supporting both desktop operating systems (Windows, macOS, Linux) and mobile devices (Android, iOS). It is written in Python and Cython, and can use a range of windowing backends.

Kivy is free and open source software distributed under the MIT license.

7.2.6 FLTK

Python bindings for the [FLTK toolkit](#), a simple yet powerful and mature cross-platform windowing system, are available from the [PyFLTK project](#).

7.2.7 OpenGL

For OpenGL bindings, see [PyOpenGL](#).

7.3 What platform-specific GUI toolkits exist for Python?

By installing the [PyObjc Objective-C bridge](#), Python programs can use Mac OS X's Cocoa libraries.

Pythonwin by Mark Hammond includes an interface to the Microsoft Foundation Classes and a Python programming environment that's written mostly in Python using the MFC classes.

7.4 Tkinter questions

7.4.1 How do I freeze Tkinter applications?

Freeze is a tool to create stand-alone applications. When freezing Tkinter applications, the applications will not be truly stand-alone, as the application will still need the Tcl and Tk libraries.

One solution is to ship the application with the Tcl and Tk libraries, and point to them at run-time using the `TCL_LIBRARY` and `TK_LIBRARY` environment variables.

To get truly stand-alone applications, the Tcl scripts that form the library have to be integrated into the application as well. One tool supporting that is SAM (stand-alone modules), which is part of the Tix distribution (<http://tix.sourceforge.net/>).

Build Tix with SAM enabled, perform the appropriate call to `Tclsam_init()`, etc. inside Python's `Modules/tkappinit.c`, and link with `libtclsam` and `libtkSAM` (you might include the Tix libraries as well).

7.4.2 Can I have Tk events handled while waiting for I/O?

On platforms other than Windows, yes, and you don't even need threads! But you'll have to restructure your I/O code a bit. Tk has the equivalent of Xt's `XtAddInput()` call, which allows you to register a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. See `tkinter-file-handlers`.

7.4.3 I can't get key bindings to work in Tkinter: why?

An often-heard complaint is that event handlers bound to events with the `bind()` method don't get handled even when the appropriate key is pressed.

The most common cause is that the widget to which the binding applies doesn't have "keyboard focus". Check out the Tk documentation for the `focus` command. Usually a widget is given the keyboard focus by clicking in it (but not for labels; see the `takefocus` option).

“ Python ”

8.1 Python

Python [Google NASA](#)
[Python Beginner's Guide to Python.](#)

8.2 Python

- Python
- Python Python GUI
- Windows Python Hewlett-Packard Compaq Python HP/Compaq Python
- Unix Mac OS X Linux Python

8.3 Python

Python
 Python Windows “ ” “ / ”
 Python Python
 Python Python Python

>>> Python
... Python

2to3 Python 2.x Python 3.x
2to3 lib2to3 Tools/scripts/2to3 2to3-reference

abstract base class – ABC *duck-typing* hasattr() ABC
isinstance() issubclass() abc Python ABC collections.
abc numbers io importlib.abc abc ABC

annotation – *type hint*
__annotations__
variable annotation function annotation **PEP 484** **PEP 526**

argument – *function method*

- : name= ** 3 5 complex() :

`complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})`
- : / * *iterable* 3 5 :

`complex(3, 5)
complex(*(3, 5))`

calls
parameter **PEP 362**

asynchronous context manager – __aenter__() __aexit__() async with
PEP 492

asynchronous generator – *asynchronous generator iterator* async def yield
 async for

await async for async with

asynchronous generator iterator – *asynchronous generator*
asynchronous iterator __anext__() yield
 yield (try) __anext__() PEP 492 PEP 525

asynchronous iterable – async for __aiter__() *asynchronous iterator* PEP 492

asynchronous iterator – __aiter__() __anext__() __anext__ awaitable async for
 __anext__() StopAsyncIteration PEP 492

attribute – o a o.a

awaitable – await *coroutine* __await__() PEP 492

BDFL “ ” Guido van Rossum Python

binary file – *file object* 'rb', 'wb' or 'rb+' sys.stdin.buffer sys.stdout.
 buffer io.BytesIO gzip.GzipFile
text file str

bytes-like object – bufferobjects C-contiguous bytes bytearray array.array
 memoryview

