# [How to use FTP in Python](http://www.pythonforbeginners.com/code-snippets-source-code/how-to-use-ftp-in-python/)

## Overview

This article will show how you can use FTP in Python with the help of the ftplib module.

## Ftplib

The ftplib module in Python allows you to write Python programs that perform a variety of automated FTP jobs. You can easily connect to a FTP server to retrieve files and process them locally.

To use the ftplib module in Python, you first have to import it into your script.

## Open a Connection

To "open" a connection to the FTP Server, you have to create the object.

Once the connection is made (opened), you can use the methods in the ftplib module.

Several methods are available in two flavors: one for handling text files and another for binary files.

You can easily navigate the directory structure, manage and download files.

**PROGRAM:**

import ftplib

ftp = ftplib.FTP('ftp.sunet.se', 'anonymous', ‘anonymous@sunet.se')

print "File List: "

files = ftp.dir()

print files

ftp.cwd("/pub/unix") #changing to /pub/unix

## Common FTP Methods

###### FTP.connect(host[, port[, timeout]])

Connect to the given host and port.

The default port number is 21, as specified by the FTP protocol specification.

It is rarely needed to specify a different port number.

This function should be called only once for each instance

It should not be called at all if a host was given when the instance was created.

All other methods can only be used after a connection  
has been made.

The optional timeout parameter specifies a timeout in seconds for the connectionattempt.

If no timeout is passed, the global default timeout setting will be used.

###### FTP.getwelcome()

Return the welcome message sent by the server in reply to the initial connection.

This message sometimes contains disclaimers or help information that may be relevant to the user

###### FTP.login([user[, passwd[, acct]]])

Log in as the given user.

The passwd and acct parameters are optional and default to the empty string.

If no user is specified, it defaults to 'anonymous'.

If user is 'anonymous', the default passwd is 'anonymous@'.

This function should be called only once for each instance, after a connection has been established.

It should not be called at all if a host and user were given when the instance was created.

Most FTP commands are only allowed after the client has logged in.

The acct parameter supplies “accounting information”; few systems implement this.

###### FTP.retrbinary(command, callback[, maxblocksize[, rest]])

Retrieve a file in binary transfer mode.

Command should be an appropriate RETR command: 'RETR filename'.

The callback function is called for each block of data received, with a single string argument giving the data block.

The optional maxblocksize argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer.

A reasonable default is chosen. rest means the same thing as in the transfercmd() method.

###### FTP.retrlines(command[, callback])

Retrieve a file or directory listing in ASCII transfer mode.

Command should be an appropriate RETR command or a command such as LIST, NLST or MLSD.

LIST retrieves a list of files and information about those files.

NLST retrieves a list of file names.

On some servers, MLSD retrieves a machine readable list of files and information about those files.

The callback function is called for each line with a string argument containing the line with the trailing CRLF stripped.

The default callback prints the line to sys.stdout.

###### FTP.dir(argument[, ...])

Produce a directory listing as returned by the LIST command, printing it to standard output.

The optional argument is a directory to list (default is the current server directory).

Multiple arguments can be used to pass non-standard options to the LIST command.

If the last argument is a function, it is used as a callback function as for retrlines(); the default prints to sys.stdout.

This method returns None.

###### FTP.delete(filename)

Remove the file named filename from the server.

If successful, returns the text of the response, otherwise raises error\_perm on permission errors or error\_reply on other errors.

###### FTP.cwd(pathname)

Set the current directory on the server.

###### FTP.mkd(pathname)

Create a new directory on the server.

###### FTP.pwd()

Return the pathname of the current directory on the server.

###### FTP.quit()

Send a QUIT command to the server and close the connection.

This is the “polite” way to close a connection, but it may raise an exception if the server responds with an error to the QUIT command.

This implies a call to the close() method which renders the FTP instance useless for subsequent calls.

###### FTP.close()

Close the connection unilaterally.

This should not be applied to an already closed connection such as after a successful call to quit().

After this call the FTP instance should not be used any more.

After a call to close() or quit() you cannot reopen the connection by issuing another login() method).

# The urllib module

This module provides a unified client interface for HTTP, FTP, and gopher. It automatically picks the right protocol handler based on the uniform resource locator (URL) passed to the library.

Fetching data from an URL is extremely easy. Just call the **urlopen** method, and read from the returned stream object.

**Example: Using the urllib module to fetch a remote resource**

# File: [urllib-example-1.py](http://effbot.org/librarybook/urllib-example-1.py)

import urllib

fp = urllib.urlopen("http://www.python.org")

op = open("out.html", "wb")

n = 0

while 1:

s = fp.read(8192)

if not s:

break

op.write(s)

n = n + len(s)

fp.close()

op.close()

for k, v in fp.headers.items():

print k, "=", v

print "copied", n, "bytes from", fp.url

server = Apache/1.3.6 (Unix)

content-type = text/html

accept-ranges = bytes

date = Mon, 11 Oct 1999 20:11:40 GMT

connection = close

etag = "741e9-7870-37f356bf"

content-length = 30832

last-modified = Thu, 30 Sep 1999 12:25:35 GMT

copied 30832 bytes from http://www.python.org

Note that stream object provides some non-standard attributes. **headers** is a **Message** object (as defined by the **mimetools** module), and **url** contains the actual URL. The latter is updated if the server redirects the client to a new URL.

The **urlopen** function is actually a helper function, which creates an instance of the **FancyURLopener** class, and calls its **open** method. To get special behavior, you can subclass that class. For example, the following class automatically logs in to the server, when necessary:

**Example: Using the urllib module with automatic authentication**

# File: [urllib-example-3.py](http://effbot.org/librarybook/urllib-example-3.py)

import urllib

class myURLOpener(urllib.FancyURLopener):

# read an URL, with automatic HTTP authentication

def setpasswd(self, user, passwd):

self.\_\_user = user

self.\_\_passwd = passwd

def prompt\_user\_passwd(self, host, realm):

return self.\_\_user, self.\_\_passwd

urlopener = myURLOpener()

urlopener.setpasswd("mulder", "trustno1")

fp = urlopener.open("http://www.secretlabs.com")

print fp.read()

# Packaging Python Libraries

## Diving In[#](http://www.diveinto.org/python3/packaging.html#divingin)

Python 3 comes with a packaging framework called Distutils. Distutils is many things: a build tool (for you), an installation tool (for your users), a package metadata format (for search engines), and more. It integrates with the [Python Package Index](http://pypi.python.org/) (“PyPI”), a central repository for open source Python libraries.

All of these facets of Distutils center around the *setup script*, traditionally called setup.py.

⁂

## Directory Structure[#](http://www.diveinto.org/python3/packaging.html#structure)

To start packaging your Python software, you need to get your files and directories in order. The httplib2 directory looks like this:

httplib2/ ①

|

+--README.txt ②

|

+--setup.py ③

|

+--httplib2/ ④

|

+--\_\_init\_\_.py

|

+--iri2uri.py

1. Make a root directory to hold everything. Give it the same name as your Python module.
2. To accomodate Windows users, your “read me” file should include a .txt extension, and it should use Windows-style carriage returns. Just because you use a fancy text editor that runs from the command line and includes its own macro language, that doesn’t mean you need to make life difficult for your users. (Your users use Notepad. Sad but true.) Even if you’re on Linux or Mac OS X, your fancy text editor undoubtedly has an option to save files with Windows-style carriage returns.
3. Your Distutils setup script should be named setup.py unless you have a good reason not to. You do not have a good reason not to.
4. If your Python software is a single .py file, you should put it in the root directory along with your “read me” file and your setup script. But httplib2 is not a single .py file; it’s [a multi-file module](http://www.diveinto.org/python3/case-study-porting-chardet-to-python-3.html#multifile-modules). But that’s OK! Just put the httplib2 directory in the root directory, so you have an \_\_init\_\_.py file within an httplib2/ directory within the httplib2/ root directory. That’s not a problem; in fact, it will simplify your packaging process.

The chardet directory looks slightly different. Like httplib2, it’s [a multi-file module](http://www.diveinto.org/python3/case-study-porting-chardet-to-python-3.html#multifile-modules), so there’s a chardet/ directory within the chardet/ root directory. In addition to the README.txt file, chardet has HTML-formatted documentation in the docs/ directory. The docs/ directory contains several .html and .css files and an images/ subdirectory, which contains several .png and .gif files. (This will be important later.) Also, in keeping with the convention for (L)GPL-licensed software, it has a separate file called COPYING.txt which contains the complete text of the LGPL.

chardet/

|

+--COPYING.txt

|

+--setup.py

|

+--README.txt

|

+--docs/

| |

| +--index.html

| |

| +--usage.html

| |

| +--images/ ...

|

+--chardet/

|

+--\_\_init\_\_.py

|

+--big5freq.py

|

+--...

⁂

## Writing Your Setup Script[#](http://www.diveinto.org/python3/packaging.html#setuppy)

The Distutils setup script is a Python script. In theory, it can do anything Python can do. In practice, it should do as little as possible, in as standard a way as possible. Setup scripts should be boring. The more exotic your installation process is, the more exotic your bug reports will be.

The first line of every Distutils setup script is always the same:

from distutils.core import setup

This imports the setup() function, which is the main entry point into Distutils. 95% of all Distutils setup scripts consist of a single call to setup() and nothing else. (I totally just made up that statistic, but if your Distutils setup script is doing more than calling the Distutils setup() function, you should have a good reason. Do you have a good reason? I didn’t think so.)

The setup() function [can take dozens of parameters](http://docs.python.org/3.1/distutils/apiref.html#distutils.core.setup). For the sanity of everyone involved, you must use [named arguments](http://www.diveinto.org/python3/your-first-python-program.html#optional-arguments) for every parameter. This is not merely a convention; it’s a hard requirement. Your setup script will crash if you try to call the setup() function with non-named arguments.

The following named arguments are required:

* **name**, the name of the package.
* **version**, the version number of the package.
* **author**, your full name.
* **author\_email**, your email address.
* **url**, the home page of your project. This can be your [PyPI](http://pypi.python.org/) package page if you don’t have a separate project website.

Although not required, I recommend that you also include the following in your setup script:

* **description**, a one-line summary of the project.
* **long\_description**, a multi-line string in [reStructuredText format](http://docutils.sourceforge.net/rst.html). [PyPI](http://pypi.python.org/) converts this to HTML and displays it on your package page.
* **classifiers**, a list of specially-formatted strings described in the next section.

☞Setup script metadata is defined in [PEP 314](http://www.python.org/dev/peps/pep-0314/).

