Intermediate Python 3

John Strickler

Version 1.0, March 2018

Table of Contents

About this course

Welcome!

- We're glad you're here
- Class has hands-on labs for nearly every chapter
- Please make a name tent

Instructor name:

Instructor e-mail:

Have Fun!

Classroom etiquette

- Noisemakers off
- No phone conversations
- Come and go quietly during class.

Please turn off cell phone ringers and other noisemakers.

If you need to have a phone conversation, please leave the classroom.

We're all adults here; feel free to leave the clasroom if you need to use the restroom, make a phone call, etc. You don't have to wait for a lab or break, but please try not to disturb others.

IMPORTANT Please do not bring any exploding penguins to class. They might maim, dismember, or otherwise disturb your fellow students.

Course Outline

Day 1

Chapter 1 Python Refresher
Chapter 2 OS Services
Chapter 3 Dates and Times
Chapter 4 Binary Data

Day 2

Chapter 5 Pythonic Programming
Chapter 6 Functions, Modules, and Packages
Chapter 7 Intermediate Classes
Chapter 8 Metaprogramming

Day 3

Chapter 9 Developer tools
Chapter 10 Unit Tests
Chapter 11 Database access
Chapter 12 PyQt

Day 4

Chapter 13 Network Programming
Chapter 14 Multiprogramming
Chapter 15 Scripting for System Administration
Chapter 16 Serializing Data

NOTE The actual schedule varies with circumstances. The last day may include *ad hoc* topics requested by students

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **py3interm**.

What's in the files?

py3interm contains data and other files needed for the exercises
py3interm/EXAMPLES contains the examples from the course manuals.
py3interm/ANSWERS contains sample answers to the labs.

NOTE	The student files do not contain Python itself. It will need to be installed separately. This has probably already been done for you.

Extracting the student files

Windows

Open the file **py3interm.zip**. Extract all files to your desktop. This will create the folder **py3interm**.

Non-Windows (includes Linux, OS X, etc)

Copy or download py3interm.tgz to your home directory. In your home directory, type

```
tar xzvf py3interm.tgz
```

This will create the **py3interm** directory under your home directory.

If your version of Unix is elderly, its tar command may not support the z option. If this is so, use this command line instead:

```
gzip -dc py3interm.tgz | tar xvf -
```

Examples

Nearly all examples from the course manual are provided in EXAMPLES subdirectory. Many of the examples have callouts — numbers that refer to notes just below the code.

It will look like this:

Example

cmd_line_args.py

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

import sys   ①

print(sys.argv) ②

name = sys.argv[1]  ③
print("name is", name)
```

① Import the **sys** module

② Print all parameters, including script itself

③ Get the first actual parameter

cmd_line_args.py apple mango 123

```
['/Users/jstrick/Documents/curr/courses/python/examples3/cmd_line_args.py', 'apple',
'mango', '123']
name is apple
```

Lab Exercises

- Relax – the labs are not quizzes
- Feel free to modify labs
- Ask the instructor for help
- Work on your own scripts or data
- Answers are in py3interm/ANSWERS

Appendices

- Appendix A: Python Bibliography
- Appendix B: String formatting

Chapter 1: Python Refresher

Objectives

- Refresh basic (intro-level) Python concepts

Variables

- Declared by assignment
- Dynamic typing

Variables are declared by assigning to them. Python does not require explicit type specifiers, but sets the type implicitly by examining the value that was assigned.

Thus, assigning a literal integer to a variable creates a variable of type int, while assigning quoted text to a variable creates a variable of type string. Once a variable is assigned to, it will cause an error if the variable is used with an operator or function that is inappropriate for the type.

A variable cannot be used before it is assigned to.

Variables must be assigned **some** value. A value of **None** may be assigned if no particular value is needed.

Names may contain only letters, digits, and underscores, and may not start with a digit. The Python convention for variable names is `all_lower_case_words_with_underscores`.

Example

```
name = 'Fred Flintstone'
count = 0
name = 'Fred Flintstone'
colors = [ 'red', 'purple', 'green' ]
```

Basic Python Data types

- Python has many data types
- Use builtin functions to convert

Python has many data types. There are builtin functions to convert from one type to another. If the source type cannot be converted to the target type, a TypeError is thrown.

Numeric types

- bool
- int
- float
- complex

Sequence types

- str
- bytes
- list
- tuple

Mapping types

- dict
- set
- frozenset

Sequence Types

Strings are text (arrays of Unicode characters)

```
s = "text";
```

Bytes are arrays of bytes

```
b = b"text";
```

Lists are sequences of values

```
my_list = []
sequence[start:limit:stride]
```

Tuples are *readonly* sequences of values (used as records)

```
my_tuple = 'Mary', 'Poppins', 'London'
```

Python supports four types of sequences – strings, bytes, lists, and tuples. All sequences share a common set of operations, methods, and builtin functions; each type also has operations specific to that type.

A str object is a list of Unicode characters. A **bytes** object is a list of bytes.

All sequences support slicing, which means returning a subset of the sequence using the [**start**:*limit*:*step*] syntax.

Example

```
colors = [ 'red', 'green', 'blue', 'purple', 'pink', 'yellow', 'black' ]
c1 = colors[0]    #  'red'
c2 = colors[1:4]  #  'green', 'blue', 'purple'
c3 = colors[-1]   # 'black'
c4 = colors[:3]   # 'red', 'green', 'blue'
c5 = colors[3:]   # 'purple', 'pink', 'yellow', 'black'
```

Table 1. Slicing syntax

sequence[START:STOP]	START to STOP - 1
sequence[START:]	START to end
sequence[:STOP]	beginning to STOP - 1
sequence[START:STOP:STEP]	START to STOP -1 counting by STEP
sequence[:]	all elements
sequence[::]	all elements
sequence[::STEP]	all elements counting by STEP

NOTE Remember that the starting value of a slice is ***IN***clusive, while the ending value is ***EX***clusive.

Mapping Types

- Dictionaries are mapped sets of values
- Sets are similar to dictionaries but contain only keys
- Syntax

```
d = { }
s = set()
f = frozenset()
```

Python also supports mapping types — dictionaries and sets.

A dictionary (**dict**) is a set of values indexed by an immutable keyword. Dictionaries are used for many tasks, including mapping one set of values to another, and counting occurrences of values. . Prior to version 3.6, dictionaries were unordered, but beginning with 3.6, dictionaries preserve the order in which items are added.

Dictionary keys must be *hashable*, which means in general that they must be immutable. This means that most dictionary keys are strings, but can be numbers, or tuples of immutable types.

A set is an unique collection of values. There are two types — the normal set is dynamic (mutable), and a frozenset is fixed (immutable), like a tuple.

Program structure

- All imports at top
- Variables, functions, and classes must be declared before use
- Main function goes at top
- Main function *called* at bottom

In Python, modules must be imported before their contents may be accessed. Variables, Functions, and classes must be declared before they can be used. Thus most scripts are ordered in this way:

1. import statements
2. global variables
3. main function
4. functions
5. call to main function

You may want to make a template for your Python scripts. Most editors and IDEs support templates or code snippets.

In PyCharm, go to **Settings→Editor→File and Code Templates** to create a new file template.

Example Script Format

script_template.py

```
#!/usr/bin/env python
"""
This is the doc string for the module/script.
"""
import sys

# other imports  (standard library, standard non-library, local)

# constants (AKA global variables -- keep these to a minimum)

# main function
def main(args):
    """
    This is the docstring for the main() function

    :param args: Command line arguments.
    :return: None
    """
    function1()

# other functions
def function1():
    """
    This is the docstring for function1().

    :return: None
    """
    pass

if __name__ == '__main__':
    main(sys.argv[1:])  # Pass command line args (minus script name) to main()
```

TIP copy/paste this script to create new scripts

Files and console I/O

- print()
- open()
- input()

Screen output

To output to the screen, use the **print()** function. print() normally outputs a newline after its arguments, this can be controlled with the **end** parameter.

print() puts spaces between its arguments by default. To use a different separator, set the **sep** parameter to the desired separator, which might be an empty string.

Reading files

To read a file, open it with the open() function as part of a **with** statement.

To read it line by line, iterate through the file with a **for** loop. To read the entire file, use file.read(); to read all the lines into a list, use file.readlines(). To read a specified number of bytes, use file.read(n).

To navigate within a file, use file.seek(offset, whence); to get the current location, use file.tell().

User input

To get input from the user, use input(). It provides a prompt to the user, and returns a string, with the newline already trimmed.

```
file_name = input("What file name? ")
```

Conditionals

- Test a Boolean value
- if-elif-else

The conditional statement in Python, like most languages, is if. There are several variations on how if is used. All depend on testing a value to see whether it is true or false.

The following values are false:

- False
- Empty collections (empty string, empty list, empty dictionary, empty set, etc.)
- Numeric zero (0 or 0.0)

Just about everything else is true. (User-defined objects, and many builtin objects are true. If you create a class, you can control when it it true, and when it is false.)

Python has a shortcut if-else that is something like the A?B:C operator in C, Perl and other curly-brace languages

```
value1 if condition else value2
```

Example

```
if name == 'root:
    print("do not run this utility as root")
elif name == 'guest':
    print("sorry - guests are not allowed to run this utility")
else:
    print("starting processing")

limit = sys.args[1] if len(sys.args) > 1 else 100
```

Loops

- Two kinds of loops
  - **while** waits for condition
  - **for** iterates over a sequence (iterable)

Python has two kinds of loops.

The **while** loop is used for reading data, typically from a database or other data source, or when waiting for user input to end a loop.

The **for** loop is used to iterate through a sequence of data. Because Python uses iterators to simplify access to many kinds of data, the for loop is used in places that would use while in most languages. The most frequent example of this is in reading lines from a file.

while and **for** loops can also have an **else** block, which is always executed unless a **break** statement is executed.

Example

loops_ex.py

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

colors = [ 'red', 'green', 'blue', 'purple', 'pink', 'yellow', 'black' ]  ①

for color in colors:  ②
    print(color)
print()

with open('../DATA/mary.txt') as MARY:  ③
    for line in MARY:    ④
        print(line,end='')  ⑤
print()

while True:  ⑥
    name = input("What is your name? ")  ⑦
    if name.lower() == 'q':
       break  ⑧
    print("Welcome,",name)
```

① create a list

② loop over list

③ open text file for reading

④ loop over lines in file

⑤ print line with extra newline

⑥ loop "forever"

⑦ read input from keyboard

⑧ exit loop

loops_ex.py

```
red
green
blue
purple
pink
yellow
black

Mary had a little lamb,
Its fleece was white as snow,
And everywhere that mary went
The lamb was sure to go

What is your name?  Fred
Welcome, Fred
What is your name?  Amir
Welcome, Amir
What is your name?  Jacinto
Welcome, Jacinto
What is your name?  q
```

Builtins

- 72 builtin functions (as of Python 3.2)
- Not called from an object or package
- Can work on many different data types

Python has many *builtin* functions. These provide generic functionality that is not tied to a particular type or package.

They can be applied to many different data types, but not all functions can be applied to all data types.

Chapter 1 Exercises

Exercise 1-1 (pres_by_state.py)

Using the file presidents.txt (in the DATA folder), count the number of Presidents who were born in each state. In other words, the output of your script should be a list, sorted by state name, with the state and the number of presidents that were born in that state:

TIP First declare a dictionary to hold the data. Then read the file in one line at a time. Split each line into fields using a colon as the separator. Add/update the element of the dictionary where the key is the state. Add 1 each time the state occurs.

expected output

```
Arkansas            1
California          1
Connecticut         1
Georgia             1
etc
```

Exercise 1-2 (pres_dates.py, pres_dates_amb.py)

Write an interactive script that asks for a president's last name. For each president whose last name matches, print out their date of birth and date of death. For presidents who are still alive, print three asterisks for the date.

NOTE Dates of death and term end date might be the string "NONE".

For the ambitious

1. Make the name search case-insensitive
2. Change your script to print out matches for partial names – so "jeff" would find "Jefferson", e.g.

Chapter 2: OS Services

Objectives

- Working with the OS
- Running external programs
- Walking through a directory tree
- Working with path names

The os module

- Provides OS-specific services

The **os** module provides many basic services from the operating system. The interface is the same for different operating systems. These services include file and folder utilities, as well as working with dates and times, running external programs, and many others.

Table 2. The os module

Method or Data	Description
os.path	either posixpath or ntpath
ctermid()	Return name of the controlling terminal
device_encoding()	Return string describing the encoding of the device
dup()	Return a duplicate of a file descriptor.
dup2()	Duplicate file descriptor.
exec…()	Execute file, with different configurations of arguments, environment, etc.
fchdir()	Change to directory of given file descriptor.
fchmod()	Change permissions of file given by file descriptor
fchown()	Change owner/group id of the file given by file descriptor
fdatasync()	force write of file with file descriptor to disk.
fork()	Fork a child process.
forkpty()	Fork a new process with a new pseudo-terminal
fpathconf()	Return the configuration limit name for the file descriptor
fstat()	Return stat result for an open file descriptor.
fstatvfs()	Return stat result for open file descriptor on virtual file system
fsync()	force write of file with filedescriptor to disk.
ftruncate()	Truncate a file to a specified length.
getcwd()	Return unicode string representing current working directory.
getegid()	Return the current process’s effective group id.
getenv()	Get specified environment variable or None/Default (returns string)
getenvb()	Get specified environment variable or None/Default (returns bytes)
geteuid()	Return the current process’s effective user id.
getgid()	Return the current process’s group id.
getgroups()	Return list of supplemental group IDs for the process.
getloadavg()	Return number of processes averaged over 1, 5, and 15 minutes

Method or Data	Description
getlogin()	Return the actual login name.
getpgid()	Call the system call getpgid().
getpgrp()	Return the current process group id.
getpid()	Return the current process id
getppid()	Return the parent's process id.
getresgid()	Return tuple of real, effective, saved group IDs
getresuid()	Return tuple of real, effective, saved user IDs
getsid()	Call the system call getsid().
getuid()	Return the current process's user id.
initgroups()	Initialize the group access list with all groups of which the specified username is a member, plus the specified group id.
isatty()	Return True if file descriptor is an open file descriptor
kill()	Kill a process with a signal.
killpg()	Kill a process group with a signal.
lchown()	Change owner/group of path (don't follow symlinks)
link()	Create a hard link to a file.
listdir()	Return list of all entries in the directory.
lseek()	Set the current position of a file descriptor.
lstat()	Like stat(path), but do not follow symbolic links.
major()	Extracts device major number from a raw device number.
makedev()	Composes a raw device number from major/minor device numbers.
makedirs()	Super-mkdir (like unix mkdir -p)
minor()	Extracts device minor number from a raw device number.
mkdir()	Create a directory.
mkfifo()	Create a FIFO (a POSIX named pipe).
mknod()	Create a filesystem node
nice()	Decrease priority of process by inc and return new priority.

Method or Data	Description
open()	Open a file (for low level IO).
openpty()	Open a pseudo-terminal
pathconf()	Return configuration limit name for file or directory path.
pipe()	Create a pipe.
putenv()	Change or add an environment variable.
read()	Read a file descriptor.
readlink()	Return string representation of symlink target
remove()	Remove a file (same as unlink(path)).
removedirs(name)	Super-rmdir; remove leaf directory and all empty intermediate ones
rename()	Rename a file or directory.
renames()	Super-rename; create directories as necessary
rmdir()	Remove a directory.
setegid()	Set the current process's effective group id.
seteuid()	Set the current process's effective user id.
setgid()	Set the current process's group id.
setgroups()	Set the groups of the current process to list.
setpgid()	Call the system call setpgid().
setpgrp()	Make this process a session leader.
setregid()	Set the current process's real and effective group ids.
setresgid()	Set the current process's real, effective, and saved group ids.
setresuid()	Set the current process's real, effective, and saved user ids.
setreuid()	Set the current process's real and effective user ids.
setsid()	Call the system call setsid().
setuid()	Set the current process's user id.
spawn...()	Execute file with arguments from args in a subprocess.
stat()	Perform a stat system call on the given path.

Method or Data	Description
stat_float_times()	Determine whether os.[lf]stat represents time stamps as float objects.
statvfs()	Perform a statvfs system call on the given path.
strerror()	Translate an error code to a message string.
symlink()	Create a symbolic link
sysconf()	Return an integer-valued system configuration variable.
system()	Execute the command (a string) in a subshell.
tcgetpgrp()	Return the process group associated with the terminal given by a fd.
tcsetpgrp()	Set the process group associated with the terminal given by a fd.
times()	Return tuple of floats indicating process times.
ttyname()	Return the name of the terminal device
umask()	Set the current numeric umask and return the previous umask.
uname()	Return a tuple identifying the current operating system.
unlink()	Remove a file (same as remove(path)).
unsetenv()	Delete an environment variable.
utime()	Set the access and modified time of file
wait...()	Wait for completion of a child process.
walk()	Directory tree generator.
write()	Write a string to a file descriptor.

Paths, directories and file names

- import os.path module
- Many routines for working with file and folder attributes

The os.path module provides many functions for working with file and directory names and paths. These are all about the file and directories *attributes*, not the contents.

Some of the more common methods are

os.path.abspath()
os.path.basename
os.path.dirname()
os.path.getmtime()
os.path.getsize()
os.path.isdir()
os.path.isfile()
os.path.join()
os.path.exists()

Example

paths.py

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

import sys
import os.path

unix_p1 = "bin/spam.txt"  ①
unix_p2 = "/usr/local/bin/ham"  ②

win_p1 = r"spam\ham.doc"    ③
win_p2 = r"\\spam\ham\eggs\toast\jam.doc"  ④

if sys.platform == 'win32': ⑤
    print("win_p1:", win_p1)
    print("win_p2:", win_p2)
    print("dirname(win_p1):", os.path.dirname(win_p1))  ⑥
    print("dirname(win_p2):", os.path.dirname(win_p2))
    print("basename(win_p1):", os.path.basename(win_p1)) ⑦
    print("basename(win_p2):", os.path.basename(win_p2))
    print("isabs(win_p1):", os.path.isabs(win_p1)) ⑧
    print("isabs(win_p2):", os.path.isabs(win_p2))
else:
    print("unix_p1:", unix_p1)
    print("unix_p2:", unix_p2)
    print("dirname(unix_p1):", os.path.dirname(unix_p1)) ⑥
    print("dirname(unix_p2):", os.path.dirname(unix_p2))
    print("basename(unix_p1):", os.path.basename(unix_p1)) ⑦
    print("basename(unix_p2):", os.path.basename(unix_p2))
    print("isabs(unix_p1):", os.path.isabs(unix_p1))  ⑧
    print("isabs(unix_p2):", os.path.isabs(unix_p2))
    print(
        'format("cp spam.txt {}".format(os.path.expanduser("~"))):',  ⑨
        format("cp spam.txt {}".format(os.path.expanduser("~"))),
    )
    print(
        'format("cd {}".format(os.path.expanduser("~root"))):', ⑩
        format("cd {}".format(os.path.expanduser("~root"))),
    )
```

① Unix relative path

② Unix absolute path

③ Windows relative path

④ Windows UNC path

⑤ What platform are we on?

⑥ Just the folder name

⑦ Just the file (or folder) name

⑧ Is it an absolute path?

⑨ ~ is current user's home

⑩ ~NAME is NAME's home

paths.py

```
unix_p1: bin/spam.txt
unix_p2: /usr/local/bin/ham
dirname(unix_p1): bin
dirname(unix_p2): /usr/local/bin
basename(unix_p1): spam.txt
basename(unix_p2): ham
isabs(unix_p1): False
isabs(unix_p2): True
format("cp spam.txt {}".format(os.path.expanduser("~"))): cp spam.txt /Users/jstrick
format("cd {}".format(os.path.expanduser("~root"))): cd /var/root
```

Table 3. os.path methods

Method	Description
abspath(path)	Return normalized absolutized version of the pathname path.
basename(path)	Return the base name of pathname path.
commonprefix(list)	Return the longest path prefix (taken character-by-character) that is a prefix of all paths in list. If list is empty, return the empty string ('').
dirname(path)	Return the directory name of pathname path.
exists(path)	Return True if path refers to an existing path. Returns False for broken symbolic links. May be subject to permissions
lexists(path)	Return True if path refers to an existing path. Returns True for broken symbolic links.
expanduser(path)	On Unix, return the argument with an initial component of "~" or "~user" replaced by that user's home directory. Only "~" is supported on Windows.
expandvars(path)	Return the argument with environment variables expanded. Substrings of the form "$name" or "${name}" are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged.
getatime(path)	Return the time of last access of path. (seconds since epoch).
getmtime(path)	Return the time of last modification of path. (seconds since epoch).
getctime(path)	Return the system's ctime. (seconds since epoch).
getsize(path)	Return the size, in bytes, of path. Raise os.error if path does not exist or cannot be accessed.
isabs(path)	Return True if path is an absolute pathname (begins with a slash).
isfile(path)	Return True if path is an existing regular file. This follows symbolic links.
isdir(path)	Return True if path is an existing directory. Follows symbolic links.
islink(path)	Return True if path refers to a directory entry that is a symbolic link. Always False on Windows.
ismount(path)	Return True if pathname path is a mount point (Unix only).
join(path1[, path2[, ...]])	Join one or more path components intelligently.

Method	Description
normcase(path)	Normalize the case of a pathname. On Unix, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes.
normpath(ph)	Normalize a pathname. This collapses redundant separators and up-level references so that A//B, A/./B and A/foo/../B all become A/B.
realpath(path)	Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.
samefile(path1, path2)	Return True if both pathname arguments refer to the same file or directory (as indicated by device number and i-node number). Raise an exception if a os.stat() call on either pathname fails. Availability: Macintosh, Unix.
sameopenfile(fp1, fp2)	Return True if the file descriptors fp1 and fp2 refer to the same file. Availability: Macintosh, Unix.
samestat(stat1, stat2)	Return True if the stat tuples stat1 and stat2 refer to the same file. These structures may have been returned by fstat(), lstat(), or stat(). Availability: Macintosh, Unix.
split(path)	Split the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash.
splitdrive(path)	Split the pathname path into a pair (drive, tail) where drive is either a drive specification or the empty string. On systems which do not use drive specifications, drive will always be the empty string..
splitext(path)	Split the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period.
splitunc(path)	Split the pathname path into a pair (unc, rest) so that unc is the UNC mount point (such as r'\\host\mount'), if present, and rest the rest of the path (such as r'\path\file.ext'). For paths containing drive letters, unc will always be the empty string. Availability: Windows.
walk(path, visit, arg)	Calls the function visit with arguments (arg, dirname, names) for each directory in the directory tree rooted at path (including path itself, if it is a directory). Note: The newer os.walk() generator supplies similar functionality and can be easier to use. (Like File::Find in Perl)

Method	Description
supports_unicode_filenames()	True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system), and if os.listdir() returns Unicode strings for a Unicode argument. New in version 2.3.

Environment variables

- Shell or OS variables
- Same for Windows and non-Windows
- Syntax

```
value = os.environ[varname]
ivalue = os.environ.get(varname)
value = os.getenv(varname)
value = os.getenv(varname,default)
str2 = os.path.expandvars(str1)
```

There are several ways to access environment variables from Python.

The most direct is to use os.environ, which is a dictionary of the current environment. If a non-existent variable name is specified, a KeyError will be raised, so it is safer to use os.environ.get(varname[,default]) than os.environ[varname].

You can also use the os.getenv(varname[,default]) method. It takes the name of an environment variable and returns that variable's value. An optional second argument provides a default value if the variable is not defined.

Another way to use environment variables is to expand a string that contains them, using the expandvars(string) method of the os.path object. This takes a string containing one or more environment variables and returns the strings with environment variables expanded to their values.

If the variables do not exist in the environment, they are left unexpanded.

Example

getenv_ex.py

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

import sys
import os.path

if sys.platform == 'win32':
    user_key = 'USERNAME'
else:
    user_key = 'USER'

count_key = 'COUNT'

os.environ[count_key] = "42"  ①
print("count is",os.environ[count_key],"user is",os.environ[user_key]) ②
print("count is",os.environ.get(count_key),"user is",os.environ.get(user_key)) #
<3>set
user = os.getenv(user_key)  ④
count = os.getenv(count_key)
print("count is",count,"user is",user)
cmd = "count is ${} user is ${}".format(count_key, user_key)
print("cmd:", cmd)
print(os.path.expandvars(cmd)) ⑤
```

① set environment variable

② os.environ is a dictionary

③ os.environ.get() is safer than os.environ[]

④ os.getenv() is shortcut for os.environ.get()

⑤ expand variables in place; handy for translating shell scripts

getenv_ex.py

```
count is 42 user is jstrick
count is 42 user is jstrick
count is 42 user is jstrick
cmd: count is $COUNT user is $USER
count is 42 user is jstrick
```

Launching external programs

- Different ways to launch programs
  - Just launch (use system())
  - Capture output (use popen())
- import os module
- Use system() or popen() methods

In Python, you can launch an external command using the os module functions **os.system()** and **os.popen()**.

os.system() launches any external command, as though you had typed it at a command prompt. popen() opens a command, returning a file-like object. You can read the output of the command with any of the methods used for a file.

You can open a process for writing as well, by specifying a mode of "w".

TIP For more sophisticated control of processes, see the **subprocess** module.

Example

external_programs.py

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

import sys
import os

os.system("hostname")   ①

with os.popen('netstat -an') as netstat_in:  ②
    for entry in netstat_in:  ③
        if 'ESTAB' in entry:   ④
            print(entry, end='')
print()
```

① Just run "hostname"

② Open command line "netstat -an" as a file-like object

③ Iterate over lines in outout of "netstat -an"

④ Check to see if line contains "ESTAB"

external_programs.py

```
MacBook-Pro-7.local
tcp6       0      0  2606:a000:1120:4.50067 2607:f8b0:4004:8.443   ESTABLISHED
tcp6       0      0  2606:a000:1120:4.50066 2607:f8b0:4004:8.443   ESTABLISHED
tcp6       0      0  2606:a000:1120:4.50065 2607:f8b0:4004:8.443   ESTABLISHED
tcp4       0      0  192.168.1.137.50063    54.148.248.31.443      ESTABLISHED
tcp6       0      0  2606:a000:1120:4.50062 2607:f8b0:4004:8.443   ESTABLISHED
tcp6       0      0  2606:a000:1120:4.50051 2607:f8b0:4004:8.443   ESTABLISHED
tcp6       0      0  2606:a000:1120:4.50048 2607:f8b0:4004:8.443   ESTABLISHED
tcp6       0      0  2606:a000:1120:4.50043 2607:f8b0:400d:c.443   ESTABLISHED
tcp6       0      0  2606:a000:1120:4.50038 2607:f8b0:4004:8.443   ESTABLISHED
tcp4       0      0  192.168.1.137.50020    162.125.18.133.443     ESTABLISHED
tcp4       0      0  192.168.1.137.49993    192.168.1.123.8008     ESTABLISHED
tcp6       0      0  2606:a000:1120:4.49954 2a03:2880:f011:1.443   ESTABLISHED
tcp4       0      0  192.168.1.137.49941    162.125.18.133.443     ESTABLISHED
tcp4       0      0  192.168.1.137.49891    199.195.144.123.80     ESTABLISHED
```

...

Walking directory trees

- Use os.walk()
- Returns tuple for each directory
- Tuple contains directory path, subdirectories, and files

The os.walk method provides a way to easily walk a directory tree. It provides an iterator for a directory and all its subdirectories. For each directory, it returns a tuple with three values.

The first element is the full (absolute) path to the directory; the second element is a list of the directory's subdirectories (relative names); the third element is a list of the non-directory entries in the subdirectory (also relative names).

Be sure to use os.path.join() to put together the directory and the file or subdirectory name.

Do not use "dir" as a variable when looping through the iterator, because it will overwrite Python's builtin **dir** function.

Example

walk_ex.py

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

import os
'''print size of every python file whose name starts with "m" '''

START_DIR = ".."   # start in root of student files  ①

def main():
    for currdir,subdirs,files in os.walk(START_DIR): ②
        for file in files: ③
            if file.endswith('.py') and file.startswith('m'):
                fullpath = os.path.join(currdir,file) ④
                fsize = os.path.getsize(fullpath)
                print("{:8d} {:s}".format(fsize, fullpath))

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

① starting location

② walk folder tree

③ loop over file names

④ get file path

walk_ex.py

```
     228 ../custom/pynavy/1.0/StudentFiles/unix/pynavy/EXAMPLES/mary_gen.py
     300 ../custom/pynavy/1.0/StudentFiles/unix/pynavy/EXAMPLES/meta_functions.py
     167 ../custom/pynavy/1.0/StudentFiles/unix/pynavy/EXAMPLES/multi_ex.py
     400 ../custom/pynavy/1.0/StudentFiles/unix/pynavy/EXAMPLES/monkeypatch.py
     469 ../custom/pynavy/1.0/StudentFiles/unix/pynavy/EXAMPLES/modtest.py
     938 ../custom/pynavy/1.0/f5_week2/EXAMPLES/mammal.py
     828 ../custom/pynavy/1.0/f5_week2/EXAMPLES/moreindex.py
     175 ../custom/pynavy/1.0/f5_week2/EXAMPLES/mathop.py
     167 ../custom/pynavy/1.0/f5_week2/EXAMPLES/multi_ex.py
     469 ../custom/pynavy/1.0/f5_week2/EXAMPLES/modtest.py
```

...

Chapter 2 Exercises

Exercise 2-1 (path_files.py)

List each component of your PATH environment variable, together with the number of files it contains. This is the set of files you can execute from the command line without specifying a their path. Output should look something like this (for Windows, the paths will look different, but the idea is the same):