“ ” bytearray bytearray memoryview (“ ”)
 bytes bytes memoryview

bytecode – Python CPython Python .pyc “ ”
virtual machine Python Python

dis

class –

class variable – ()

coercion – int(3.15) 3 3+4.5 int, float
 TypeError float(3)+4.5 3+4.5

complex number – -1 i j Python j
 3+1j math cmath

context manager – with __enter__() __exit__() PEP 343

contiguous – C- Fortran C Fortran C-

coroutine – async def PEP 492

coroutine function – *coroutine* async def await async for async with
 PEP 492

CPython Python python.org “CPython” Jython IronPython

decorator – @wrapper classmethod() staticmethod()
 :


```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

descriptor – `__get__()`, `__set__()`, `__delete__()` *a.b* *a*
b *b* Python

descriptors

dictionary – `__hash__()`, `__eq__()` Perl hash

dictionary view – `dict.keys()`, `dict.values()`, `dict.items()`
`list(dictview)` dict-views

docstring – `__doc__`

duck-typing – “ ” `type()`
`isinstance()` () `hasattr()` *EAFP*

EAFP “ ” Python `try except` *LBYL* C

expression – *statement* `while`

extension module – C C++ Python C API

f-string – `f' ' 'F'` “f- ” **PEP 498**

file object – API `read()` `write()` /

`:` , `io` `open()`

file-like object – *file object*

finder – *loader*

Python 3.3 : `sys.meta_path` *path entry finders* `sys.path_hooks`

PEP 302, PEP 420 PEP 451

floor division – `//` `11 // 4` `2` `2.75` `(-11) // 4` `-3`
`-2.75` **PEP 238**

function – *parameter, method* function

function annotation – *annotation*

`int` `int` :

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

function

variable annotation **PEP 484**

__future__

__future__ :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection – Python gc

generator – *generator iterator* yield for- next()

generator iterator – *generator*

yield try

generator expression – for if :

```
>>> sum(i*i for i in range(10)) # sum of squares 0, 1, 4, ... 81
285
```

generic function –

single dispatch functools.singledispatch() PEP 443

GIL *global interpreter lock*

global interpreter lock – CPython Python *bytecode* dict
CPython

GIL I/O GIL

“ ”

hash-based pyc – pyc pyc-invalidation

hashable – __hash__() __eq__()

Python id()

IDLE Python IDE “ ” Python

immutable –

import path – *path based finder* sys.path __path__

importing – Python Python

importer – *finder loader*

interactive – Python python
help(x)

interpreted – Python / *in-teractive*

interpreter shutdown – Python

__main__

iterable – `list` `str` `tuple` `dict` `__iter__()` *Sequence*
`__getitem__()`
`for` `zip()` `map()` ... `iter()` `iter()`
`for` *iterator sequence* *generator*

iterator – `__next__()` `next()` `StopIteration`
`__next__()` `StopIteration` `__iter__()`
`list` `iter()` `for`

`typeiter`

key function – `locale.strxfrm()`
`Python` `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` `itertools.groupby()`
`str.lower()` `lambda` `lambda r: (r[0], r[2])` `operator`
`attrgetter()` `itemgetter()` `methodcaller()`

keyword argument – *argument*

lambda *expression* `lambda` `lambda [parameters]: expression`

LBYL “ ” *EAFP* `if`
`LBYL` “ ” “ ” `if key in mapping: return mapping[key]` *mapping*
`key` `EAFP`

list – `Python` *sequence* `O(1)`

list comprehension – `result = [{':#04x}'.format(x) for x in range(256)]`
`if x % 2 == 0]` `0 255` `0x..` `if` `range(256)`

loader – `load_module()` *finder* **PEP 302** *abstract base class* `importlib.abc.Loader`

mapping – `Mapping` `MutableMapping` `dict`, `collections.defaultdict`,
`collections.OrderedDict` `collections.Counter`

meta path finder – `sys.meta_path` *finder* *path entry finders*
`importlib.abc.MetaPathFinder`

metaclass – `Python`
`metaclasses`

method – *argument* (`self`) *function* *nested scope*

method resolution order – `Python 2.3` `2.3` `Python`

module – `Python` `Python` *importing* `Python`
package

module spec – `importlib.machinery.ModuleSpec`

MRO *method resolution order*

mutable – `id()` *immutable*

named tuple – `time.localtime()` *year* `t[0]` `t.`
`tm_year`

time.struct_time collections.namedtuple()
Employee(name='jones', title='programmer')

namespace – builtins.open os.open()
random.seed() itertools.islice() random itertools

namespace package – PEP 420 package regular package __init__.
py module

nested scope – nonlocal

new-style class – Python Python
__slots__ __getattr__()

object – object new-style class

package – Python module __path__ Python
regular package namespace package

parameter – function argument

- positional-or-keyword foo bar:

```
def func(foo, bar=None): ...
```
- positional-only Python abs()
• keyword-only * kw_only1 kw_only2:

```
def func(arg, *, kw_only1, kw_only2): ...
```
- var-positional * args:

```
def func(*args, **kwargs): ...
```
- var-keyword ** kwargs

argument inspect.Parameter function PEP 362

path entry – import path path based finder

path entry finder – sys.path_hooks (path entry hook) finder path entry
importlib.abc.PathEntryFinder

path entry hook – path entry sys.path_hook path entry finder

path based finder – import path

path-like object – str bytes os.PathLike os.PathLike
os.fspath() str bytes os.fsdecode() os.fsencode() str bytes
PEP 519

PEP “Python ” PEP Python Python PEP
PEP Python PEP
PEP 1

portion – zip PEP 420

positional argument – *argument*

provisional API – API API –

API “ ” ____

PEP 411

provisional package – *provisional API*

Python 3000 Python 3.x 3 “Py3k”

Pythonic Python Python for Python
:

```
for i in range(len(food)):
    print(food[i])
```

Pythonic :

```
for piece in food:
    print(piece)
```

qualified name – “ ” PEP 3155 :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

email.mime.text:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count – Python CPython sys
getrefcount()

regular package – *package* __init__.py
namespace package

__slots__

sequence – *iterable* __getitem__() __len__() list str tuple
bytes dict __getitem__() __len__() *immutable*
collections.abc.Sequence __getitem__() __len__() count(), index(),
__contains__() __reversed__() register()

single dispatch – *generic function*

slice – *sequence* [] *variable_name*[1:3:5] *slice*

special method – Python *specialnames*

statement – “ ” *expression* if while for

struct sequence – *named tuple* _make() _asdict()
sys.float_info os.stat()

text encoding – Unicode

text file – str *file object* *text encoding* 'r' 'w' sys.
stdin sys.stdout io.StringIO
binary file

triple-quoted string – “ ” ’

type – Python __class__ type(obj)

type alias –

:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

typing **PEP 484**

type hint – *annotation*

Python IDE

typing.get_type_hints()

typing **PEP 484**

universal newlines – Unix '\n' Windows '\r\n' Macintosh '\r'
PEP 278 PEP 3116 bytes.splitlines()

variable annotation – *annotation*

:

```
class C:
    field: 'annotation'
```

int :

count: `int` = 0

annassign

function annotation **PEP 484** **PEP 526**

virtual environment – Python Python Python
venv

virtual machine – Python *bytecode*

Zen of Python – Python "import this"

Sphinx Python reStructuredText
Python reporting-bugs

- Fred L. Drake, Jr. [Python](#)
- [reStructuredText](#) [Docutils](#)
- Fredrik Lundh [Alternative Python Reference](#) [Sphinx](#)

B.1 Python

Python	Python	Python	Misc/ACKS	Python
Python	Python	-		

History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <https://www.cwi.nl/>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <https://www.cnri.reston.va.us/>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <http://www.zope.com/>). In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see <https://opensource.org/> for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.7.2 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.7.2 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2019 Python Software Foundation; All Rights Reserved" are retained in Python 3.7.2 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.7.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.7.2.
4. PSF is making Python 3.7.2 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.7.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 3.7.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191

()

()

- ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
 3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
 4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
 5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
 6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
 7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

()

()

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`
or `init_by_array(init_key, key_length)`.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions

()

()

are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, <http://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND

()

()

ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.3 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.4 Cookie management

The `http.cookies` module contains the following notice:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of

()

()

Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.5 Execution tracing

The `trace` module contains the following notice:

portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
<http://zooko.com/>
<mailto:zooko@zooko.com>

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of neither Automatrix, Bioreason or Mojam Media be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

C.3.6 UUencode and UUdecode functions

The `uu` module contains the following notice:

Copyright 1994 by Lance Ellinghouse
 Cathedral City, California Republic, United States of America.
 All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:

- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
 Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,

()

()

WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.8 test_epoll

The `test_epoll` module contains the following notice:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.9 Select kqueue

The `select` module contains the following notice for the `kqueue` interface:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE

()

()

```
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski's implementation of Dan Bernstein's SipHash24 algorithm. The contains the following note:

```
<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
  https://github.com/majek/csiphash/

Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/siphash24/little)
  djb (supercop/crypto_auth/siphash24/little2)
  Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <http://www.netlib.org/fp/>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/* *****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
```

()

()

```

* purpose without fee is hereby granted, provided that this entire notice
* is included in all copies of any software which is or includes a copy
* or modification of this software and in all copies of the supporting
* documentation for such software.
*
* THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
* WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
*****/

```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```

LICENSE ISSUES
=====

```

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

```

OpenSSL License
-----

```

```

/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to

```

()

()

```

*   endorse or promote products derived from this software without
*   prior written permission. For written permission, please contact
*   openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*   nor may "OpenSSL" appear in their names without prior written
*   permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*   acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to.  The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code.  The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.

```

()

()

```

* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*   notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*   notice, this list of conditions and the following disclaimer in the
*   documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*   must display the following acknowledgement:
*   "This product includes cryptographic software written by
*    Eric Young (eay@cryptsoft.com)"
*   The word 'cryptographic' can be left out if the routines from the library
*   being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*   the apps directory (application specific code) you must include an acknowledgement:
*   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