Now let’s look at the chardet setup script. It has all of these required and recommended parameters, plus one I haven’t mentioned yet: packages.

from distutils.core import setup  
setup(  
    name = 'chardet',  
    packages = ['chardet'],  
    version = '1.0.2',  
    description = 'Universal encoding detector',  
    author='Mark Pilgrim',  
    ...  
)

The packages parameter highlights an unfortunate vocabulary overlap in the distribution process. We’ve been talking about the “package” as the thing you’re building (and potentially listing in The Python “Package” Index). But that’s not what this packages parameter refers to. It refers to the fact that the chardet module is [a multi-file module](http://www.diveinto.org/python3/case-study-porting-chardet-to-python-3.html#multifile-modules), sometimes known as… a “package.” The packages parameter tells Distutils to include the chardet/ directory, its \_\_init\_\_.py file, and all the other .py files that constitute the chardet module. That’s kind of important; all this happy talk about documentation and metadata is irrelevant if you forget to include the actual code!

⁂

## Classifying Your Package[#](http://www.diveinto.org/python3/packaging.html#trove)

The Python Package Index (“PyPI”) contains thousands of Python libraries. Proper classification metadata will allow people to find yours more easily. PyPI lets you [browse packages by classifier](http://pypi.python.org/pypi?:action=browse). You can even select multiple classifiers to narrow your search. Classifiers are not invisible metadata that you can just ignore!

To classify your software, pass a classifiers parameter to the Distutils setup() function. The classifiers parameter is a list of strings. These strings are not freeform. All classifier strings should come from [this list on PyPI](http://pypi.python.org/pypi?:action=list_classifiers).

Classifiers are optional. You can write a Distutils setup script without any classifiers at all. **Don’t do that.** You should always include at least these classifiers:

* **Programming Language**. In particular, you should include both "Programming Language :: Python" and "Programming Language :: Python :: 3". If you do not include these, your package will not show up in [this list of Python 3-compatible libraries](http://pypi.python.org/pypi?:action=browse&c=533&show=all), which linked from the sidebar of every single page of pypi.python.org.
* **License**. This is the absolute first thing I look for when I’m evaluating third-party libraries. Don’t make me hunt for this vital information. Don’t include more than one license classifier unless your software is explicitly available under multiple licenses. (And don’t release software under multiple licenses unless you’re forced to do so. And don’t force other people to do so. Licensing is enough of a headache; don’t make it worse.)
* **Operating System**. If your software only runs on Windows (or Mac OS X, or Linux), I want to know sooner rather than later. If your software runs anywhere without any platform-specific code, use the classifier "Operating System :: OS Independent". Multiple Operating System classifiers are only necessary if your software requires specific support for each platform. (This is not common.)

I also recommend that you include the following classifiers:

* **Development Status**. Is your software beta quality? Alpha quality? Pre-alpha? Pick one. Be honest.
* **Intended Audience**. Who would download your software? The most common choices are Developers, End Users/Desktop, Science/Research, and System Administrators.
* **Framework**. If your software is a plugin for a larger Python framework like [Django](http://www.djangoproject.com/) or [Zope](http://www.zope.org/), include the appropriate Framework classifier. If not, omit it.
* **Topic**. There are [a large number of topics to choose from](http://pypi.python.org/pypi?:action=list_classifiers); choose all that apply.

## Specifying Additional Files With A Manifest[#](http://www.diveinto.org/python3/packaging.html#manifest)

By default, Distutils will include the following files in your release package:

* README.txt
* setup.py
* The .py files needed by the multi-file modules listed in the packages parameter
* The individual .py files listed in the py\_modules parameter

That will cover [all the files in the httplib2 project](http://www.diveinto.org/python3/packaging.html#structure). But for the chardet project, we also want to include the COPYING.txt license file and the entire docs/ directory that contains images and HTML files. To tell Distutils to include these additional files and directories when it builds the chardet release package, you need a *manifest file*.

A manifest file is a text file called MANIFEST.in. Place it in the project’s root directory, next to README.txt and setup.py. Manifest files are not Python scripts; they are text files that contain a series of “commands” in a Distutils-defined format. Manifest commands allow you to include or exclude specific files and directories.

This is the entire manifest file for the chardet project:

include COPYING.txt ①

recursive-include docs \*.html \*.css \*.png \*.gif ②

1. The first line is self-explanatory: include the COPYING.txt file from the project’s root directory.
2. The second line is a bit more complicated. The recursive-include command takes a directory name and one or more filenames. The filenames aren’t limited to specific files; they can include wildcards. This line means “See that docs/ directory in the project’s root directory? Look in there (recursively) for .html, .css, .png, and .gif files. I want all of them in my release package.”

All manifest commands preserve the directory structure that you set up in your project directory. That recursive-include command is not going to put a bunch of .html and .png files in the root directory of the release package. It’s going to maintain the existing docs/ directory structure, but only include those files inside that directory that match the given wildcards. (I didn’t mention it earlier, but the chardet documentation is actually written in XML and converted to HTML by a separate script. I don’t want to include the XML files in the release package, just the HTML and the images.)

☞Manifest files have their own unique format. See [Specifying the files to distribute](http://docs.python.org/3.1/distutils/sourcedist.html#manifest) and [the manifest template commands](http://docs.python.org/3.1/distutils/commandref.html#sdist-cmd) for details.

To reiterate: you only need to create a manifest file if you want to include files that Distutils doesn’t include by default. If you do need a manifest file, it should only include the files and directories that Distutils wouldn’t otherwise find on its own.

## Checking Your Setup Script for Errors[#](http://www.diveinto.org/python3/packaging.html#check)

There’s a lot to keep track of. Distutils comes with a built-in validation command that checks that all the required metadata is present in your setup script. For example, if you forget to include the version parameter, Distutils will remind you.

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check running check

warning: check: missing required meta-data: version

Once you include a version parameter (and all the other required bits of metadata), the check command will look like this:

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check running check

⁂

## Creating a Source Distribution[#](http://www.diveinto.org/python3/packaging.html#sdist)

Distutils supports building multiple types of release packages. At a minimum, you should build a “source distribution” that contains your source code, your Distutils setup script, your “read me” file, and whatever [additional files you want to include](http://www.diveinto.org/python3/packaging.html#manifest). To build a source distribution, pass the sdist command to your Distutils setup script.

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py sdist

running sdist

running check

reading manifest template 'MANIFEST.in'

writing manifest file 'MANIFEST'

creating chardet-1.0.2

creating chardet-1.0.2\chardet

creating chardet-1.0.2\docs

creating chardet-1.0.2\docs\images

copying files to chardet-1.0.2...

copying COPYING -> chardet-1.0.2

copying README.txt -> chardet-1.0.2

copying setup.py -> chardet-1.0.2

copying chardet\\_\_init\_\_.py -> chardet-1.0.2\chardet

copying chardet\big5freq.py -> chardet-1.0.2\chardet

...

copying chardet\universaldetector.py -> chardet-1.0.2\chardet

copying chardet\utf8prober.py -> chardet-1.0.2\chardet

copying docs\faq.html -> chardet-1.0.2\docs

copying docs\history.html -> chardet-1.0.2\docs

copying docs\how-it-works.html -> chardet-1.0.2\docs

copying docs\index.html -> chardet-1.0.2\docs

copying docs\license.html -> chardet-1.0.2\docs

copying docs\supported-encodings.html -> chardet-1.0.2\docs

copying docs\usage.html -> chardet-1.0.2\docs

copying docs\images\caution.png -> chardet-1.0.2\docs\images

copying docs\images\important.png -> chardet-1.0.2\docs\images

copying docs\images\note.png -> chardet-1.0.2\docs\images

copying docs\images\permalink.gif -> chardet-1.0.2\docs\images

copying docs\images\tip.png -> chardet-1.0.2\docs\images

copying docs\images\warning.png -> chardet-1.0.2\docs\images

creating dist

creating 'dist\chardet-1.0.2.zip' and adding 'chardet-1.0.2' to it

adding 'chardet-1.0.2\COPYING'

adding 'chardet-1.0.2\PKG-INFO'

adding 'chardet-1.0.2\README.txt'

adding 'chardet-1.0.2\setup.py'

adding 'chardet-1.0.2\chardet\big5freq.py'

adding 'chardet-1.0.2\chardet\big5prober.py'

...

adding 'chardet-1.0.2\chardet\universaldetector.py'

adding 'chardet-1.0.2\chardet\utf8prober.py'

adding 'chardet-1.0.2\chardet\\_\_init\_\_.py'

adding 'chardet-1.0.2\docs\faq.html'

adding 'chardet-1.0.2\docs\history.html'

adding 'chardet-1.0.2\docs\how-it-works.html'

adding 'chardet-1.0.2\docs\index.html'

adding 'chardet-1.0.2\docs\license.html'

adding 'chardet-1.0.2\docs\supported-encodings.html'

adding 'chardet-1.0.2\docs\usage.html'

adding 'chardet-1.0.2\docs\images\caution.png'

adding 'chardet-1.0.2\docs\images\important.png'

adding 'chardet-1.0.2\docs\images\note.png'

adding 'chardet-1.0.2\docs\images\permalink.gif'

adding 'chardet-1.0.2\docs\images\tip.png'

adding 'chardet-1.0.2\docs\images\warning.png'

removing 'chardet-1.0.2' (and everything under it)

Several things to note here:

* Distutils noticed the manifest file (MANIFEST.in).
* Distutils successfully parsed the manifest file and added the additional files we wanted — COPYING.txt and the HTML and image files in the docs/ directory.
* If you look in your project directory, you’ll see that Distutils created a dist/ directory. Within the dist/ directory the .zip file that you can distribute.

c:\Users\pilgrim\chardet> dir dist

Volume in drive C has no label.

Volume Serial Number is DED5-B4F8

Directory of c:\Users\pilgrim\chardet\dist

07/30/2009 06:29 PM <DIR> .

07/30/2009 06:29 PM <DIR> ..

07/30/2009 06:29 PM 206,440 chardet-1.0.2.zip

1 File(s) 206,440 bytes

2 Dir(s) 61,424,635,904 bytes free

⁂

## Creating a Graphical Installer[#](http://www.diveinto.org/python3/packaging.html#bdist)

In my opinion, every Python library deserves a graphical installer for Windows users. It’s easy to make (even if you don’t run Windows yourself), and Windows users appreciate it.

Distutils can [create a graphical Windows installer for you](http://docs.python.org/3.1/distutils/builtdist.html#creating-windows-installers), by passing the bdist\_wininst command to your Distutils setup script.