```
/usr/bin            2376
/usr/local/bin        17
/usr/local/sbin        1
/usr/sbin            263
```

TIP Use os to get the pathsep value; then use os.listdir to get the contents of each directory after splitting PATH.

Exercise 2-2 (oldest_file.py)

Write a script that, given a directory on the command line, prints out the oldest file in that directory. If there is more than one file sharing the oldest timestamp, print any one of them.

TIP Use os.path.getmtime()

Exercise 2-3 (all_python_lines.py)

Write a script that finds all the Python files (.py) in the student files (starting at py3interm), and counts the total number of lines in all of them.

Chapter 3: Dates and Times

Objectives

- Creating date and time objects
- Making date/time calculations
- Getting Unix-style timestamps
- Formatting dates
- Parsing dates from strings
- Working with calendars

Python modules for dates and times

- standard library
  - datetime
  - time
  - calendar
- included with Anaconda
  - dateutil
- other
  - arrow

Python provides several modules for working with dates and times.

The standard library includes the datetime module, which provides the date, datetime, and time classes. These basic classes cover most date/time related needs.

The time module in the standard library works with times (including dates) in either epoch time or time tuple format.

Finally, the calendar module has tools for generating calendars in text and HTML.

If you need to parse dates from text, or work with time zones, the dateutil module, which is not in the standard library, but which is included with Anaconda, is very useful.

The **arrow** module (http://crsmithdev.com/arrow) attempts to consolidate all date/time related functionality into a single, easy-to-use module.

NOTE In the ideal world, there would be one comprehensive module in the standard library to handle all time-related issues, but that may not ever happen.

Ways to store dates and times

- date, datetime, time classes from datetime module
- large integer (epoch time)
- time tuple

There are three ways (at least) in which dates and times can be stored.

The classes from the datetime module store time data as date, time, or datetime objects.

Routines in the time module (not datetime.time) generally return a given date/time as a Unix epoch time (number of seconds since December 31, 1969), but some functions return a 9-element tuple of (year, month, day, hour, minute, second, weekday, days since December 31, and DST flag).

The dateutil module uses the classes from datetime, as does arrow.

Basic dates and times

- datetime module
- Provides four classes
  - datetime
  - date
  - time
  - timedelta

Python provides the datetime module for manipulating dates and times. Once you have created date or time objects, you can combines them and extract the time units you need.

The date object contains a valid year, month and day. The time object contains a valid hour, minute, second, and microsecond. The datetime object combines a date with a time, and the timedelta object is the difference between two of the preceding; a timedelta contains days, seconds, and microseconds.

To prevent conflicts with the separate time module, it's not a bad idea to import time from datetime with an alias: from datetime import time as Time

Example

datetime_ex.py

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

from datetime import datetime, date, timedelta

print("date.today():",date.today()) ①

now = datetime.now()  ②
print("now.day:", now.day)  ③
print("now.month:", now.month)
print("now.year:", now.year)
print("now.hour:", now.hour)
print("now.minute:", now.minute)
print("now.second:", now.second)

d1 = datetime(2007, 6, 13) ④
d2 = datetime(2007, 8, 24)

d3 = d2 - d1  ⑤

print("raw time delta:", d3)
print("time delta days:", d3.days)  ⑥

interval = timedelta(10)  ⑦
print("interval:", interval)

d4 = d2 + interval  ⑧
d5 = d2 - interval
print("d2 + interval:", d4)
print("d2 - interval:", d5)
print()

t1 = datetime(2013, 8, 24, 10, 4, 34)  ⑨
t2 = datetime(2015, 8, 24, 22, 8, 1)
t3 = t2 - t1

print("datetime(2007, 8, 24, 10, 4, 34):", t1)
print("datetime(2007, 8, 24, 22, 8, 1):", t2)
print("time diff (t2 - t1):", t3)
```

① get today's date

② get today's date and time

③ a datetime object has attributes for date/time parts

④ create a date object

⑤ date objects can be subtracted from other date objects

⑥ timedelta has days, seconds, and microseconds

⑦ create a timedelta of 10 days

⑧ add timedelta to datetime

⑨ create a datetime object

datetime_ex.py

```
date.today(): 2018-03-26
now.day: 26
now.month: 3
now.year: 2018
now.hour: 16
now.minute: 38
now.second: 7
raw time delta: 72 days, 0:00:00
time delta days: 72
interval: 10 days, 0:00:00
d2 + interval: 2007-09-03 00:00:00
d2 - interval: 2007-08-14 00:00:00

datetime(2007, 8, 24, 10, 4, 34): 2013-08-24 10:04:34
datetime(2007, 8, 24, 22, 8, 1): 2015-08-24 22:08:01
time diff (t2 - t1): 730 days, 12:03:27
```

Formatting dates and times

- All date/time classes in datetime implement strftime()
- Returns formatted string
- Template contains literal text plus "directives"

All of the classes in datetime except timedelta implement strftime() for creating a custom time format. The usage is similar to legacy string formatting; the "directives", or placeholders, start with a percent sign.

Example

datetime_fmt.py

```
#!/usr/bin/env python
from datetime import datetime as DateTime

# Bill Gates's birthday
gates_bd = DateTime(1955,10,28, 22, 0, 0) ①

print(gates_bd) ②
print()

print(gates_bd.strftime('Bill was born on %B %d, %Y at %I:%M %p')) ③
print()

print(gates_bd.strftime('BG: %m/%d/%y')) ③
print()

print(gates_bd.strftime('Mr. Gates was born %d-%b-%Y')) ③
print()

print(gates_bd.strftime('log entry: %Y-%m-%d')) ③
print()
```

① Create a datetime object

② Print using default format

③ Format using strftime()

datetime_fmt.py

```
1955-10-28 22:00:00

Bill was born on October 28, 1955 at 10:00 PM

BG: 10/28/55

Mr. Gates was born 28-Oct-1955

log entry: 1955-10-28
```

Table 4. Date Format Directives

Directive	Meaning	Notes
%a	Locale's abbreviated weekday name	
%A	Locale's full weekday name	
%b	Locale's abbreviated month name	
%B	Locale's full month name	
%c	Locale's appropriate date and time representation	
%d	Day of the month as a decimal number [01,31]	
%f	Microsecond as a decimal number [0,999999], zero-padded on the left	(1)
%H	Hour (24-hour clock) as a decimal number [00,23]	
%I	Hour (12-hour clock) as a decimal number [01,12]	
%j	Day of the year as a decimal number [001,366]	
%m	Month as a decimal number [01,12]	
%M	Minute as a decimal number [00,59]	
%p	Locale's equivalent of either AM or PM.	(2)
%S	Second as a decimal number [00,61]	(3)
%U	Week number (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0	(4)
%w	Weekday as a decimal number [0(Sunday),6]	
%W	Week number (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0	(4)
%x	Locale's appropriate date representation	
%X	Locale's appropriate time representation	
%y	Year without century as a decimal number [00,99]	
%Y	Year with century as a decimal number	
%z	UTC offset in the form +HHMM or -HHMM (empty string if the the object is naive)	(5)

Directive	Meaning	Notes
%Z	Time zone name (empty string if the object is naive)	
%%	A literal '%' character	

NOTE

1. When used with the strptime() method, the %f directive accepts from one to six digits and zero pads on the right.
2. When used with the strptime() method, the %p directive only affects the output hour field if the %I directive is used to parse the hour.
3. The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.
4. When used with the strptime() method, %U and %W are only used in calculations when the day of the week and the year are specified.
5. For example, if utcoffset() returns timedelta(hours=-3, minutes=-30), %z is replaced with the string '-0330'.

Parsing date/time strings

- time.strptime() returns time tuple
- datetime.datetime.strptime returns datetime
- Use same directives as strftime()

To parse dates from strings, use one of the strptime() methods. There are two different versions, which return different representations of the date and time. In both cases, you pass strptime() the time string, followed by a template that uses the same directives as strftime().

time.strptime() returns a time tuple. It is a named tuple, so to get the year, for instance, you can use either tuple[0] or tuple.tm_year. datetime.strptime() returns a new datetime object.

Example

datetime_parse.py

```
#!/usr/bin/env python
from datetime import datetime as DateTime
import time

data = (
    ('Jan 1, 1970','%b %d, %Y'),
    ('01/01/70','%m/%d/%y'),
    ('1970-01-01', '%Y-%m-%d'),
    ('Jan 1, 1970 at 3:14 pm', '%b %d, %Y at %I:%M %p'),
)

for date_str, parse_template in data:
    time_tuple = time.strptime(date_str, parse_template)  ①
    print(time_tuple)
    print()

    parsed_date = DateTime.strptime(date_str, parse_template) ②
    print(parsed_date)
    print('-' * 20)
```

① Parse into timetuple using time module

② Parse into datetime.datetime using datetime.datetime class

datetime_parse.py

```
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=-1)

1970-01-01 00:00:00
--------------------
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=-1)

1970-01-01 00:00:00
--------------------
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=-1)

1970-01-01 00:00:00
--------------------
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=15, tm_min=14, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=-1)

1970-01-01 15:14:00
--------------------
```


Parsing dates the easier way

- Use dateutil.parser
- Understands many different formats
- No need for directives

The dateutil.parser module provides a parse() function that will parse most any date or date/time string, without the need to put together directives, as with strptime().

The parse() function has several flags to deal with variations in date format, such as European-style day first ("1/4/15").

The value returned by parse() is a normal python datetime.datetime.

Example

date_parsing_easy.py

```
#!/usr/bin/env python
#
from dateutil import parser

date_strings = [  ①
    'April 1, 2015',
    '4/1/2015',
    'Apr 1, 2015',
    'Apr 1 2015',
    '04/01/2015',
    '1 Apr 2015',
    'April 1st, 2015',
    'April 1, 2015 8:09',
    '4/1/2015 8:09 PM',
    'Apr 1, 2015 5 AM',
    'Apricot 4, 341',
]

for date_string in date_strings:
    try:
        dt = parser.parse(date_string)  ②
        print(dt)
    except ValueError as err:
        print("Can't parse", date_string)
```

① list of assorted date strings

② parse with dateutil.parser.parse() — doesn't need a template

date_parsing_easy.py

```
2015-04-01 00:00:00
2015-04-01 00:00:00
2015-04-01 00:00:00
2015-04-01 00:00:00
2015-04-01 00:00:00
2015-04-01 00:00:00
2015-04-01 00:00:00
2015-04-01 08:09:00
2015-04-01 20:09:00
2015-04-01 05:00:00
Can't parse Apricot 4, 341
```

Converting timestamps

- Use datetime.fromtimestamp() to get datetime object
- Use time.localtime() to get time tuple
- time.time() returns current time as timestamp

Certain functions, such as os.getmtime() or os.getatime(), return a Unix-style timestamp. This is the number of seconds since January 1, 1970, a point in time usually called the "Unix Epoch". You may also get this version of a date and time from other sources.

To convert this to a useful format, use datetime.fromtimestamp() to convert to a datetime object, or time.localtime() to convert to a time tuple.

The time.time() method returns the current date and time as a timestamp.

Converting to a timestamp takes two steps. First, get the date and time to a time tuple, and then call time.mktime().

```
adate = datetime.datetime(1975, 4, 2, 12, 9, 55)
timestamp = time.mktime(adate.timetuple())
```

Example

conv_timestamp.py

```
#!/usr/bin/env python
from datetime import datetime
import time

adate = datetime(1975, 4, 2, 12, 9, 55) ①
timestamp = time.mktime(adate.timetuple()) ②
today = time.time() ③


for ts in 86400, timestamp, today: ④
    tm = time.localtime(ts)  ⑤
    print(tm.tm_year, tm.tm_mon, tm.tm_mday)
    print()

    dt = datetime.fromtimestamp(ts)  ⑥
    print(dt.year, dt.month, dt.day)
    print('-' * 20)
```

① create a Python datetime object

② extract a timetuple, and use it to make a Unix epoch time value (# seconds since 1/1/70)

③ get epoch time for right now

④ loop through the 3 epoch times. NOTE: Windows requires minimum value of 86400; non-Windows platforms could use 0

⑤ convert epoch time to timetuple

⑥ convert epoch time to Python datetime object

conv_timestamp.py

```
1970 1 1

1970 1 1
--------------------
1975 4 2

1975 4 2
--------------------
2018 3 26

2018 3 26
--------------------
```

Time zones

- Not as easy as you might expect
- datetime.datetime has UTC-oriented functions
  - utcnow()
  - utctimetuple()
  - utcfromtimestamp()
- Use dateutil module (not part of standard library)

Support for time zones is fairly weak in the standard library. The datetime.datetime class has some UTC-based methods.

The dateutil module supports time zones "out of the box". In addition, it has features that combine the most useful parts of the datetime and calendar packages.

If you use dateutil to parse a date/time string that has a timezone, it will automatically add that to the date or datetime object. The time zone is usually specified as the offset from UTC, so EDT would be '-04:00' and PST would be '-08:00'.

Generating calendars

- Use calendar module
- Output can be text or HTML
- Create TextCalendar or HTMLCalendar object
- formatmonth(), formatyear() generate formatted calendars

The calendar module provides two classes for generating calendars. TextCalendar generates calendars as plain text, and HTMLCalendar generates calendars as HTML.

Both support formatmonth() and formatyear() methods. TextCalendar, for convenience, also provides prmonth() and pryear() methods, which call print on the strings returned by formatmonth() and formatyear(). For year calendars, an option argument specifies the number of months per row.

HTMLCalendar adds formatyearpage(), which returns an entire Web page, complete with headers and optional CSS filename. You can add optional parameters for the width of the day column, and the number of lines for each week. The defaults are 2 for the day, and 1 for the lines per week.

Example

calendar_ex.py

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

import os
import calendar
import webbrowser

tcal = calendar.TextCalendar()  ①
print(tcal.formatmonth(2012,1)) ②

print()

hcal = calendar.HTMLCalendar()   ③
formatted_month = hcal.formatmonth(2012,1)  ④

html_file_name = 'sample_calendar.html'

with open(html_file_name, 'w') as calendar_out:
    calendar_out.write(formatted_month)
    webbrowser.open("file://" + os.path.realpath(html_file_name))
```

① create a text calendar object

② format one month as text

③ create an HTML calendar object

④ format one month as HTML

calendar_ex.py

```
    January 2012
Mo Tu We Th Fr Sa Su
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Table 5. Calendar methods

Methods	Returns
isleap(year)	True if leap year, False otherwise
leapdays(year1,year2)	Number of leap years between year1 (inclusive) and year2 (exclusive)
weekday(year,month,day)	Weekday (0 is Monday) for given date
weekheader(width)	String of abbreviated weekday names; width is width of one weekday (longer widths give more characters, up to 3 – 'M', 'Mo', 'Mon', ' Mon', etc.)
monthrange(year,month)	Weekday of first day of month and number of days in month, as a tuple
monthcalendar(year,month)	A two-dimensional list representing a month's calendar. Each row represents a week; days outside of the month are represented by zeros.
prmonth(year,month)	Prints calendar for specified month (prints return value of calendar.month())
month(year,month)	Return a multi-line string representing one month
prcal(year)	Prints calendar for specified year (prints return value of calendar.year())
calendar(year)	Return a multi-line string representing one year
timegm(tuple)	Converts timetuple tuple to a Unix timestamp (this is the opposite of time.gmtime())

Table 6. Calendar Data

Property	List of
day_name	Full names of days of the week ('Monday', 'Tuesday', etc.)
day_abbr	Abbreviations of days of the week ('Mon', 'Tue', etc.)
month_name	Full names of months (index 0 is empty string) — ('', 'January', 'February', etc.)
month_abbr	Abbreviations of months ('', 'Jan', 'Feb', etc)

Chapter 3 Exercises

Exercise 3-1 (file_date.py)

Write a script which accepts one or more filenames on the command line, and prints out each file name with its date of last modification in the format 'Mar 12, 2013'.

Exercise 3-2 (ww2_days.py)

Write a script to calculate the number of days between December 7, 1941 and August 15, 1945.

Exercise 3-3 (fun_with_dates.py)

Given the following dates, write a script to print out the date, the day of the week it fell on, and True if it occurs in a leap year or False otherwise.

October 31, 1956
September 22, 1952
August 27, 1990

Exercise 3-4 (youngest_pres.py)

Write a script to print out the presidents sorted by the age at which they took office, youngest first. Get the data from the presidents.txt file in DATA.

TIP Subtract the date were born from the date they took office.

Chapter 4: Binary Data

Objectives

- Know the difference between text and binary data
- Open files in text or binary mode
- Use Struct to process binary data streams

"Binary" (raw, or non-delimited) data

- Open file in binary mode
- Use read()
- Specify number of bytes to read
- Read entire file when no size given
- Returns **bytes** object

A file can be opened in binary mode. This allows for "raw", or non-delimited reads, which do not treat newlines and carriage returns as special.

In binary mode, read() will return a **bytes** object (array of 8-bit integers), not a Python string (array of Unicode characters). Use **.decode()** to convert the bytes object to a string.

Use **write()** to write raw data to a file.

Use **seek()** to position the next read, and **tell()** to determine the current location within the file.

Binary vs Text data

- Networks use binary data
- Convert to standard if mixing platforms
- Need to know layout of data

When you read data from a network application, such as getting the HTML source of a web page, it is retrieved as binary data, even thought it is "text". It is typically encoded as ASCII or UTF-8. This is represented by a **bytes** object, which is an array of bytes.

To convert a bytes object to a string, call the **decode()** method. When going the other direction, as in writing some text out to a network application, you will need to convert from a Python string, which is an in-memory representation, to a string of bytes. Do do this, call the string's **encode()** method.

Using Struct

- Struct class from struct module
- Translates between Python to native/standard formats
- Format string describes data layout

If you need to process a **raw** binary file

The **struct** module provides the Struct class. You can instantiate a Struct with a format string representing the binary data layout. From the instance, you can call **unpack()** to decode a binary stream, or **pack()** to encode it.

The **size** property is the number of bytes needed for the data.

The format string describes the data layout using format codes. Each code is a letter representing a data type, and can be preceded with a size or repeat count (depending on data type), and a prefix which specifies the byte order and alignment.

"Native" byte order or alignment refers to the same byte order or alignment used by the C compiler on the current platform. "Standard" refers to a standard set of sizes for typical numerical objects, such as shorts, ints, longs, floats and doubles. The default is native.

Table 7. Struct format codes

Format	C Type	Python Type	Standard size	Notes
x	pad byte	no value	n/a	
c	char	bytes of length 1	1	
b	signed char	integer	1	(1),(3)
B	unsigned char	integer	1	(3)
?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2),(3)
Q	unsigned long long	integer	8	(2), (3)
n	ssize_t	integer		(4)
N	size_t	integer		(4)
f	float	float	4	(5)
d	double	float	8	(5)
s	char[]	bytes		
p	char[]	bytes		
P	void *	integer		(6)

Notes

1. The '?' conversion code corresponds to the _Bool type defined by C99. If this type is not available, it is simulated using a char. In standard mode, it is always represented by one byte.
2. The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes.

3. When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a *index*() method then that method is called to convert the argument to an integer before packing.
4. The 'n' and 'N' conversion codes are only available for the native size (selected as the default or with the '@' byte order character). For the standard size, you can use whichever of the other integer formats fits your application.
5. For the 'f' and 'd' conversion codes, the packed representation uses the IEEE 754 binary32 (for 'f') or binary64 (for 'd') format, regardless of the floating-point format used by the platform.
6. The 'P' format character is only available for the native byte ordering (selected as the default or with the '@' byte order character). The byte order character '=' chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the 'P' format is not available.

Table 8. Struct byte order/size/alignment flags

Flag	**Byte order**	**Size and byte alignment**
@ default	Native	Native
=	Native	Standard
<	Little-endian	Standard
> or !	Big-endian	Standard

Example

binary_data.py

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

from struct import Struct

values = 7, 6, 42.3, b'Guido' ①

demo = Struct('iif10s')  ②

print("Size of data: {} bytes".format(demo.size)) ③

binary_stream = demo.pack(*values) ④

int1, int2, float1, raw_bytes = demo.unpack(binary_stream) ⑤
str1 = raw_bytes.decode().rstrip('\x00')  ⑥

print(raw_bytes)
print(int1, int2, float1, str1)
```

① create some assoted values

② create Struct object with desired data layout

③ size property gives size of data in bytes

④ pack() converts values into binary stream using format

⑤ unpack() converts binary stream into list of values

⑥ decode the raw bytes into a string, and strip off trailing null bytes (that were added by pack())

binary_data.py

```
Size of data: 22 bytes
b'Guido\x00\x00\x00\x00\x00'
7 6 42.29999923706055 Guido
```

Example

parse_bmp.py

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

from struct import Struct

# short int short short int (native, unsigned)
s = Struct('=HIHHI') ①

with open('../DATA/chimp.bmp','rb') as CHIMP:
    chimp_bmp = CHIMP.read(14)  ②

(signature,size,reserved1,reserved2,offset) = s.unpack(chimp_bmp)  ③

print("signature:",signature)  ④
print('size:',size)
print('reserved1:',reserved1)
print('reserved2:',reserved2)
print('offset:',offset)
```

① define layout of bitmap header

② read the first 14 bytes of bitmap file in binary mode

③ unpack the binary header into individual values

④ output the individual values

parse_bmp.py

```
signature: 19778
size: 5498
reserved1: 0
reserved2: 0
offset: 1074
```

Example

read_binary.py

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

# print out a file 10 bytes at a time

with open("../DATA/parrot.txt", "rb") as parrot_in: ①
    while True:
        chunk = parrot_in.read(10)  ②
        if chunk == b"":   ③
            break
        print(chunk.decode()) ④
```

① Add "b" to "r", "w", or "a" for binary mode

② Use read() to read a specified number of bytes

③ Read returns **bytes**, not **str**

④ Use decode() to convert bytes to str

read_binary.py | head -10

```
So there's
 this fell
a with a p
arrot. And
 this parr
ot swears

like a sai
lor, I mea
n he's a p
```

Chapter 4 Exercises

Exercise 4-1 (demystify.py)

Write a program that prints out every third byte (starting with the first byte) of the file named **mystery**. The output will be an ASCII art picture.

TIP read the file into a bytes object (be sure to use binary mode), then use slice notation to select the bytes, then decode into a string for printing.

Exercise 4-2 (pypuzzle.py)

The file **puzzle.data** has a well-known name encoded in it. The ASCII values of the characters in the name are represented by a series of integers and floats of various sizes.

The layout is: float, int, float, int, float, short, unsigned short, float, unsigned int, double, float, double, unsigned int, int, unsigned int, short

To decode, read the file into a bytes object and use a Struct object to decode the raw data into the values. Then convert the values into integers, and use chr() to convert the integers into ASCII characters. Finally, you can join the characters together and print them out.

Chapter 5: Pythonic Programming

Objectives

- Learn what makes code "Pythonic"
- Understand some Python-specific idioms
- Create lambda functions
- Perform advanced slicing operations on sequences
- Distinguish between collections and generators

The Zen of Python

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one-- and preferably only one --obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than **right** now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea — let's do more of those!
>
> — Tim Peters, from PEP 20

Tim Peters is a longtime contributor to Python. He wrote the standard sorting routine, known as "timsort".

The above text is printed out when you execute the code "import this". Generally speaking, if code follows the guidelines in the Zen of Python, then it's Pythonic.

Tuples

- Fixed-size, read-only
- Collection of related items
- Supports some sequence operations
- Think 'struct' or 'record'

A **tuple** is a collection of related data. While on the surface it seems like just a read-only list, it is used when you need to pass multiple values to or from a function, but the values are not all the same type

To create a tuple, use a comma-separated list of objects. Parentheses are not needed around a tuple unless the tuple is nested in a larger data structure.

A tuple in Python might be represented by a struct or a "record" in other languages.

While both tuples and lists can be used for any data:

- Use a list when you have a collection of similar objects.
- Use a tuple when you have a collection of related objects, which may or may not be similar.

TIP To specify a one-element tuple, use a trailing comma, otherwise it will be interpreted as a single object: color = 'red',

Example

```
hostinfo = ( 'gemini','linux','ubuntu','hardy','Bob Smith' )

birthday = ( 'April',5,1978 )
```

Iterable unpacking

- Copy iterable to list of variables
- Can have one wild card
- Frequently used with list of tuples

When you have an iterable such as a tuple or list, you access individual elements by index. However, `spam[0]` and `spam[1]` are not so readable compared to `first_name` and `company`. To copy an iterable to a list of variable names, just assign the iterable to a comma-separated list of names:

```
birthday = ( 'April',5,1978 )
month, day, year = birthday
```

You may be thinking "why not just assign to the variables in the first place?". For a single tuple or list, this would be true. The power of unpacking comes in the following areas:

- Looping over a sequence of tuples
- Passing tuples (or other iterables) into a function

Example

unpacking_people.py

```
#!/usr/bin/env python
# (c) 2018 CJ Associates
#


people = [  ①
    ('Melinda', 'Gates', 'Gates Foundation'),
    ('Steve', 'Jobs', 'Apple'),
    ('Larry', 'Wall', 'Perl'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Bill', 'Gates', 'Microsoft'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey','Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linus', 'Torvalds', 'Linux'),
]


for first_name, last_name, org in people:  ②
    print("{} {}".format(first_name, last_name))
```

① A list of 3-element tuples

② The for loop unpacks each tuple into the three variables.

unpacking_people.py

```
Melinda Gates
Steve Jobs
Larry Wall
Paul Allen
Larry Ellison
Bill Gates
Mark Zuckerberg
Sergey Brin
Larry Page
Linus Torvalds
```

Unpacking function arguments

- Go from iterable to list of items
- Use * or **

Sometimes you need the other end of iterable unpacking. What do you do if you have a list of three values, and you want to pass them to a method that expects three positional arguments? One approach is to use the individual items by index. A more Pythonic approach is to use * to *unpack* the iterable into individual items:

Use a single asterisk to unpack a list or tuple (or similar iterable); use two asterisks to unpack a dictionary or similar.

In the example, see how the list **HEADINGS** is passed to .format(), which expects individual parameters, not *one parameter* containing multiple values.

Example

unpacking_function_args.py

```
#!/usr/bin/env python
# (c) 2018 CJ Associates
#

people = [ ①
    ('Joe', 'Schmoe', 'Burbank', 'CA'),
    ('Mary', 'Rattburger', 'Madison', 'WI'),
    ('Jose', 'Ramirez', 'Ames', 'IA'),
]

def person_record(first_name, last_name, city, state): ②
    print("{} {} lives in {}, {}".format(first_name, last_name, city, state))

for person in people:  ③
    person_record(*person)  ③
```

① list of 4-element tuples

② function that takes 4 parameters

③ person is a tuple (one element of people list)

④ *person unpacks the tuple into four individual parameters

unpacking_function_args.py

```
Joe Schmoe lives in Burbank, CA
Mary Rattburger lives in Madison, WI
Jose Ramirez lives in Ames, IA
```

Example

shoe_sizes.py

```
#!/usr/bin/env python
#
BARLEYCORN = 1 / 3.0
CM_TO_INCH = 2.55
MENS_START_SIZE = 12
WOMENS_START_SIZE = 10.5

FMT = '{:6.1f} {:8.2f} {:8.2f}'
HEADFMT = '{:>6s} {:>8s} {:>8s}'

HEADINGS = ['Size', 'Inches', 'CM']

SIZE_RANGE = []
for i in range(6,14):
    SIZE_RANGE.extend([i, i + .5])

def main():
    for heading, flag in [("MEN'S", True), ("WOMEN'S", False)]:
        print(heading)
        print((HEADFMT.format(*HEADINGS))) ①
        for size in SIZE_RANGE:
            inches, cm = get_length(size)
            print(FMT.format(size, inches, cm))

        print()

def get_length(size, mens=True):
    if mens:
        start_size = MENS_START_SIZE
    else:
        start_size = WOMENS_START_SIZE

    inches = start_size - ((start_size - size) * BARLEYCORN)
    cm = inches * CM_TO_INCH
    return inches, cm

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

① format expects individual arguments for each placeholder; the asterisk unpacks HEADINGS into individual strings

shoe_sizes.py

```
MEN'S
  Size   Inches       CM
   6.0    10.00    25.50
   6.5    10.17    25.92
   7.0    10.33    26.35
   7.5    10.50    26.77
   8.0    10.67    27.20
   8.5    10.83    27.62
```

...

The sorted() function

- Returns a sorted copy of any collection
- Customize with named keyword parameters

```
key=
reverse=
```

The sorted() builtin function returns a sorted copy of its argument, which can be any iterable.

You can customize sorted with the **key** parameter.

Example

basic_sorting.py

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

"""Basic sorting example"""

fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",
"ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear", "banana",
"Tamarind", "persimmon", "elderberry", "peach", "BLUEberry", "lychee",
 "grape" ]

sorted_fruit = sorted(fruit) ①

print(sorted_fruit)
```

① sorted() returns a list

basic_sorting.py

```
['Apple', 'BLUEberry', 'FIG', 'Kiwi', 'ORANGE', 'Tamarind', 'Watermelon', 'apricot',
'banana', 'cherry', 'date', 'elderberry', 'grape', 'guava', 'lemon', 'lime',
'lychee', 'papaya', 'peach', 'pear', 'persimmon', 'pomegranate']
```

Custom sort keys

- Use **key** parameter
- Specify name of function to use
- Key function takes exactly one parameter
- Useful for case-insensitive sorting, sorting by external data, etc.

You can specify a function with the **key** parameter of the sorted() function. This function will be used once for each element of the list being sorted, to provide the comparison value. Thus, you can sort a list of strings case-insensitively, or sort a list of zip codes by the number of Starbucks within the zip code.

The function must take exactly one parameter (which is one element of the sequence being sorted) and return either a single value or a tuple of values. The returned values will be compared in order.

You can use any builtin Python function or method that meets these requirements, or you can write your own function.

TIP The lower() method can be called directly from the builtin object str. It takes one string argument and returns a lower case copy.