C.3.13 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured --with-system-expat:

```

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

```

```

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the

```

()

()

"Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.14 libffi

The `_ctypes` extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ``Software''), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.15 zlib

The `zlib` extension is built using an included copy of the `zlib` sources if the `zlib` version found on the system is too old to be used for the build:

Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

C.3.16 cfuhash

The implementation of the hash table used by the tracemalloc is based on the cfuhash project:

Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS

()

()

FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

APPENDIX D

Python

Copyright © 2001-2019 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

History and License

Non-alphabetical

..., 83
2to3, 83
>>>, 83
__future__, 86
__slots__, 89

PATH, 54
PYTHONDONTWRITEBYTECODE, 35
TCL_LIBRARY, 79
TK_LIBRARY, 79

A

abstract base class -- , 83
annotation -- , 83
argument
 difference from parameter, 15
argument -- , 83
asynchronous context manager -- , 83
asynchronous generator -- , 84
asynchronous generator iterator -- , 84
asynchronous iterable -- , 84
asynchronous iterator -- , 84
attribute -- , 84
awaitable -- , 84

B

BDFL, 84
binary file -- , 84
bytecode -- , 84
bytes-like object -- , 84

C

C-contiguous, 84
class -- , 84
class variable -- , 84
coercion -- , 84
complex number -- , 84
context manager -- , 84

contiguous -- , 84
coroutine -- , 84
coroutine function -- , 84
CPython, 84

D

decorator -- , 84
descriptor -- , 85
dictionary -- , 85
dictionary view -- , 85
docstring -- , 85
duck-typing -- , 85

E

EAFP, 85
expression -- , 85
extension module -- , 85

F

file object -- , 85
file-like object -- , 85
finder -- , 85
floor division -- , 85
Fortran contiguous, 84
f-string -- f- , 85
function -- , 85
function annotation -- , 85

G

garbage collection -- , 86
generator, 86
generator -- , 86
generator expression, 86
generator expression -- , 86
generator iterator -- , 86
generic function -- , 86
GIL, 86
global interpreter lock -- , 86

H

hashable -- , 86
hash-based pyc -- pyc, 86

I

IDLE, 86
immutable -- , 86
import path -- , 86
importer -- , 86
importing -- , 86
interactive -- , 86
interpreted -- , 86
interpreter shutdown -- , 86
iterable -- , 87
iterator -- , 87

K

key function -- , 87
keyword argument -- , 87

L

lambda, 87
LBYL, 87
list -- , 87
list comprehension -- , 87
loader -- , 87

M

mapping -- , 87
meta path finder -- , 87
metaclass -- , 87
method -- , 87
method resolution order -- , 87
module -- , 87
module spec -- , 87
MRO, 87
mutable -- , 87

N

named tuple -- , 87
namespace -- , 88
namespace package -- , 88
nested scope -- , 88
new-style class -- , 88

O

object -- , 88

P

package -- , 88
parameter
 difference from argument, 15
parameter -- , 88

PATH, 54

path based finder -- , 88
path entry -- , 88
path entry finder -- , 88
path entry hook -- , 88
path-like object -- , 88
PEP, 88
portion -- , 89
positional argument -- , 89
provisional API -- API, 89
provisional package -- , 89
Python 3000, 89

Python

 PEP 1, 88
 PEP 5, 6
 PEP 6, 2
 PEP 8, 10, 74
 PEP 238, 85
 PEP 275, 43
 PEP 278, 90
 PEP 302, 85, 87
 PEP 343, 84
 PEP 362, 83, 88
 PEP 411, 89
 PEP 420, 85, 88, 89
 PEP 443, 86
 PEP 451, 85
 PEP 484, 83, 85, 90, 91
 PEP 492, 83, 84
 PEP 498, 85
 PEP 519, 88
 PEP 525, 84
 PEP 526, 83, 91
 PEP 570, 21
 PEP 3116, 90
 PEP 3147, 35
 PEP 3155, 89

PYTHONDONTWRITEBYTECODE, 35

Pythonic, 89

Q

qualified name -- , 89

R

reference count -- , 89
regular package -- , 89

S

sequence -- , 89
single dispatch -- , 90
slice -- , 90
special method -- , 90
statement -- , 90
struct sequence -- , 90

T

TCL_LIBRARY, [79](#)
text encoding -- , [90](#)
text file -- , [90](#)
TK_LIBRARY, [79](#)
triple-quoted string -- , [90](#)
type -- , [90](#)
type alias -- , [90](#)
type hint -- , [90](#)

U

universal newlines -- , [90](#)

V

variable annotation -- , [90](#)
virtual environment -- , [91](#)
virtual machine -- , [91](#)

Z

Zen of Python -- Python , [91](#)