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py bdist\_wininst running bdist\_wininst

running build

running build\_py

creating build

creating build\lib

creating build\lib\chardet

copying chardet\big5freq.py -> build\lib\chardet

copying chardet\big5prober.py -> build\lib\chardet

...

copying chardet\universaldetector.py -> build\lib\chardet

copying chardet\utf8prober.py -> build\lib\chardet

copying chardet\\_\_init\_\_.py -> build\lib\chardet

installing to build\bdist.win32\wininst

running install\_lib

creating build\bdist.win32

creating build\bdist.win32\wininst

creating build\bdist.win32\wininst\PURELIB

creating build\bdist.win32\wininst\PURELIB\chardet

copying build\lib\chardet\big5freq.py -> build\bdist.win32\wininst\PURELIB\chardet

copying build\lib\chardet\big5prober.py -> build\bdist.win32\wininst\PURELIB\chardet

...

copying build\lib\chardet\universaldetector.py -> build\bdist.win32\wininst\PURELIB\chardet

copying build\lib\chardet\utf8prober.py -> build\bdist.win32\wininst\PURELIB\chardet

copying build\lib\chardet\\_\_init\_\_.py -> build\bdist.win32\wininst\PURELIB\chardet

running install\_egg\_info

Writing build\bdist.win32\wininst\PURELIB\chardet-1.0.2-py3.1.egg-info

creating 'c:\users\pilgrim\appdata\local\temp\tmp2f4h7e.zip' and adding '.' to it

adding 'PURELIB\chardet-1.0.2-py3.1.egg-info'

adding 'PURELIB\chardet\big5freq.py'

adding 'PURELIB\chardet\big5prober.py'

...

adding 'PURELIB\chardet\universaldetector.py'

adding 'PURELIB\chardet\utf8prober.py'

adding 'PURELIB\chardet\\_\_init\_\_.py'

removing 'build\bdist.win32\wininst' (and everything under it)

c:\Users\pilgrim\chardet> dir dist

c:\Users\pilgrim\chardet>dir dist

Volume in drive C has no label.

Volume Serial Number is AADE-E29F

Directory of c:\Users\pilgrim\chardet\dist

07/30/2009 10:14 PM <DIR> .

07/30/2009 10:14 PM <DIR> ..

07/30/2009 10:14 PM 371,236 chardet-1.0.2.win32.exe

07/30/2009 06:29 PM 206,440 chardet-1.0.2.zip

2 File(s) 577,676 bytes

2 Dir(s) 61,424,070,656 bytes free

## Adding Your Software to The Python Package Index[#](http://www.diveinto.org/python3/packaging.html#pypi)

Uploading software to the Python Package Index is a three step process.

1. Register yourself
2. Register your software
3. Upload the packages you created with setup.py sdist and setup.py bdist\_\*

To register yourself, go to [the PyPI user registration page](http://pypi.python.org/pypi?:action=register_form). Enter your desired username and password, provide a valid email address, and click the Register button. (If you have a PGP or GPG key, you can also provide that. If you don’t have one or don’t know what that means, don’t worry about it.) Check your email; within a few minutes, you should receive a message from PyPI with a validation link. Click the link to complete the registration process.

Now you need to register your software with PyPI and upload it. You can do this all in one step.

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist bdist\_wininst upload ①

running register

We need to know who you are, so please choose either:

1. use your existing login,

2. register as a new user,

3. have the server generate a new password for you (and email it to you), or

4. quit

Your selection [default 1]: 1 ②

Username: MarkPilgrim ③

Password:

Registering chardet to http://pypi.python.org/pypi ④

Server response (200): OK

running sdist ⑤

... output trimmed for brevity ...

running bdist\_wininst ⑥

... output trimmed for brevity ...

running upload ⑦

Submitting dist\chardet-1.0.2.zip to http://pypi.python.org/pypi

Server response (200): OK

Submitting dist\chardet-1.0.2.win32.exe to http://pypi.python.org/pypi

Server response (200): OK

I can store your PyPI login so future submissions will be faster.

(the login will be stored in c:\home\.pypirc)

Save your login (y/N)?n ⑧

1. When you release your project for the first time, Distutils will add your software to the Python Package Index and give it its own URL. Every time after that, it will simply update the project metadata with any changes you may have made in your setup.py parameters. Next, it builds a source distribution (sdist) and a Windows installer (bdist\_wininst), then uploads them to PyPI (upload).
2. Type 1 or just press ENTER to select “use your existing login.”
3. Enter the username and password you selected on the [the PyPI user registration page](http://pypi.python.org/pypi?:action=register_form). Distuils will not echo your password; it will not even echo asterisks in place of characters. Just type your password and press ENTER.
4. Distutils registers your package with the Python Package Index…
5. …builds your source distribution…
6. …builds your Windows installer…
7. …and uploads them both to the Python Package Index.
8. If you want to automate the process of releasing new versions, you need to save your PyPI credentials in a local file. This is completely insecure and completely optional.

Congratulations, you now have your own page on the Python Package Index! The address is http://pypi.python.org/pypi/*NAME*, where *NAME* is the string you passed in the name parameter in your setup.py file.

If you want to release a new version, just update your setup.py with the new version number, then run the same upload command again:

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist bdist\_wininst upload

⁂

**NETWORK PROGRAMMING**

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

This tutorial gives you understanding on most famous concept in Networking - Socket Programming

## What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary:

|  |  |
| --- | --- |
| **Term** | **Description** |
| domain | The family of protocols that will be used as the transport mechanism. These values are constants such as AF\_INET, PF\_INET, PF\_UNIX, PF\_X25, and so on. |
| type | The type of communications between the two endpoints, typically SOCK\_STREAM for connection-oriented protocols and SOCK\_DGRAM for connectionless protocols. |
| protocol | Typically zero, this may be used to identify a variant of a protocol within a domain and type. |
| hostname | The identifier of a network interface:   * A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation * A string "<broadcast>", which specifies an INADDR\_BROADCAST address. * A zero-length string, which specifies INADDR\_ANY, or * An Integer, interpreted as a binary address in host byte order. |
| port | Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service. |

## The *socket* Module:

To create a socket, you must use the *socket.socket()* function available in *socket* module, which has the general syntax:

s = socket.socket (socket\_family, socket\_type, protocol=0)

Here is the description of the parameters:

* **socket\_family:** This is either AF\_UNIX or AF\_INET, as explained earlier.
* **socket\_type:** This is either SOCK\_STREAM or SOCK\_DGRAM.
* **protocol:** This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use required functions to create your client or server program. Following is the list of functions required:

## Server Socket Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.bind() | This method binds address (hostname, port number pair) to socket. |
| s.listen() | This method sets up and start TCP listener. |
| s.accept() | This passively accept TCP client connection, waiting until connection arrives (blocking). |

## Client Socket Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.connect() | This method actively initiates TCP server connection. |

## General Socket Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.recv() | This method receives TCP message |
| s.send() | This method transmits TCP message |
| s.recvfrom() | This method receives UDP message |
| s.sendto() | This method transmits UDP message |
| s.close() | This method closes socket |
| socket.gethostname() | Returns the hostname. |

## A Simple Server:

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call **bind(hostname, port** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

#!/usr/bin/python # This is server.py file

import socket # Import socket module

s = socket.socket() # Create a socket object

host = socket.gethostname() # Get local machine name

port = 12345 # Reserve a port for your service.

s.bind((host, port)) # Bind to the port

s.listen(5) # Now wait for client connection.

while True:

c, addr = s.accept() # Establish connection with client.

print 'Got connection from', addr

c.send('Thank you for connecting')

c.close() # Close the connection

## A Simple Client:

Now we will write a very simple client program which will open a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function.

The **socket.connect(hosname, port )** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits:

#!/usr/bin/python # This is client.py file

import socket # Import socket module

s = socket.socket() # Create a socket object

host = socket.gethostname() # Get local machine name

port = 12345 # Reserve a port for your service.

s.connect((host, port))

print s.recv(1024)

s.close # Close the socket when done

Now run this server.py in background and then run above client.py to see the result.

# Following would start a server in background.

$ python server.py &

# Once server is started run client as follows:

$ python client.py

This would produce following result:

Got connection from ('127.0.0.1', 48437)

Thank you for connecting

## Python Internet modules

A list of some important modules which could be used in Python Network/Internet programming.

|  |  |  |  |
| --- | --- | --- | --- |
| **Protocol** | **Common function** | **Port No** | **Python module** |
| HTTP | Web pages | 80 | httplib, urllib, xmlrpclib |
| NNTP | Usenet news | 119 | nntplib |
| FTP | File transfers | 20 | ftplib, urllib |
| SMTP | Sending email | 25 | smtplib |
| POP3 | Fetching email | 110 | poplib |
| IMAP4 | Fetching email | 143 | imaplib |
| Telnet | Command lines | 23 | telnetlib |
| Gopher | Document transfers | 70 | gopherlib, urllib |

Please check all the libraries mentioned above to work with FTP, SMTP, POP, and IMAP protocols

Running several threads is similar to running several different programs concurrently, but with the following benefits:

* Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
* Threads sometimes called light-weight processes and they do not require much memory overhead; they care cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

* It can be pre-empted (interrupted)
* It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

## Starting a New Thread:

To spawn another thread, you need to call following method available in *thread* module:

thread.start\_new\_thread ( function, args[, kwargs] )

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of *agrs*. When function returns, the thread terminates.

Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

### Example:

#!/usr/bin/python

import thread

import time

# Define a function for the thread

def print\_time( threadName, delay):

count = 0

while count < 5:

time.sleep(delay)

count += 1

print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows

try:

thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )

thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )

except:

print "Error: unable to start thread"

while 1:

pass

When the above code is executed, it produces the following result:

Thread-1: Thu Jan 22 15:42:17 2009

Thread-1: Thu Jan 22 15:42:19 2009

Thread-2: Thu Jan 22 15:42:19 2009

Thread-1: Thu Jan 22 15:42:21 2009

Thread-2: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:25 2009

Thread-2: Thu Jan 22 15:42:27 2009

Thread-2: Thu Jan 22 15:42:31 2009

Thread-2: Thu Jan 22 15:42:35 2009

Although it is very effective for low-level threading, but the *thread* module is very limited compared to the newer threading module.

## The *Threading* Module:

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

* **threading.activeCount():** Returns the number of thread objects that are active.
* **threading.currentThread():** Returns the number of thread objects in the caller's thread control.
* **threading.enumerate():** Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

* **run():** The run() method is the entry point for a thread.
* **start():** The start() method starts a thread by calling the run method.
* **join([time]):** The join() waits for threads to terminate.
* **isAlive():** The isAlive() method checks whether a thread is still executing.
* **getName():** The getName() method returns the name of a thread.
* **setName():** The setName() method sets the name of a thread.

## Creating Thread using *Threading* Module:

To implement a new thread using the threading module, you have to do the following:

* Define a new subclass of the *Thread* class.
* Override the *\_\_init\_\_(self [,args])* method to add additional arguments.
* Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which will in turn call *run()* method.