```
sorted_strings = sorted(unsorted_strings, key=str.lower)
```

Example

custom_sort_keys.py

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

fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon",
    "Kiwi", "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG",
    "pear", "banana", "Tamarind", "persimmon", "elderberry", "peach",
    "BLUEberry", "lychee", "grape" ]

def ignore_case(e):    ①
    return e.lower()  ②

fs1 = sorted(fruit, key=ignore_case)  ③
print("Ignoring case:")
print(" ".join(fs1), end="\n\n")

def by_length_then_name(e):
    return (len(e), e.lower())  ④

fs2 = sorted(fruit,key=by_length_then_name)
print("By length, then name:")
print(" ".join(fs2))
print()

nums = [800, 80, 1000, 32, 255, 400, 5, 5000]

n1 = sorted(nums)    ⑤
print("Numbers sorted numerically:")
for n in n1:
    print(n, end=' ')
print("\n")

n2 = sorted(nums, key=str)   ⑥
print("Numbers sorted as strings:")
for n in n2:
    print(n, end=' ')
print()
```

① Parameter is *one* element of iterable to be sorted

② Return value to sort on

③ Specify function with named parameter **key**

④ Key functions can return tuple of values to compare, in order

⑤ Numbers sort numerically by default

⑥ Sort numbers as strings

custom_sort_keys.py

```
Ignoring case:
Apple apricot banana BLUEberry cherry date elderberry FIG grape guava Kiwi lemon
lime lychee ORANGE papaya peach pear persimmon pomegranate Tamarind Watermelon

By length, then name:
FIG date Kiwi lime pear Apple grape guava lemon peach banana cherry lychee ORANGE
papaya apricot Tamarind BLUEberry persimmon elderberry Watermelon pomegranate

Numbers sorted numerically:
5 32 80 255 400 800 1000 5000

Numbers sorted as strings:
1000 255 32 400 5 5000 80 800
```

Example

sort_holmes.py

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

import re

books = [
    "A Study in Scarlet",
    "The Sign of the Four",
    "The Hound of the Baskervilles",
    "The Valley of Fear",
    "The Adventures of Sherlock Holmes",
    "The Memoirs of Sherlock Holmes",
    "The Return of Sherlock Holmes",
    "His Last Bow",
    "The Case-Book of Sherlock Holmes",
]

rx_article = re.compile(r'^(the|a|an)\s+', re.I)  ①

def strip_articles(title):  ②
    stripped_title = rx_article.sub('', title.lower())  ③
    return stripped_title

for book in sorted(books, key=strip_articles):  ④
    print(book)
```

① compile regex to match leading articles

② create function which takes element to compare and returns comparison key

③ strip off article and convert title to lower case

④ sort using custom function

sort_holmes.py

```
The Adventures of Sherlock Holmes
The Case-Book of Sherlock Holmes
His Last Bow
The Hound of the Baskervilles
The Memoirs of Sherlock Holmes
The Return of Sherlock Holmes
The Sign of the Four
A Study in Scarlet
The Valley of Fear
```

Lambda functions

- Short cut function definition
- Useful for functions only used in one place
- Frequently passed as parameter to other functions
- Function body is an expression; it cannot contain other code

A **lambda function** is a brief function definition that makes it easy to create a function on the fly. This can be useful for passing functions into other functions, to be called later. Functions passed in this way are referred to as "callbacks". Normal functions can be callbacks as well. The advantage of a lambda function is solely the programmer's convenience. There is no speed or other advantage.

One important use of lambda functions is for providing sort keys; another is to provide event handlers in GUI programming.

The basic syntax for creating a lambda function is

```
lambda parameter-list: expression
```

where parameter-list is a list of function parameters, and expression is an expression involving the parameters. The expression is the return value of the function.

A lambda function could also be defined in the normal manner

```
    def function-name(param-list):
        return expr
```

But it is not possible to use the normal syntax as a function parameter, or as an element in a list.

Example

lambda_examples.py

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

fruits = ['watermelon','Apple','Mango','KIWI','apricot','LEMON','guava']

sfruits = sorted(fruits,key=lambda e: e.lower()) ①

print(" ".join(sfruits))
```

① The lambda function takes one fruit and returns it in lower case

lambda_examples.py

```
Apple apricot guava KIWI LEMON Mango watermelon
```

List comprehensions

- Filters or modifies elements
- Creates new list
- Shortcut for a for loop

A list comprehension is a Python idiom that creates a shortcut for a for loop. It returns a copy of a list with every element transformed via an expression. Functional programmers refer to this as a mapping function.

A loop like this:

```
results = []
for var in sequence:
    results.append(expr)   # where expr involves var
```

can be rewritten as

```
results = [ expr for var in sequence ]
```

A conditional if may be added to filter values:

```
results = [ expr for var in sequence if expr ]
```

Example

listcomp.py

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

fruits = ['watermelon','apple','mango','kiwi','apricot','lemon','guava']

values = [ 2, 42, 18, 92, "boom", ['a','b','c'] ]

ufruits = [ fruit.upper() for fruit in fruits ]  ①

afruits = [ fruit for fruit in fruits if fruit.startswith('a') ] ②

doubles = [ v * 2 for v in values ]  ③


print("ufruits:"," ".join(ufruits))
print("afruits:"," ".join(afruits))
print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print()
```

① Copy each fruit to upper case

② Select each fruit if it starts with 'a'

③ Copy each number, doubling it

listcomp.py

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: apple apricot
doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']
```

Dictionary comprehensions

- Expression is key/value pair
- Transform iterable to dictionary

A dictionary comprehension has syntax similar to a list comprehension. The expression is a key:value pair, and is added to the resulting dictionary. If a key is used more than once, it overrides any previous keys. This can be handy for building a dictionary from a sequence of values.

Example

dict_comprehension.py

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


animals = ['OWL', 'Badger', 'bushbaby', 'Tiger', 'Wombat', 'GORILLA', 'AARDVARK']
# dictionary comprehension
d = {a.lower(): len(a) for a in animals} ①
print(d)
```

① Create a dictionary with key/value pairs derived from an iterable

dict_comprehension.py

```
{'owl': 3, 'badger': 6, 'bushbaby': 8, 'tiger': 5, 'wombat': 6, 'gorilla': 7,
'aardvark': 8}
```

Set comprehensions

- Expression is added to set
- Transform iterable to set — with modifications

A set comprehension is useful for turning any sequence into a set. Items can be modified or skipped as the set is built.

If you don't need to modify the items, it's probably easier to just past the sequence to the **set()** constructor.

Example

set_comprehension.py

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

import re

with open("../DATA/mary.txt") as mary_in:
    s = {w.lower()  for ln in mary_in  for w in re.split(r'\W+', ln) if w} ①
print(s)
```

① Get unique words from file. Only one line is in memory at a time. Skip "empty" words.

set_comprehension.py

```
{'go', 'a', 'lamb', 'snow', 'sure', 'and', 'had', 'as', 'was', 'to', 'everywhere',
'white', 'its', 'little', 'the', 'went', 'that', 'mary', 'fleece'}
```

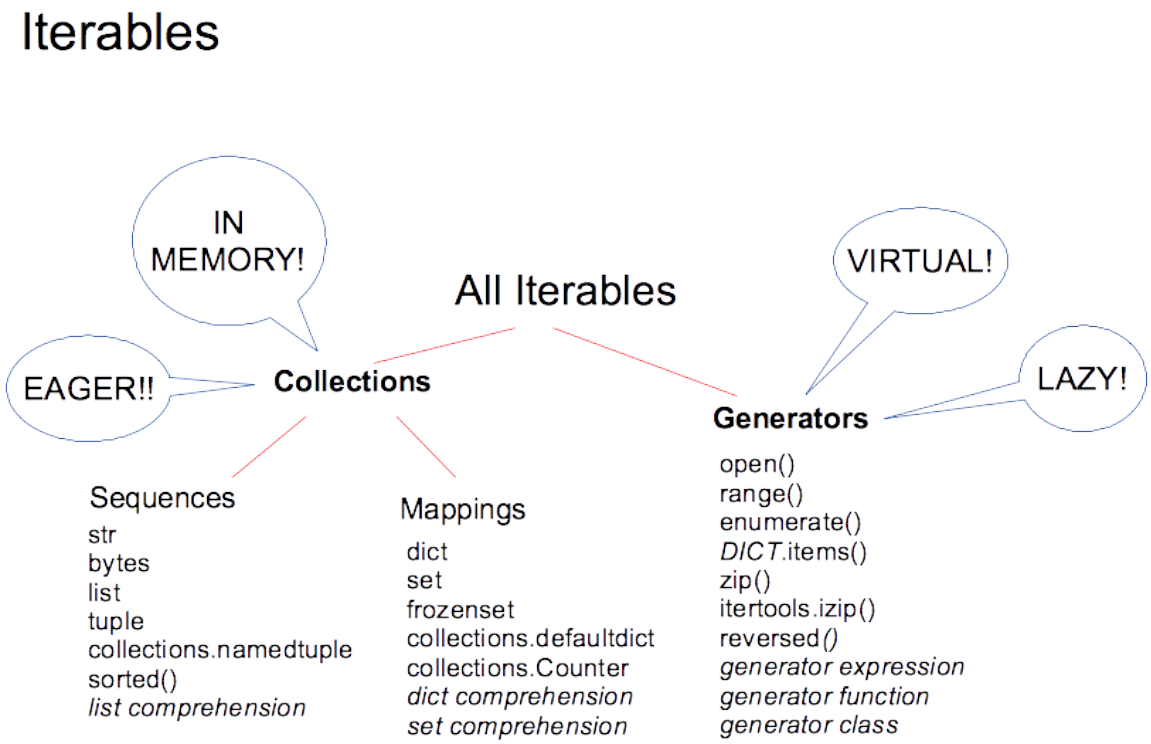
Iterables

- Expression that can be looped over
- Can be collections *e.g.* list, tuple, str, bytes
- Can be generators *e.g.* range(), file objects, enumerate(), zip(), reversed()

Python has many builtin iterables – a file object, for instance, which allows iterating through the lines in a file.

All builtin collections (list, tuple, str, bytes) are iterables. They keep all their values in memory. Many other builtin iterables are *generators*.

A generator does not keep all its values in memory – it creates them one at a time as needed, and feeds them to the for-in loop. This is a Good Thing, because it saves memory.


Iterables
IN MEMORY!
All Iterables
VIRTUAL!
EAGER!!
Collections
LAZY!
Generators
Sequences
str
bytes
list
tuple
collections.namedtuple
sorted()
list comprehension
Mappings
dict
set
frozenset
collections.defaultdict
collections.Counter
dict comprehension
set comprehension
open()
range()
enumerate()
DICT.items()
zip()
itertools.izip()
reversed()
generator expression
generator function
generator class

Generator Expressions

- Like list comprehensions, but create a generator object
- More efficient
- Use parentheses rather than brackets

A generator expression is similar to a list comprehension, but it provides a generator instead of a list. That is, while a list comprehension returns a complete list, the generator expression returns one item at a time.

The main difference in syntax is that the generator expression uses parentheses rather than brackets.

Generator expressions are especially useful with functions like sum(), min(), and max() that reduce an iterable input to a single value:

NOTE There is an implied **yield** statement at the beginning of the expression.

Example

gen_ex.py

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

# sum the squares of a list of numbers
# using list comprehension, entire list is stored in memory
s1 = sum([x*x for x in range(10)])  ①

# only one square is in memory at a time with generator expression
s2 = sum(x*x for x in range(10))  ②
print(s1,s2)
print()

page = open("../DATA/mary.txt")
m = max(len(line) for line in page) ③
page.close()
print(m)
```

① using list comprehension, entire list is stored in memory

② with generator expression, only one square is in memory at a time

③ only one line in memory at a time. max() iterates over generated values

gen_ex.py

```
285 285

30
```

Generator functions

- Mostly like a normal function
- Use yield rather than return
- Maintains state

A generator is like a normal function, but instead of a return statement, it has a yield statement. Each time the yield statement is reached, it provides the next value in the sequence. When there are no more values, the function calls return, and the loop stops. A generator function maintains state between calls, unlike a normal function.

Example

sieve_gen.py

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

def next_prime(limit):
    flags = [False] * (limit+1)  #  ①

    for i in range(2,limit):
        if flags[i]:
            continue
        for j in range(2*i,limit+1,i):
            flags[j] = True
        yield i  ②

for p in next_prime(200):  ③
    print(p, end=' ')
```

① initialize flags

② execution stops here until next value is requested by for-in loop

③ next_prime() returns a generator object

sieve_gen.py

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107
109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

Example

trimmed.py

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

def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            if line.endswith('\n'):
                line = line.rstrip('\n\r')
            yield line  ①

for line in trimmed('../DATA/mary.txt'):  ②
    print(line)
```

① 'yield' causes this function to return a generator object

② looping over the a generator object returned by trimmed()

trimmed.py

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that mary went
The lamb was sure to go
```

String formatting

- Numbered placeholders
- Add width, padding
- Access elements of sequences and dictionaries
- Access object attributes

The traditional (i.e., old) way to format strings in Python was with the % operator and a format string containing fields designated with percent signs. The new, improved method of string formatting uses the format() method. It takes a format string and one or more arguments. The format strings contains placeholders which consist of curly braces, which may contain formatting details. This new method has much more flexibility.

By default, the placeholders are numbered from left to right, starting at 0. This corresponds to the order of arguments to format().

Formatting information can be added, preceded by a colon.

```
{:d}        format the argument as an integer +
{:03d}      format as an integer, 3 columns wide, zero padded +
{:>25s}  same, but right-justified +
{:.3f}     format as a float, with 3 decimal places
```

Placeholders can be manually numbered. This is handy when you want to use a format() parameter more than once.

```
"Try one of these: {0}.jpg {0}.png {0}.bmp {0}.pdf".format('penguin')
```

See appendix String Formatting for more details on string formatting.

Example

stringformat_ex.py

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

from datetime import date

color = 'blue'
animal = 'iguana'

print('{} {}'.format(color,animal)) ①

fahr = 98.6839832
print('{:.1f}'.format(fahr)) ②

value = 12345
print('{0:d} {0:04x} {0:08o} {0:016b}'.format(value)) ③

data = {'A': 38, 'B': 127, 'C': 9}

for letter, number in sorted(data.items()):
    print("{} {:4d}".format(letter, number)) ④
```

① {} placeholders are autonumbered, starting at 0; this corresponds to the parameters to format()

② Formatting directives start with ':'; .1f means format floating point with one decimal place

③ {} placeholders can be manually numbered to reuse parameters

④ :4d means format decimal integer in a field 4 characters wide

stringformat_ex.py

```
blue iguana
98.7
12345 3039 00030071 0011000000111001
A   38
B  127
C    9
```

f-strings

- Shorter syntax for string formatting
- Only available Python 3.6 and later
- Put **f** in front of string

A great new feature, f-strings, was added to Python 3.6. These are strings that contain placeholders, as used with normal string formatting, but the expression to be formatted is also placed in the placeholder. This makes formatting strings more readable, with less typing. As with formatted strings, any expression can be formatted.

Other than putting the value to be formatted directly in the placeholder, the formatting directives are the same as normal Python 3 string formatting.

In normal 3.x formatting:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Bill Gates'
print("{} founded {}.format(name, company)"
print("{:10s} {:.2f}".format(x, y)
```

f-strings let you do this:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Bill Gates'
print(f"{name} founded {company}"
print(f"{x:10s} {y:.2f}"
```

Example

f_strings.py

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

import sys

if sys.version_info.major == 3 and sys.version_info.minor >= 6:

    name = "Tim"
    count = 5
    avg = 3.456
    info = 2093
    result = 38293892

    print(f"Name is [{name:<10s}]")   ①
    print(f"Name is [{name:>10s}]")   ②
    print(f"count is {count:03d} avg is {avg:.2f}") ③

    print(f"info is {info} {info:d} {info:o} {info:x}".format(info))      ④


    print(f"${result:,d}".format(result))     ⑤


    city='Orlando'
    temp=85

    print(f"It is {temp} in {city}")  ⑥

else:
    print("Sorry -- f-strings are only supported by Python 3.6+")
```

① < means left justify (default for non-numbers), 10 is field width, s formats a string

② > means right justify

③ .2f means round a float to 2 decimal points

④ d is decimal, o is octal, x is hex

⑤ , means add commas to numeric value

⑥ parameters can be selected by name instead of position

f_strings.py

```
Name is [Tim       ]
Name is [       Tim]
count is 005 avg is 3.46
info is 2093 2093 4055 82d
$38,293,892
It is 85 in Orlando
```

Chapter 5 Exercises

Exercise 5-1 (pres_upper.py)

Read the file **presidents.txt**, creating a list of of the presidents' last names. Then, use a list comprehension to make a copy of the list of names in upper case. Finally, loop through the list returned by the list comprehension and print out the names one per line.

Exercise 5-2 (pres_by_death.py)

Print out all the presidents first and last names, date of birth, and their political affiliations, sorted by date of birth.

Read the **presidents.txt** file, putting the four fields into a list of tuples.

Loop through the list, sorting by date of birth, and printing the information for each president. Use **sorted()** and a lambda function.

Exercise 5-3 (pres_gen.py)

Write a generator function to provide a sequence of the names of presidents (in "FIRSTNAME MIDDLENAME LASTNAME" format) from the presidents.txt file. They should be provided in the same order they are in the file. You should not read the entire file into memory, but one-at-a-time from the file.

Then iterate over the the generator returned by your function and print the names.

Chapter 6: Functions, Modules and Packages

Objectives

- Define functions
- Learn the four kinds of function parameters
- Create new modules
- Load modules with **import**
- Set module search locations
- Organize modules into packages
- Alias module and package names

Functions

- Defined with **def**
- Accept parameters
- Return a value

Functions are a way of isolating code that is needed in more than one place, refactoring code to make it more modular. They are defined with the **def** statement.

Functions can take various types of parameters, as described on the following page. Parameter types are dynamic.

Functions can return one object of any type, using the **return** statement. If there is no return statement, the function returns **None**.

TIP Be sure to separate your business logic (data and calculations) from your presentation logic (the user interface).

Example

function_basics.py

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

def say_hello():   ①
    print("Hello, world")
    print()
    ②

say_hello()  ③

def get_hello():
    return "Hello, world" ④

h = get_hello()   ⑤
print(h)
print()

def sqrt(n):   ⑥
    return n ** .5


m = sqrt(1234) ⑦
n = sqrt(2)

print("m is {:.3f} n is {:.3f}".format(m, n))
```

① Function takes no parameters

② If no **return** statement, return None

③ Call function (arguments, if any, in ())

④ Function returns value

⑤ Store return value in h

⑥ Function takes exactly one argument

⑦ Call function with one argument

function_basics.py

```
Hello, world

Hello, world

m is 35.128 n is 1.414
```

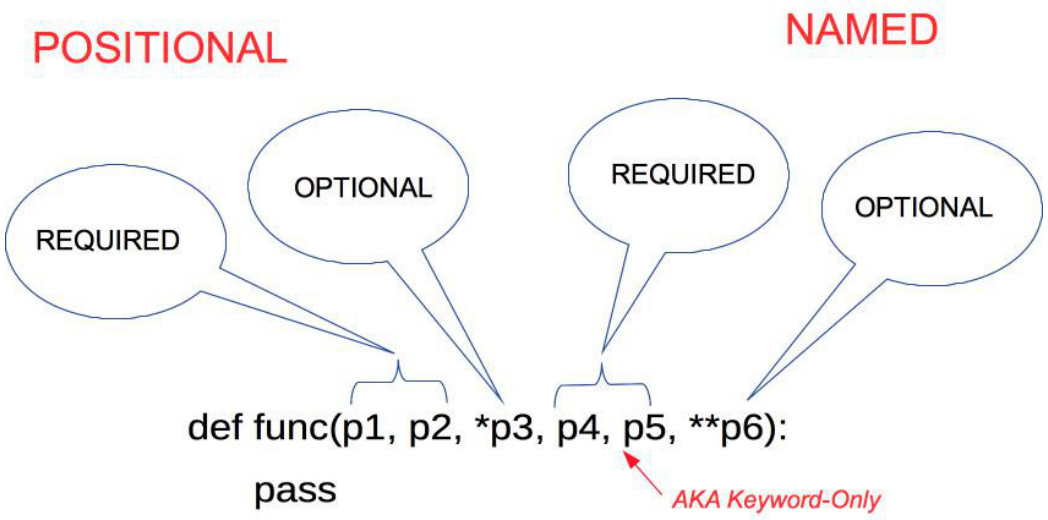
Function parameters

- Positional or named
- Required or optional
- Can have default values

Functions can accept both positional and named parameters. Furthermore, parameters can be required or optional. They must be specified in the order presented here.

The first set of parameters, if any, is a set of comma-separated names. These are all required. Next you can specify a variable preceded by an asterisk — this will accept any optional parameters.

After the optional positional parameters you can specify required named parameters. These must come after the optional parameters. If there are no optional parameters, you can use a plain asterisk as a placeholder. Finally, you can specify a variable preceded by two asterisks to accept optional named parameters.



Example

function_parameters.py

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

def fun_one():              ①
    print("Hello, world")

print("fun_one():", end=' ')
fun_one()
print()

def fun_two(n):             ②
    return n ** 2

x = fun_two(5)
print("fun_two(5) is {}\n".format(x))

def fun_three(count=3):  ③
    for i in range(count):
        print("spam", end=' ')
    print()

fun_three()
fun_three(10)
print()

def fun_four(n, *opt):   ④
    print("fun_four():")
    print("n is ", n)
    print("opt is", opt)
    print('-' * 20)

fun_four('apple')
fun_four('apple',"blueberry","peach","cherry")

def fun_five(*, spam=0, eggs=0): ⑤
    print("fun_five():")
    print("spam is:", spam)
    print("eggs is:", eggs)
    print()

fun_five(spam=1, eggs=2)
```

```
fun_five(eggs=2, spam=2)
fun_five(spam=1)
fun_five(eggs=2)
fun_five()

def fun_six(**named_args): ⑥
    print("fun_six():")
    for name in named_args:
        print(name,"==> ",named_args[name])

fun_six(name="Lancelot",quest="Grail",color="red")
```

① no parameters

② one required parameter

③ one required parameter with default value

④ one fixed, plus optional parameters

⑤ keyword-only parameters

⑥ keyword (named) parameters

function_parameters.py

```
fun_one(): Hello, world

fun_two(5) is 25

spam spam spam
spam spam spam spam spam spam spam spam spam spam

fun_four():
n is  apple
opt is ()
--------------------
fun_four():
n is  apple
opt is ('blueberry', 'peach', 'cherry')
--------------------
fun_five():
spam is: 1
eggs is: 2

fun_five():
spam is: 2
eggs is: 2

fun_five():
spam is: 1
eggs is: 0

fun_five():
spam is: 0
eggs is: 2

fun_five():
spam is: 0
eggs is: 0

fun_six():
name ==>  Lancelot
quest ==>  Grail
color ==>  red
```

Default parameters

- Assigned with equals sign
- Used if no values passed to function

Required parameters can have default values. They are assigned to parameters with the equals sign. Parameters without defaults cannot be specified after parameters with defaults.

Example

default_parameters.py

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

def spam(greeting, whom='world'):  ①
    print("{}, {}".format(greeting, whom))


spam("Hello")  ②
spam("Hello", "Mom")  ③
print()

def ham(*, file_name, format='txt'):  ④
    print("Processing {} as {}".format(file_name, format))

ham(file_name='eggs')  ⑤
ham(file_name='toast', format='csv')
```

① 'world' is default value for positional parameter **whom**

② parameter supplied; default not used

③ parameter not supplied; default is used

④ 'world' is default value for named parameter **format**

⑤ parameter **format** not supplied; default is used

default_parameters.py

```
Hello, world
Hello, Mom

Processing eggs as txt
Processing toast as csv
```

Name resolution (AKA Scope)

- What is "scope"
- Scopes used dynamically
- Four levels of scope
- Assignments always go into the innermost scope (starting with local)

A scope is the area of a Python program where an unqualified (not preceded by a module name) name can be looked up.

Scopes are used dynamically. There are four nested scopes that are searched for names in the following order:

local	local names bound within a function
nonlocal	local names plus local names of outer function(s)
global	the current module's global names
builtin	built-in functions (contents of ***_builtins_*** module)

Within a function, all assignments and declarations create local names. All variables found outside of local scope (that is, outside of the function) are read-only.

Inside functions, local scope references the local names of the current function. Outside functions, local scope is the same as the global scope – the module's namespace. Class definitions also create a local scope.

Nested functions provide another scope. Code in function B which is defined inside function A has read-only access to all of A's variables. This is called **nonlocal** scope.

Example

scope_examples.py

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

x = 42 ①

def function_a():
    y = 5  ②

    def function_b():
        z = 32  ③
        print("function_b(): z is", z)      ④
        print("function_b(): y is", y)      ⑤
        print("function_b(): x is", x)      ⑥
        print("function_b(): type(x) is", type(x)) ⑦

    return function_b;

f = function_a()   ⑧
f() ⑨
```

① global variable

② local variable to function_a(), or nonlocal to function_b()

③ local variable

④ local scope

⑤ nested (nonlocal) scope

⑥ global scope

⑦ builtin scope

⑧ calling function_a, which returns function_b

⑨ calling function_b

scope_examples.py

```
function_b(): z is 32
function_b(): y is 5
function_b(): x is 42
function_b(): type(x) is <class 'int'>
```

The global statement

- global statement allows function to change globals
- nonlocal statement allows function to change nonlocals

The **global** keyword allows a function to modify a global variable. This is universally acknowledged as a BAD IDEA. Mutating global data can lead to all sorts of hard-to-diagnose bugs, because a function might change a global that affects some other part of the program. It's better to pass data into functions as parameters and return data as needed. Mutable objects, such as lists, sets, and dictionaries can be modified in-place.

The **nonlocal** keyword can be used like **global** to make nonlocal variables in an outer function writable.

Modules

- Files containing python code
- End with .py
- No real difference from scripts

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable *name*.

To use a module named spam.py, say `import spam`

This does not enter the names of the functions defined in spam directly into the symbol table; it only adds the module name **spam**. Use the module name to access the functions or other attributes.

Python uses modules to contain functions that can be loaded as needed by scripts. A simple module contains one or more functions; more complex modules can contain initialization code as well. Python classes are also implemented as modules.

A module is only loaded once, even there are multiple places in an application that import it.

Modules and packages should be documented with *docstrings*.

Using import

- import statement loads modules
- Three variations
  - import module
  - from module import function-list
  - from module import * use with caution!

There are three variations on the **import** statement:

Variation 1

import module
loads the module so its data and functions can be used, but does not put its attributes (names of classes, functions, and variables) into the current namespace.

Variation 2

from module import function, ...
imports only the function(s) specified into the current namespace. Other functions are not available (even though they are loaded into memory).

Variation 3

from module import *
loads the module, and imports all functions that do not start with an underscore into the current namespace. This should be used with caution, as it can pollute the current namespace and possibly overwrite builtin attributes or attributes from a different module.

NOTE

The first time a module is loaded, the interpreter creates a version compiled for faster loading. This version has platform information embedded in the name, and has the extension .pyc. These .pyc files are put in a folder named **__pycache__**.

Example

samplelib.py

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

"""
Example module.
"""

def spam():
    print("Hello from spam()")

def ham():
    print("Hello from ham()")

def _eggs():
    print("Hello from _eggs()")
```

use_samplelib1.py

```
#!/usr/bin/env python
import samplelib  ①

samplelib.spam()  ②
samplelib.ham()
```

① import samplelib module (samplelib.py) — creates object named **samplelib** of type "Module"

② call function spam() in module samplelib

use_samplelib1.py

```
Hello from spam()
Hello from ham()
```

use_samplelib2.py

```
#!/usr/bin/env python
from samplelib import spam, ham ①

spam()  ②
ham()
```

① import functions spam and ham from samplelib module into current namespace — does not create the module object

② module name not needed to call function spam()

use_samplelib2.py

```
Hello from spam()
Hello from ham()
```

use_samplelib3.py

```
#!/usr/bin/env python
from samplelib import * ①

spam()  ②
ham()
```

① import all functions (that do not start with _) from samplelib module into current namespace

② module name not needed to call function spam()

use_samplelib3.py

```
Hello from spam()
Hello from ham()
```

use_samplelib4.py

```
#!/usr/bin/env python
from samplelib import spam as pig, ham as hog  ①

pig()
hog()
```

① import functions spam and ham, aliased to pig and hog

use_samplelib4.py

```
Hello from spam()
Hello from ham()
```

How *import* * can be dangerous

- Imported names may overwrite existing names
- Be careful to read the documentation

Using `import *` to import all public names from a module has a bit of a risk. While generally harmless, there is the chance that you will unknowingly import a module that overwrites some previously-imported module.

To be 100% certain, always import the entire module, or else import names explicitly.

Example

electrical.py

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

default_amps = 10
default_voltage = 110
default_current = 'AC'

def amps():
    return default_amps

def voltage():
    return default_voltage

def current():
    return default_current
```

navigation.py

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

current_types = 'slow medium fast'.split()

def current():
    return current_types[0]
```

why_import_star_is_bad.py

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

from electrical import *  ①
from navigation import *  ②

print(current())  ③
print(voltage())
print(amps())
```

① import current *explicitly* from electrical

② import current *implicitly* from navigation

③ calls navigation.current(), not electrical.current()

why_import_star_is_bad.py

```
slow
110
10
```

how_to_avoid_import_star.py

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

import electrical as elec ①
import navigation as nav  ②

print(elec.current()) ③
print(nav.current())  ④
```

how_to_avoid_import_star.py

```
AC
slow
```

Module search path

- Searches current folder first, then predefined locations
- Add custom locations to PYTHONPATH
- Paths stored in sys.path

When you specify a module to load with the import statement, it first looks in the current directory, and then searches the directories listed in sys.path.

```
>>> import sys
>>> sys.path
```

To add locations, put one or more directories to search in the PYTHONPATH environment variable. Separate multiple paths by semicolons for Windows, or colons for Unix/Linux. This will add them to **sys.path**, after the current folder, but before the predefined locations.

Windows

set PYTHONPATH=C:"\Documents and settings\Bob\Python"

Linux/OS X

export PYTHONPATH="/home/bob/python"

You can also append to sys.path in your scripts, but this can result in non-portable scripts, and scripts that will fail if the location of the imported modules changes.

