Library and Extension FAQ

Contents

General Library Questions

How do I find a module or application to perform task X?

Check the Library Reference to see if there's a relevant standard library module. (Eventually you'll learn what's in the standard library and will be able to skip this step.)

For third-party packages, search the Python Package Index or try Google or another Web search engine. Searching for "Python" plus a keyword or two for your topic of interest will usually find something helpful.

Where is the math.py (socket.py, regex.py, etc.) source file?

If you can't find a source file for a module it may be a built-in or dynamically loaded module implemented in C, C++ or other compiled language. In this case you may not have the source file or it may be something like `mathmodule.c`, somewhere in a C source directory (not on the Python Path).

There are (at least) three kinds of modules in Python:

1. modules written in Python (.py);
2. modules written in C and dynamically loaded (.dll, .pyd, .so, .sl, etc);
3. modules written in C and linked with the interpreter; to get a list of these, type:

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

How do I make a Python script executable on Unix?

You need to do two things: the script file's mode must be executable and the first line must begin with `#!` followed by the path of the Python interpreter.

The first is done by executing `chmod +x scriptfile` or perhaps `chmod 755 scriptfile`.

The second can be done in a number of ways. The most straightforward way is to write

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

as the very first line of your file, using the pathname for where the Python interpreter is installed on your platform.

If you would like the script to be independent of where the Python interpreter lives, you can use the **env** program. Almost all Unix variants support the following, assuming the Python interpreter is in a directory on the user's `PATH`:

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

Don't do this for CGI scripts. The `PATH` variable for CGI scripts is often very minimal, so you need to use the actual absolute pathname of the interpreter.

Occasionally, a user's environment is so full that the **/usr/bin/env** program fails; or there's no env program at all. In that case, you can try the following hack (due to Alex Rezinsky):

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

The minor disadvantage is that this defines the script's __doc__ string. However, you can fix that by adding

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

Is there a curses/termcap package for Python?

For Unix variants: The standard Python source distribution comes with a curses module in the Modules subdirectory, though it's not compiled by default. (Note that this is not available in the Windows distribution – there is no curses module for Windows.)

The `curses` module supports basic curses features as well as many additional functions from ncurses and SYSV curses such as colour, alternative character set support, pads, and mouse support. This means the module isn't compatible with operating systems that only have BSD curses, but there don't seem to be any currently maintained OSes that fall into this category.

For Windows: use the consolelib module.

Is there an equivalent to C's onexit() in Python?

The `atexit` module provides a register function that is similar to C's `onexit()`.

Why don’t my signal handlers work?

The most common problem is that the signal handler is declared with the wrong argument list. It is called as

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

so it should be declared with two arguments:

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

Common tasks

How do I test a Python program or component?

Python comes with two testing frameworks. The `doctest` module finds examples in the docstrings for a module and runs them, comparing the output with the expected output given in the docstring.

The `unittest` module is a fancier testing framework modelled on Java and Smalltalk testing frameworks.

To make testing easier, you should use good modular design in your program. Your program should have almost all functionality encapsulated in either functions or class methods – and this sometimes has the surprising and delightful effect of making the program run faster (because local variable accesses are faster than global accesses). Furthermore the program should avoid depending on mutating global variables, since this makes testing much more difficult to do.

The “global main logic” of your program may be as simple as

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

at the bottom of the main module of your program.

Once your program is organized as a tractable collection of functions and class behaviours you should write test functions that exercise the behaviours. A test suite that automates a sequence of tests can be associated with each module. This sounds like a lot of work, but since Python is so terse and flexible it’s surprisingly easy. You can make coding much more pleasant and fun by writing your test functions in parallel with the “production code”, since this makes it easy to find bugs and even design flaws earlier.

“Support modules” that are not intended to be the main module of a program may include a self-test of the module.

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

Even programs that interact with complex external interfaces may be tested when the external interfaces are unavailable by using “fake” interfaces implemented in Python.

How do I create documentation from doc strings?

The `pydoc` module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is epydoc. Sphinx can also include docstring content.

How do I get a single keypress at a time?

For Unix variants there are several solutions. It’s straightforward to do this using curses, but curses is a fairly large module to learn.

Threads

How do I program using threads?

Be sure to use the `threading` module and not the `_thread` module. The `threading` module builds convenient abstractions on top of the low-level primitives provided by the `_thread` module.

Aahz has a set of slides from his threading tutorial that are helpful; see http://www.pythoncraft.com/OSCON2001/.