### Example:

#!/usr/bin/python

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print "Starting " + self.name

print\_time(self.name, self.counter, 5)

print "Exiting " + self.name

def print\_time(threadName, delay, counter):

while counter:

if exitFlag:

thread.exit()

time.sleep(delay)

print "%s: %s" % (threadName, time.ctime(time.time()))

counter -= 1

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

print "Exiting Main Thread"

When the above code is executed, it produces the following result:

Starting Thread-1

Starting Thread-2

Exiting Main Thread

Thread-1: Thu Mar 21 09:10:03 2013

Thread-1: Thu Mar 21 09:10:04 2013

Thread-2: Thu Mar 21 09:10:04 2013

Thread-1: Thu Mar 21 09:10:05 2013

Thread-1: Thu Mar 21 09:10:06 2013

Thread-2: Thu Mar 21 09:10:06 2013

Thread-1: Thu Mar 21 09:10:07 2013

Exiting Thread-1

Thread-2: Thu Mar 21 09:10:08 2013

Thread-2: Thu Mar 21 09:10:10 2013

Thread-2: Thu Mar 21 09:10:12 2013

Exiting Thread-2

## Synchronizing Threads:

The threading module provided with Python includes a simple-to-implement locking mechanism that will allow you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object would be used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread will wait to acquire the lock.

If *blocking* is set to 0, the thread will return immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread will block and wait for the lock to be released.

The *release()* method of the the new lock object would be used to release the lock when it is no longer required.

### Example:

#!/usr/bin/python

import threading

import time

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print "Starting " + self.name

# Get lock to synchronize threads

threadLock.acquire()

print\_time(self.name, self.counter, 3)

# Free lock to release next thread

threadLock.release()

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print "%s: %s" % (threadName, time.ctime(time.time()))

counter -= 1

threadLock = threading.Lock()

threads = []

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

# Add threads to thread list

threads.append(thread1)

threads.append(thread2)

# Wait for all threads to complete

for t in threads:

t.join()

print "Exiting Main Thread"

When the above code is executed, it produces the following result:

Starting Thread-1

Starting Thread-2

Thread-1: Thu Mar 21 09:11:28 2013

Thread-1: Thu Mar 21 09:11:29 2013

Thread-1: Thu Mar 21 09:11:30 2013

Thread-2: Thu Mar 21 09:11:32 2013

Thread-2: Thu Mar 21 09:11:34 2013

Thread-2: Thu Mar 21 09:11:36 2013

Exiting Main Thread

## Multithreaded Priority Queue:

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue:

* **get():** The get() removes and returns an item from the queue.
* **put():** The put adds item to a queue.
* **qsize() :** The qsize() returns the number of items that are currently in the queue.
* **empty():** The empty( ) returns True if queue is empty; otherwise, False.
* **full():** the full() returns True if queue is full; otherwise, False.

### Example:

#!/usr/bin/python

import Queue

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, q):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.q = q

def run(self):

print "Starting " + self.name

process\_data(self.name, self.q)

print "Exiting " + self.name

def process\_data(threadName, q):

while not exitFlag:

queueLock.acquire()

if not workQueue.empty():

data = q.get()

queueLock.release()

print "%s processing %s" % (threadName, data)

else:

queueLock.release()

time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]

nameList = ["One", "Two", "Three", "Four", "Five"]

queueLock = threading.Lock()

workQueue = Queue.Queue(10)

threads = []

threadID = 1

# Create new threads

for tName in threadList:

thread = myThread(threadID, tName, workQueue)

thread.start()

threads.append(thread)

threadID += 1

# Fill the queue

queueLock.acquire()

for word in nameList:

workQueue.put(word)

queueLock.release()

# Wait for queue to empty

while not workQueue.empty():

pass

# Notify threads it's time to exit

exitFlag = 1

# Wait for all threads to complete

for t in threads:

t.join()

print "Exiting Main Thread"

When the above code is executed, it produces the following result:

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three

Thread-1 processing Four

Thread-2 processing Five

Exiting Thread-3

Exiting Thread-1

Exiting Thread-2

Exiting Main Thread

**High Concurrency with Python and Oracle Database**

**Learn how to improve the throughput and responsiveness of Oracle Database-backed Python applications with the help of threading and concurrency.**

With the trend toward more, rather than faster, cores, exploiting concurrency is increasing in importance. Concurrency creates a new paradigm shift in programming, making it possible to write asynchronous code that separates tasks into a set of threads or processes working in parallel. If you are not new to programming and are somewhat familiar with C or C++, you probably already have some idea of threads and processes and know how they differ from each other: threads provide a lightweight alternative to processes when it comes to concurrent programming, which makes multithreading preferable to multiprocessing in most cases. This article therefore discusses the multithreaded approach to concurrency.

As with many other programming languages, separating CPU-intensive tasks into multiple processes in Python (this can be done using the multiprocessing module of the Python standard librarymay give you some performance benefits when utilizing a multiple-CPU machine. It can really run multiple operations in parallel, as opposed to just switching between tasks while performing only one task at any given time, which is true for single-processor machines. In contrast, you may not notice any performance improvement when moving a multithreaded Python program to a multiple-CPU machine, because of the global interpreter lock (GIL), which is used by Python to protect internal data structures, ensuring that only one thread at a time runs the CPython virtual machine.

However, you may still be interested in adding threads to a database-backed Python program to speed it up. The point is that the underlying database your Python program interacts with is most likely installed on a high-performance server that processes submitted queries in parallel. This means that you can benefit from submitting several queries to the database server in separate threads in parallel, rather than issuing them sequentially, one after another, in a single thread.

One word of warning: although utilizing the natural parallelism of tasks may significantly improve application performance, it’s important to realize that not all tasks can be performed in parallel. For example, you cannot issue confirmation e-mails to customers until the operation they requested—say, a money transfer—is complete. It’s fairly obvious that such tasks must be performed sequentially in a certain order.

Another important thing to keep in mind when building multithreaded code is that some threads running in parallel may try to change shared objects at the same time, which may lead to loss of data, data malformation, or even corruption of the object being changed. To avoid this problem, it would be helpful to control access to shared objects so that only one thread can use such an object at once. Fortunately, Python lets you implement a locking mechanism to synchronize access to shared objects utilizing locking tools available in the thread module.

The downside to using locks is that it hurts scalability. When designing for scalability, it’s important to remember that putting a lock on a resource within one thread makes that resource unavailable in all the other running threads and processes until the lock is released. Thus, to ensure efficient resource management, you should not overuse locks, avoiding them whenever possible or releasing them as soon as possible when they are required.

Luckily, you don’t need to worry about locking when you’re dealing with resources stored in an Oracle database. This is because Oracle Database uses its own locking mechanisms behind the scenes when it comes to providing access to shared data in concurrent environments. So it’s often a good idea to keep shared data in the database, thus letting Oracle Database take care of the concurrency problems.

Another good way to achieve scalability and benefit from concurrency is to perform operations asynchronously. In asynchronous programming, the blocking code is queued for completion later by a separate thread, enabling your application to proceed with other tasks. Using an asynchronous framework such as Twisted may greatly simplify the task of building asynchronous applications.

This article provides a brief introduction to building concurrent applications with Python and Oracle Database, describing how to utilize threads in Python code interacting with an Oracle database and explaining how to submit SQL queries to the database server in parallel instead of processing them sequentially. You will also learn how to make Oracle Database take care of concurrency problems as well as how to take advantage of Twisted, a Python event-driven framework.

### Multithreaded Programming in Python

Threads are a very useful feature when it comes to parallel processing. If you have a program that is performing time-consuming operations and can divide it into several independent tasks to be performed in parallel, using threads can help you build more-efficient, faster code. Another interesting use of multithreading can be to improve the responsiveness of your application—the main program remains responsive while time-consuming operations are performed in the background.

Enclosing long-running SQL statements in separate threads in Python can be a good idea when these statements do not depend on each other and can be executed in parallel. For example, you might significantly reduce the loading time of a Web page if it is submitting initial SQL queries to the database server in parallel instead of processing them sequentially, thus making them wait for one another.

Another situation in which you might find threads useful is if you have to upload some large objects (LOBs) into the database. Doing this in parallel can not only reduce the overall time it takes to upload your LOBs into the database but can also keep the main thread of the program responsive while parallel uploading is happening in the background.

Suppose you need to upload a couple of binary large objects (BLOBs) to the database and save them to the blob\_tab table that you might have created in a custom database schema, as follows:

CREATE TABLE blob\_tab(

id NUMBER PRIMARY KEY,

blobdoc BLOB

);

CREATE SEQUENCE blob\_seq;

First, let’s look at how you might store BLOBs into the blob\_tab table one after another without utilizing threads. The following is the Python script that serves this purpose, persisting two input images obtained by use of the filename and URL, respectively. The example assumes that you have created a blob\_tab table and a blob\_seq sequence in a usr/pswd custom database schema:

#File: singlethread.py

#Storing BLOBs in a single thread sequentially, one after another

import cx\_Oracle

from urllib import urlopen

inputs = []

#if you’re a Windows user, the path could be 'c:/temp/figure1.bmp'

inputs.append(open('/tmp/figure1.bmp', 'rb'))

inputs.append(urlopen('http://localhost/mypictures/figure2.bmp', 'rb'))

#obtaining a connection and predefining a memory area for a BLOB

dbconn = cx\_Oracle.connect('usr', 'pswd', '127.0.0.1/XE')

dbconn.autocommit = True

cur = dbconn.cursor()

cur.setinputsizes(blobdoc=cx\_Oracle.BLOB)

#executing INSERT statements saving BLOBs to the database

for input in inputs:

blobdoc = input.read()

cur.execute("INSERT INTO blob\_tab (ID, BLOBDOC) VALUES(blob\_seq.NEXTVAL, :blobdoc)", {'blobdoc':blobdoc})

input.close()

dbconn.close()

Although the task of obtaining and storing figure1.bmp and the similar task for figure2.bmp are going on one after another here, these tasks, as you might guess, are not actually sequentially dependent. So you might refactor the above code so that it reads and stores each image within a separate thread, thus improving performance through parallel processing. It’s important to note that in this particular case, you won’t need to coordinate the threads running in parallel, which significantly simplifies coding.

The following example shows how you might rewrite the above script to use threads, utilizing an object-oriented approach. In particular, it illustrates how you can extend the Thread class from the threading module, customizing it for a particular task.

#File: multithread.py

#Storing BLOBs in separate threads in parallel

import cx\_Oracle

import threading

from urllib import urlopen

#subclass of threading.Thread

class AsyncBlobInsert(threading.Thread):

def \_\_init\_\_(self, cur, input):

threading.Thread.\_\_init\_\_(self)

self.cur = cur

self.input = input

def run(self):

blobdoc = self.input.read()

self.cur.execute("INSERT INTO blob\_tab (ID, BLOBDOC) VALUES(blob\_seq.NEXTVAL, :blobdoc)", {'blobdoc':blobdoc})

self.input.close()

self.cur.close()

#main thread starts here

inputs = []

inputs.append(open('/tmp/figure1.bmp', 'rb'))

inputs.append(urlopen('http://localhost/\_figure2.bmp', 'rb'))

dbconn = cx\_Oracle.connect('usr', 'pswd', '127.0.0.1/XE',threaded=True)

dbconn.autocommit = True

for input in inputs:

cur = dbconn.cursor()

cur.setinputsizes(blobdoc=cx\_Oracle.BLOB)

th = AsyncBlobInsert(cur, input)

th.start()

In the above code, note the use of the threaded attribute passed to the cx\_Oracle.connect method as a parameter. By setting it to true, you instruct Oracle Database to use the OCI\_THREADED mode, also known as the threaded mode, thus specifying that the application is running in a multithreaded environment. It’s interesting to note here that using the threaded mode with a single-threaded application would not be a good idea. According to the cx\_Oracle documentation, turning the threaded parameter to true in a single-threaded application may impose a performance penalty of 10 percent to 15 percent.