```
import sys
sys.path.extend("/usr/dev/python/libs","/home/bob/pylib")
import module1
import module2
```

Executing modules as scripts

- *_name_* is current module.
  - set to **__main__** if run as script
  - set to module_name if imported
- test with `if name == "__main__"`
- Module can be run directly or imported

It is sometimes convenient to have a module also be a runnable script. This is handy for testing and debugging, and for providing modules that also can be used as standalone utilities.

Since the interpreter defines its own name as '__main__', you can test the current namespace's *name* attribute. If it is '__main__', then you are at the main (top) level of the interpreter, and your file is being run as a script; it was not loaded as a module.

Any code in a module that is not contained in function or method is executed when the module is imported.

This can include data assignments and other startup tasks, for example connecting to a database or opening a file.

Many modules do not need any initialization code.

Example

using_main.py

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

# other imports  (standard library, standard non-library, local)

# constants (AKA global variables)

# main function
def main(args):  ①
    function1()
    function2()

# other functions
def function1():
    print("hello from function1()")

def function2():
    print("hello from function2()")

if __name__ == '__main__':
    main(sys.argv[1:])  ②
```

① Program entry point. While **main** is not a reserved word, it is a strong convention

② Call main() with the command line parameters (omitting the script itself)

Packages

- Package is folder containing modules or packages
- Startup code goes in __init__.py (optional)

A package is a group of related modules or subpackages. The grouping is physical – a package is a folder that contains one or more modules. It is a way of giving a hierarchical structure to the module namespace so that all modules do not live in the same folder.

A package may have an initialization script named **__init__.py**. If present, this script is executed when the package or any of its contents are loaded. (In Python 2, **__init__.py** was required).

Modules in packages are accessed by prefixing the module with the package name, using the dot notation used to access module attributes.

Thus, if Module **eggs** is in package **spam**, to call the **scramble()** function in **eggs**, you would say `spam.eggs.scramble()`.

By default, importing a package name by itself has no effect; you must explicitly load the modules in the packages. You should usually import the module using its package name, like `from spam import eggs`, to import the eggs module from the spam package.

Packages can be nested.

Example

```
sound/                  Top-level package
    __init__.py         Initialize the sound package (optional)
    formats/            Subpackage for file formats
        __init__.py         Initialize the formats package (optional)
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    effects/            Subpackage for sound effects
        __init__.py         Initialize the formats package (optional)
        echo.py
        surround.py
        reverse.py
        ...
    filters/            Subpackage for filters
        __init__.py         Initialize the formats package (optional)
        equalizer.py
```

```
from sound.formats import aiffread
import sound.effects
import sound.filters.equalizer
```

Configuring import with __init__.py

- load modules into package's namespace
- specify modules to load when * is used

For convenience, you can put import statements in a package's **__init__.py** to autoload the modules into the package namespace, so that import PKG imports all the (or just selected) modules in the package.

__init__.py can also be used to setup data or other resources that will be used by multiple modules within a package.

If the variable ***_all_*** in **__init__.py** is set to a list of module names, then only these modules will be loaded when the import is

```
from PKG import *
```

Given the following package and module layout, the table on the next page describes how **__init__.py** affects imports.

```
my_package
   |------__init__.py
   |------module_a.py
   |         function_a()
   |------module_b.py
   |         function_b()
   |------module_c.py
             function_c()
```

Import statement	What it does
If **__init__.py** is empty	
`import my_package`	Imports **my_package** only, but not contents. No modules are imported. This is not useful.
`import my_package.module_a`	Imports **module_a** into **my_package** namespace. Objects in **module_a** must be prefixed with **my_package.module_a**
`from my_package import module_a`	Imports **module_a** into main namespace. Objects in **module_a** must be prefixed with **module_a**
`from my_package import module_a, module_b`	Imports **module_a** and **module_b** into main namespace.
`from my_package import *`	Does not import anything!
`from my_package.module_a import *`	Imports all contents of **module_a** (that do not start with an underscore) into main namespace. Not generally recommended.
If **__init__.py** contains: `all = ['module_a', 'module_b']`	
`import my_package`	Imports my_package only, but not contents. No modules are imported. This is still not useful.
`from my_package import module_a`	As before, imports **module_a** into main namespace. Objects in **module_a** must be prefixed with **module_a**
`from my_package import *`	Imports **module_a** and **module_b**, but not **module_c** into main namespace.

If **__init__.py** contains:
`all = ['module_a', 'module_b']`
`import module_a`
`import module_b`	

`import my_package`	Imports **module_a** and **module_b** into the my_package namespace. Objects in **module_a** must be prefixed with **my_package.module_a**. *Now this is useful.*
`from my_package import module_a`	Imports **module_a** into main namespace. Objects in **module_a** must be prefixed with **module_a**
`from my_package import *`	Only imports **module_a** and **module_b** into main namespace.
`from my_package import module_c`	Imports **module_c** into the main namespace.

Documenting modules and packages

- Use docstrings
- Described in PEP 257
- Generate docs with Sphinx (optional)

In addition to comments, which are for the *maintainer* of your code, you should add docstrings, which provide documentation for the *user* of your code.

If the first statement in a module, function, or class is an unassigned string, it is assigned as the *docstring* of that object. It is stored in the special attribute *__doc__*, and so is available to code.

The docstring can use any form of literal string, but triple double quotes are preferred, for consistency.

See **PEP 257** for a detailed guide on docstring conventions.

Tools such as pydoc, and many IDEs will use the information in docstrings. In addition, the Sphinx tool will gather docstrings from an entire project and format them as a single HTML, PDF, or EPUB document.

Example

script_template.py

```
#!/usr/bin/env python
"""
This is the doc string for the module/script.
"""
import sys

# other imports  (standard library, standard non-library, local)

# constants (AKA global variables -- keep these to a minimum)

# main function
def main(args):
    """
    This is the docstring for the main() function

    :param args: Command line arguments.
    :return: None
    """
    function1()

# other functions
def function1():
    """
    This is the docstring for function1().

    :return: None
    """
    pass

if __name__ == '__main__':
    main(sys.argv[1:])  # Pass command line args (minus script name) to main()
```

Python style

- Code is read more often than it is written!
- Style guides enforce consistency and readability

- Indent 4 spaces (do not use tabs)
- Keep lines ⇐ 79 characters
- Imports at top of script, and on separate lines
- Surround operators with space
- Comment thoroughly to explain why and how code works when not obvious
- Use docstrings to explain how to use modules, classes, methods, and functions
- Use lower_case_with_underscores for functions, methods, and attributes
- Use UPPER_CASE_WITH_UNDERSCORES for globals
- Use StudlyCaps (mixed-case) for class names
- Use _leading_underscore for internal (non-API) attributes

Guido van Rossum, Python's BDFL (Benevolent Dictator For Life), once said that code is read much more often than it is written. This means that once code is written, it may be read by the original developer, users, subsequent developers who inherit your code. Do them a favor and make your code readable. This in turn makes your code more maintainable.

To make your code readable, it is import to write your code in a consistent manner. There are several Python style guides available, including PEP (Python Enhancement Proposal) 8, Style Guide for Python Code, and PEP 257, Docstring Conventions.

If you are part of a development team, it is a good practice to put together a style guide for the team. The team will save time not having to figure out each other's style.

Chapter 6 Exercises

Exercise 6-1 (potus.py, potus_main.py)

Create a module named to provide information from the presidents.txt file. It should provide the following function:

```
get_info(term#) -> dict  provide dictionary of info for a specified president
```

Write a script to use the module.

For the ambitious (potus_amb.py, potus_amb_main.py)

Add the following functions to the module

```
get_oldest() -> string   return the name of oldest president
get_youngest()-> string  return the name of youngest president
```

'youngest' and 'oldest' refer to age at beginning of first term and age at end of last term.

Chapter 7: Intermediate Classes

Objectives

- Defining a class and its constructor
- Creating object methods
- Adding properties to a class
- Working with class data and methods
- Leveraging inheritance for code reuse
- Implementing special methods
- Knowing when NOT to use classes

What is a class?

- Represents a *thing*
- Encapsulates functions and variables
- Creator of object *instances*
- Basic unit of object-oriented programming

A class is definition that represents a *thing*. The thing could be a file, a process, a database record, a strategy, a string, a person, or a truck.

The class describes both data, which represents one instance of the thing, and methods, which are functions that act upon the data. There can be both class data, which is shared by all instances, and instance data, which is only accessible from the instance.

TIP Classes are a very powerful tool to organize code. However, there are some circumstances in Python where classes are not needed. If you just need some functions, and they don't need to share or remember data, just put the functions in a module. If you just need some data, but you don't need functions to process it, just used a nested data structure built out of dictionaries, lists, and tuples, as needed.

Defining Classes

- Syntax

```
class ClassName(base_class,...):
    # class body - methods and data
```

- Specify base classes
- Use StudlyCaps for name

The **class** statement defines a class and assigns it to a name.

The simplest form of class definition looks like this:

```
class ClassName():
    pass
```

Normally, the contents of a class definition will be method definitions and shared data.

A class definition creates a new local namespace. All variable assignments go into this new namespace. All methods are called via the instance or the class name.

A list of base classes may be specified in parentheses after the class name.

Object Instances

- Call class name as a function
- ***self*** contains attributes
- Syntax

```
obj = ClassName(args...)
```

An object instance is an object created from a class. Each object instance has its own private attributes, which are usually created in the __init__ method.

Instance attributes

- Methods and data
- Accessed using dot notation
- Privacy by convention (_name)

An instance of a class (AKA object) normally contains methods and data. To access these attributes, use "dot notation": object.attribute.

Instance attributes are dynamic; they can be accessed directly from the object. You can create, update, and delete attributes in this way.

Attributes cannot be made private, but names that begin with an underscore are understood by convention to be for internal use only. Users of your class will not consider methods that begin with an underscore to be part of your class's API.

Example

```
class Spam():
    def eggs(self):
        pass

    def _beverage(self):   # private!
        pass

s = Spam()
s.eggs()
s.toast = 'buttered'
print(s.toast)

s._beverage()   # legal, but wrong!
```

In most cases, it is better to use properties (described later) to access data attributes.

Instance Methods

- Called from objects
- Object is implicit parameter

An instance method is a function defined in a class. When a method is called from an object, the object is passed in as the implicit first parameter, named **self** by strong convention.

Example

rabbit.py

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

class Rabbit:

    def __init__(self,size,danger): ①
        self._size = size
        self._danger = danger
        self._victims = []

    def threaten(self): ②
        print("I am a {} bunny with {}!".format(self._size, self._danger))

r1 = Rabbit('large',"sharp, pointy teeth")  ③
r1.threaten() ④

r2 = Rabbit('small','fluffy fur')
r2.threaten()
```

① constructor, passed **self**

② instance method, passed **self**

③ pass parameters to constructor

④ instance method has access to variables via **self**

rabbit.py

```
I am a large bunny with sharp, pointy teeth!
I am a small bunny with fluffy fur!
```

Constructors

- Named __init__.py
- Implicitly called when object is created
- **self** is object it*self*

If a class defines a method named **__init__.py**, it will be automatically called when an object instance is created. This is the *constructor*.

The object being created is implicitly passed as the first parameter to **__init__.py** . This parameter is named **self** by very strong convention. Data attributes can be assigned to **self**. These attributes can then be accessed by other methods.

Example

```
class Rabbit:

    def __init__(self, size, danger):
        self._size = size
        self._danger = danger
        self._victims = []
```

TIP In C++, Java, and C#, **self** might be called **this**.

Getters and setters

- Used to access data
- AKA *accessors* and *mutators*
- Most people prefer **properties** (see next topic)

Getter and setter methods can be used to access an object's data. These are traditional in object-oriented programming.

A *getter* retrieves a data (private variable) from self. A *setter* assigns a value to a variable.

NOTE Most Python developers use *properties*, described next, instead of getters and setters.

Example

```
class Knight(object):
    def __init__(self,name):
        self._name = name

    def set_name(self,name):
        self._name = name

    def get_name(self):
        return self._name

k = Knight("Lancelot")
print( k.get_name() )
```

Properties

- Accessed like variables
- Invoke implicit getters and setters
- Can be read-only

While object attributes can be accessed directly, in many cases the class needs to exercise some control over the attributes.

A more elegant approach is to use properties. A property is a kind of managed attribute. Properties are accessed directly, like normal attributes (variables), but getter, setter, and deleter functions are implicitly called, so that the class can control what values are stored or retrieved from the attributes.

You can create getter, setter, and deleter properties.

To create the getter property (which must be created first), apply the **@property** decorator to a method with the name you want. It receives no parameters other than **self**.

To create the setter property, create another function with the property name (yes, there will be two function definitions with the same name). Decorate this with the propety name plus ".setter". In other words, if the property is named "spam", the decorator will be "@spam.setter". The setter method will take one parameter (other than self), which is the value assigned to the property.

It is common for a setter property to raise an error if the value being assigned is invalid.

While you seldom need a deleter property, creating it is the same as for a setter property, but use "@*propertyname*.deleter".

Example

knight.py

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

class Knight(object):
    def __init__(self, name, title, color):
        self._name = name
        self._title = title
        self._color = color

    @property  ①
    def name(self):  ②
        return self._name

    @property
    def color(self):
        return self._color

    @color.setter  ③
    def color(self, color):
        self._color = color

    @property
    def title(self):
        return self._title


if __name__ == '__main__':
    k = Knight("Lancelot", "Sir", 'blue')

    # Bridgekeeper's question
    print('Sir {}, what is your...favorite color?'.format( k.name )) ④

    # Knight's answer
    print("red, no -- {}!".format( k.color ))

    k.color = 'red' ⑤

    print("color is now:", k.color)
```

① getter property decorator

② property implemented by name() method

③ setter property decorator

④ use property

⑤ set property

knight.py

```
Sir Lancelot, what is your...favorite color?
red, no -- blue!
color is now: red
```

Class Data

- Attached to class, not instance
- Shared by all instances

Data can be attached to the class itself, and shared among all instances. Class data can be accessed via the class name from inside or outside of the class.

Any class attribute not overwritten by an instance attribute is also available through the instance.

Example

class_data.py

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

class Rabbit:
    LOCATION = "the Cave of Caerbannog" ①

    def __init__(self, weapon):
        self.weapon = weapon

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon))  ②

r1 = Rabbit("a nice cup of tea")
r1.display() ③

r1 = Rabbit("big pointy teeth")
r1.display() ③
```

① class data

② look up class data via instance

③ instance method uses class data

class_data.py

```
This rabbit guarding the Cave of Caerbannog uses a nice cup of tea as a weapon
This rabbit guarding the Cave of Caerbannog uses big pointy teeth as a weapon
```

Class Methods

- Called from class or instance
- Use @classmethod to define
- First (implicit) parameter named "cls" by convention

If a method only needs class attributes, it can be made a class method via the @classmethod decorator. This alters the method so that it gets a copy of the class object rather than the instance object. This is true whether the method is called from the class or from an instance.

The parameter to a class method is named **cls** by strong convention.

Example

class_methods_and_data.py

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

class Rabbit:
    LOCATION = "the Cave of Caerbannog" ①

    def __init__(self, weapon):
        self.weapon = weapon

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon))  ②

    @classmethod ③
    def get_location(cls):  ④
        return cls.LOCATION   ⑤

r = Rabbit("a nice cup of tea")
print(Rabbit.get_location()) ⑥
print(r.get_location())  ⑦
```

① constructor

② increment class data for each instance

③ cls is class object ("Animal")

④ returns Animal.count

⑤ create instance of Animal

class_methods_and_data.py

```
the Cave of Caerbannog
the Cave of Caerbannog
```

Inheritance

- Specify base class in class definition
- Call base class constructor explicitly

Any language that supports classes supports *inheritance*. One or more base classes may be specified as part of the class definition. All of the previous examples in this chapter have used the default base class, object.

The base class must already be imported, if necessary. If a requested attribute is not found in the class, the search looks in the base class. This rule is applied recursively if the base class itself is derived from some other class. For instance, all classes inherit the implementation from **object**, unless a class explicitly implements it.

Classes may override methods of their base classes. (For Java and C++ programmers: all methods in Python are effectively virtual.)

To extend rather than simply replace a base class method, call the base class method directly: BaseClassName.methodname(self, arguments).

You can also use the super() function, which stands in for the base class:

```
class Foo(Bar):
    def __init__(self):
        super().__init__()   # same as Bar._init__(self)
```

The advantage of super() is that you don't have to specify the base class explicitly, so if you change the base class, it automatically does the right thing.

Using Super

- Follows MRO (method resolution order) to find function
- Great for single inheritance tree
- Use explicit base class names for multiple inheritance
- Syntax:

```
super().method()
```

The **super()** function can be used in a class to invoke methods in base classes. It searches the base classes and their bases, recursively, from left to right until the method is found.

For classes that have a single inheritance tree, this works great. For classes that have a diamond-shaped tree, super() may not do what you expect. In this case, using the explicit base class name is best.

Example

animal.py

```
#!/usr/bin/env python
class Animal(object):
    count = 0    ①

    def __init__(self, species, name, sound):
        self._species = species
        self._name = name
        self._sound = sound
        Animal.count += 1

    @property
    def species(self):
        return self._species

    @classmethod
    def kill(cls):
        cls.count -= 1

    @property
    def name(self):
        return self._name

    def make_sound(self):
        print(self._sound)

    @classmethod
    def remove(cls):
        cls.count -= 1  ②

    @classmethod
    def zoo_size(cls):  ③
        return cls.count

if __name__ == "__main__":
    leo = Animal("African lion", "Leo", "Roarrrrrrr")
    garfield = Animal("cat", "Garfield", "Meowwwww")
    felix = Animal("cat", "Felix", "Meowwwww")

    print(leo.name, "is a", leo.species, "--", end=' ')
    leo.make_sound()
```

```
    print(garfield.name, "is a", garfield.species, "--", end=' ')
    garfield.make_sound()

    print(felix.name, "is a", felix.species, "--", end=' ')
    felix.make_sound()
```

① class data

② update class data from instance

③ zoo_size gets class object when called from instance or class

insect.py

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

from  animal import Animal

class Insect(Animal):
    '''
        An animal with 2 sets of wings and 3 pairs of legs
    '''
    def __init__(self, species, name, sound, can_fly=True): ①
        super().__init__(species, name, sound) ②
        self._can_fly = can_fly

    @property
    def can_fly(self): ③
        return self._can_fly

if __name__ == '__main__':
    mon = Insect('monarch butterfly', 'Mary', None) ④
    scar = Insect('scarab beetle', 'Rupert', 'Bzzz', False)

    for insect in mon, scar:
        flying_status = 'can' if insect.can_fly else "can't"
        print("Hi! I am {} the {} and I {} fly!".format( ⑤
                insect.name, insect.species, flying_status
            ),
        )
        insect.make_sound() ⑥
        print()
```

① constructor (AKA initializer)

② call base class constructor

③ "getter" property

④ defaults to can_fly being True

⑤ .name and .species inherited from base class (Animal)

⑥ .make_sound inherited from Animal

insect.py

```
Hi! I am Mary the monarch butterfly and I can fly!
None

Hi! I am Rupert the scarab beetle and I can't fly!
Bzzz
```

Multiple Inheritance

- More than one base class
- All data and methods are inherited
- Methods resolved left-to-right, depth-first

Python classes can inherit from more than one base class. This is called "multiple inheritance".

Classes designed to be added to a base class are sometimes called "mixin classes", or just "mixins".

Methods are searched for in the first base class, then its parents, then the second base class and parents, and so forth.

Put the "extra" classes before the main base class, so any methods in those classes will override methods with the same name in the base class.

TIP To find the exact method resolution order (MRO) for a class, call the class's **mro()** method.

Example

multiple_inheritance.py

```
#!/usr/bin/env python
class AnimalBase():  ①
    def __init__(self, name):
        self._name = name

    def id(self):
        print(self._name)

class CanBark():  ②
    def bark(self):
        print("woof-woof")

class CanFly(): ②
    def fly(self):
        print("I'm flying")


class Dog(CanBark, AnimalBase): ③
    pass

class Sparrow(CanFly, AnimalBase): ③
    pass

d = Dog('Dennis')
d.id()    ④
d.bark()  ⑤
print()

s = Sparrow('Steve')
s.id()
s.fly()  ⑥
print()

print("Sparrow mro:", Sparrow.mro())
```

① create primary base class

② create additional (mixin) base class

③ inherit from primary base class plus mixin

④ all animals have id()

⑤ dogs can bark() (from mixin)

⑥ sparrows can fly() (from mixin)

multiple_inheritance.py

```
Dennis
woof-woof

Steve
I'm flying

Sparrow mro: [<class '__main__.Sparrow'>, <class '__main__.CanFly'>, <class
'__main__.AnimalBase'>, <class 'object'>]
```

Abstract base classes

- Designed for inheritance
- Abstract methods *must* be implemented
- Non-abstract methods *may* be overwritten

The **abc** module provides abstract base classes. When a method in an abstract class is designated **abstract**, it must be implemented in any derived class. If a method is not marked abstract, it may be overwritten or extended.

To create an abstract class, import ABCMeta and abstractmethod. Create the base (abstract) class normally, but assign ABCMeta to the class option **metaclass**. Then decorated any desired abstract methods with *@abstractmethod.

Now, any classes that inherit from the base class must implement any abstract methods. Non-abstract methods do not have to be implemented, but of course will be inherited.

NOTE abc also provides decorators for abstract properties and abstract class methods.

Example

abstract_base_classes.py

```
#!/usr/bin/env python
#
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):  ①

    @abstractmethod   ②
    def speak(self):
        pass

class Dog(Animal):  ③
    def speak(self):   ④
        print("woof! woof!")

class Cat(Animal):  ③
    def speak(self):  ④
        print("Meow meow meow")

class Duck(Animal): ③
    pass  ⑤

d = Dog()
d.speak()

c = Cat()
c.speak()

try:
    d = Duck()  ⑥
    d.speak()
except TypeError as err:
    print(err)
```

① metaclasses control how classes are created; ABCMeta adds restrictions to classes that inherit from Animal

② when decorated with @abstractmethod, speak() becomes an abstract method

③ Inherit from abstract base class Animal

④ speak() **must** be implemented

⑤ Duck does not implement speak()

⑥ Duck throws a TypeError if instantiated

abstract_base_classes.py

```
woof! woof!
Meow meow meow
Can't instantiate abstract class Duck with abstract methods speak
```

Special Methods

- User-defined classes emulate standard types
- Define behavior for builtin functions
- Override operators

Python has a set of special methods that can be used to make user-defined classes emulate the behavior of builtin classes. These methods can be used to define the behavior for builtin functions such as str(), len() and repr(); they can also be used to override many Python operators, such as +, *, and ==.

These methods expect the self parameter, like all instance methods. They frequently take one or more additional methods. self. Is the object being called from the builtin function, or the left operand of a binary operator such as ==.

For instance, if your object represented a database connection, you could have str() return the hostname, port, and maybe the connection string. The default for str() is to call repr(), which returns something like <*main*.DBConn object at 0xb7828c6c>, which is not nearly so user-friendly.

TIP See http://docs.python.org/reference/datamodel.html#special-method-names for detailed documentation on the special methods.

Table 9. Special Methods and Variables

Method or Variables	Description
__new__(cls,...)	Returns new object instance; Called before __init__()
__init__(self,...)	Object initializer (constructor)
__del__(self)	Called when object is about to be destroyed
__repr__(self)	Called by repr() builtin
__str__(self)	Called by str() builtin
__eq__(self, other) __ne__(self, other) __gt__(self, other) __lt__(self, other) __ge__(self, other) __le__(self, other)	Implement comparison operators ==, !=, >, <, >=, and ⇐. self is object on the left.
__cmp__(self, other)	Called by comparison operators if __eq__, etc., are not defined
__hash__(self)	Called by hash() builtin, also used by dict, set, and frozenset operations
__bool__(self)	Called by bool() builtin. Implements truth value (boolean) testing. If not present, bool() uses len()
__unicode__(self)	Called by unicode() builtin
__getattr__(self, name) __setattr__(self, name, value) __delattr__(self, name)	Override normal fetch, store, and deleter
__getattribute__(self, name)	Implement attribute access for new-style classes
__get__(self, instance)	__set__(self, instance, value)
__del__(self, instance)	Implement descriptors
__slots__ = variable-list	Allocate space for a fixed number of attributes.
__metaclass__ = callable	Called instead of type() when class is created.
__instancecheck__(self, instance)	Return true if instance is an instance of class
__subclasscheck__(self, instance)	Return true if instance is a subclass of class

Method or Variables	Description
__call__(self, ...)	Called when instance is called as a function.
__len__(self)	Called by len() builtin
__getitem__(self, key)	Implements self[key]
__setitem__(self, key, value)	Implements self[key] = value
__selitem__(self, key)	Implements del self[key]
__iter__(self)	Called when iterator is applied to container
__reversed__(self)	Called by reversed() builtin
__contains__(self, object)	Implements in operator
__add__(self, other) __sub__(self, other) __mul__(self, other) __floordiv__(self, other) __mod__(self, other) __divmod__(self, other) __pow__(self, other[, modulo]) __lshift__(self, other) __rshift__(self, other) __and__(self, other) __xor__(self, other) __or__(self, other)	Implement binary arithmetic operators +, -, *, //, %, **, <<, >>, &, ^, and |. Self is object on left side of expression.
__div__(self,other) __truediv__(self,other)	Implement binary division operator /. __truediv__ is called if __future__.division is in effect.

Method or Variables	Description
__radd__(self, other) __rsub__(self, other) __rmul__(self, other) __rdiv__(self, other) __rtruediv__(self, other) __rfloordiv__(self, other) __rmod__(self, other) __rdivmod__(self, other) __rpow__(self, other) __rlshift__(self, other) __rrshift__(self, other) __rand__(self, other) __rxor__(self, other) __ror__(self, other)	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)
__iadd__(self, other) __isub__(self, other) __imul__(self, other) __idiv__(self, other) __itruediv__(self, other) __ifloordiv__(self, other) __imod__(self, other) __ipow__(self, other[, modulo]) __ilshift__(self, other) __irshift__(self, other) __iand__(self, other) __ixor__(self, other) __ior__(self, other)	Implement augmented (+=, -=, etc.) arithmetic operators
__neg__(self) __pos__(self) __abs__(self) __invert__(self)	Implement unary arithmetic operators -, +, abs(), and ~
__oct__(self) __hex__(self)	Implement oct() and hex() builtins
__index__(self)	Implement operator.index()
__coerce__(self, other)	Implement "mixed-mode" numeric arithmetic.

specialmethods.py

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

class Special(object):

    def __init__(self,value):
        self._value = str(value) ①

    def __add__(self,other):  ②
        return self._value + other._value

    def __mul__(self,num): ③
        return ''.join((self._value for i in range(num)))

    def __str__(self): ④
        return self._value.upper()

    def __eq__(self,other): ⑤
        return self._value == other._value

if __name__ == '__main__':
    s = Special('spam')
    t = Special('eggs')
    u = Special('spam')
    v = Special(5)  ⑥
    w = Special(22)

    print("s + s", s + s) ⑦
    print("s + t", s + t)
    print("t + t", t + t)
    print("s * 10", s * 10) ⑧
    print("t * 3", t * 3)
    print("str(s)={}  str(t)={}".format( str(s), str(t) ))
    print("id(s)={} id(t)={} id(u)={}".format( id(s), id(t), id(u) ))
    print("s == s", s == s)
    print("s == t", s == t)
    print("s == u", s == u)
    print("v + v", v + v)
    print("v + w", v + w)
    print("w + w", w + w)
    print("v * 10", v * 10)
    print("w * 3", w * 3)
```

① all Special instances are strings

② define what happens when a Special instance is added to another Special object

③ define what happens when a Special instance is multiplied by a value

④ define what happens when str() called on a Special instance

⑤ define equality between two Special valuess

⑥ parameter to Special() is converted to a string

⑦ add two Special instances

⑧ multiply a Special instance by an integer

specialmethods.py

```
s + s spamspam
s + t spameggs
t + t eggseggs
s * 10 spamspamspamspamspamspamspamspamspamspam
t * 3 eggseggseggs
str(s)=SPAM  str(t)=EGGS
id(s)=4546221952 id(t)=4546222008 id(u)=4546222176
s == s True
s == t False
s == u True
v + v 55
v + w 522
w + w 2222
v * 10 5555555555
w * 3 222222
```

Static Methods

- Related to class, but doesn't need instance or class object
- Use @staticmethod decorator

A static method is a utility method that is related to the class, but does not need the instance or class object. Thus, it has no automatic parameter.

One use case for static methods is to factor some kind of logic out of several methods, when the logic doesn't require any of the data in the class.

NOTE Static methods are seldom needed.

Chapter 7 Exercises

Exercise 7-1 (president.py, president_main.py)

Create a module that implements a **President** class. This class has a constructor that takes the index number of the president (1-45) and creates an object containing the associated information from the presidents.txt file.

Provide the following properties (types indicated after →):

```
term_number -> int
first_name -> string
last_name -> string
birth_date -> date object
death_date -> date object (or None, if still alive)
birth_place -> string
birth_state -> string
term_start_date -> date object
term_end_date -> date object (or None, if still in office)
party -> string
```

Write a main script to exercise some or all of the properties. It could look something like

```
from president import President

p = President(1)   # George Washington
print("George was born at {0}, {1} on {2}".format(
    p.birth_place, p.birth_state, p.birth_date
)
```

Chapter 8: Metaprogramming

Objectives

- Learn what metaprogramming means
- Access local and global variables by name
- Inspect the details of any object
- Use attribute functions to manipulate an object
- Create decorators for classes and functions

Metaprogramming

- Writing code that writes (or at least modifies) code
- Can simplify some kinds of programs
- Not as hard as you think!
- Considered deep magic in other languages

Metaprogramming is writing code that generates or modifies other code. It includes fetching, changing, or deleting attributes, and writing functions that return functions (AKA factories).