None of my threads seem to run: why?

As soon as the main thread exits, all threads are killed. Your main thread is running too quickly, giving the threads no time to do any work.

A simple fix is to add a sleep to the end of the program that’s long enough for all the threads to finish:

```
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)  # <--------------------------!
```

But now (on many platforms) the threads don't run in parallel, but appear to run sequentially, one at a time! The reason is that the OS thread scheduler doesn't start a new thread until the previous thread is blocked.

A simple fix is to add a tiny sleep to the start of the run function:

```
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)
```

Instead of trying to guess a good delay value for `time.sleep()`, it's better to use some kind of semaphore mechanism. One idea is to use the `queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

How do I parcel out work among a bunch of worker threads?

The easiest way is to use the new `concurrent.futures` module, especially the `ThreadPoolExecutor` class.

Or, if you want fine control over the dispatching algorithm, you can write your own logic manually. Use the `queue` module to create a queue containing a list of jobs. The `Queue` class maintains a list of objects and has a `.put(obj)` method that adds items to the queue and a `.get()` method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Here's a trivial example:

```
import threading, queue, time

# The worker thread gets jobs off the queue.  When the queue is empty,
```

```
# 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)
```

When run, this will produce the following output:

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

Consult the module's documentation for more details; the `Queue` class provides a featureful interface.

What kinds of global value mutation are thread-safe?

A global interpreter lock (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via `sys.setswitchinterval()`. Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that "look atomic" really are.

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):

```
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()
```

These aren't:

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

Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

Can’t we get rid of the Global Interpreter Lock?

The global interpreter lock (GIL) is often seen as a hindrance to Python’s deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the “free threading” patches) that removed the GIL and replaced it with fine-grained locking. Adam Olsen recently did a similar experiment in his python-safethread project. Unfortunately, both experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL.

This doesn’t mean that you can’t make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. The `ProcessPoolExecutor` class in the new `concurrent.futures` module provides an easy way of doing so; the `multiprocessing` module provides a lower-level API in case you want more control over dispatching of tasks.

Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done. Some standard library modules such as `zlib` and `hashlib` already do this.

It has been suggested that the GIL should be a per-interpreter-state lock rather than truly global; interpreters then wouldn’t be able to share objects. Unfortunately, this isn’t likely to happen either. It would be a tremendous amount of work, because many object implementations currently have global state. For example, small integers and short strings are cached; these caches would have to be moved to the interpreter state. Other object types have their own free list; these free lists would have to be moved to the interpreter state. And so on.

And I doubt that it can even be done in finite time, because the same problem exists for 3rd party extensions. It is likely that 3rd party extensions are being written at a faster rate than you can convert them to store all their global state in the interpreter state.

And finally, once you have multiple interpreters not sharing any state, what have you gained over running each interpreter in a separate process?

Input and Output

How do I delete a file? (And other file questions…)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the `os` module. The two functions are identical; `unlink()` is simply the name of the Unix system call for this function.

To remove a directory, use `os.rmdir()`; use `os.mkdir()` to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don't exist. `os.removedirs(path)` will remove intermediate directories as long as they're empty; if you want to delete an entire directory tree and its contents, use `shutil.rmtree()`.

To rename a file, use `os.rename(old_path, new_path)`.

To truncate a file, open it using `f = open(filename, "rb+")`, and use `f.truncate(offset)`; offset defaults to the current seek position. There's also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where *fd* is the file descriptor (a small integer).

The `shutil` module also contains a number of functions to work on files including `copyfile()`, `copytree()`, and `rmtree()`.

How do I copy a file?

The `shutil` module contains a `copyfile()` function. Note that on MacOS 9 it doesn't copy the resource fork and Finder info.

How do I read (or write) binary data?