In this example, you share a single connection between two threads, creating a separate cursor object for each thread, though. The operation of reading and then inserting a BLOB into the database is implemented here within the overridden run method of the AsyncBlobInsert custom subclass of the threading.Thread standard Python class. So all you need to do to start uploading a BLOB in a separate thread is to create an AsyncBlobInsert instance and then call its start method.

There is one problem with the script discussed here. When executed, it will not wait until the threads being launched are completed—the main thread will finish after launching the child threads, not waiting for their completion. If this is not the desired behavior and you want the program to complete only when all the threads have been completed, you might call the join method of each AsyncBlobInsert instance at the end of the script. This will block the main thread, making it wait for completion of the child threads. Here is how you might modify the preceding script so that it waits for all the threads launched in the for loop to finish:

...

th = []

for i, input in enumerate(inputs):

cur = dbconn.cursor()

cur.setinputsizes(blobdoc=cx\_Oracle.BLOB)

th.append(AsyncBlobInsert(cur, input))

th[i].start()

#main thread waits until all child threads are done

for t in th:

t.join()

The next section provides an example in which forcing the main thread to wait for completion of the child threads is required.

### Synchronizing Access to Shared Resources

The preceding example showed a multithreaded Python application dealing with a couple of tasks that did not depend on each other and thus could be easily decoupled and put into separate threads to be processed in parallel. In practice, though, you often have to deal with operations that depend on each other and require synchronization at some point.

Being part of a single process, threads share the same global memory and therefore can pass information to each other through shared resources such as variables, class instances, streams, and files. However, this simple way of exchanging information between threads comes at a price—you really need to be careful when it comes to modifying an object that can be accessed and/or modified in another thread at the same time. So it would be useful to be able to avoid collisions, employing a mechanism for synchronizing access to shared data.

To help with this problem, Python lets you allocate a lock that can be then acquired by a thread to ensure exclusive access to the data structures you work with in that thread. The threading module comes with the Lock method, which you can use to allocate a lock. Be warned, though, that a lock allocated with the threading.Lock method is initially in an unlocked state. To lock an allocated lock, you have to explicitly call the acquire method of that lock object. After that, you can perform operations on the objects that need locking. For example, you might have to use a lock when writing to the stdout standard output stream in a thread, to avoid potential overlap with the other threads working with stdout. Once you’ve done that, you have to release the lock with the release method of the lock object, thus making the released data structures available for further processing in other threads.

An interesting thing about locks is that they are not bound to a single thread. A lock allocated within one thread can then be acquired by another and released by a third thread. The following script represents a simple example of locks in action. Here you allocate a lock in the main thread to subsequently utilize it in the child threads, acquiring it before writing to a DOM document and then immediately releasing it.

#File: synchmultithread.py

#Using locks for synchronization in a multithreaded script

import sys

import cx\_Oracle

import threading

from xml.dom.minidom import parseString

from urllib import urlopen

#subclass of threading.Thread

class SynchThread(threading.Thread):

def \_\_init\_\_(self, cur, query, dom):

threading.Thread.\_\_init\_\_(self)

self.cur = cur

self.query = query[1]

self.tag = query[0]

self.dom = dom

def run(self):

self.cur.execute(self.query)

rslt = self.cur.fetchone()[0]

self.cur.close()

mutex.acquire()

sal = self.dom.getElementsByTagName('salary')[0]

newtag = self.dom.createElement(self.tag)

newtext = self.dom.createTextNode('%s'%rslt)

newtag.appendChild(newtext)

sal.appendChild(newtag)

mutex.release()

#main thread starts here

domdoc = parseString('<employees><salary/></employees>')

dbconn = cx\_Oracle.connect('hr', 'hr', '127.0.0.1/XE',threaded=True)

mutex = threading.Lock()

queries = {}

queries['avg'] = "SELECT AVG(salary) FROM employees"

queries['max'] = "SELECT MAX(salary) FROM employees"

th = []

for i, query in enumerate(queries.items()):

cur = dbconn.cursor()

th.append(SynchThread(cur, query, domdoc))

th[i].start()

#forcing the main thread to wait until all child threads are done

for t in th:

t.join()

#printing out the result xml document

domdoc.writexml(sys.stdout)

In the above script, you first create a Document Object Model (DOM) document object in the main thread and then modify this document in the child threads running in parallel, adding tags containing information obtained from the database. Here you use two simple queries against the employees table in the HR demonstration schema. To avoid potential collisions during parallel writing to the DOM object, you, in each child thread, acquire the lock allocated in the main thread. Once the lock is acquired by one child thread, the other one will not be able to modify the DOM object processed here until the first thread releases the lock.

Then you synchronize the updates made to the DOM object in the child threads with the main thread, calling the join method of each of the child thread objects in the main thread. After that you further process the DOM document object in the main stream. In this particular example, you simply write it to the stdout standard output stream.

As you might notice, the example shown here doesn’t actually discuss how to lock database access operations, such as issuing queries or updates against the same database table in parallel threads. The fact is that Oracle Database has its own powerful locking mechanisms to ensure data integrity in concurrent environments. Your job is to use those mechanisms correctly. The next section discusses how you can utilize Oracle Database features to control concurrent access to shared data, thus making the database take care of the concurrency problems.

### Making Oracle Database Take Care of Concurrency

As mentioned, you don’t need to manually implement resource locking in your Python code when it comes to accessing or manipulating shared data stored in an Oracle database. To address concurrency issues, Oracle Database uses different types of locks and multiversion concurrency control system behind the scenes, features based on the concept of a transaction. What this means in practice is that the only thing you have to worry about is to correctly utilize transactions to guarantee that the database data is accessed, updated, or changed properly. In particular, you must be careful when choosing between the autocommit and manual-commit transaction mode and when grouping several SQL statements into a transaction. Finally, you must avoid destructive interactions between concurrent transactions.

One important thing to remember here is that transactions you use in your Python code are associated with connections rather than cursors, meaning that you can easily logically group together statements executed with different cursors but through the same connection into a single transaction. However, if you want to implement two concurrent transactions, you will need to create two separate connection objects.

In the multithreaded sample discussed in the “Multithreaded Programming in Python” section earlier, you set the autocommit mode of the connection object to true, thus instructing the cx\_Oracle module to implicitly issue a COMMIT after each INSERT statement. In that particular case, using the autocommit mode was reasonable, because this let you avoid synchronization between the child threads and the main thread, which would otherwise be required so that you could manually issue a COMMIT in the main thread, as shown below:

...

#main thread waits until all child threads are done

for t in th:

t.join()

#and then issues a commit

dbconn.commit()

There are cases, though, where you have to use the above scenario. Consider the following example. Suppose you perform the following two operations, each in a parallel thread. In one thread, you save a purchase order document into the database, including the order details. In the other thread, you modify the table containing information about the products included in the order, updating the quantity of products left available for purchase.

It’s fairly obvious that the above two operations must be wrapped in a single transaction. To achieve this, you must have the autocommit mode off, which is the default. Also you will have to synchronize the parallel threads with the main thread and then explicitly issue a COMMIT, as shown in the above code snippet.

Although the above scenario can be easily implemented, in practice you will most likely want to implement the second operation—updating the quantity of products left available for purchase—inside the database, placing the BEFORE INSERT trigger on the table storing order details so that it automatically updates the corresponding record in the table containing information about the products. This would simplify the code on the Python side and eliminate the need to write a multithreaded Python script, making Oracle Database worry about data integrity issues. The fact is that Oracle Database will automatically roll back the operation of inserting a new row into the details table if a problem occurs while the products table is being updated within the BEFORE INSERT trigger placed on the details table. On the Python side, all that’s left is to wrap in a transaction all the INSERTs used to save the details of an order, as follows:

...

dbconn = cx\_Oracle.connect('hr', 'hr', '127.0.0.1/XE',threaded=True)

dbconn.autocommit = False

cur = dbconn.cursor()

...

for detail in details:

id = detail['id']

quantity = person['quantity']

cur.execute("INSERT INTO details(id, quantity) VALUES(:id, :quantity)", {'id':id, 'quantity':quantity})

dbconn.commit()

...

### Using Twisted, Python Event-Driven Framework

Twisted makes multithreaded programming in Python simpler and safer, providing a nice way of coding event-driven applications while hiding the complexity. The Twisted concurrency model is based on the concept of nonblocking calls. You call a function to request some data and specify a callback function to be called when the requested data is ready. In the meantime, the program can continue with other tasks.

The twisted.enterprise.adbapi module, an asynchronous wrapper for any DB-API-compatible Python module, enables you to perform database-related tasks in a nonblocking mode. With it, your application, for example, won’t wait until a connection to the database is established or a query is completed, instead performing other tasks in parallel. This section looks at a couple of simple examples of Twisted applications interacting with an Oracle database.

Twisted doesn’t come with Python—you need to download and install it after Python has been installed in your system. You can download the Twisted installation package suitable for your Python version and operating system from the Twisted Matrix Labs Web site at [http://twistedmatrix.com](http://twistedmatrix.com/). Once you have downloaded the package, the installation process, a matter of a couple clicks in the Twisted setup wizard, takes about one minute.

Twisted is an event-driven framework, so it is to have an event loop that, once started, runs uninterruptedly, waiting for events to dispatch. In Twisted, the event loop is implemented with the object called reactor. You start the Twisted event loop with reactor.run method and stop it with reactor.stop. Another Twisted object, called Deferred, is used to manage callbacks. The following is a simplified example of the Twisted event loop and a callback in action. The \_\_name\_\_ test is used to guarantee that the solution will run only if the module is called as a main script but not imported (that is, it must be called from a command line, with the IDLE Python GUI, or by an icon click).

#File: twistedsimple.py

#A simple example of a Twisted app

from twisted.internet import reactor

from twisted.enterprise import adbapi

def printResult(rslt):

print rslt[0][0]

reactor.stop()

if \_\_name\_\_ == "\_\_main\_\_":

dbpool = adbapi.ConnectionPool('cx\_Oracle', user='hr', password ='hr', dsn='127.0.0.1/XE')

empno = 100

deferred = dbpool.runQuery("SELECT last\_name FROM employees WHERE employee\_id = :empno", {'empno':empno})

deferred.addCallback(printResult)

reactor.run()

It’s important to realize that the twisted.enterprise.adbapi module is built on top of the standard DB-API interface and, behind the scenes, utilizes the Python database module you specify when calling the adbapi.ConnectionPool method. Even a set of allowed keywords you can use when specifying the adbapi.ConnectionPool input parameters will depend on the database module type you’re using.