Metaprogramming is easier in Python than many other languages. Python provides explicit access to objects, even the parts that are hidden or restricted in other languages.

For instance, you can easily replace one method with another in a Python class, or even in an object instance. In Java, this would be deep magic requiring many lines of code.

globals() and locals()

- Contain all variables in a namespace
- globals() returns all global objects
- locals() returns all local variables

The **globals()** builtin function returns a dictionary of all global objects. The keys are the object names, and the values are the objects values. The dictionary is "live" — changes to the dictionary affect global variables.

The **locals()** builtin returns a dictionary of all objects in local scope.

Example

globals_locals.py

```
#!/usr/bin/env python
from pprint import pprint  ①

spam = 42  ②
ham = 'Smithfield'

def eggs(fruit):  ③
    name = 'Lancelot'   ④
    idiom = 'swashbuckling' ④
    print("Globals:")
    pprint(globals())  ⑤
    print()
    print("Locals:")
    pprint(locals())  ⑥

eggs('mango')
```

① import prettyprint function

② global variable

③ function parameters are local

④ local variable

⑤ globals() returns dict of all globals

⑥ locals() returns dict of all locals

globals_locals.py

```
Globals:
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__':
'/Users/jstrick/Documents/curr/courses/python/examples3/globals_locals.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x10376fef0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'eggs': <function eggs at 0x1036eeea0>,
 'ham': 'Smithfield',
 'pprint': <function pprint at 0x1039c3ea0>,
 'spam': 42}

Locals:
{'fruit': 'mango', 'idiom': 'swashbuckling', 'name': 'Lancelot'}
```

Working with attributes

- Objects are dictionaries of attributes
- Special functions can be used to access attributes
- Attributes specified as strings
- Syntax

```
getattr(object,  attribute [,defaultvalue] )
hasattr(object, attribute)
setattr(object, attribute, value)
delattr(object, attribute)
```

All Python objects are essentially dictionaries of attributes. There are four special builtin functions for managing attributes.

getattr() returns the value of a specified attribute, or None if the object does not have that attribute. a.spam is the same as getattr(a,'spam'). An optional third argument to getattr() provides a default value for nonexistent attributes.

hasattr() returns the value of a specified attribute, or None if the object does not have that attribute.

setattr() an attribute to a specified value.

delattr() deletes an attribute and its corresponding value.

Example

attributes.py

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

class Spam(object):

    def eggs(self,msg):   ①
        print("eggs!",msg)

s = Spam()

s.eggs("fried")

print("hasattr()",hasattr(s,'eggs'))  ②

e  = getattr(s,'eggs')  ③
e("scrambled")

def toast(self,msg):
    print("toast!",msg)

setattr(Spam,'eggs',toast)  ④

s.eggs("buttered!")

delattr(Spam,'eggs')  ⑤

try:
    s.eggs("shirred")
except AttributeError as err:  ⑥
    print(err)
```

① create attribute

② check whether attribute exists

③ retrieve attribute

④ set (or overwrite) attribute

⑤ remove attribute

⑥ missing attribute raises error

attributes.py

```
eggs! fried
hasattr() True
eggs! scrambled
toast! buttered!
'Spam' object has no attribute 'eggs'
```

The inspect module

- Simplifies access to metadata
- Provides user-friendly functions for testing metadata

The **inspect** module provides user-friendly functions for accessing Python metadata.

Example

inspect_ex.py

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

import inspect

class Spam: ①
    pass

def Ham(p1, p2='a', *p3, p4, p5='b', **p6):  ②
    pass

for thing in (inspect, Spam, Ham):
    print("{}: Module? {}. Function? {}. Class? {}".format(
        thing.__name__,
        inspect.ismodule(thing),   ③
        inspect.isfunction(thing), ④
        inspect.isclass(thing),    ⑤
    ))

print()

print("Function spec for Ham:",inspect.getfullargspec(Ham)) ⑥
print()

print("Current frame:",inspect.getframeinfo(inspect.currentframe())) ⑦
```

① define a class

② define a function

③ test for module

④ test for function

⑤ test for class

⑥ get argument specifications for a function

⑦ get frame (function call stack) info

inspect_ex.py

```
inspect: Module? True. Function? False. Class? False
Spam: Module? False. Function? False. Class? True
Ham: Module? False. Function? True. Class? False

Function spec for Ham: FullArgSpec(args=['p1', 'p2'], varargs='p3', varkw='p6',
defaults=('a',), kwonlyargs=['p4', 'p5'], kwonlydefaults={'p5': 'b'},
annotations={})

Current frame:
Traceback(filename='/Users/jstrick/Documents/curr/courses/python/examples3/inspect_e
x.py', lineno=24, function='<module>', code_context=['print("Current
frame:",inspect.getframeinfo(inspect.currentframe())) # <7>\n'], index=0)
```

Table 10. inspect module convenience functions

Function(s)	Description
ismodule(), isclass(), ismethod(), isfunction(), isgeneratorfunction(), isgenerator(), istraceback(), isframe(), iscode(), isbuiltin(), isroutine()	check object types
getmembers()	get members of an object that satisfy a given condition
getfile(), getsourcefile(), getsource()	find an object's source code
getdoc(), getcomments()	get documentation on an object
getmodule()	determine the module that an object came from
getclasstree()	arrange classes so as to represent their hierarchy
getargspec(), getargvalues()	get info about function arguments
formatargspec(), formatargvalues()	format an argument spec
getouterframes(), getinnerframes()	get info about frames
currentframe()	get the current stack frame
stack(), trace()	get info about frames on the stack or in a traceback

Decorators

- Classic design pattern
- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take parameters (but not required to)
- functools.wraps() preserves function's properties

In Python, many decorators are provided by the standard library, such as property() or classmethod(), with a special syntax. The @ sign is used to apply a decorator to a function or class.

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the @app.route() decorator in Flask maps a URL to a view function.

Table 11. Decorators in the standard library

Decorator	Description
@abc.abstractmethod	Indicate abstract method (must be implemented).
@abc.abstractproperty	Indicate abstract property (must be implemented).
@asyncio.coroutine	Mark generator-based coroutine.
@atexit.register	Register function to be executed when interpreter (script) exits.
@classmethod	Indicate class method (receives class object, not instance object)
@contextlib.contextmanager	Define factory function for **with** statement context managers (no need to create *enter*() and *exit*() methods)
@functools.lru_cache	Wrap a function with a memoizing callable
@functools.singledispatch	Transform function into a single-dispatch generic function.
@functools.total_ordering	Supply all other comparison methods if class defines at least one.
@functools.wraps	Invoke update_wrapper() so decorator's replacement function keeps original function's name and other properties.
@property	Indicate a class property.
@staticmethod	Indicate static method (passed neither instance nor class object).
@types.coroutine	Transform generator function into a coroutine function.
@unittest.mock.patch	Patch target with a new object. When the function/with statement exits patch is undone.
@unittest.mock.patch.dict	Patch dictionary (or dictionary-like object), then restore to original state after test.
@unittest.mock.patch.multiple	Perform multiple patches in one call.
@unittest.mock.patch.object	Patch object attribute with mock object.

Decorator	Description
@unittest.skip()	Skip test unconditionally
@unittest.skipIf()	Skip test if condition is true
@unittest.skipUnless()	Skip test unless condition is true
@unittest.expectedFailure()	Mark Test as expected failure
@unittest.removeHandler()	Remove Control-C handler

Decorator functions

- Provide a wrapper around a function
- Add functionality
- Syntax

```
@decorator
def function():
    pass
```

- Same as

```
function = decorator(function)
```

A decorator function acts as a wrapper around some function. It allows you to add features to a function without changing the function itself. For instance, the @property, @classmethod, and @staticmethod decorators are used in classes.

A decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code.

The new function should be defined with generic arguments so it can handle the original function's arguments.

The **wraps** decorator from the functools module in the standard library should be used with the function that returns the replacement function. This makes sure the replacement function keeps the same properties (especially the name) as the original (target) function.

Example

deco_debug.py

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

from functools import wraps

def debugger( old_func ):  ①

    @wraps(old_func)  ②
    def new_func( *args, **kwargs ): ③
        print("*" * 40)  ④
        print("** function", old_func.__name__,"**") ④

        if args: ④
            print("\targs are ", args)
        if kwargs: ④
            print("\tkwargs are ", kwargs)

        print("*" * 40) ④

        return old_func( *args, **kwargs )  ⑤

    return new_func   ⑥


@debugger  ⑦
def hello( greeting, whom='world'):
    print("{}, {}".format( greeting, whom ))

hello('hello','world') ⑧
print()

hello('hi','Earth')
print()

hello('greetings')
```

① decorator function — expects decorated (original) function as a parameter

② @wraps preserves name of original function after decoration

③ replacement function; takes generic parameters

④ new functionality added by decorator

⑤ call the original function

⑥ return the new function object

⑦ apply the decorator to a function

⑧ call new function

deco_debug.py

```
****************************************
** function hello **
    args are  ('hello', 'world')
****************************************
hello, world

****************************************
** function hello **
    args are  ('hi', 'Earth')
****************************************
hi, Earth

****************************************
** function hello **
    args are  ('greetings',)
****************************************
greetings, world
```

Decorator Classes

- Same purpose as decorator functions
- *_init_* method expects original function
- *_call_* method replaces original function

A class can also be used to implement a decorator. The class must implement two methods: *init* is passed the original function, and can perform any setup needed. The *call* method replaces the original function.

Example

deco_debug_class.py

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

class debugger(object): ①
    def __init__(self,func):  ②
        self._func = func

    def __call__( self, *args, **kwargs ):  ③

        print("*" * 40) ④
        print("** function", self._func.__name__,"**") ④

        if args:
          print("\targs are ", args) ④
        if kwargs:
          print("\tkwargs are ", kwargs) ④

        print("*" * 40) ④

        return self._func( *args, **kwargs )  ⑤

@debugger ⑥
def hello( greeting, whom="world"):
    print("{}, {}".format( greeting, whom ))

hello('hello','world') ⑦
print()

hello('hi','Earth')
print()

hello('greetings')
```

① decorator implemented as a class

② original function passed into decorator's constructor

③ *call*() is replacement function

④ add functionality to original function

⑤ call the original function

⑥ apply debugger to function

⑦ call replacement function

deco_debug_class.py

```
****************************************
** function hello **
    args are  ('hello', 'world')
****************************************
hello, world

****************************************
** function hello **
    args are  ('hi', 'Earth')
****************************************
hi, Earth

****************************************
** function hello **
    args are  ('greetings',)
****************************************
greetings, world
```

Decorator parameters

- Decorator functions require two nested functions
- Method *call* returns replacement function in classes

A decorator can be passed parameters. This requires a little extra work.

For decorators implemented as functions, the decorator itself is passed the parameters; it contains a nested function that is passed the decorated function (the target), and it returns the replacement function.

For decorators implemented as classes, init is passed the parameters, *call* is passed the decorated function (the target), and *call* returns the replacement function.

There are many combinations of decorators (8 total, to be exact). This is because decorators can be implemented as either functions or classes, they may take parameters, or not, and they can decorate either functions or classes. For an example of all 8 approaches, see the file **decorama.py** in the EXAMPLES folder.

Example

deco_params.py

```
#!/usr/bin/env python
# (c) 2018 CJ Associates
#

from functools import wraps  ①


def multiply(multiplier): ②

    def deco(old_func): ③

        @wraps(old_func)  ④
        def new_func(*args, **kwargs): ⑤
            result = old_func(*args, **kwargs) ⑥
            return result * multiplier ⑦

        return new_func ⑧

    return deco ⑨



@multiply(4)
def spam():
    return 5


@multiply(10)
def ham():
    return 8

a = spam()
b = ham()
print(a, b)
```

① wrapper to preserve properties of original function

② actual decorator — receives decorator parameters

③ "inner decorator" — receives function being decorated

④ retain name, etc. of original function

⑤ replacement function — this is called instead of original

⑥ call original function and get return value

⑦ multiple result of original function by multiplier

⑧ deco() returns new_function

⑨ multiply returns deco

deco_params.py

```
20 80
```

Creating classes at runtime

- Use the type() function
- Provide dictionary of attributes

For advanced needs, a class can be created programmatically, without the use of the class statement. The syntax is

```
type("name", (base_class, ⋯), {attributes})
```

The first argument is the name of the class, the second is a tuple of base classes (use object if you are not inheriting from a specific class), and the third is a dictionary of the class's attributes.

Example

creating_classes.py

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

def f1(self): ①
    print("Hello from f1()")

def f2(self): ①
    print("Hello from f2()")

new_class = type("new_class", (object,), {  ②
    'hello1': f1,
    'hello2': f2,
    'color': 'red',
    'state': 'Ohio',
})

n1 = new_class()  ③

n1.hello1()  ④
n1.hello2()
print(n1.color) ⑤
print()

sub_class = type("sub_class", (new_class,), {'fruit': 'banana'}) ⑥
s1 = sub_class() ⑦
s1.hello1()     ⑧
print(s1.color)  ⑨
print(s1.fruit)
```

① create method (not inside a class — could be a lambda)

② create class using type() — parameters are class name, base classes, dictionary of attributes

③ create instance of new class

④ call instance method

⑤ access class data

⑥ create subclass of first class

⑦ create instance of subclass

⑧ call method on subclass

⑨ access class data

creating_classes.py

```
Hello from f1()
Hello from f2()
red

Hello from f1()
red
banana
```

Monkey Patching

- Modify existing class or object
- Useful for enabling/disabling behavior
- Can cause problems

"Monkey patching" refers to technique of changing the behavior of an object by adding, replacing, or deleting attributes from outside the object's class definition.

It can be used for

Replacing methods, attributes, or functions
Modifying a third-party object for which you do not have access
Adding behavior to objects in memory

If you are not careful when creating monkey patches, some hard-to-debug problems can arise

- If the object being patched changes after a software upgrade, the monkey patch can fail in unexpected ways.
- Conflicts may occur if two different modules monkey-patch the same object.
- Users of a monkey-patched object may not realize which behavior is original and which comes from the monkey patch.

Monkey patching defeats object encapsulation, and so should be used sparingly.

Example

meta_monkey.py

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

class Spam(object): ①

    def __init__(self,name):
        self._name = name

    def eggs(self):  ②
        print("Good morning, {}. Here are your delicious fried
eggs.".format(self._name,))

s = Spam('Mrs. Higgenbotham') ③
s.eggs() ④

def scrambled(self): ⑤
    print("Hello, {}. Enjoy your scrambled eggs".format(self._name,))

setattr(Spam, "eggs", scrambled)   ⑥

s.eggs() ⑦
```

① create normal class

② add normal method

③ create instance of class

④ call method

⑤ define new method outside of class

⑥ monkey patch the class with the new method

⑦ call the monkey-patched method from the instance

meta_monkey.py

```
Good morning, Mrs. Higgenbotham. Here are your delicious fried eggs.
Hello, Mrs. Higgenbotham. Enjoy your scrambled eggs
```

Chapter 8 Exercises

Exercise 8-1 (pres_attr.py)

Instantiate the President class. Get the first name, last name, and party attributes using getattr().

Exercise 8-2 (pres_monkey.py, pres_monkey_amb.py)

Monkey-patch the President class to add a method get_full_name which returns a single string consisting of the first name and the last name, separated by a space.

*Try This_*: Instead of a method, make full_name a property.

Exercise 8-3 (sillystring.py)

Without using the **class** statement, create a class named SillyString, which is initialized with any string. Include an instance method called every_other which returns every other character of the string.

Instantiate your string and print the result of calling the **every_other()** method. Your test code should look like this:

```
ss = SillyString('this is a test')
print(ss.every_other())
```

It should output

```
ti sats
```

Exercise 8-4 (doubledeco.py)

Write a decorator to double the return value of any function. If a function returns 5, after decoration it should return 10. If it returns "spam", after decoration it should return "spamspam", etc.

Chapter 9: Developer Tools

Objectives

- Run pylint to check source code
- Debug scripts
- Find speed bottlenecks in code
- Compare algorithms to see which is faster

Program development

- More than just coding
  - Design first
  - Consistent style
  - Comments
  - Debugging
  - Testing
  - Documentation

Comments

- Keep comments up-to-date
- Use complete sentences
- Block comments describe a section of code
- Inline comments describe a line
- Don't state the obvious

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period. Use two spaces after a sentence-ending period.

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement; they should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1       # Increment x
```

Only use an inline comment if the reason for the statement is not obvious.

```
x = x + 1       # Compensate for border
```

The above was adapted from PEP 8

TIP | See Pep 257 for detailed suggestions for doc strings

pylint

- Checks many aspects of code
- Finds mistakes
- Rates your code for standards compliance
- Don't worry if your code has a lower rating!
- Can be highly customized

pylint is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells (as defined in Martin Fowler's Refactoring book)

from the pylint documentation

pylint can be very helpful in identifying errors and pointing out where your code does not follow standard coding conventions. It was developed by Python coders at Logilab http://www.logilab.fr.

It has very verbose output, which can be modified via command line options.

pylint can be customized to reflect local coding conventions. To use pylint, just say pylint filename, or pylint directory. Most Python IDEs have pylint, or the equivalent, built in.

Other tools for analyzing Python code: * pyflakes * pychecker

Customizing pylint

- Use pylint --generate-rcfile
- Redirect to file
- Edit as needed
- Knowledge of regular expressions useful
- Name file \~/.pylintrc on Linux/Unix/OS X
- Use –rcfile file to specify custom file on Windows

To customize pylint, run pylint with only the -generate-rcfile option. This will output a well-commented configuration file to STDOUT, so redirect it to a file.

Edit the file as needed. The comments describe what each part does. You can change the allowed names of variables, functions, classes, and pretty much everything else. You can even change the rating algorithm.

Windows

Put the file in a convenient location (name it something like pylintrc). Invoke pylint with the –rcfile option to specify the location of the file.

pylint will also find a file named pylintrc in the current directory, without needing the -rcfile option.

Non-Windows systems

On Unix-like systems (Unix, Mac OS, Linux, etc.), /etc/pylintrc and ~/.pylintrc will be automatically loaded, in that order.

See docs.pylint.org for more details.

Using pyreverse

- Source analyzer
- Reverse engineers Python code
- Part of pylint
- Generates UML diagrams

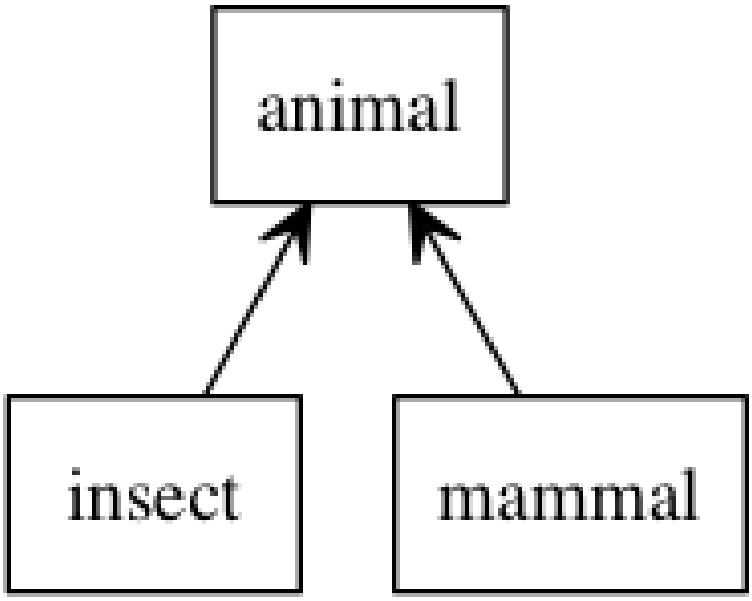
pyreverse is a Python source code analyzer. It reads a script, and the modules it depends on, and generates UML diagrams. It is installed as part of the pylint package.

There are many options to control what it analyzes and what kind of output it produces.

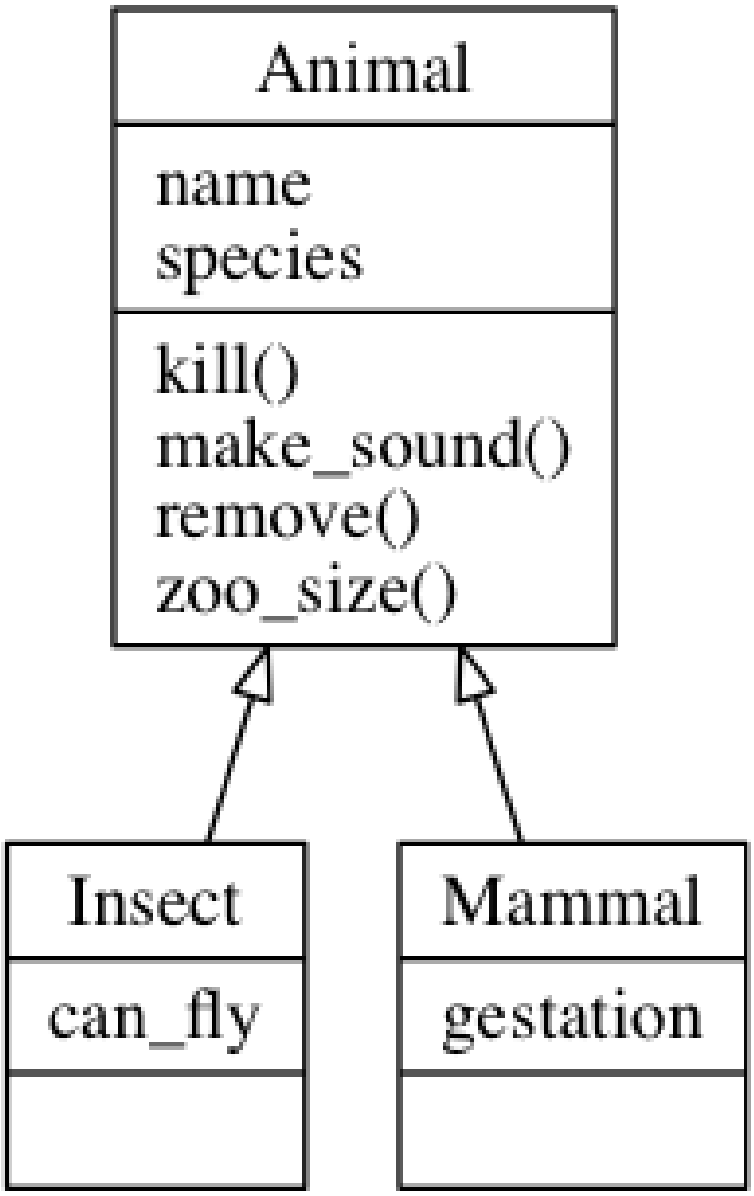
Example

```
pyreverse -o png -p MyProject -A animal.py mammal.py insect.py
```

packages_MyProject.png



classes_MyProject.png



NOTE **pyreverse** requires the (non-Python) Graphviz utility to be installed.

The Python debugger

- Implemented via pdb module
- Supports breakpoints and single stepping
- Based on gdb

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The pdb module provides debugging facilities for Python.

The usual way to use pdb is from the command line:

```
python -mpdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

Starting debug mode

- Syntax

```
python -m pdb script
```

or

```
import pdb
pdb.run('function')
```

pdb is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import some_module
>>> pdb.run('some_module.function_to_text()')
> <string>(0)?()
(Pdb) c    # (c)ontinue
> <string>(1)?()
(Pdb) c    # (c)ontinue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

To get help, type **h** at the debugger prompt.

Stepping through a program

- s single-step, stepping into functions
- n single-step, stepping over functions
- r return from function
- c run to next breakpoint or end

The debugger provides several commands for stepping through a program. Use **s** to step through one line at a time, stepping into functions.

Use **n** to step over functions; use **r** to return from a function; use **c** to continue to next breakpoint or end of program.

Pressing Enter repeats most commands; if the previous command was list, the debugger lists the next set of lines.

Setting breakpoints

- Syntax

```
b list all breakpoints
b linenumber (, condition)
b file:linenumber (, condition)
b function name (, condition)
```

Breakpoints can be set with the b command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The **tbreak** command creates a one-time breakpoint that is deleted after it is hit the first time.

Profiling

- Use the **profile** module from the command line
- Shows where program spends the most time
- Output can be tweaked via options

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

To use the profiler, execute the following at the command line:

```
python -m profile scriptname.py
```

This will output a simple report to STDOUT. You can also specify an output file with the -o option, and the sort order with the -s option. See the docs for more information.

Example

```
python -m profile count_with_dict.py
...script output...
         19 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       14    0.000    0.000    0.000    0.000 :0(get)
        1    0.000    0.000    0.000    0.000 :0(items)
        1    0.000    0.000    0.000    0.000 :0(open)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 count_with_dict.py:3(<module>)
        1    0.000    0.000    0.000    0.000 profile:0(<code object <module> at
0xb74c36e0, file "count_with_dict.py", line 3>)
        0    0.000             0.000          profile:0(profiler)
```

TIP The **pycallgraph** module (not in the standard library) will create a graphical representation of an application's profile, indicating visually where the application is spending the most time.

Benchmarking

- Use the timeit module
- Create a timer object with specified # of repetitions

Use the timeit module to benchmark two or more code snippets. To time code, create a Timer object, which takes two strings of code. The first is the code to test; the second is setup code, that is only run once per timer .

Call the timeit() method with the number of times to call the test code, or call the repeat() method which repeats timeit() a specified number of times.

Example

bm_range_vs_while.py

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

import timeit

setup_code = 'values = []' ①

test_code_one = '''
for i in range(10000):
    values.append(i)
'''  ②
test_code_two = '''
i = 0
while i < 10000:
    values.append(i)
    i += 1
'''  ②

t1 = timeit.Timer(test_code_one, setup_code) ③
t2 = timeit.Timer(test_code_two, setup_code) ③

print("test one:")
print(t1.timeit(1000))  ④
print()

print("test two:")
print(t2.timeit(1000)) ④
print()
```

① setup code is only executed once

② code fragment executed many times

③ Timer object creates time-able code

④ timeit() runs code fragment N times

bm_range_vs_while.py

```
test one:
1.4417963339947164

test two:
1.7840413979720324
```

Chapter 9 Exercises

Exercise 9-1

Pick several of your scripts (from class, or from real life) and run pylint on them.

Exercise 9-2

Use any available debugger to step through any of the scripts you have written so far.

Chapter 10: Unit Tests

Objectives

- Understand the purpose of unit tests
- Design and implement unit tests
- Run tests in different ways
- Learn how to mock data for tests

The unittest Module

- Provides automation for testing
- Test classes inherit from unittest.TestCase

A unit test is a test which asserts that an isolated piece of code (one function, method, or class) has some expected behavior. It is a way of making sure that code provides repeatable results.

The **unittest** module provides base classes and tools for creating, running, and managing unit tests.

There are three main components of a unit testing system:

1. Unit tests – individual assertions that an expected condition has been met
2. Test cases – collections of related unit tests
3. Test runners – utilities to execute the tests in one or more test cases

Unit tests should each test one aspect of your code, and each test should be independent of all other tests, including the order in which tests are run.

Unit tests are collected into a test case, which is a related group of unit tests. You can create test cases by inheriting from unittest.TestCase.

Another component of a unit test system is a test suite, which is a collection of test cases or other test suites. unittest also provides test suite builders. While you can create test suites in any arrangement, it is more common to let unittest find the tests for you using auto-discovery.

NOTE PyCharm can automatically detect a script that contains test cases. When you run it the first time, it will ask whether you want to run it normally or use its builtin test runner. Use Edit Configurations to modify how the script is run.

Creating a test case

- Inherit from unittest.TestCase
- Create one or more tests
- Add fixtures

To create a test case, define a class that inherits from **unittest.TestCase**. This will contain a collection of tests.

Each test is an instance method in the test case class. Individual test names must begin with "test", so they can be discovered by automated test runners. It is conventional to make test names verbose, so when test names are output, it is clear which tests are being run.

Each test makes an **assertion**; that is, it asserts that some condition is true. In addition to the builtin assert function, unittest.TestCase provides many specialized assertion functions to give better reporting when a test fails.

A test case my contain *fixtures*, which are methods that contain shared code and data, so it doesn't have to be duplicated in many individual tests.

Table 12. unittest Assertions

Methods	Assert that…
assertAlmostEqual assertNotAlmostEqual	Two expressions are equal [unequal] — difference between objects is less than the given delta (default zero).
assertDictContainsSubset	First dictionary is a superset of the second.
assertDictEqual	Two dictionaries have the same keys and values
assertEqual assertNotEqual	Two expressions are equal [unequal] as determined by the '==' operator.
assertGreater	The first expression is greater than the second
assertGreaterEqual	The first expression is greater than or equal to the second
assertIn assertNotIn	The first expression is [not] a member of the second
assertIs assertIsNot	The first expression is [not] the same object as the second
assertIsInstance assertNotIsInstance	The first expression is [not] an instance of the second
assertIsNone assertIsNotNone	The expression is [not] None
assertItemsEqual	The first expression and second expression have the same element counts (and the same elements, but not necessarily in the same order)
assertLess	The first expression is less than the second
assertLessEqual	The first expression is less than or equal to the second
assertListEqual	The first list is equal to the second
assertMultiLineEqual	The first multi-line strings is equal to the second
assertRaises	The specified exception is raised when the specified callable is invoked
assertRegexpMatches assertNotRegexpMatches	The expression matches [does not match] the specified regular expression (can be string or re instance)
assertSequenceEqual	The first ordered sequence (lists or tuples) is equal to the second
assertSetEqual	The first set is equal to the second

Methods	Assert that...
AssertTrue, assert_ assertFalse	The expression is True [False]
assertTupleEqual	The first tuple is equal to the second +2

Running tests

- Run test script directly (unittest.main())
- Use IDE's builtin test runner
- Use unittest

There are many ways to run tests.

If you have unittest.main() in the test script, you can run it like any other script:

```
python test_wombats.py
```

Many IDEs, such as PyCharm and Eclipse, have builtin test runners.

You can also use the unittest module from the command line:

```
python -m unittest test_wombats
python -m unittest test_wombats.TestWombats
python -m unittest test_wombats.TestWombats.test_wombat_shuffles_noisily
```

Fixtures

- Run before and after each test
- Factor out common code
- Predefined names: setUp() and tearDown()
- Class-level fixtures: setUpClass and tearDownClass()

To avoid duplicating code across many tests, unittest provides fixtures. These are methods that are called before or after each individual test. They can be used for some common task, such as initializing an array or creating a class instance.

If implemented in a test case, methods setUp() and tearDown() are called before and after each test.

You can also implement class-level fixtures. SetUpClass() and tearDownClass() are called at the beginning and end of the entire test case. These methods must be decorated with the **@classmethod** decorator.

Example

testrandom.py

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

# Test three functions from the random module:
import random
import unittest

class TestSequenceFunctions(unittest.TestCase): ①
    @classmethod
    def setUpClass(cls):  ②
        print("Starting all tests")

    def setUp(self):  ③
        print("Hello!")
        self.seq = list(range(10))

    def testshuffle(self):  ④
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, list(range(10))) ⑤

    def testchoice(self):
        element = random.choice(self.seq)
        self.assertIn(element, self.seq)  ⑥

    def testsample(self):
        self.assertRaises(ValueError, random.sample, self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertIn(element, self.seq)

    def tearDown(self):  ⑦
        print("Goodbye!")

    @classmethod
    def tearDownClass(cls):  ⑧
        print("Ending all tests")

if __name__ == '__main__':

    unittest.main()  ⑨
```

① TestCase objects contain one or more tests

② setUpClass is called once at the beginning of the test case, before any tests have run. Any data created by setUpClass should not be modified by tests

③ setUp is called before each individual test

④ all tests must start with "test"

⑤ all tests must assert some expression

⑥ there are many assertion types

⑦ tearDown is called after every test

⑧ tearDownClass is called after all tests

⑨ when this script is run, it will execute the default text-based test runner

testrandom.py

```
Starting all tests
Hello!
Goodbye!
Hello!
Goodbye!
Hello!
Goodbye!
Ending all tests
```

Skipping tests

- Skip tests depending on circumstances
- Decorate with
  - @unittest.skip(reason)
  - @unittest.skipIf(true-condition, reason)
  - @unittest.skipUnless(false-condition, reason)

To skip tests conditionally (or unconditionally), unittest provides three decorators. unittest.skip() skips a test unconditionally. unittest.skipIf() skips a test if the condition is true, and unittest.skipUnless() skips a test if the condition is false. The first parameter to all three decorators is a string with the reason for skipping the test.