To read or write complex binary data formats, it's best to use the `struct` module. It allows you to take a string containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 2-byte integers and one 4-byte integer in big-endian format from a file:

```
import struct

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

The '>' in the format string forces big-endian data; the letter 'h' reads one "short integer" (2 bytes), and 'l' reads one "long integer" (4 bytes) from the string.

For data that is more regular (e.g. a homogeneous list of ints or floats), you can also use the `array` module.

> **Note:** To read and write binary data, it is mandatory to open the file in binary mode (here, passing `"rb"` to `open()`). If you use `"r"` instead (the default), the file will be open in text mode and `f.read()` will return `str` objects rather than `bytes` objects.

I can't seem to use os.read() on a pipe created with os.popen(); why?

`os.read()` is a low-level function which takes a file descriptor, a small integer representing the opened file. `os.popen()` creates a high-level file object, the same type returned by the built-in `open()` function. Thus, to read *n* bytes from a pipe *p* created with `os.popen()`, you need to use `p.read(n)`.

How do I access the serial (RS232) port?

For Win32, POSIX (Linux, BSD, etc.), Jython:

> http://pyserial.sourceforge.net

For Unix, see a Usenet post by Mitch Chapman:

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

Why doesn't closing sys.stdout (stdin, stderr) really close it?

Python file objects are a high-level layer of abstraction on low-level C file descriptors.

For most file objects you create in Python via the built-in `open()` function, `f.close()` marks the Python file object as being closed from Python's point of view, and also arranges to close the underlying C file descriptor. This also happens automatically in `f`'s destructor, when `f` becomes garbage.

But stdin, stdout and stderr are treated specially by Python, because of the special status also given to them by C. Running `sys.stdout.close()` marks the Python-level file object as being closed, but does *not* close the associated C file descriptor.

To close the underlying C file descriptor for one of these three, you should first be sure that's what you really want to do (e.g., you may confuse extension modules trying to do I/O). If it is, use `os.close()`:

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

Or you can use the numeric constants 0, 1 and 2, respectively.

Network/Internet Programming

What WWW tools are there for Python?

See the chapters titled Internet Protocols and Support and Internet Data Handling in the Library Reference Manual. Python has many modules that will help you build server-side and client-side web systems.

A summary of available frameworks is maintained by Paul Boddie at https://wiki.python.org/moin/WebProgramming.

Cameron Laird maintains a useful set of pages about Python web technologies at http://phaseit.net/claird/comp.lang.python/web_python.

How can I mimic CGI form submission (METHOD=POST)?

I would like to retrieve web pages that are the result of POSTing a form. Is there existing code that would let me do this easily?

Yes. Here's a simple example that uses 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()
```

Note that in general for percent-encoded POST operations, query strings must be quoted using `urllib.parse.urlencode()`. For example, to send `name=Guy Steele, Jr.`:

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

See also: HOWTO Fetch Internet Resources Using The urllib Package for extensive examples.

What module should I use to help with generating HTML?

You can find a collection of useful links on the Web Programming wiki page.

How do I send mail from a Python script?

Use the standard library module `smtplib`.

Here's a very simple interactive mail sender that uses it. This method will work on any host that supports an SMTP listener.

```
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()
```

A Unix-only alternative uses sendmail. The location of the sendmail program varies between systems; sometimes it is `/usr/lib/sendmail`, sometimes `/usr/sbin/sendmail`. The sendmail manual page will help you out. Here's some sample code:

```
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)
```

How do I avoid blocking in the connect() method of a socket?

The `select` module is commonly used to help with asynchronous I/O on sockets.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno`. `errno.EINPROGRESS` indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the errno value. To poll, you can call `connect_ex()` again later – `0` or `errno.EISCONN` indicate that you're connected – or you can pass this socket to select to check if it's writable.

> **Note:** The `asyncore` module presents a framework-like approach to the problem of writing non-blocking networking code. The third-party Twisted library is a popular and feature-rich alternative.

Databases

Are there any interfaces to database packages in Python?

Yes.

Interfaces to disk-based hashes such as `DBM` and `GDBM` are also included with standard Python. There is also the `sqlite3` module, which provides a lightweight disk-based relational database.

Support for most relational databases is available. See the DatabaseProgramming wiki page for details.

How do you implement persistent objects in Python?

The `pickle` library module solves this in a very general way (though you still can't store things like open files, sockets or windows), and the `shelve` library module uses pickle and (g)dbm to create persistent mappings containing arbitrary Python objects.

Mathematics and Numerics

How do I generate random numbers in Python?

The standard module `random` implements a random number generator. Usage is simple:

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

This returns a random floating point number in the range [0, 1).

There are also many other specialized generators in this module, such as:

- `randrange(a, b)` chooses an integer in the range [a, b).
- `uniform(a, b)` chooses a floating point number in the range [a, b).
- `normalvariate(mean, sdev)` samples the normal (Gaussian) distribution.

Some higher-level functions operate on sequences directly, such as:

- `choice(S)` chooses random element from a given sequence
- `shuffle(L)` shuffles a list in-place, i.e. permutes it randomly

There's also a `Random` class you can instantiate to create independent multiple random number generators.