Despite some differences in syntax when it’s used with different Python database modules, twisted.enterprise.adbapi provides the ability to write asynchronous code that will continue with the flow of your program while safely processing database-related tasks in the background. The following example represents a simple Twisted Web application, querying the database asynchronously. This example assumes that you have created the blob\_tab table and populated it with data, as discussed in the “Multithreaded Programming in Python” section at the beginning of the article.

#File: twistedTCPServer.py

#Querying database asynchronously with Twisted

from twisted.web import resource, server

from twisted.internet import reactor

from twisted.enterprise import adbapi

class BlobLoads(resource.Resource):

def \_\_init\_\_(self, dbconn):

self.dbconn = dbconn

resource.Resource.\_\_init\_\_(self)

def \_getBlobs(self, txn, query):

txn.execute(query)

return txn.fetchall()

def render\_GET(self, request):

query = "select id, blobdoc from blob\_tab"

self.dbconn.runInteraction(self.\_getBlobs, query).addCallback(

self.\_writeBlobs, request).addErrback(

self.\_exception, request)

return server.NOT\_DONE\_YET

def \_writeBlobs(self, results, request):

request.write("""

<html>

<head><title>BLOBs manipulating</title></head>

<body>

<h2>Writing BLOBs from the database to your disk</h2>

""")

for id, blobdoc in results:

request.write("<i>/tmp/picture%s.bmp</i><br/>" % id)

blob = blobdoc.read()

output = open("/tmp/picture%s.bmp" % id, 'wb')

output.write(blob)

output.close()

request.write("""

<p>Operation completed</p>

</body>

</html>

""")

request.finish( )

def \_exception(self, error, request):

request.write("Error obtaining BLOBs: %s" % error.getErrorMessage())

request.write("""

<p>Could not complete operation</p>

</body>

</html>

""")

request.finish( )

class SiteResource(resource.Resource):

def \_\_init\_\_(self, dbconn):

resource.Resource.\_\_init\_\_(self)

self.putChild('', BlobLoads(dbconn))

if \_\_name\_\_ == "\_\_main\_\_":

dbconn = adbapi.ConnectionPool('cx\_Oracle', user='usr', password ='pswd', dsn='127.0.0.1/XE')

site = server.Site(SiteResource(dbconn))

print "Listening on port 8000"

reactor.listenTCP(8000, site)

reactor.run()

When executed, this script starts a TCP server listening on port 8000. Upon accepting a client connection, the script will download all the images stored in the blob\_tab database and store them in the /tmp folder as separate files, sending appropriate messages back to the client. To test the application, you will need to run the script and then point your browser to [http://localhost:8000](http://localhost:8000/).

The most interesting thing about the above code is that it runs the query being issued against the database in a nonblocking mode, continuing with the flow of the program. To make sure it works this way, you might enhance the render\_GET method with some code inserted below the call to runInteraction that instructs Twisted to make an asynchronous call to \_getBlobs and then to \_writeBlobs. The newly inserted code should send something back to the client, using the request.write method, so that you can see that this output appears before the output generated within \_writeBlobs in the client’s browser.

PYTHON DEBUGGER:

# Debugging in Python

As a programmer, one of the first things that you need for serious program development is a debugger.

Python has a debugger, which is available as a module called pdb (for “Python DeBugger”, naturally!). Unfortunately, most discussions of pdb are not very useful to a Python newbie — most are very terse and simply rehash the [description of pdb in the Python library reference manual](http://www.python.org/doc/current/lib/module-pdb.html). The discussion that I have found most accessible is in the first four pages of Chapter 27 of the [Python 2.1 Bible](http://www.amazon.com/exec/obidos/ASIN/0764548077/qid%3D1005424462/ref%3Dsr%5F11%5F0%5F1/102-2202009-5284143).

**Getting started — pdb.set\_trace()**

To start, I’ll show you the very simplest way to use the Python debugger.

1. Let’s start with a simple program, epdb1.py.

*# epdb1.py -- experiment with the Python debugger, pdb*

*a = "aaa"*

*b = "bbb"*

*c = "ccc"*

*final = a + b + c*

*print final*

2. Insert the following statement at the beginning of your Python program. This statement imports the Python debugger module, pdb.

*import pdb*

3. Now find a spot where you would like tracing to begin, and insert the following code:

*pdb.set\_trace()*

So now your program looks like this.

*# epdb1.py -- experiment with the Python debugger, pdb*

*import pdb*

*a = "aaa"*

*pdb.set\_trace()*

*b = "bbb"*

*c = "ccc"*

*final = a + b + c*

*print final*

4. Now run your program from the command line as you usually do, which will probably look something like this:

*PROMPT> python epdb1.py*

When your program encounters the line with pdb.set\_trace() it will start tracing. That is, it will (1) stop, (2) display the “current statement” (that is, the line that will execute next) and (3) wait for your input. You will see the pdb prompt, which looks like this:

*(Pdb)*

**Execute the next statement… with “n” (next)**

At the (Pdb) prompt, press the lower-case letter “n” (for “next”) on your keyboard, and then press the ENTER key. This will tell pdb to execute the current statement. Keep doing this — pressing “n”, then ENTER.

Eventually you will come to the end of your program, and it will terminate and return you to the normal command prompt.

Congratulations! You’ve just done your first debugging run!

**Repeating the last debugging command… with ENTER**

This time, do the same thing as you did before. Start your program running. At the (Pdb) prompt, press the lower-case letter “n” (for “next”) on your keyboard, and then press the ENTER key.

But this time, after the first time that you press “n” and then ENTER, don’t do it any more. Instead, when you see the (Pdb) prompt, just press ENTER. You will notice that pdb continues, just as if you had pressed “n”. So this is Handy Tip #1:

If you press ENTER without entering anything, pdb will re-execute the last command that you gave it.

In this case, the command was “n”, so you could just keep stepping through the program by pressing ENTER.

Notice that as you passed the last line (the line with the “print” statement), it was executed and you saw the output of the print statement (“aaabbbccc”) displayed on your screen.

**Quitting it all… with “q” (quit)**

The debugger can do all sorts of things, some of which you may find totally mystifying. So the most important thing to learn now — before you learn anything else — is how to quit debugging!

It is easy. When you see the (Pdb) prompt, just press “q” (for “quit”) and the ENTER key. Pdb will quit and you will be back at your command prompt. Try it, and see how it works.

**Printing the value of variables… with “p” (print)**

The most useful thing you can do at the (Pdb) prompt is to print the value of a variable. Here’s how to do it.

When you see the (Pdb) prompt, enter “p” (for “print”) followed by the name of the variable you want to print. And of course, you end by pressing the ENTER key.

Note that you can print multiple variables, by separating their names with commas (just as in a regular Python “print” statement). For example, you can print the value of the variables a, b, and c this way:

*p a, b, c*

**When does pdb display a line?**

Suppose you have progressed through the program until you see the line

*final = a + b + c*

and you give pdb the command

*p final*

You will get a NameError exception. This is because, although you are seeing the line, it has not yet executed. So the **final** variable has not yet been created.

Now press “n” and ENTER to continue and execute the line. Then try the “p final” command again. This time, when you give the command “p final”, pdb will print the value of **final**, which is “aaabbbccc”.

**Turning off the (Pdb) prompt… with “c” (continue)**

You probably noticed that the “q” command got you out of pdb in a very crude way — basically, by crashing the program.

If you wish simply to stop debugging, but to let the program continue running, then you want to use the “c” (for “continue”) command at the (Pdb) prompt. This will cause your program to continue running normally, without pausing for debugging. It may run to completion. Or, if the **pdb.set\_trace()** statement was inside a loop, you may encounter it again, and the (Pdb) debugging prompt will appear once more.

**Seeing where you are… with “l” (list)**

As you are debugging, there is a lot of stuff being written to the screen, and it gets really hard to get a feeling for where you are in your program. That’s where the “l” (for “list”) command comes in. (Note that it is a lower-case “L”, not the numeral “one” or the capital letter “I”.)

“l” shows you, on the screen, the general area of your program’s souce code that you are executing. By default, it lists 11 (eleven) lines of code. The line of code that you are about to execute (the “current line”) is right in the middle, and there is a little arrow “–>” that points to it.

So a typical interaction with pdb might go like this

* The pdb.set\_trace() statement is encountered, and you start tracing with the (Pdb) prompt
* You press “n” and then ENTER, to start stepping through your code.
* You just press ENTER to step again.
* You just press ENTER to step again.
* You just press ENTER to step again. etc. etc. etc.
* Eventually, you realize that you are a bit lost. You’re not exactly sure where you are in your program any more. So…
* You press “l” and then ENTER. This lists the area of your program that is currently being executed.
* You inspect the display, get your bearings, and are ready to start again. So….
* You press “n” and then ENTER, to start stepping through your code.
* You just press ENTER to step again.
* You just press ENTER to step again. etc. etc. etc.

**Stepping into subroutines… with “s” (step into)**

Eventually, you will need to debug larger programs — programs that use subroutines. And sometimes, the problem that you’re trying to find will lie buried in a subroutine. Consider the following program.

*# epdb2.py -- experiment with the Python debugger, pdb*

*import pdb*

*def combine(s1,s2): # define subroutine combine, which...*

*s3 = s1 + s2 + s1 # sandwiches s2 between copies of s1, ...*

*s3 = '"' + s3 +'"' # encloses it in double quotes,...*

*return s3 # and returns it.*

*a = "aaa"*

*pdb.set\_trace()*

*b = "bbb"*

*c = "ccc"*

*final = combine(a,b)*

*print final*

As you move through your programs by using the “n” command at the (Pdb) prompt, you will find that when you encounter a statement that invokes a subroutine — the final = combine(a,b) statement, for example — pdb treats it no differently than any other statement. That is, the statement is executed and you move on to the next statement — in this case, to print final.

But suppose you suspect that there is a problem in a subroutine. In our case, suppose you suspect that there is a problem in the combine subroutine. What you want — when you encounter the final = combine(a,b) statement — is some way to step into the combine subroutine, and to continue your debugging inside it.

Well, you can do that too. Do it with the “s” (for “step into”) command.

When you execute statements that do not involve function calls, “n” and “s” do the same thing — move on to the next statement. But when you execute statements that invoke functions, “s”, unlike “n”, will step into the subroutine. In our case, if you executed the

*final = combine(a,b)*

statement using “s”, then the next statement that pdb would show you would be the first statement in the combine subroutine:

*def combine(s1,s2):*

and you will continue debugging from there.