You can also create your own decorator by creating a function that returns unittest.skip() if the test should be skipped, and the function itself otherwise.

Automated test discovery

- Find tests (classes inheriting from unittest.TestCase)
- Create suites
- Run all tests

The previous topic showed how to create suites of tests to run all the tests in a project. Because this is both tedious and straightforward, Python 2.7 (and 3.2) added tools to automate finding running tests.

While you can use it programmatically, the usual approach is to run the discover method of unittest from the command line as follows:

```
python -m unittest discover
```

This will find and run all tests in the specified folder and subfolders.

The following options can be added:

```
    -v, --verbose       verbose output
    -s, --start-directory   starting directory (defaults to .)
    -p, --pattern       pattern for test modules (defaults to test*.py)
    -t, --top-level-directory   top level of project (defaults to start directory)
```

Mocking data

- Simulate behavior of actual objects
- Replace expensive dependencies (time/resources)
- Use unittest.mock (formerly pymock)

Some objects have dependencies which can make unit testing difficult. These dependencies may be expensive in terms of time or resources.

The solution is to use a **mock** object, which pretends to be the real object. A mock object behaves like the original object, but is restricted and controlled in its behavior.

For instance, a class may have a dependency on a database query. A mock object may accept the query, but always returns a hard-coded set of results.

A mock object can record the calls made to it, and assert that the calls were made with correct parameters.

A mock object can be preloaded with a return value, or a function that provides dynamic (or random) return values.

A *stub* is an object that returns minimal information, and is also useful in testing. However, a mock object is more elaborate, with record/playback capability, assertions, and other features.

Mock objects

- Use unittest.mock.Mock
- Callable, so emulate functions or classes
- Return themselves by default

The unittest module provides a submodule, mock, which contains classes and helper functions for mocking.

The main class to use is Mock. A Mock object acts like any Python callable (i.e., class or function). You can pass any parameters to it. It will record those parameters for playback.

NOTE The MagicMock object is like Mock, but also has all of the "magic" methods (*str*(), *len*(), etc.) defined.

Example

mock_basics.py

```
#!/usr/bin/env python
#
import unittest
from unittest.mock import Mock

ham = Mock()  ①

# system under test
class Spam():
    def __init__(self, param):
        self._value = ham(param) ②

# dependency to be mocked -- not used in test
# def ham(n):
#     pass

class TestSpam(unittest.TestCase): ③

    def test_spam_calls_ham(self):
        spam = Spam(42)   ④
        ham.assert_called_once_with(42) ⑤

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

① create a Mock object — it will be used as a function here, but can be used as anything

② call the mock "ham()" function — works like the real ham() function

③ test case that will use the mock

④ Spam constructor calls ham()

⑤ assert that ham was called with the correct argument

mock_basics.py

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Return values

- Create mock function with specified return value
- Use return_value param to Mock()

You can have a mocked function or method return a specified value via the return_value parameter.

Example

mock_return_values.py

```
#!/usr/bin/env python
#
import unittest
from unittest.mock import Mock

RETURN_VALUE = 99

ham = Mock(return_value=RETURN_VALUE) ①

class Spam(): ②
    def __init__(self):
        self._value = ham() ③

    @property
    def value(self):  ④
        return self._value

# dependency to be mocked -- not used in test
# def ham():
#     return 42

class TestSpam(unittest.TestCase): ⑤

    def test_whatever(self):
        spam = Spam()  ⑥
        self.assertEqual(RETURN_VALUE, spam.value, "value is not
{}".format(RETURN_VALUE)) ⑦

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

① create a mock function that returns 99; this simulates the real-life ham() function

② system under test

③ call mock function and get return value (Spam() does not know that it's not the "real" ham())

④ Property to return value returned by ham()

⑤ Test case for Spam

⑥ Call Spam(), which in turn calls ham

⑦ Check that spam.value (which was returned by ham()) is equal to the correct value.

mock_return_values.py

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Chapter 10 Exercises

Exercise 10-1 (test_president.py)

Create a unit test for the President class you created earlier.

Suggestions for tests:

- What happens when an out-of-range term number is given
- President 1's first name is "George"
- Date fields return an object of type **datetime.date**

Chapter 11: Database Access

Objectives

- Understand the Python DB API architecture
- Connect to a database
- Execute simple and parameterized queries
- Fetch single and multiple row results
- Get metadata about a query
- Execute non-query statements
- Start transactions and commit or rollback as needed

The DB API

- Several ways to access DBMSs from Python
- DB API is most popular
- DB API is sort of an "abstract class"
- Many modules for different DBMSs using DB API
- Hides actual DBMS implementation

To make database programming simpler, Python has the DB API. This is an API to standardize working with databases. When a package is written to access a database, it is written to conform to the API, and thus programmers do not have to learn a new set of methods and functions.

Table 13. Available Interfaces (using Python DB API-2.0)

Database	**Python package**
Firebird (and Interbase)	KInterbasDB
IBM DB2	PyDB2
Informix	informixdb
Ingres	ingmod
Microsoft SQL Server	pymssql
MySQL	pymysql
ODBC	pyodbc
Oracle	cx_oracle
PostgreSQL	psycopg2
SAP DB (also known as "MaxDB")	sapdbapi
SQLite	sqlite3
Sybase	Sybase

NOTE There may be other interfaces to some of the listed DBMSs as well.

Connecting to a Server

- Import appropriate library
- Use connect() to get a database object
- Specify host, database, username, password

To connect to a database server, import the package for the specific database. Use the package's **connect()** method to get a database object, specifying the host, initial database, username, and password. If the username and password are not needed, use None.

Some database modules have nonstandard parameters to the connect() method.

When finished with the connection, call the **close()** method on the connection object.

Many database modules support the context manager (**with** statement), and will automatically close the database when the with block is exited. Check the documentation to see how this is implemented for a specific database.

Example

```
import sqlite3

slconn = sqlite3.connect('web_content')

import pymysql

myconn = pymysql.connect (host = "myserver1",
                          user = "adeveloper",
                          passwd = "s3cr3t",
                          db = "web_content")
#  make queries, etc. here ...
myconn.close()
```

NOTE Argument names for the connect() method may not be consistent. For instance, pymysql supports the above parameter names, while pymssql does not.

Table 14. connect() examples

Package	Database	Connection

cx_oracle	Oracle	`ip = 'localhost'`
		`port = 1521`
		`SID = 'YOURSIDHERE'`
		`dsn_tns = cx_Oracle.makedsn(ip, port, SID)`
		
		`db = cx_Oracle.connect('adeveloper', '$3cr3t', dsn_tns)`

psychopg	PostgreSQL	`psycopg2.connect ('''`
		`host='localhost'`
		`user='adeveloper'`
		`password='$3cr3t'`
		`dbname='testdb'`
		`''')`
		note: connect() has one (string) parameter, not multiple parameters

pymssql	MS-SQL	`pymssql.connect (`
		`host="localhost",`
		`user="adeveloper",`
		`passwd="$3cr3t",`
		`db="testdb",`
		`)`
		
		`pymssql.connect (`
		`dsn="DSN",`
		`)`

pymysql	MySQL	`pymysql.connect (`
		`host="localhost",`
		`user="adeveloper",`
		`passwd="$3cr3t",`
		`db="testdb",`
		`)`

pyodbc	Any ODBC-compliant DB	`pyodbc.connect('''`
		`DRIVER={SQL Server};`
		`SERVER=localhost;`
		`DATABASE=testdb;`
		`UID=adeveloper;`
		`PWD=$3cr3t`
		`''')`
		
		`pyodbc.connect('DSN=testdsn;PWD=$3cr3t')`
		note: connect() has one (string) parameter, not multiple parameters

sqlite3	SqlLite3	`sqlite3.connect('testdb')`
		
		`sqlite3.connect(':memory:')`

Creating a Cursor

- Cursor can execute SQL statements
- Multiple cursors available
  - Standard cursor
    - Returns tuples
  - Other cursors
    - Returns dictionaries
    - Leaves data on server

Once you have a database object, you can create one or more cursors. A cursor is an object that can execute SQL code and fetch results.

The default cursor for most packages returns each row as a tuple of values. There are different types of cursors that can return data in different formats, or that control whether data is stored on the client or the server.

Example

```
myconn = pymysqlconnect (host="myserver1",user="adeveloper", passwd="s3cr3t",
db="web_content")
mycursor = myconn.cursor()
```

Executing a Statement

- Executing cursor sends SQL to server
- Data not returned until asked for
- Returns number of lines in result set for queries
- Returns lines affected for other statements

Once you have a cursor, you can use it to perform queries, or to execute arbitrary SQL statements via the execute() method. The first argument to execute() is a string containing the SQL statement to run.

Example

```
cursor.execute("select hostname,ostype,user from hostinfo")
cursor.execute('insert into hostinfo values
("foo",5,"2.6","arch","net",2055,3072,"bob",0)')
```

Fetching Data

- Use one of the fetch methods from the cursor object
- Syntax
  - `rec = cursor.fetchone()`
  - `recs = cursor.fetchall()`
  - `recs = cursor.fetchmany()`

Cursors provide three methods for returning query results.

fetchone() returns the next available row from the query results.

fetchall() returns a tuple of all rows.

fetchmany(n) returns up to n rows. This is useful when the query returns a large number of rows.

Example

```
cursor.execute("select color, quest from knights where name = 'Robin'")
(color,quest) = cursor.fetchone()

cursor.execute("select color, quest from knights")
rows = cursor.fetchall()

cursor.execute("select * from huge_table)
while True:
    rows = cursor.fetchmany(1000)
    if rows == []:
        break
    for row in rows:
        # process row
```

Example

db_sqlite_basics.py

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

import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn: ①

    s3cursor = s3conn.cursor()  ②

    # select first name, last name from all presidents
    s3cursor.execute('''
        select firstname, lastname
        from presidents
    ''')  ③

    print("Sqlite3 does not provide a row count\n")  ④

    for row in s3cursor.fetchall():  ⑤
        print(' '.join(row))  ⑥
```

① connect to the database

② get a cursor object

③ execute a SQL statement

④ (included for consistency with other DBMS modules)

⑤ fetchall() returns all rows

⑥ each row is a tuple

db_sqlite_basics.py

```
Lyndon Baines Johnson
Richard Milhous Nixon
Gerald Rudolph Ford
James Earl 'Jimmy' Carter
Ronald Wilson Reagan
George Herbert Walker Bush
William Jefferson 'Bill' Clinton
George Walker Bush
Barack Hussein Obama
Donald J Trump
```

NOTE See db_mysql_basics.py and db_postgres_basics.py for examples using those modules. In general, all the sqlite3 examples are also implemented for MySQL and Postgres.

SQL Injection

- "Hijacks" SQL code
- Result of string formatting
- Always use parameterized statements

One kind of vulnerability in SQL code is called SQL injection. This occurs when an attacker embeds SQL commands in input data. This can happen when naively using string formatting to build SQL statements:

Example

db_sql_injection.py

```
#!/usr/bin/env python
#
good_input = 'Google'
malicious_input  = "'; drop table customers; -- "  ①

naive_format = "select * from customers where company_name = '{}' and company_id !=
0"

good_query = naive_format.format(good_input)  ②
malicious_query = naive_format.format(malicious_input)  ②

print("Good query:")
print(good_query)  ③
print()

print("Bad query:")
print(malicious_query)  ④
```

① input would come from a web form, for instance

② string formatting naively adds the user input to a field, expecting only a customer name

③ non-malicious input works fine

④ query now drops a table ('--' is SQL comment)

db_sql_injection.py

```
Good query:
select * from customers where company_name = 'Google' and company_id != 0

Bad query:
select * from customers where company_name = ''; drop table customers; -- ' and
company_id != 0
```

Parameterized Statements

- More efficient updates
- Use placeholders in query
  - Placeholders vary by DB
- Pass iterable of parameters
- Prevent SQL injection
- Use cursor.execute() or cursor.executemany()

For efficiency, you can iterate over of sequence of input datasets when performing a non-query SQL statement. The execute() method takes a query, plus an iterable of values to fill in the placeholders. The database manager will only parse the query once, then reuse it for subsequent calls to execute().

Parameterized queries also protect against SQL injection attacks.

Different database modules use different placeholders. To see what kind of placeholder a module uses, check MODULE.paramstyle. Types include 'pyformat', meaning '%s', and 'qmark', meaning '?'.

The executemany() method takes a query, plus an iterable of iterables. It will call execute() once for each nested iterable.

Example

```
single_row = ("Smith","John","green"),

multi_rows= [
    ("Smith","John","green"),
    ("Douglas","Sam","pink"),
    ("Robinson","Alberta","blue"),
]

query = "insert into people (lname,fname,color) values (%s,%s,%s)"

rows_added = cursor.execute(query, single_row)
rows_added = cursor.executemany(query, multi_rows)
```

Table 15. Placeholders for SQL Parameters

Python package	Placeholder for parameters
pymysql	%s
cx_oracle	:param_name
pyodbc	?
pymssql	%d for int, %s for str, etc.
Psychopg	%s or %(param_name)s
sqlite3	? or :param_name

TIP with the exception of **pymssql** the same placeholder is used for all column types.

Example

db_sqlite_parameterized.py

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

import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn:

    s3cursor = s3conn.cursor()

    party_query = '''
    select firstname, lastname
    from presidents
        where party = ?
    '''   ①

    for party in 'Federalist', 'Whig':
        print(party)
        s3cursor.execute(party_query, (party,))  ②
        print(s3cursor.fetchall())
        print()
```

① ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use different placeholders

② second argument to execute() is iterable of values to fill in placeholders from left to right

db_sqlite_parameterized.py

```
Federalist
[('John', 'Adams')]

Whig
[('William Henry', 'Harrison'), ('John', 'Tyler'), ('Zachary', 'Taylor'),
('Millard', 'Fillmore')]
```

Example

db_sqlite_bulk_insert.py

```
#!/usr/bin/env python
import os
import sqlite3
import random

FRUITS = ["pomegranate", "cherry", "apricot", "date", "apple",
"lemon", "kiwi", "orange", "lime", "watermelon", "guava",
"papaya", "fig", "pear", "banana", "tamarind", "persimmon",
"elderberry", "peach", "blueberry", "lychee", "grape" ]

DB_NAME = 'fruitprices.db'  ①

CREATE_TABLE = """
create table fruit (
    name varchar(30),
    price decimal
)
"""  ②

INSERT = '''
insert into fruit (name, price) values (?, ?)
'''  ③

def main():
    """
    Program entry point.

    :return: None
    """
    conn = get_connection()
    create_database(conn)
    populate_database(conn)

    read_database()


def get_connection():
    """
    Get a connection to the PRODUCE database
```

```
    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)  ④

    s3conn = sqlite3.connect(DB_NAME) ⑤
    return s3conn

def create_database(conn):
    """
    Create the fruit table

    :param conn: The database connection
    :return: None
    """
    conn.execute(CREATE_TABLE) ⑥

def populate_database(conn):
    """
    Add rows to the fruit table

    :param conn: The database connection
    :return: None
    """

    fruit_data = get_fruit_data()

    conn.executemany(INSERT, fruit_data) ⑦

    conn.commit()  ⑧

def get_fruit_data():
    """
    Create iterable of fruit records.

    :return: Generator of name/price tuples.
    """
    return ((f, round(random.random() * 10 + 5, 2)) for f in FRUITS) ⑨

def read_database():
    conn = sqlite3.connect(DB_NAME)
    for name, price in conn.execute('select name, price from fruit'):
        print('{:12s} {:6.2f}'.format(name, price))
```

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

① set name of database

② SQL statement to create table

③ parameterized SQL statement to insert one record

④ remove existing database if it exists

⑤ connect to (new) database

⑥ run SQL to create table

⑦ iterate over list of pairs and add each pair to the database

⑧ commit the inserts; without this, no data would be saved

⑨ build list of tuples containing fruit, price pairs

db_sqlite_bulk_insert.py

```
pomegranate   13.80
cherry         6.39
apricot        5.87
date           9.69
apple          5.53
lemon          7.41
kiwi          10.06
orange         6.02
lime           5.54
watermelon    14.62
guava         10.30
papaya        14.76
fig           10.50
pear           8.30
banana         7.41
tamarind      10.21
persimmon     13.71
elderberry    11.26
peach          5.62
blueberry      7.12
lychee        13.89
grape         14.72
```

Dictionary Cursors

- Indexed by column name
- Not standardized in the DB API

The standard cursor provided by the DB API returns a tuple for each row. Most DB packages provide other kinds of cursors, including user-defined versions.

A very common cursor is a dictionary cursor, which returns a dictionary for each row, where the keys are the column names. Each package that provides a dictionary cursor has its own way of providing the dictionary cursor, although they all work the same way.

For the packages that don't have a dictionary cursor, you can make a generator function that will emulate one.

Table 16. Dictionary Cursors

Python package	How to get a dictionary cursor

pymysql	`import pymysql.cursors`
	`conn = pymysql.connect(…,`
	`cursorclass = pymysql.cursors.DictCursor`
	`)`
	`dcur = conn.cursor()`
	all cursors will be dict cursors
	
	`dcur = conn.cursor(`
	`pymysql.cursors.DictCursor)`
	only this cursor will be a dict cursor

cx_oracle	*Not available*
pyodbc	*Not available*
pgdb	*Not available*
pymssql	`conn = pymssql.connect (…, as_dict=True)` `dcur = conn.cursor()`

psychopg	`import psycopg2.extras`
	`dcur =`
	`conn.cursor(cursor_factory=psycopg.extras.Dic`
	`tCursor)`

sqlite3	`conn = sqlite3.connect (…,`
	`row_factory=sqlite3.Row)`
	`dcur = conn.cursor()`
	`conn.row_factory = sqlite3.Row`
	`dcur = conn.cursor()`

Example

db_sqlite_extras.py

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

import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")
# uncomment to make _all_ cursors dictionary cursors
# conn.row_factory = sqlite3.Row

NAME_QUERY = '''
    select firstname, lastname
    from presidents
    where termnum < 5
'''

cur = s3conn.cursor()

# select first name, last name from all presidents
cur.execute(NAME_QUERY)

for row in cur.fetchall():
    print(row)
print('-' * 50)

dcur = s3conn.cursor()  ①

# make _this_ cursor a dictionary cursor
dcur.row_factory = sqlite3.Row  ②

# select first name, last name from all presidents
dcur.execute(NAME_QUERY)

for row in dcur.fetchall():
    print(row['firstname'], row['lastname']) ③

print('-' * 50)
```

① default cursor returns tuple for each row

② Row object is tuple/dict hybrid; can be indexed by position OR column name

③ selecting by column name

db_sqlite_extras.py

```
('George', 'Washington')
('John', 'Adams')
('Thomas', 'Jefferson')
('James', 'Madison')
------------------------------------------------
George Washington
John Adams
Thomas Jefferson
James Madison
------------------------------------------------
```

Metadata

- cursor.description returns tuple of tuples
- Fields
  - name
  - type_code
  - display_size
  - internal_size
  - precision
  - scale
  - null_ok

Once a query has been executed, the cursor's description() method returns metadata about the columns in the query as a tuple of tuples.

There is one tuple for each column in the query; each tuple contains a tuple of 7 values describing the column.

For instance, to get the names of the columns, you could say:

```
names = [ d[0] for d in cursor.description ]
```

For non-query statements, cursor.description returns None.

The names are based on the query (with possible aliases), and not necessarily on the names in the table.

NOTE Most database modules, including pymysql, have a dictionary cursor built in — this is just for an example you could use with any DB API module that does not have this capability. The example uses the metadata from the cursor to get the column names, and forms a dictionary by zipping the column names with the column values. Another approach would be to use a named tuple. __

Example

db_sqlite_emulate_dict_cursor.py

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

import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")

c = s3conn.cursor()

def row_as_dict(cursor):
    '''Generate rows as dictionaries'''
    column_names = [desc[0] for desc in cursor.description]
    for row in cursor.fetchall():
        row_dict = dict(zip(column_names, row))
        yield row_dict


# select first name, last name from all presidents
num_recs = c.execute('''
    select lastname, firstname
    from presidents
''')

for row in row_as_dict(c):
    print(row['firstname'], row['lastname'])
```

db_sqlite_emulate_dict_cursor.py

```
Lyndon Baines Johnson
Richard Milhous Nixon
Gerald Rudolph Ford
James Earl 'Jimmy' Carter
Ronald Wilson Reagan
George Herbert Walker Bush
William Jefferson 'Bill' Clinton
George Walker Bush
Barack Hussein Obama
Donald J Trump
```

See db_sqlite_named_tuple_cursor.py for a similar example that creates named tuples rather than dictionaries for each row.

Transactions

- Transactions allow safer control of updates
- commit() to save transactions
- rollback() to discard
- Default is autocommit off
- autocommit\=True to turn on

Sometimes a database task involves more than one change to your database (i.e., more than one SQL statement). You don't want the first SQL statement to succeed and the second to fail; this would leave your database in a corrupt state.

To be certain of data integrity, use **transactions**. This lets you make multiple changes to your database and only commit the changes if all the SQL statements were successful. For all packages using the Python DB API, a transaction is started when you connect. At any point, you can call *CONNECTION*.commit() to save the changes, or *CONNECTION*.rollback() to discard the changes. For most packages, if you don't call commit() after modify a table, the data will not be saved.

NOTE You can also turn on autocommit, which calls commit() after every statement.

Example

```
try:
    for info in list_of_tuples:
         cursor.execute(query,info)
except SQLError:
    dbconn.rollback()
else:
    dbconn.commit()
```

NOTE **pymysql** only supports transaction processing when using the **InnoDB** engine

Object-relational Mappers

- No SQL required
- Maps a class to a table
- All DB work is done by manipulating objects
- Most popular Python ORMs
  - SQLAlchemy
  - Django (which is a complete web framework)

An Object-relational mapper is a module or framework that creates a level of abstraction above the actual database tables and SQL queries. As the name implies, a Python class (object) is mapped to the actual table.

The two most popular Python ORMs are SQLAlchemy which is a standalone ORM, and Django ORM. Django is a comprehensive Web development framework, which provides an ORM as a subpackage. SQLAlchemy is the most fully developed package, and is the ORM used by Flask and some other Web development frameworks.

Instead of querying the database, you call a search method on an object representing a table. To add a row to the table, you create a new instance of the table class, populate it, and call a method like save(). You can create a large, complex database system, complete with foreign keys, composite indices, and all the other attributes near and dear to a DBA, without writing the first line of SQL.

You can use Python ORMs in two ways.

One way is to design the database with the ORM. To do this, you create a class for each table in the database, specifying the columns with predefined classes from the ORM. Then you run an ORM command which executes the queries needed to build the database. If you need to make changes, you update the class definitions, and run an ORM command to synchronize the actual DBMS to your classes.

The second way is to map tables to an existing database. You create the classes to match the schemas that have already been defined in the database. Both SQLAlchemy and the Django ORM have tools to automate this process.

NoSQL

- Non-relational database
- Document-oriented
- Can be hierarchical (nested)
- Examples
  - MongoDB
  - Cassandra
  - Redis

A current trend in data storage are called "NoSQL" or non-relational databases. These databases consist of *documents*, which are indexed, and may contain nested data.

NoSQL databases don’t contain tables, and do not have relations.

While relational databases are great for tabular data, they are not as good a fit for nested data. Geo-spatial, engineering diagrams, and molecular modeling can have very complex structures. It is possible to shoehorn such data into a relational database, but a NoSQL database might work much better. Another advantage of NoSQL is that it can adapt to changing data structures, without having to rebuild tables if columns are added, deleted, or modified.

Some of the most common NoSQL database systems are MongoDB, Cassandra and Redis.

Example

mongodb_example.py

```
#!/usr/bin/env python
import re
from pymongo import MongoClient

FIELD_NAMES = (
    'termnumber lastname firstname '
    'birthdate '
    'deathdate birthplace birthstate '
    'termstartdate '
    'termenddate '
    'party'
).split() ①

mc = MongoClient() ②

try:
    mc.drop_database("presidents") ③
except:
    pass

db = mc["presidents"] ④

coll = db.presidents  ⑤

with open('../DATA/presidents.txt') as PRES: ⑥
    for line in PRES:
        flds = line[:-1].split(':')
        kvpairs = zip(FIELD_NAMES, flds)
        record_dict = dict(kvpairs)
        coll.insert(record_dict)  ⑦

print(db.collection_names())  ⑧
print()

abe = coll.find_one({'termnumber': '16'})  ⑨
for field in FIELD_NAMES:
    print("{0:15s} {1}".format(field.upper(), abe[field])) ⑩

print('-' * 50)
```

```
for president in coll.find(): ⑪
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

rx_lastname = re.compile('^roo', re.IGNORECASE)
for president in coll.find({ 'lastname': rx_lastname }): ⑫
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

for president in coll.find({"birthstate": 'Virginia',  'party': 'Whig'}): ⑬
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))

print('-' * 50)
print("removing Millard Fillmore")
result = coll.remove({'lastname': 'Fillmore'})  ⑭
print(result)
result = coll.remove({'lastname': 'Roosevelt'}) ⑭
print(result)
print('-' * 50)
result = coll.count()  ⑮
print(result)
```

① define some field name

② get a Mongo client

③ delete 'presidents' database if it exists

④ create a new database named 'presidents'

⑤ get the collection from presidents db

⑥ open a data file

⑦ insert a record into collection

⑧ get list of collections

⑨ search collection for doc where termnumber == 16

⑩ print all fields for one record

⑪ loop through all records in collection

⑫ find record using regular expression

⑬ find record searching multiple fields

⑭ delete record

⑮ get count of records

mongodb_example.py

```
Dwight David              Eisenhower
John Fitzgerald           Kennedy
Lyndon Baines             Johnson
Richard Milhous           Nixon
Gerald Rudolph            Ford
James Earl 'Jimmy'        Carter
Ronald Wilson             Reagan
George Herbert Walker     Bush
William Jefferson 'Bill'  Clinton
George Walker             Bush
Barack Hussein            Obama
Donald John               Trump
-------------------------------------------------
Theodore                  Roosevelt
Franklin Delano           Roosevelt
-------------------------------------------------
William Henry             Harrison
John                      Tyler
Zachary                   Taylor
-------------------------------------------------
removing Millard Fillmore
{'ok': 1, 'n': 1}
{'ok': 1, 'n': 2}
-------------------------------------------------
42
```

Chapter 11 Exercises

Exercise 11-1 (president_sqlite.py)

For this exercise, you can use the SQLite3 database provided, or use your own DBMS. The mkpres.sql script is generic and should work with any DBMS to create and populate the presidents table. The SQLite3 database is named **presidents.db** and is created in the DATA folder of the student files.

The data has the following layout

Table 17. Layout of President Table

Field Name	Data Type	Null	Default
termnum	int(11)	YES	NULL
lastname	varchar(32)	YES	NULL
firstname	varchar(64)	YES	NULL
dstart	date	YES	NULL
dend	date	YES	NULL
birthplace	varchar(128)	YES	NULL
dbirth	date	YES	NULL
ddeath	date	YES	NULL
party	varchar(32)	YES	NULL

Refactor your **president.py** module to get its data from this table, rather than from a file. Re-run your previous scripts that used president.py; now they should get their data from the database, rather than from the flat file.

Exercise 11-2 (add_pres_sqlite.py)

Assuming your favorite candidate wins (or has won) the next election, add them to the table.

HINT: Just make up the data as necessary (or Google it)

SQL syntax for adding a record is

```
INSERT INTO table ("COL1-NAME",..) VALUES ("VALUE1",...)
```

To do a parameterized insert (the right way!):

```
INSERT INTO table ("COL1-NAME",..) VALUES (%s,%s,...)  # MySQL
INSERT INTO table ("COL1-NAME",..) VALUES (?,?,...)    # SQLite
```

or whatever your database uses as placeholders

NOTE There are also MySQL versions of the answers.

Chapter 12: PyQt

Objectives

- Explore PyQt programming
- Understand event-driven programming
- Code a minimal PyQt application
- Use the Qt designer to create GUIs
- Wire up the generated GUI to event handlers
- Validate input
- Use predefined dialogs
- Design and use custom dialogs

What is PyQt?

- Python Bindings for Qt library
- Qt written in C++ but ported to many languages
- Complete GUI library
- Cross-platform (OS X, Windows, Linux, et al.)
- Hides platform-specific details

PyQt is a package for Python that provides binding to the generic Qt graphics programming library. Qt provides a complete graphics programming framework, and looks "native" across various platforms. In addition to graphics components, it includes database access and many other tools.

It hides the platform-specific details, so PyQt programs are portable.

Matplotlib can be integrated with PyQT for data visualizations.

NOTE The current version of PyQt is PyQt 5. There are Python packages to support both PyQt4 and PyQt5. In the EXAMPLES folder, there are subfolders for each package. The only difference in the scripts is in the PyQt imports.

Event Driven Applications

- Application starts event loop
- When event occurs, goes to handler function, then back to loop
- Terminate event ends the loop (and the app)

GUI programs are different from conventional, procedural applications. Instead of the programmer controlling the order of execution via logic, the user controls the order of execution by manipulating the GUI.

To accomplish this, starting a GUI app launches an event loop, which "listens" for user-generated events, such as key presses, mouse clicks, or mouse motion. If there is a method associated with the event (AKA "event handler") (AKA "slot" in PyQt), the method is invoked.

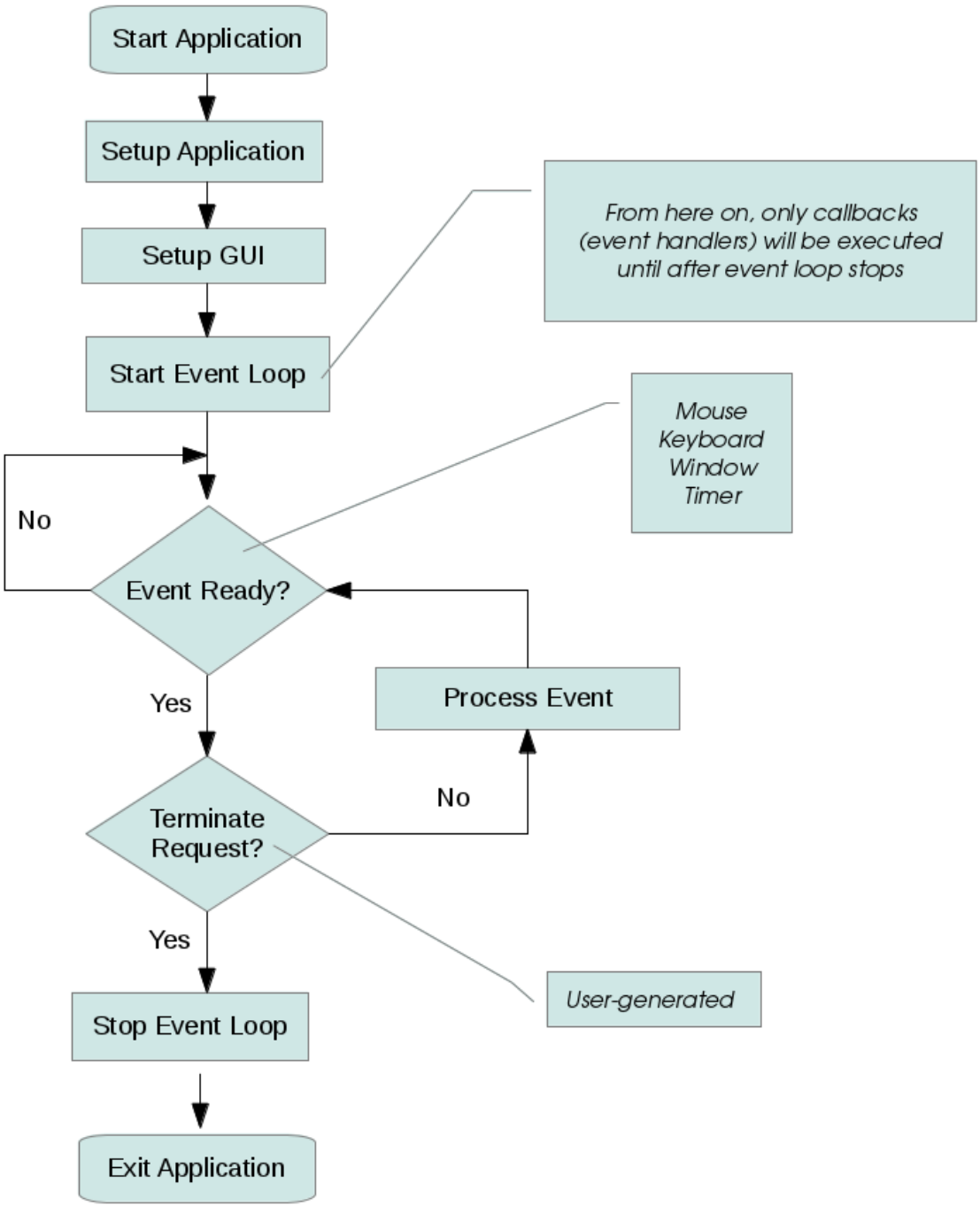
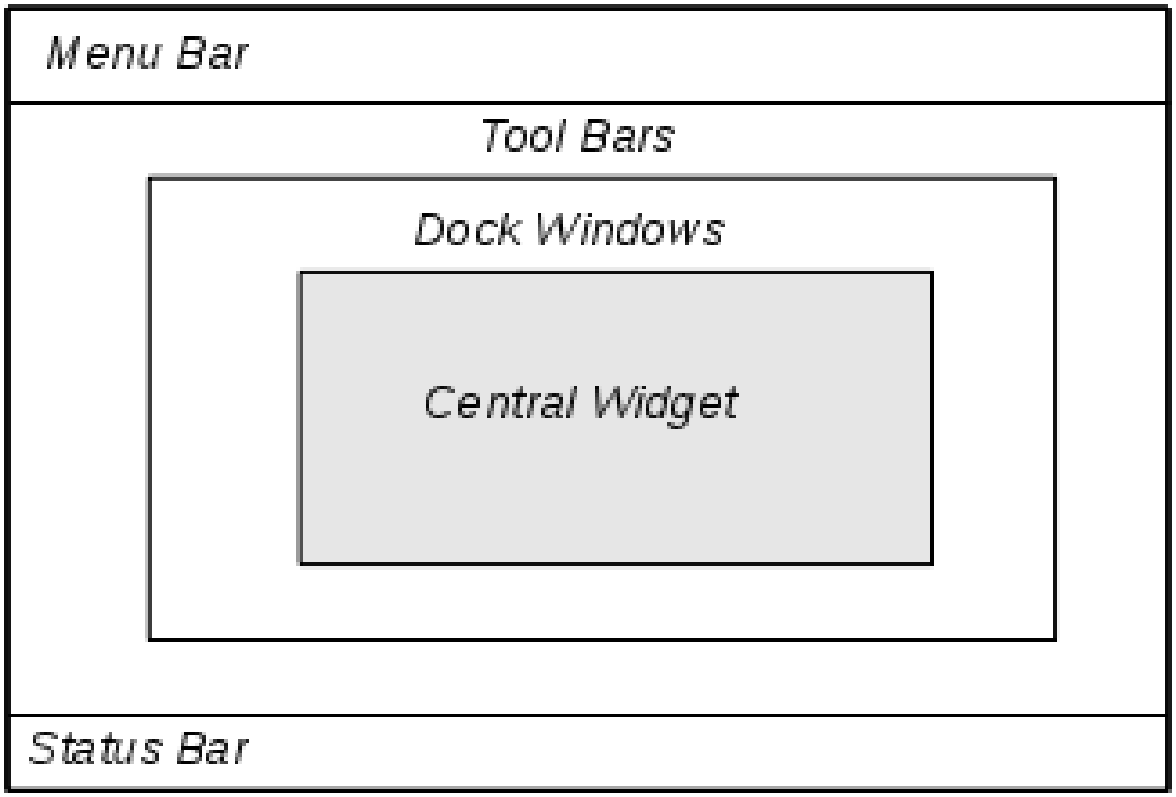


Figure 1. GUI Application Flow Chart

External Anatomy of a PyQt Application



The main window widget has several predefined areas:

- The menu bar area contains the usual File, Edit, and other standard menus.
- The tool bar areas can contain any tool bar buttons.
- The dock windows area contains any docked windows, which can be docked in any of the doc areas, or which can float freely. They have their own title bar with close, minimize and maximize buttons.
- The status bar can contain any other widgets, typically labels, but anything is fair game.
- None of the above are required, and if not present will not take up any screen space.
- The central widget is the main widget of the application. It is typically a QWidget layout object such as VBoxLayout, HBoxLayour, or GridLayout.

Internal Anatomy of a PyQt Application

- Extend (subclass) QmainWindow
- Call show() on main class

The normal (and convenient) approach is to subclass QMainWindow to create a custom main window for your application. Within this class, ***self*** is the main window of the application. You can attach widgets to and call setup methods from self.

QApplication is an object which acts as the application itself. You need to create a QApplication object and pass the command line arguments to it. To start your program call the exec_() method on your application object, after calling **show()** on the main window to make it visible.

Example

qt5/qt_hello.py

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

import sys
from PyQt5.QtWidgets import QMainWindow, QApplication, QLabel ①

class HelloWindow(QMainWindow): ②

    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)  ③
        self._label = QLabel("Hello PyQt5 World")
        self.setCentralWidget(self._label)

if __name__ == "__main__":
    app = QApplication(sys.argv)  ④
    main_window = HelloWindow()
    main_window.show()
    sys.exit(app.exec_())
```

① Standard PyQt5 imports

② Main class inherits from QMainWindow to have normal application behavior

③ Must call QMainWindow constructor

④ These 4 lines are always required. Only the name of the main window object changes.

Using designer

- GUI for building GUIs
- Builds applications, dialogs, and widgets
- Outputs generic XML that describes GUI
- pyuic5 (or pyuic4) generates Python module from XML
- Import generated module in main script

The **designer** tool makes it fast and easy to generate any kind of GUI.

To get started with designer, choose **New...** from the File menu.

Select Main Window. (You can also use designer to create dialogs and widgets). This will create a blank application window. It will already have a menu bar and a status bar. It is ready for you to drag layouts and widgets as needed.

Be sure to change the objectName property of the QMainWindow object in the Property Editor. This (with "Ui_" prefixed) will be the name of the GUI class generated by pyuic4.

To set the title of your application, which will show up on the title bar, select the QmainWindow object in the Object Inspector, then open the QWidget group of properties in the Property Editor. Enter the title in the windowTitle property.

Be sure to give your widgets meaningful names, so when you have to use them in your main program, you know which widget is which. A good approach is a short prefix that describes the kind of widget (such as "bt" for QPushButton or "cb" for QComboBox) followed by a descriptive name. Good examples are 'bt_open_file', 'cb_select_state', and 'lab_hello'. There are no standard prefixes, so use whatever makes sense to you.

You can just drag widgets onto the main window and position them, but in general you will use layouts to contain widgets.

For each of the example programs, the designer file (.ui) and the generated module are provided in addition to the source of the main script.

PyQt designer

Designer-based application workflow

- Using designer
  - Create GUI
  - Name MainWindow Hello2 (for example)
  - Save from designer as hello2.ui
- Using pyuic5 (or pyuic4)
  - generate ui_hello2 .py from hello2.ui
- Using your IDE
  - Import PyQt5 widgets
  - Subclass QMainWindow
  - Add instance of Ui_Hello2 to main window
  - Call setupUi() from Ui_Hello2
  - Instantiate main window and call show()
- Start application

Example

qt5/qt_hello2.py

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

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_hello2 import Ui_Hello2  ①

class Hello2Main(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        ②
        self.ui = Ui_Hello2()  ③
        self.ui.setupUi(self)  ④

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = Hello2Main()
    main.show()
    sys.exit(app.exec_())
```

① import generated interface

② Set up the user interface generated from Designer.

③ Attribute name does not have to be "ui"

④ Create the widgets

Naming conventions

- Keep names consistent across applications
- Use "application name" throughout
- Pay attention to case

Using consistent names for the main window object can pay off in several ways. First, you'll be less confused. Second, you can write scripts or IDE macros to generate Python code from the designer output. Third, you can create standard templates which contain the boilerplate PyQt code to set up and run the GUI.

If application name is **Spam**:

- set QMainWindow object name to **Spam**
- set windowTitle to **Spam** (or as desired)
- Save designer file as **spam.ui**
- Redirect output of `pyuic5 spam.ui` to **ui_spam.py**
- In main program, use this import
  - `from ui_spam import Ui_Spam`

The file and object names are not required to be the same, or even similar. You can name any of these anything you like, but consistency can really help simplify code maintenance.

In **EXAMPLES/qt5** there is a file called **qt5_template.py** that you can copy, replacing AppName or appname with your app's name. This contains all of the required code for a normal Qt5 app.

NOTE

If you are using PyCharm, you can add a *live template* via **Settings(Preferences on Macs)→Editor→File and Code Templates**. Copy and paste the contents of the file **EXAMPLES/qt5/qt5_template_pycharm.py** into a live template. Be sure to set the extension to "py". The template will now ask for the app name, and insert it into the appropriate places.

Common Widgets

- QLabel, QPushButton, QLineEdit, QComboBox
- There are many more

The Qt library has many widgets; some are simple, and some are complex. Some of the most basic are Qlabel, QPushButton, QLineEdit, and QComboBox.

QLabel is a widget with some text. QPushButton is just a clickable button. QLineEdit is a one-line entry blank. QComboBox shows a list of values, and allows a new value to be entered.

QLineEdit widgets are typically paired with QLabels. Using the property editor in designer, set buddy property to be the matching QLineEdit. This allows the accelerator (specified with &letter in the label text) of the label to place focus on the paired widget.

Table 18. Common PyQt Widgets

QLabel	**Display non-editable text, image, or video**
QLineEdit	Single line input field
QPushButton	Clickable button with text or image
QRadioButton	Selectable button; only one of a radio button group can be pushed at a time
QCheckBox	Individual selectable box
QComboBox	Dropdown list of items to select; allows new entry to be added
QSpinBox	Text box for integers with up/down arrow for incrementing/decrementing
QSlider	Line with a movable handle to control a bounded value
QMenuBar	A row of QMenu widgets displayed below the title bar
QMenu	A selectable menu (can have sub-menus)
QToolBar	Panel containing buttons with text, icons or widgets
QInputDialog	Dialog with text field plus OK and Cancel buttons
QFontDialog	Font selector dialog
QColorDialog	Color selector dialog
QFileDialog	Provides several methods for selecting files and folders for opening or saving
QTab	A tabbed window. Only one QTab is visible at a time
QStacked	Stacked windows, similar to QTab
QDock	Dockable, floating, window
QStatusBar	Horizontal bar at the bottom of the main window for displaying permanent or temporary information. May contain other widgets
QList	Item-based interface for updating a list. Can be multiselectable
QScrollBar	Add scrollbars to another widget
QCalendar	Date selector

Example

qt5/qt_commonwidgets.py

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

import sys
from PyQt5.QtWidgets import QMainWindow, QApplication

from ui_commonwidgets import Ui_CommonWidgets

class CommonWidgetsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_CommonWidgets()
        self.ui.setupUi(self)

        for k,v in (('apple',1),('banana',2),('mango',3)):  ①
            self.ui.cbFruits.insertItem(v,k,v) ①

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = CommonWidgetsMain()
    main.show()
    sys.exit(app.exec_())
```

① populate the combo box

Layouts

QVBoxLayout (vertical, like pancakes) QHBoxLayout (horizontal, like books on a shelf) QGridLayout (rows and columns) QFormLayout (2 columns)

Most applications use more than one widget. To easily organize widgets into rows and columns, use layouts. There are four layout types: QVBoxLayout, QHBoxLayout, QGridLayout, and QFormLayout. Drag layouts to your Deigner canvas to create the arrangement you need; of course they may be nested.

QVBoxlayout and QHBoxlayout lay out widgets vertical or horizontally. Widgets will automatically be centered and evenly spaced.

QGridLayout lays out widgets in specified rows and columns. QFormLayout is a 2-column-wide grid, for labels and input widgets.

Layouts can be resized; widgets attached to them will grow and shrink as needed (by default – all PyQt behavior can be changed in the Property editor, or programmatically).

Example

qt5/qt_layouts.py

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

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_layouts import Ui_Layouts

class LayoutsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Layouts()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = LayoutsMain()
    main.show()
    sys.exit(app.exec_())
```

Selectable Buttons

- QRadioButton, QCheckBox
- Can be grouped

There are two kinds of selectable buttons. QRadioButtons are used in a group, where only one of the group can be checked. QCheckBoxes are individual, and can be checked or unchecked.

By default, radio buttons are auto-exclusive – all radio buttons that have the same parent are grouped. If you need more than one group of radio buttons to share a parent, use QButtonGroup to group them.

Use the isChecked() method to determine whether a button has been selected. Use the checked property to mark a button as checked.

Example

qt5/qt_selectables.py

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

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_selectables import Ui_Selectables

class SelectablesMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Selectables()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = SelectablesMain()
    main.show()
    sys.exit(app.exec_())
```

Actions and Events

- Widgets have predefined actions that can be handled
- Handler is ordinary method
- Use widget.action.connect(method)
- Pass in function object; don't call function
- Action is clicked, triggered, etc.
- Event names are predefined; override in main class

An event is something that changes in a GUI app, such as a mouse click, mouse movement, key press (or release), etc. To do something when an event occurs, you associate a function with an *action*, which represents the event. Actions are verbs that end with -ed, such as clicked, connected, triggered, etc.

To add an action to a widget, use the connect() method of the appropriate action, which is an attribute of the widget. For instance, if you have a QPushButton object pb, you can set the action with pb.clicked.connect(method).

Other events can be handled by using implementing predfined methods, such as keyPressEvent. Event handlers are passed the event object, which has more detail about the event, such as the key pressed, the mouse position, or the number of mouse clicks.

To get the widget that generated the event (AKA the sender), use self.sender() in the handler function.

If the action needs parameters, the simplest thing to do is use a lambda function as the handler (AKA slot), which can then call some other function with the desired parameters. NOTE: Actions and events are also called "signals" and "slots" in Qt.

Example

qt5/qt_events.py

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

import sys
import types

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_events import Ui_Events

def fprint(*args):
    """ print and flush the output buffer so text
        shows up immediately
    """
    print(*args)
    sys.stdout.flush()

class EventsMain(QMainWindow):
    FRUITS = dict(A='Apple', B='Banana', C='Cherry', D='Date')

    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Events()
        self.ui.setupUi(self)

        # set the File->Quit handler
        self.ui.actionQuit.triggered.connect(self.close) ①

        # set the Edit->Clear Name Field handler
        self.ui.actionClear_name_field.triggered.connect(self._clear_field)

        # use the same handler for all 4 buttons
        self.ui.pb_A.clicked.connect(self._mkfunc('red',self.ui.pb_A))  ②
        self.ui.pb_B.clicked.connect(self._mkfunc('blue',self.ui.pb_B))
        self.ui.pb_C.clicked.connect(self._mkfunc('yellow',self.ui.pb_C))
        self.ui.pb_D.clicked.connect(self._mkfunc('purple',self.ui.pb_D))

        self.setup_mouse_move_event_handler()

        self.ui.checkBox.toggled.connect(self._toggled)
        self.ui.checkBox.clicked.connect(self._clicked)
```

```
    def setup_mouse_move_event_handler(self):
        def mme(self, mouse_ev):
            self.ui.statusbar.showMessage("Motion: {0},{1}".format(   ③
                mouse_ev.x(), mouse_ev.y(), 0)  # 2nd param is timeout
            )
        # add method instance to label dynamically
        self.ui.label.mouseMoveEvent = types.MethodType(mme, self)

    def keyPressEvent(self, key_ev):
        """ Generated on keypresses """
        key_code = key_ev.key()   ④
        char = chr(key_code) if key_code < 128 else 'Special'  ⑤
        fprint("Key press: {0} ({1})".format(key_code, char))

    def mousePressEvent(self, mouse_ev):    ⑥
        """ generated when mouse button is pressed """
        fprint("Press:", mouse_ev.x(), mouse_ev.y())


    def mouseReleaseEvent(self, mouse_ev):
        fprint("Release:", mouse_ev.x(), mouse_ev.y())

    def _toggled(self, mouse_ev):   ⑦
        fprint("Toggle")

    def _clicked(self, mouse_ev):
        fprint("Click")

    def _checked(self, mouse_ev):
        fprint("Toggle")

    def _pushed(self):
        sender = self.sender()
        button_text = str(sender.text())
        if button_text in EventsMain.FRUITS:
            sender.setText(EventsMain.FRUITS[button_text])

    def _mkfunc(self, color, widget):  ⑧
        def pushed(stuff):
            button_text = str(widget.text())
            fprint("HI I AM BUTTON {0} and I AM {1}".format(button_text, color))
            if button_text in EventsMain.FRUITS:
                widget.setText(EventsMain.FRUITS[button_text])
```

```
        return pushed  ⑨

    def _clear_field(self):
        self.ui.leName.setText('')   ⑩

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = EventsMain()
    main.show()
    sys.exit(app.exec_())
```

① Add an event handler callback for when **Quit** is selected from the the FIle menu

② Add a handler for button A

③ Update status bar

④ Get the key that was pressed

⑤ See if it's a "normal" key

⑥ Overload mouse press event

⑦ Handler when check box is toggled

⑧ Function factory to make button click event handlers

⑨ Function factory returns....a function

⑩ Update text on the LineEntry

Signal/Slot Editor

- Event manager
  - Signal is event
  - Slot is handler
- Two ways to edit
  - Signal/Slot mode
  - Signal/Slot editor

The designer has an editor for connection signals (events) to builtin slots (handlers). You can select the widget that generates the event (emits the signal), and select which signal you want to handle. Then you can select the widget to receive the signal, and finally, select the method (on the receiving widget) to handle the event.

This is very handy for tying, for example, actionQuit.triggered to MainWindow.close()

It can't be used for custom handlers. Do that in your main script.

Editing modes

- Widgets
- Signals/Slots
- Tab Order
- Buddies

By default **designer** is in *widget editing* mode, which allows dragging new widgets onto the window and arranging them. There are three other modes.

Signal/slot editing mode lets you drag and drop to connect signals (events generated by widgets) to slots (error handling methods). You can also use the separate signal/slot editor to do this.

Tab Order editing mode lets you set the tab order of the widgets in your interface. To do this, click on the widgets in the preferred order. You can right-click on widgets for other options. Tab order is the order in which the **Tab** key will traverse the widgets.

Buddies lets you pair labels with input widgets. Accelerator keys for the labels will jump to the paired input widgets.

Menu Bar

- Add menus and sub-menus to menu bar
- Add actions to sub-menus

In the Designer, you can just start typing on the menus in the menu bar to add menu items, separators, and sub-menus. To make the menus do something, see the examples in the previous topics. Note this section of code:

```
self.ui.actionQuit.triggered.connect(lambda:self.close())
       self.ui.actionClear_name_field.triggered.connect(self._clear_field)
```

This is how to attach a callback function to a menu item. By default, the designer will name menu choices based on the text in the menu. You can name the menu items anything, however.

TIP You can also use the Signal/Slot editor in the Designer to handle menu events (signals) using builtin handlers (slots).

Status Bar

- Displays at bottom of main window

A status bar is a row at the bottom of the main window which can be used for text messages. A default status bar is automatically part of a GUI based on QmainWindow. It is named "statusbar".

To put a message on the status bar, use the showMessage() method of the status bar object. The first parameter is a string containing the message; the second parameter is the timeout in milliseconds. A timeout of 0 means display until overwritten. To clear the message, use either removeMessage(), or display an empty string.

Example

qt5/qt_statusbar.py

```
#!/usr/bin/env python
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_statusbar import Ui_StatusBar

class StatusBarMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self._count = 0

        # Setup the user interface from Designer.
        self.ui = Ui_StatusBar()
        self.ui.setupUi(self)

        self.ui.btPushMe.clicked.connect(self._pushed)
        self._update_statusbar() ①

    def _pushed(self, ev):
        self._update_statusbar()

    def _update_statusbar(self):
        self._count += 1
        msg = "Count is " + str(self._count)
        self.ui.statusbar.showMessage(msg, 0) ②


if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = StatusBarMain()
    main.show()
    sys.exit(app.exec_())
```

① do initial status bar update

② show message, 0 means no timeout, >= 0 means timeout in seconds

Forms and validation

- Use form layout
- Validate input
- Use form data

PyQt makes it easy to create fill-in forms. Use a form layout (QFormLayout) to create a two-column form. Labels go in the left column and an entry widget goes in the right column. The entry widget can be any of a variety of widget types.

The Line Edit (QLineEdit) widget is typically used for a single-line text entry. You can add validators to this widget to accept or reject user input.

THere are three kinds of validators — QRegExpValidator, QIntValidator, and QDoubleValidator. To use them, create an instance of the validator, and attach it to the Line Edit widget with the `setValidator()` method.

TIP for QRexExpValidator, you need to create a QRexExp object to pass to it.

Example

qt5/qt_validators.py

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

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5.QtGui import QRegExpValidator, QIntValidator
from PyQt5.QtGui import QDoubleValidator
from PyQt5.QtCore import QRegExp

from ui_validators import Ui_Validators

class ValidatorsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        # Set up the user interface from Designer.
        self.ui = Ui_Validators()
```

```
        self.ui.setupUi(self)
        self._set_validators()

        self.ui.bt_save.clicked.connect(self._save_pushed)

    def _set_validators(self):
        # Set up the validators (could be in separate function or module)
        reg_ex = QRegExp(r"[A-Za-z0-9]{1,10}") ①
        val_alphanum = QRegExpValidator(reg_ex, self.ui.le_alphanum)  ②
        self.ui.le_alphanum.setValidator(val_alphanum) ③

        reg_ex = QRegExp(r"[a-z ]{0,30}") ①
        val_lcspace = QRegExpValidator(reg_ex, self.ui.le_lcspace)  ②
        self.ui.le_lcspace.setValidator(val_lcspace) ③

        val_nums_1_100 = QIntValidator(1, 100, self.ui.le_nums_1_100)  ④
        self.ui.le_nums_1_100.setValidator(val_nums_1_100)  ⑤

        val_float = QDoubleValidator(0.0, 20.0, 2, self.ui.le_float)  ⑥
        self.ui.le_float.setValidator(val_float)  ⑦

    def _save_pushed(self):
        alphanum = self.ui.le_alphanum.text()
        lcspace = self.ui.le_lcspace.text()
        nums = self.ui.le_nums_1_100.text()
        fl = self.ui.le_float.text()
        msg = '{}/{}/{}/{}'.format(alphanum, lcspace, nums, fl)
        self.ui.statusbar.showMessage(msg, 0) ⑧

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = ValidatorsMain()
    main.show()
    sys.exit(app.exec_())
```

① create Qt regular expression object (note use of raw string)

② create regex validator from regex object

③ attach validator to line entry field

④ create integer validator

⑤ attach validator to line entry field

⑥ create double (large float) validator

⑦ attach validator to line entry field

⑧ display valid date on status bar

Using Predefined Dialogs

- Predefined dialogs for common tasks
- Files, Messages, Colors, Fonts, Printing
- Use static methods for convenience.

PyQt defines several standard dialogs for common GUI tasks. The following chart lists them. Some of the standard dialogs can be invoked directly; however, most also provide some convenient static methods that provide more fine-grained control. These static methods return an appropriate value, typically a user selection.

Example

qt5/qt_standard_dialogs.py

```
#!/usr/bin/env python
import sys
import os

from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog, QColorDialog,
QErrorMessage, QInputDialog
from PyQt5.QtGui import QColor

from ui_standard_dialogs import Ui_StandardDialogs

class StandardDialogsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_StandardDialogs()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        # Connect up the buttons.
        self.ui.btFile.clicked.connect(self._choose_file)  ①
        self.ui.btColor.clicked.connect(self._choose_color)
        self.ui.btMessage.clicked.connect(self._show_error)
```

```
        self.ui.btInput.clicked.connect(self._get_input)
       # self.ui.BUTTON_NAME.clicked.connect(self._pushed)

    def _choose_file(self):
        full_path, _ = QFileDialog.getOpenFileName(self, 'Open file', os.getcwd())
②
        file_name = os.path.basename(full_path)
        self.ui.statusbar.showMessage("You chose: " + file_name) ③

    def _choose_color(self):
        result = QColorDialog.getColor() ④
        self.ui.statusbar.showMessage(
            "You chose #{0:02x}{1:02x}{2:02x} ({0},{1},{2})".format(
                result.red(),  ⑤
                result.green(),
                result.blue()
            )
        )


    def _show_error(self):
        em = QErrorMessage(self)  ⑥
        em.showMessage("There is a problem")
        self.ui.statusbar.showMessage('Diplaying Error')

    def _get_input(self):
        text, ok = QInputDialog.getText(self, 'Input Dialog',
            'Enter your name:')  ⑦
        if ok:
            self.ui.statusbar.showMessage("Your name is " + text)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = StandardDialogsMain()
    main.show()
    sys.exit(app.exec_())
```

① setup buttons to invoke builtin dialogs

② invoke open-file dialog (starts in current directory); returns tuple of selected file path and an empty string

③ update statusbar with chosen filename

④ invoke color selector dialog and return result

⑤ result has methods to retrieve color values

⑥ invoke error message dialog with specified message

⑦ invoke input dialog with prompt; returns entered text and boolean flag — True if user pressed OK, False if user pressed Cancel

Table 19. Standard Dialogs (with Convenience Methods)

Dialog	Description
QColorDialog.getColor	Select a color
QErrorMessage.showMessage	Display an error messages
QFileDialog.getExistingDirectory QFileDialog.getOpenFileName QFileDialog.getOpenFileNameAndFilter QFileDialog.getOpenFileNames QFileDialog.getSaveFileName QFileDialog.getSaveFileNameAndFilter	Select files or folders
QFontDialog	Select a font
QInputDialog.getText QInputDialog.getInteger QInputDialog.getDouble QInputDialog.getItem	Get input from user
QMessageBox	Display a modal message
QPageSetupDialog	Select page-related options for printer
QPrintPreviewDialog	Display print preview
QProgressDialog	Display a progress windows
QWizard	Guide user through step-by-step process

Tabs

- Use a QTab Widget
- In designer under "Containers"/a
- Each tab can have a name

A QTab Widget contains one or more tabs, each of which can contain a widget, either a single widget or some kind of container.

As usual, tabs can be created in the designer. You should give each tab a unique name, so that in your main code you can access them programmatically.

Drag a Tab Widget to your application and place it. Then you can add more tabs by right-clicking on a tab and selecting Insert Page. You can then go to the properties of the tab widget and set the properties for the currently select tab. Select other tabs and change their properties as appropriate.

The actual tabs can be on any of the 4 sides of the tab widget, and they can be left-justified, right-justified, or centered. In addition, you can modify the shape of the tabs, and of course the color and font of the labels.

Whichever tab is selected when you generate the UI file will be selected when you start your application.

NOTE The QStacked widget is similar to QTab, but can stack *any* kind of widget.

Example

qt5/qt_tabs.py

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

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_tabs import Ui_Tabs

class TabsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Tabs()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())
        self.ui.actionA.triggered.connect(lambda: self._show_tab('A'))
        self.ui.actionB.triggered.connect(lambda: self._show_tab('B'))
        self.ui.actionC.triggered.connect(lambda: self._show_tab('C'))

    def _show_tab(self, which_tab):
        if which_tab == 'A':
            self.ui.tabWidget.setCurrentIndex(0)  ①
            self.ui.labA.setText('Aardvark')  ②
        elif which_tab == 'B':
            self.ui.tabWidget.setCurrentIndex(1)  ①
            self.ui.labB.setText('Bonobo')  ②
        elif which_tab == 'C':
            self.ui.tabWidget.setCurrentIndex(2)  ①
            self.ui.labC.setText('Coatimundi')  ②

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = TabsMain()
    main.show()
    sys.exit(app.exec_())
```

① choose tab programmatically

② set text on label widget on tab

Niceties

- Styling Text
- Tooltips

Fonts can be configured via the designer. Choose any widget in the Object inspector, then search for the font property group in the Property Editor. You can change the font family, the point size, and weight and slant of the text.