**Continuing… but just to the end of the current subroutine… with “r” (return)**

When you use “s” to step into subroutines, you will often find yourself trapped in a subroutine. You have examined the code that you’re interested in, but now you have to step through a lot of uninteresting code in the subroutine.

In this situation, what you’d like to be able to do is just to skip ahead to the end of the subroutine. That is, you want to do something like the “c” (“continue”) command does, but you want just to continue to the end of the subroutine, and then resume your stepping through the code.

You can do it. The command to do it is “r” (for “return” or, better, “continue until return”). If you are in a subroutine and you enter the “r” command at the (Pdb) prompt, pdb will continue executing until the end of the subroutine. At that point — the point when it is ready to return to the calling routine — it will stop and show the (Pdb) prompt again, and you can resume stepping through your code.

**You can do anything at all at the (Pdb) prompt …**

Sometimes you will be in the following situation — You think you’ve discovered the problem. The statement that was assigning a value of, say, “aaa” to variable var1 was wrong, and was causing your program to blow up. It should have been assigning the value “bbb” to var1.

… at least, you’re pretty sure that was the problem…

What you’d really like to be able to do, now that you’ve located the problem, is to assign “bbb” to var1, and see if your program now runs to completion without bombing.

It can be done!

One of the nice things about the (Pdb) prompt is that you can do anything at it — you can enter any command that you like at the (Pdb) prompt. So you can, for instance, enter this command at the (Pdb) prompt.

*(Pdb) var1 = "bbb"*

You can then continue to step through the program. Or you could be adventurous — use “c” to turn off debugging, and see if your program will end without bombing!

**… but be a little careful!**

[Thanks to Dick Morris for the information in this section.]

Since you can do anything at all at the (Pdb) prompt, you might decide to try setting the variable b to a new value, say “BBB”, this way:

*(Pdb) b = "BBB"*

If you do, pdb produces a strange error message about being unable to find an object named ‘= “BBB” ‘. Why???

What happens is that pdb attempts to execute the pdb **b** command for setting and listing breakpoints (a command that we haven’t discussed). It interprets the rest of the line as an argument to the **b** command, and can’t find the object that (it thinks) is being referred to. So it produces an error message.

So how can we assign a new value to **b**? The trick is to start the command with an exclamation point (!).

*(Pdb)!b = "BBB"*

An exclamation point tells pdb that what follows is a Python statement, not a pdb command.

Python has a strange behaviour - in comparison with other programming languages - when assigning and copying simple data types like integers and strings. The difference between shallow and deep copying is only relevant for compound objects, i.e. objects containing other objects, like lists or class instances.   
  
In the following code snippet y points to the same memory location than X. This changes, when we assign a different value to y. In this case y will receive a separate memory location, as we have seen in the chapter "Data Types and Variables".

>>> x = 3

>>> y = x

But even if this internal behaviour appears strange compared to programming languages like C, C++ and Perl, yet the observable results of the assignments answer our expectations. But it can be problematic, if we copy mutable objects like lists and dictionaries.   
  
Python creates only real copies, if it has to, i.e. if the user, the programmer, explicitly demands it.   
  
We will introduce you to the most crucial problems, which can occur when copying mutable objects, i.e. when copying lists and dictionaries.

**Copying a list**

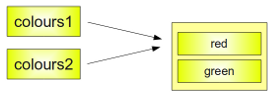
>>> colours1 = ["red", "green"]

>>> colours2 = colours1

>>> colours2 = ["rouge", "vert"]

>>> print colours1

['red', 'green']

In the example above a simple list is assigned to colours1. In the next step we assign colour1 to colours2. After this, a new list is assigned to colours2.   
  
As we have expected, the values of colours1 remained unchanged. Like it was in our example in the chapter "Data types and variables", a new memory location had been allocated for colours2, because we have assigned a complete new list to this variable.

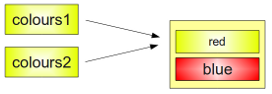
>>> colours1 = ["red", "green"]

>>> colours2 = colours1

>>> colours2[1] = "blue"

>>> colours1

['red', 'blue']

But the question is, what will happen, if we change an element of the list of colours2 or colours1?   
In the example above, we assign a new value to the second element of colours2. Lots of beginners will be astonished, that the list of colours1 has been "automatically" changed as well.   
The explanation is, that there has been no new assignment to colours2, only to one of its elements.

**Copy with the Slice Operator**

It's possible to completely copy shallow list structures with the slice operator without having any of the side effects, which we have described above:

>>> list1 = ['a','b','c','d']

>>> list2 = list1[:]

>>> list2[1] = 'x'

>>> print list2

['a', 'x', 'c', 'd']

>>> print list1

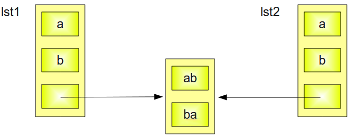
['a', 'b', 'c', 'd']

>>>

But as soon as a list contains sublists, we have the same difficulty, i.e. just pointers to the sublists.

>>> lst1 = ['a','b',['ab','ba']]

>>> lst2 = lst1[:]

This behaviour is depicted in the following diagram:   
  
  
  
  
  
  
  
  
  
  
  
If you assign a new value to the 0th Element of one of the two lists, there will be no side effect. Problems arise, if you change one of the elements of the sublist.

>>> lst1 = ['a','b',['ab','ba']]

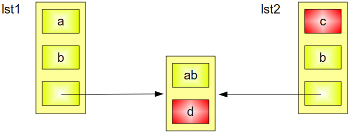
>>> lst2 = lst1[:]

>>> lst2[0] = 'c'

>>> lst2[2][1] = 'd'

>>> print(lst1)

['a', 'b', ['ab', 'd']]

The following diagram depicts what happens, if one of the elements of a sublist will be changed: Both the content of lst1 and lst2 are changed.   
  


**Using the Method deepcopy from the Module copy**

A solution to the described problems provides the module "copy". This module provides the method "copy", which allows a complete copy of a arbitrary list, i.e. shallow and other lists.   
  
The following script uses our example above and this method:

from copy import deepcopy

lst1 = ['a','b',['ab','ba']]

lst2 = deepcopy(lst1)

lst2[2][1] = "d"

lst2[0] = "c";

print lst2

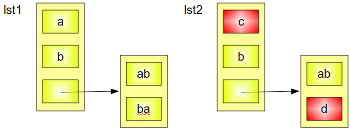
print lst1

If we save this script under the name of deep\_copy.py and if we call the script with "python deep\_copy.py", we will receive the following output:

$ python deep\_copy.py

['c', 'b', ['ab', 'd']]

['a', 'b', ['ab', 'ba']]

  
  
Generators functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.

# Simplified Code

The simplification of code is a result of generator function and generator expression support provided by Python.

To illustrate this, we will compare different implementations that implement a function, "firstn", that represents the first n non-negative integers, where n is a really big number, and assume (for the sake of the examples in this section) that each integer take up a lot of space, say 10 megabytes each.

Note: Please note that in real life, integers do not take up that much space, unless they are really, really, really, big integers. For instance you can represent a 309 digit number with 128 bytes (add some overhead, it will still be less than 150 bytes).

First, let us consider the simple example of building a list and returning it.

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_1) # Build and return a list

[2](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_2) def firstn(n):

[3](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_3) num, nums = 0, []

[4](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_4) while num < n:

[5](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_5) nums.append(num)

[6](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_6) num += 1

[7](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_7) return nums

[8](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_8)

[9](https://wiki.python.org/moin/Generators#CA-73c1ec4dcfbe9f81e8af5205a528baf15cdc1356_9) sum\_of\_first\_n = sum(firstn(1000000))

The code is quite simple and straightforward, but its builds the full list in memory. This is clearly not acceptable in our case, because we cannot afford to keep all n "10 megabyte" integers in memory.

So, we resort to the generator pattern. The following implements generator as an iterable object.

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_1) # Using the generator pattern (an iterable)

[2](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_2) class firstn(object):

[3](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_3) def \_\_init\_\_(self, n):

[4](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_4) self.n = n

[5](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_5) self.num, self.nums = 0, []

[6](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_6)

[7](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_7) def \_\_iter\_\_(self):

[8](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_8) return self

[9](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_9)

[10](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_10) # Python 3 compatibility

[11](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_11) def \_\_next\_\_(self):

[12](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_12) return self.next()

[13](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_13)

[14](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_14) def next(self):

[15](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_15) if self.num < self.n:

[16](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_16) cur, self.num = self.num, self.num+1

[17](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_17) return cur

[18](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_18) else:

[19](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_19) raise StopIteration()

[20](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_20)

[21](https://wiki.python.org/moin/Generators#CA-31c8468eb74aeab199a158c358a0dcdea389911d_21) sum\_of\_first\_n = sum(firstn(1000000))

This will perform as we expect, but we have the following issues:

* there is a lot of boilerplate
* the logic has to be expressed in a somewhat convoluted way

Furthermore, this is a pattern that we will use over and over for many similar constructs. Imagine writing all that just to get an iterator.

Python provides generator functions as a convenient shortcut to building iterators. Lets us rewrite the above iterator as a generator function:

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_1) # a generator that yields items instead of returning a list

[2](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_2) def firstn(n):

[3](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_3) num = 0

[4](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_4) while num < n:

[5](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_5) yield num

[6](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_6) num += 1

[7](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_7)

[8](https://wiki.python.org/moin/Generators#CA-d5466ac7b4355903187a3edcd0c340643d098bbd_8) sum\_of\_first\_n = sum(firstn(1000000))

Note that the expression of the number generation logic is clear and natural. It is very similar to the implementation that built a list in memory, but has the memory usage characteristic of the iterator implementation.

Note: the above code is perfectly acceptable for expository purposes, but remember that in Python 2 ***firstn()*** is equivalent to the built-in ***xrange()*** function, and in Python 3 ***range()*** is a generator. The built-ins will always be much faster. SH

Generator expressions provide an additional shortcut to build generators out of expressions similar to that of list comprehensions.

In fact, we can turn a list comprehension into a generator expression by replacing the square brackets ("[ ]") with parentheses. Alternately, we can think of list comprehensions as generator expressions wrapped in a list constructor.

Consider the following example:

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-6021c335e902d0510d5cf727f4891fa800d440d6_1) # list comprehension

[2](https://wiki.python.org/moin/Generators#CA-6021c335e902d0510d5cf727f4891fa800d440d6_2) doubles = [2 \* n for n in range(50)]

[3](https://wiki.python.org/moin/Generators#CA-6021c335e902d0510d5cf727f4891fa800d440d6_3)

[4](https://wiki.python.org/moin/Generators#CA-6021c335e902d0510d5cf727f4891fa800d440d6_4) # same as the list comprehension above

[5](https://wiki.python.org/moin/Generators#CA-6021c335e902d0510d5cf727f4891fa800d440d6_5) doubles = list(2 \* n for n in range(50))

Notice how a list comprehension looks essentially like a generator expression passed to a list constructor.

By allowing generator expressions, we don't have to write a generator function if we do not need the list. If only list comprehensions were available, and we needed to lazily build a set of items to be processed, we will have to write a generator function.

This also means that we can use the same syntax we have been using for list comprehensions to build generators.

Keep in mind that generators are a special type of iterator, and that containers like list and set are also iterables. The uniform way in which all of these are handled, adds greatly to the simplification of code.

# Improved Performance

The performance improvement from the use of generators is the result of the lazy (on demand) generation of values, which translates to lower memory usage. Furthermore, we do not need to wait until all the elements have been generated before we start to use them. This is similar to the benefits provided by iterators, but the generator makes building iterators easy.

This can be illustrated by comparing the range and xrange built-ins of Python 2.x.

Both range and xrange represent a range of numbers, and have the same function signature, but range returns a list while xrange returns a generator (at least in concept; the implementation may differ).

Say, we had to compute the sum of the first n, say 1,000,000, non-negative numbers.

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-bdc68d1c164def45d62ea764ccc6bd755bf24e0b_1) # Note: Python 2.x only

[2](https://wiki.python.org/moin/Generators#CA-bdc68d1c164def45d62ea764ccc6bd755bf24e0b_2) # using a non-generator

[3](https://wiki.python.org/moin/Generators#CA-bdc68d1c164def45d62ea764ccc6bd755bf24e0b_3) sum\_of\_first\_n = sum(range(1000000))

[4](https://wiki.python.org/moin/Generators#CA-bdc68d1c164def45d62ea764ccc6bd755bf24e0b_4)

[5](https://wiki.python.org/moin/Generators#CA-bdc68d1c164def45d62ea764ccc6bd755bf24e0b_5) # using a generator

[6](https://wiki.python.org/moin/Generators#CA-bdc68d1c164def45d62ea764ccc6bd755bf24e0b_6) sum\_of\_first\_n = sum(xrange(1000000))

Note that both lines are identical in form, but the one using range is much more expensive.

When we use range we build a 1,000,000 element list in memory and then find its sum. This is a waste, considering that we use these 1,000,000 elements just to compute the sum.

This waste becomes more pronounced as the number of elements (our n) becomes larger, the size of our elements become larger, or both.

On the other hand, when we use xrange, we do not incur the cost of building a 1,000,000 element list in memory. The generator created by xrange will generate each number, which sum will consume to accumulate the sum.

In the case of the "range" function, using it as an iterable is the dominant use-case, and this is reflected in Python 3.x, which makes the range built-in return a generator instead of a list.

Note: a generator will provide performance benefits only if we do not intend to use that set of generated values more than once.

Consider the following example:

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-7f5f127f307c9812f4165f9564a00d72bfaa9c2d_1) # Note: Python 2.x only

[2](https://wiki.python.org/moin/Generators#CA-7f5f127f307c9812f4165f9564a00d72bfaa9c2d_2) s = sum(xrange(1000000))

[3](https://wiki.python.org/moin/Generators#CA-7f5f127f307c9812f4165f9564a00d72bfaa9c2d_3) p = product(xrange(1000000))

Imagine that making a integer is a very expensive process. In the above code, we just performed the same expensive process twice. In cases like this, building a list in memory might be worth it (see example below):

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-2feb617fa11c8940f0548771255a2de723441cc5_1) # Note: Python 2.x only

[2](https://wiki.python.org/moin/Generators#CA-2feb617fa11c8940f0548771255a2de723441cc5_2) nums = list(xrange(1000000))

[3](https://wiki.python.org/moin/Generators#CA-2feb617fa11c8940f0548771255a2de723441cc5_3) s = sum(nums)

[4](https://wiki.python.org/moin/Generators#CA-2feb617fa11c8940f0548771255a2de723441cc5_4) p = product(nums)

However, a generator might still be the only way, if the storage of these generated objects in memory is not practical, and it might be worth to pay the price of duplicated expensive computations.

# Examples

For example, the [RangeGenerator](https://wiki.python.org/moin/RangeGenerator) can be used to iterate over a large number of values, without creating a massive list (like range would)

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-f41b3124950450be853a5bf278c557e6d9ff41ef_1) #the for loop will generate each i (i.e. 1,2,3,4,5, ...), add it to sum, and throw it away

[2](https://wiki.python.org/moin/Generators#CA-f41b3124950450be853a5bf278c557e6d9ff41ef_2) #before the next i is generated. This is opposed to iterating through range(...), which creates

[3](https://wiki.python.org/moin/Generators#CA-f41b3124950450be853a5bf278c557e6d9ff41ef_3) #a potentially massive list and then iterates through it.

[4](https://wiki.python.org/moin/Generators#CA-f41b3124950450be853a5bf278c557e6d9ff41ef_4) for i in irange(1000000):

[5](https://wiki.python.org/moin/Generators#CA-f41b3124950450be853a5bf278c557e6d9ff41ef_5) sum = sum+i

Generators can be composed. Here we create a generator on the squares of consecutive integers.

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-1dfdaaba2e2ab39b6b8f47ec670d97b4d44b8421_1) #square is a generator

[2](https://wiki.python.org/moin/Generators#CA-1dfdaaba2e2ab39b6b8f47ec670d97b4d44b8421_2) square = (i\*i for i in irange(1000000))

[3](https://wiki.python.org/moin/Generators#CA-1dfdaaba2e2ab39b6b8f47ec670d97b4d44b8421_3) #add the squares

[4](https://wiki.python.org/moin/Generators#CA-1dfdaaba2e2ab39b6b8f47ec670d97b4d44b8421_4) for i in square:

[5](https://wiki.python.org/moin/Generators#CA-1dfdaaba2e2ab39b6b8f47ec670d97b4d44b8421_5) sum = sum +i

Here, we compose a square generator with the takewhile generator, to generate squares less than 100

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-b42a3b3e524fd1e3cc2627775676f0474e4d10a8_1) #add squares less than 100

[2](https://wiki.python.org/moin/Generators#CA-b42a3b3e524fd1e3cc2627775676f0474e4d10a8_2) square = (i\*i for i in count())

[3](https://wiki.python.org/moin/Generators#CA-b42a3b3e524fd1e3cc2627775676f0474e4d10a8_3) bounded\_squares = takewhile(lambda x : x< 100, square)

[4](https://wiki.python.org/moin/Generators#CA-b42a3b3e524fd1e3cc2627775676f0474e4d10a8_4) for i in bounded\_squares:

[5](https://wiki.python.org/moin/Generators#CA-b42a3b3e524fd1e3cc2627775676f0474e4d10a8_5) sum += i

to be written: Generators made from classes?

## Links

* [PEP-255: Simple Iterators](http://www.python.org/peps/pep-0255.html) -- the original
* [Iterators and Simple Generators](http://www-106.ibm.com/developerworks/library/l-pycon.html?n-l-9271)
* [combinatorial functions in itertools](http://www-106.ibm.com/developerworks/linux/library/l-cpyiter.html)
* [Python Generator Tricks](http://linuxgazette.net/100/pramode.html) -- various infinite sequences, recursions, ...
* ["weightless threads"](http://www-106.ibm.com/developerworks/linux/library/l-pythrd.html) -- simulating threads using generators
* [XML processing](http://www-106.ibm.com/developerworks/library/x-tipgenr.html) -- yes, using generators
* [C2:GeneratorsAreNotCoroutines](http://c2.com/cgi/wiki?GeneratorsAreNotCoroutines) -- particulars on generators, coroutines, and continuations

See also: [Iterator](https://wiki.python.org/moin/Iterator)

# Discussion

I once saw [MikeOrr](https://wiki.python.org/moin/MikeOrr) demonstrate Before and After examples. But, I forget how they worked.

Can someone demonstrate here?

He did something like: Show how a normal list operation could be written to use generators. Something like:

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-d74760b038716a21bbd3d3a74f70c6a9c21743bb_1) def double(L):

[2](https://wiki.python.org/moin/Generators#CA-d74760b038716a21bbd3d3a74f70c6a9c21743bb_2) return [x\*2 for x in L]

[3](https://wiki.python.org/moin/Generators#CA-d74760b038716a21bbd3d3a74f70c6a9c21743bb_3)

[4](https://wiki.python.org/moin/Generators#CA-d74760b038716a21bbd3d3a74f70c6a9c21743bb_4) eggs = double([1, 2, 3, 4, 5])

...he showed how that, or something like that, could be rewritten using iterators, generators.

It's been a while since I've seen it, I may be getting this all wrong.

-- [LionKimbro](https://wiki.python.org/moin/LionKimbro) 2005-04-02 19:12:19

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_1) # explicitly write a generator function

[2](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_2) def double(L):

[3](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_3) for x in L:

[4](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_4) yield x\*2

[5](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_5)

[6](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_6) # eggs will be a generator

[7](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_7) eggs = double([1, 2, 3, 4, 5])

[8](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_8)

[9](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_9) # the above is equivalent to ("generator comprehension"?)

[10](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_10) eggs = (x\*2 for x in [1, 2, 3, 4, 5])

[11](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_11)

[12](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_12) # need to do this if you need a list

[13](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_13) eggs = list(double([1, 2, 3, 4, 5]))

[14](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_14)

[15](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_15) # the above is equivalent to (list comprehension)

[16](https://wiki.python.org/moin/Generators#CA-4867240ebda44ea3b0051b2340f465a4e480b6da_16) eggs = [x\*2 for x in [1, 2, 3, 4, 5]]

For the above example, a generator comprehension or list comprehension is sufficient unless you need to apply that in many places.

Also, a generator function will be cleaner and more clear, if the generated expressions are more complex, involve multiple steps, or depend on additional temporary state.

Consider the following example:

[Toggle line numbers](https://wiki.python.org/moin/Generators)

[1](https://wiki.python.org/moin/Generators#CA-d312f2eada1bf2fe2fd4877fad12afa8041a608d_1) def unique(iterable, key=lambda x: x):

[2](https://wiki.python.org/moin/Generators#CA-d312f2eada1bf2fe2fd4877fad12afa8041a608d_2) seen = set()

[3](https://wiki.python.org/moin/Generators#CA-d312f2eada1bf2fe2fd4877fad12afa8041a608d_3) for elem, ekey in ((e, key(e)) for e in iterable):

[4](https://wiki.python.org/moin/Generators#CA-d312f2eada1bf2fe2fd4877fad12afa8041a608d_4) if ekey not in seen:

[5](https://wiki.python.org/moin/Generators#CA-d312f2eada1bf2fe2fd4877fad12afa8041a608d_5) yield elem

[6](https://wiki.python.org/moin/Generators#CA-d312f2eada1bf2fe2fd4877fad12afa8041a608d_6) seen.add(ekey)

Here, the temporary keys collector, seen, is a temporary storage that will just be more clutter in the location where this generator will be used.

Even if we were to use this only once, it is worth writing a function (for the sake of clarity; remember that Python allows nested functions).