You can further style a widget by specifying a string containing a valid CSS (cascading style sheet). In the CSS, the selectors are the object names. Either of the following approaches can be used:

```
widget.setStyleSheet('QPushButton {color: blue}')
widget.setStyleSheet(open(cssfile).read())
```

Tooltips can be added via the designer. Search for the toolTip property and type in the desired text.

Working with Images

- Display images via QLabels
- Create a QPixMap of the image
- Assign pixmap to label

To display an image, create a Pixmap object from the graphics file. Then assign the pixmap to a QLabel. The image may be scaled or resized.

NOTE The images in the example program are scaled to fit the original label. This is why they are distorted.

Example

qt5/qt_images.py

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

from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5.QtGui import QPixmap
from ui_images import Ui_Images

class ImagesMain(QMainWindow):

    def __init__(self):
        QMainWindow.__init__(self)

        # Set up the user interface from Designer.
        self.ui = Ui_Images()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        self.ui.actionPictureA.triggered.connect(
            lambda: self._show_picture('A')
        )
        self.ui.actionPictureB.triggered.connect(
            lambda: self._show_picture('B')
        )
        self.ui.actionPictureC.triggered.connect(
            lambda: self._show_picture('C')
        )

    def _show_picture(self, which_picture):
        if which_picture == 'A':
            image_file = 'apple.png'  ①
            label = self.ui.labA  ②
        elif which_picture == 'B':
            image_file = 'banana.jpg'
            label = self.ui.labB
        elif which_picture == 'C':
            image_file = 'cherry.jpg'
            label = self.ui.labC

        img = QPixmap('../../DATA/' + image_file)  ③
```

```
        label.setPixmap(img)  ④
        label.setScaledContents(True)  ⑤

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = ImagesMain()
    main.show()
    sys.exit(app.exec_())
```

① select image

② select label for image

③ create QPixMap from image

④ assign pixmap to label

⑤ scale picture

Complete Example

This is an application that lets the user load a file of words, then shows all words that match a specified regular expression

Example

qt5/qt_wordfinder.py

```
#!/usr/bin/env python
import sys
import re

from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog
from ui_wordfinder import Ui_WordFinder

class WordFinderMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)


        # Set up the user interface from Designer.
        self.ui = Ui_WordFinder()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        self.ui.actionLoad.triggered.connect(self._load_file)

        self.ui.lePattern.returnPressed.connect(self._search)

        # the following might be too time-consuming for large files
        # self.ui.lePattern.textChanged.connect(self._search)

        self.ui.btSearch.clicked.connect(self._search)

    def _load_file(self):
        file_name, _ = QFileDialog.getOpenFileName(
            self, 'Open file for matching', '.'
        )
        if file_name:
```

```
            with open(file_name) as F:
                self._words = [ line.rstrip() for line in F ]

            self._numwords = len(self._words)
            self.ui.teText.clear()

            self.ui.teText.insertPlainText(
                '\n'.join(self._words))

    def _search(self):
        pattern = str(self.ui.lePattern.text())
        if pattern == '':
            pattern = '.'
        rx = re.compile(pattern)

        self.ui.teText.clear()
        self.ui.lePattern.setEnabled(False)
        # self.lePattern.setVisible(False)
        count = 0
        for word in self._words:
            if rx.search(word):
                self.ui.teText.insertPlainText(word + '\n')
                count += 1
        self.ui.lePattern.setEnabled(True)
        #self.ui.lePattern.setVisible(True)
        self.ui.statusbar.showMessage(
            "Matched {0} out of {1} words".format(count,self._numwords),
            0
        )

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = WordFinderMain()
    main.show()
    sys.exit(app.exec_())
```

Chapter 12 Exercises

Exercise 12-1 (gpresinfo.py, ui_gpresinfo.py, gpresinfo.ui)

Using the Qt designer, write a GUI application to display data from the **presidentsqlite** or **presidentmysql** module. It should have at least the following components:

- A text field (use Entry) for entering the term number
- A Search button for retrieving data
- An Exit button

A widget or widgets to display the president's information – you could use a Text widget, multiple Labels, or whatever suits your fancy.

Be creative.

For the ambitious Provide a combo box of all available presidents rather than making the user type in the name manually.

For the even more ambitious Implement a search function – let a user type text in a blank, and return a list of presidents which matches the text in either the first name or last name field. E.g., "jeff" would retrieve "Jefferson, Thomas", and "John" would retrieve "Adams, John", as well as "Kennedy, John", "Johnson, Lyndon" and so forth.

Chapter 13: Network Programming

Objectives

- Download web pages or file from the Internet
- Consume web services
- Send e-mail using a mail server
- Learn why requests is the best HTTP client

Grabbing a web page

- import urlopen from urllib.request
- urlopen() similar to open()
- Iterate through (or read from) response
- Use info() method for metadata

The standard library module **urllib.request** includes **urlopen()** for reading data from web pages. urlopen() returns a file-like object. You can iterate over lines of HTML, or read all of the contents with read().

The URL is opened in binary mode ; you can download any kind of file which a URL represents – PDF, MP3, JPG, and so forth – by using read().

NOTE When downloading HTML or other text, a bytes object is returned; use decode() to convert it to a string.

Example

read_html_urllib.py

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

import urllib.request

u = urllib.request.urlopen("https://www.python.org")

print(u.info())  ①
print()

print(u.read(500).decode())   ②
```

① .info() returns a dictionary of HTTP headers

② The text is returned as a bytes object, so it needs to be decoded to a string

read_html_urllib.py

```
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
x-xss-protection: 1; mode=block
X-Clacks-Overhead: GNU Terry Pratchett
Via: 1.1 varnish
Content-Length: 48942
Accept-Ranges: bytes
Date: Mon, 26 Mar 2018 20:38:19 GMT
Via: 1.1 varnish
Age: 2721
Connection: close
X-Served-By: cache-iad2140-IAD, cache-dca17721-DCA
X-Cache: HIT, HIT
X-Cache-Hits: 6, 6
X-Timer: S1522096700.692907,VS0,VE0
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains



<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]-->
<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">                 <![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr">  <!--<![endif]-->

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <link rel="prefetch" href="//ajax.googleapis.com/ajax/libs/jqu
```

Example

read_pdf_urllib.py

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

import sys
import os
from urllib.request import urlopen
from urllib.error import HTTPError

# url to download a PDF file of a NASA ISS brochure

url =
'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure%202012%20Screenres
%203-8-13.pdf'  ①

saved_pdf_file = 'nasa_iss.pdf'  ②

try:
    URL = urlopen(url)  ③
except HTTPError as e:  ④
    print("Unable to open URL:",e)
    sys.exit(1)

pdf_contents = URL.read()  ⑤
URL.close()

with open(saved_pdf_file,'wb') as pdf_in:
    pdf_in.write(pdf_contents)  ⑥


if sys.platform == 'win32': ⑦
    cmd = saved_pdf_file
elif sys.platform == 'darwin':
    cmd = 'open ' + saved_pdf_file
else:
    cmd = 'acroread ' + saved_pdf_file

os.system(cmd)   ⑧
```

① target URL

② name of PDF file for saving

③ open the URL

④ catch any HTTP errors

⑤ read all data from URL in binary mode

⑥ write data to a local file in binary mode

⑦ select platform and choose the app to open the PDF file

⑧ launch the app

Consuming Web services

- Use urllib.parse to URL encode the query.
- Use urllib.request.Request
- Specify data type in header
- Open URL with urlopen Read data and parse as needed

To consume Web services, use the urllib.request module from the standard library. Create a urllib.request.Request object, and specify the desired data type for the service to return.

If needed, add a headers parameter to the request. Its value should be a dictionary of HTTP header names and values.

For URL encoding the query, use urllib.parse.urlencode(). It takes either a dictionary or an iterable of key/value pairs, and returns a single string in the format "K1=V1&K2=V2&..." suitable for appending to a URL.

Pass the Request object to urlopen(), and it will return a file-like object which you can read by calling its read() method.

The data will be a bytes object, so to use it as a string, call decode() on the data. It can then be parsed as appropriate, depending on the content type.

NOTE the example program on the next page queries the Liquor Control Board of Ontario. It expects a search term on the command line.

TIP List of public RESTful API: http://www.programmableweb.com/apis/directory/1?protocol=REST

Example

web_content_consumer_urllib.py

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

import sys

import urllib.request, urllib.parse, urllib.error
import urllib.request, urllib.error, urllib.parse
import json

DATA_TYPE = 'application/json'

BASE_URL = 'http://lcboapi.com/products' ①
AUTH_TOKEN = 'CJAssociatesTraining'
AUTH_KEY=
'MDowYzMxMTg5Mi0yMzA5LTExZTUtODcxMC0wNzEwNDcxM2NkOTA6QVBxNklDQXU1M2RSNEkyUjBBOEpkZVN
QQVJUYXY2Q3liSzBy'

def main(args):

    if len(args) < 1:
        print("Please specify a search term")
        sys.exit(1)

    search_term = args[0]

    setup_auth()

    params = make_params(search_term)

    url = BASE_URL + '?' + params ②

    do_query(url)


def do_query(url):
    response = urllib.request.urlopen(url) ③
    json_string = response.read().decode() ④

    raw_data = json.loads(json_string) ⑤
    # print('RAW JSON:', raw_data, '\n\n')
    if raw_data['result']:  ⑥
```

```
        for result in raw_data['result']:
            print("PRODUCT NUMBER:", result['product_no'])
            print("NAME:", result['name'])
            print("PACKAGE:", result['package'])
            print("PRICE:
${:5.2f}/liter".format(result['price_per_liter_in_cents']/100))
            print()
    else:
        print("Sorry, no items matched your query.")


def make_params(term):
    query_terms = { 'q': term }
    return urllib.parse.urlencode(query_terms) ⑦


def setup_auth():
    password_mgr = urllib.request.HTTPPasswordMgrWithDefaultRealm() ⑧
    password_mgr.add_password(None, BASE_URL, AUTH_TOKEN, AUTH_KEY) ⑨
    auth_handler = urllib.request.HTTPBasicAuthHandler(password_mgr) ⑩
    opener = urllib.request.build_opener(auth_handler) ⑪
    urllib.request.install_opener(opener) ⑫


if __name__ == '__main__':
    main(sys.argv[1:])
```

① base URL of resource site

② build search URL

③ send HTTP request and get HTTP response

④ read content from web site and decode() from bytes to str

⑤ convert JSON string to Python data structure

⑥ check for results

⑦ create URL-encode string from query term(s)

⑧ create password manager

⑨ add credentials to password manager

⑩ create authentication handler

⑪ create HTTP opener from password manager

⑫ install HTTP opener to urllib.request

web_content_consumer_urllib.py dewars

```
PRODUCT NUMBER: 217992
NAME: Dewar's White Label
PACKAGE: 1140 mL bottle
PRICE: $35.17/liter

PRODUCT NUMBER: 11130
NAME: Dewar's White Label Scotch Whisky
PACKAGE: 750 mL bottle
PRICE: $35.60/liter

PRODUCT NUMBER: 438598
NAME: Dewar's 12 Year Old
PACKAGE: 750 mL bottle
PRICE: $63.00/liter

PRODUCT NUMBER: 214154
NAME: Dewar's Signature
PACKAGE: 750 mL bottle
PRICE: $364.93/liter
```

HTTP the easy way

- Use the **requests** module
- Pythonic front end to urllib, urllib2, httplib, etc
- Makes HTTP transactions simple

While **urllib** and friends are powerful libraries, their interfaces are complex for non-trivial tasks. You have to do a lot of work if you want to provide authentication, proxies, headers, or data, among other things.

The **requests** module is a much easier to use HTTP client module. It is included with Anaconda, or is readily available from PyPI via **pip**.

requests implements GET, POST, PUT, and other HTTP verbs, and takes care of all the protocol housekeeping needed to send data on the URL, to send a username/password, and to retrieve data in various formats.

Example

read_html_requests.py

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

import requests

response = requests.get("https://www.python.org")  ①

for header, value in sorted(response.headers.items()): ②
    print(header, value)
print()

print(response.content[:200].decode())   ③
print('...')
print(response.content[-200:].decode())   ④
```

① requests.get() returns HTTP response object

② response.headers is a dictionary of the headers

③ The text is returned as a bytes object, so it needs to be decoded to a string; print the first 200 bytes

④ print the last 200 bytes

Example

read_pdf_requests.py

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

import sys
import os

import requests

url =
'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure%202012%20Screenres
%203-8-13.pdf' ①
saved_pdf_file = 'nasa_iss.pdf' ②

response = requests.get(url) ③
if response.status_code == requests.codes.OK: ④

    with open(saved_pdf_file,'wb') as pdf_in:  ⑤
        pdf_in.write(response.content) ⑥

    if sys.platform == 'win32': ⑦
        cmd = saved_pdf_file
    elif sys.platform == 'darwin':
        cmd = 'open ' + saved_pdf_file
    else:
        cmd = 'acroread ' + saved_pdf_file

    os.system(cmd) ⑧
```

① target URL

② name of PDF file for saving

③ open the URL

④ check status code

⑤ open local file

⑥ write data to a local file in binary mode; response.content is data from URL

⑦ select platform and choose the app to open the PDF file

⑧ launch the app

Example

web_content_consumer_requests.py

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

from pprint import pprint
import sys
import requests

BASE_URL = 'http://lcboapi.com/products'  ①
AUTH_TOKEN = 'CJAssociatesTraining' ②
AUTH_KEY=
'MDowYzMxMTg5Mi0yMzA5LTExZTUtODcxMC0wNzEwNDcxM2NkOTA6QVBxNklDQXU1M2RSNEkyUjBBOEpkZVN
QQVJUYXY2Q3liSzBy'

def main(args):
    if len(args) < 1:
        print("Please specify a search term")
        sys.exit(1)

    response = requests.get(BASE_URL, params={ 'q': args[0] }, auth=(AUTH_TOKEN,
AUTH_KEY) ) ③

    if response.status_code == requests.codes.OK:
        raw_data = response.json() ④
        if raw_data['result']: ⑤
            for result in raw_data['result']:
                print("PRODUCT NUMBER:", result['product_no'])
                print("NAME:", result['name'])
                print("PACKAGE:", result['package'])
                print("PRICE:
${:5.2f}/liter".format(result['price_per_liter_in_cents']/100))
                print()
        else:
            print("Sorry, no items matched your query.")
    else:
        print("Sorry, HTTP response", response.status_code)

if __name__ == '__main__':
    main(sys.argv[1:])
```

① base URL of resource site

② credentials

③ send HTTP request and get HTTP response

④ convert JSON content to Python data structure

⑤ check for results

web_content_consumer_requests.py "maker's mark"

```
PRODUCT NUMBER: 103747
NAME: Maker's Mark Kentucky Bourbon
PACKAGE: 750 mL bottle
PRICE: $53.26/liter

PRODUCT NUMBER: 269563
NAME: Maker's Mark Kentucky Straight Bourbon
PACKAGE: 375 mL bottle
PRICE: $66.40/liter

PRODUCT NUMBER: 225565
NAME: Maker's Mark 46
PACKAGE: 750 mL bottle
PRICE: $86.60/liter
```

TIP See details of requests API at http://docs.python-requests.org/en/v3.0.0/api/#main-interface

Table 20. Keyword Parameters for **requests** *methods*

Option	Data Type	Description
allow_redirects	bool	set to True if PUT/POST/DELETE redirect following is allowed
auth	tuple	authentication pair (user/token,password/key)
cert	str or tuple	path to cert file or ('cert', 'key') tuple
cookies	dict or CookieJar	cookies to send with request
data	dict	parameters for a POST or PUT request
files	dict	files for multipart upload
headers	dict	HTTP headers
json	str	JSON data to send in request body
params	dict	parameters for a GET request
proxies	dict	map protocol to proxy URL
stream	bool	if False, immediately download content
timeout	float or tuple	timeout in seconds or (connect timeout, read timeout) tuple
verify	bool	if True, then verify SSL cert

sending e-mail

- import smtplib module
- Create an SMTP object specifying server
- Call sendmail() method from SMTP object

You can send e-mail messages from Python using the smtplib module. All you really need is one smtplib object, and one method – sendmail().

Create the smtplib object, then call the sendmail() method with the sender, recipient(s), and the message body (including any headers).

The recipients list should be a list or tuple, or could be a plain string containing a single recipient.

Example

email_simple.py

```
#!/usr/bin/env python
import smtplib ①

DEBUG = True # set to false for production

smtp_user = 'jstrickpython'
smtp_pwd = 'python(monty)'

sender = 'jstrick@mindspring.com'
recipients = ['jstrickler@gmail.com']
msg = '''Subject: SMTP example
Hello hello?
Testing email from Python
'''

smtpserver = smtplib.SMTP("smtpcorp.com", 2525) ②
smtpserver.login(smtp_user, smtp_pwd)  ③
smtpserver.set_debuglevel(DEBUG) ④

try:
    smtpserver.sendmail(
        sender,
        recipients,
        msg
    )   ⑤
except Exception as e:
    print("Unable to send mail:", e)
finally:
    smtpserver.quit()   ⑥
```

① module for sending email

② connect to SMTP server

③ log into SMTP server

④ turn on debugging to show exchange with SMTP server

⑤ send the message

⑥ disconnect from SMTP server

Email attachments

- Create MIME multipart message
- Create MIME objects
- Attach MIME objects
- Serialize message and send

To send attachments, you need to create a MIME multipart message, then create MIME objects for each of the attachments, and attach them to the main message. This is done with various classes provided by the **email.mime** module.

These modules include **multipart** for the main message, **text** for text attachments, **image** for image attachments, **audio** for audio files, and **application** for miscellaneous binary data.

One the attachments are created and attached, the message must be serialized with the **as_string()** method. The actual transport uses **smptlib**, just like simple email messages described earlier.

Example

email_attach.py

```
#!/usr/bin/env python
import smtplib
import os
from email.mime.multipart import MIMEMultipart  ①
from email.mime.text import MIMEText ②
from email.mime.image import MIMEImage ③

SMTP_SERVER = "smtpcorp.com"
SMTP_PORT = 2525
SMTP_USER = 'jstrickpython'
SMTP_PWD = 'python(monty)'

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']

def main():   ④
    smtp_server = create_smtp_server()
    msg = create_message(
        'Here is your attachment',
        'Testing email attachments from python class.',
    )
    add_text_attachment('../DATA/parrot.txt', msg)
    add_image_attachment('../DATA/felix_auto.jpeg', msg)
    send_message(smtp_server, msg)


def create_message(subject, body):
    msg = MIMEMultipart(body)  ⑤
    msg['Subject'] = subject   ⑥

    return msg

def add_text_attachment(file_name, message): ⑦
    add_attachment(file_name, message, MIMEText, 'r')

def add_image_attachment(file_name, message): ⑧
    add_attachment(file_name, message, MIMEImage, 'rb')

def add_attachment(file_name, message, mime_type, file_mode):
    with open(file_name, file_mode) as file_in:  ⑨
```

```
        attachment_data = file_in.read()

    short_name = os.path.basename(file_name)
    attachment = mime_type(attachment_data)  ⑩
    attachment.add_header(
        'Content-Disposition', 'attachment', filename=short_name
    )

    message.attach(attachment)   ⑪

def create_smtp_server():
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT) ⑫
    smtpserver.login(SMTP_USER, SMTP_PWD)

    return smtpserver

def send_message(server, message):
    try:
        server.sendmail(
            SENDER,
            RECIPIENTS,
            message.as_string() ⑬
        )
    finally:
        server.quit()

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

① class for message body

② class for text attachments

③ class for image attachements

④ normal Python script organization

⑤ create main message

⑥ set the subject

⑦ convenience function for attaching text

⑧ convenience function for attaching an image

⑨ read data for attachment

⑩ create MIME object of appropriate type

⑪ attach attachment to main message

⑫ connecting same as simple example

⑬ can't send MIME object; need to convert to string for sending

Remote Access

- Use paramiko (not part of standard library)
- Create ssh client
- Create transport object to use sftp

For remote access to other computers, you would usually use the ssh protocol. Python has several ways to use ssh.

The best way is to use paramiko. It is a pure-Python module for connecting to other computers using SSH. It is not part of the standard library, but is included with Anaconda.

To run commands on a remote computer, use SSHClient. Once you connect to the remote host, you can execute commands and retrieve the standard I/O of the remote program.

To avoid the "I haven't seen this host before" prompt, call **set_missing_host_key_policy()** like this:

```
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

The exec_command() method executes a command on the remote host, and returns a triple with the remote command's stdin, stdout, and stderr as file-like objects.

There is also a builtin **ssh** module, but it depends on having an external command named "ssh".

NOTE

Paramiko is used by Ansible and other sys admin tools.

Find out more about paramiko at http://www.lag.net/paramiko/
Find out more about Ansible at http://www.ansible.com/

Example

paramiko_commands.py

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

import paramiko

with paramiko.SSHClient() as ssh: ①

    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) ②

    ssh.connect('localhost', username='python', password='l0lz') ③

    stdin, stdout, stderr  = ssh.exec_command('whoami') ④
    print(stdout.read().decode()) ⑤
    print('-' * 60)

    stdin, stdout, stderr  = ssh.exec_command('ls -l') ④
    print(stdout.read().decode()) ⑤
    print('-' * 60)

    stdin, stdout, stderr = ssh.exec_command('ls -l /etc/passwd /etc/horcrux')   ④
    print("STDOUT:")
    print(stdout.read().decode())  ⑤
    print("STDERR:")
    print(stderr.read().decode())  ⑥
    print('-' * 60)
```

① create paramiko client

② ignore missing keys (this is safe)

③ connect to remote host

④ execute remote command; returns standard I/O objects

⑤ read stdout of command

⑥ read stderr of command

paramiko_commands.py

```
python

------------------------------------------------------------
total 16
drwx------+  3 python  staff    96 Apr 16  2014 Desktop
drwx------+  3 python  staff    96 Apr 16  2014 Documents
drwx------+  4 python  staff   128 Apr 16  2014 Downloads
drwx------+ 28 python  staff   896 Dec 17 19:13 Library
drwx------+  3 python  staff    96 Apr 16  2014 Movies
drwx------+  3 python  staff    96 Apr 16  2014 Music
drwx------+  3 python  staff    96 Apr 16  2014 Pictures
drwxr-xr-x+  5 python  staff   160 Apr 16  2014 Public
-rw-r--r--   1 python  staff  1436 Apr 16  2014 parrot.txt
-rw-r--r--   1 python  staff   519 Jul 27  2016 remote_processes.py
drwxr-xr-x   2 python  staff    64 Mar 26 11:11 text_files

------------------------------------------------------------
STDOUT:
-rw-r--r--  1 root  wheel  6774 Oct  2 20:29 /etc/passwd

STDERR:
ls: /etc/horcrux: No such file or directory

------------------------------------------------------------
```

Copying files with Paramiko

- Create transport
- Create SFTP client with transport

To copy files with paramiko, first create a **Transport** object. Using a **with** block will automatically close the Transport object.

From the transport object you can create an SFTPClient. Once you have this, call standard FTP/SFTP methods on that object.

Some common methods include listdir_iter(), get(), put(), mkdir(), and rmdir().

Example

paramiko_copy_files.py

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

import os
import paramiko
REMOTE_DIR = 'text_files'

with paramiko.Transport(('localhost', 22)) as transport:   ①
    transport.connect(username='python', password='l0lz')  ②
    sftp = paramiko.SFTPClient.from_transport(transport)   ③
    for item in sftp.listdir_iter():    ④
        print(item)
    print('-' * 60)

    remote_file = os.path.join(REMOTE_DIR, 'betsy.txt')  ④

    sftp.mkdir(REMOTE_DIR)      ⑤
    sftp.put('../DATA/alice.txt', remote_file)   ⑥
    sftp.get(remote_file, 'eileen.txt')   ⑦

⑧
with paramiko.SSHClient() as ssh:
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    try:
        ssh.connect('localhost', username='python', password='l0lz')
    except paramiko.SSHException as err:
        print(err)
        exit()

    stdin, stdout, stderr  = ssh.exec_command('ls -l {}'.format(REMOTE_DIR))
    print(stdout.read().decode())
    print('-' * 60)

    stdin, stdout, stderr = ssh.exec_command('rm -f
{}/betsy.txt'.format(REMOTE_DIR))
    stdin, stdout, stderr = ssh.exec_command('rmdir {}'.format(REMOTE_DIR))

    stdin, stdout, stderr = ssh.exec_command('ls -l')
    print(stdout.read().decode())
    print('-' * 60)
```

① create paramiko Transport instance

② connect to remote host

③ create SFTP client using Transport instance

④ get list of items on default (login) folder (listdir_iter() returns a generator)

⑤ create a folder on the remote host

⑥ copy a file to the remote host

⑦ copy a file from the remote host

⑧ use SSHClient to confirm operations (not needed, just for illustration)

paramiko_copy_files.py

```
drwx------   1 504      20             96 16 Apr 2014  Music
-rw-------   1 504      20              3 16 Apr 2014  .CFUserTextEncoding
-rw-r--r--   1 504      20           1436 16 Apr 2014  parrot.txt
drwx------   1 504      20             96 16 Apr 2014  Pictures
drwxr-xr-x   1 504      20             64 26 Mar 11:11 text_files
drwx------   1 504      20             96 16 Apr 2014  Desktop
drwx------   1 504      20            896 17 Dec 19:13 Library
drwxr-xr-x   1 504      20            160 16 Apr 2014  Public
drwx------   1 504      20             96 06 Feb 2015  .ssh
drwx------   1 504      20             96 16 Apr 2014  Movies
drwx------   1 504      20             96 16 Apr 2014  Documents
-rw-r--r--   1 504      20            519 27 Jul 2016  remote_processes.py
drwx------   1 504      20            128 16 Apr 2014  Downloads
-rw-------   1 504      20           1898 23 Mar 07:55 .bash_history
----------------------------------------------------------------
```

Chapter 13 Exercises

Exercise 13-1 (fetch_xkcd_requests.py, fetch_xkcd_urllib.py)

Write a script to fetch the following image from the Internet and display it. http://imgs.xkcd.com/comics/python.png

Exercise 13-2 (wiki_links_requests.py, wiki_links_urllib.py)

Write a script to count how many links are on the home page of Wikipedia. To do this, read the page into memory, then look for occurences of the string "href". (For *real* screen-scraping, you can use the Beautiful Soup module.)

You can use the string method **find()**, which can be called like S.find('text', start, stop), which finds on a slice of the string, moving forward each time the string is found.

Exercise 13-3 (send_chimp.py)

If the class conditions allow it (i.e., if you have access to the Internet, and an SMTP account), send an email to yourself with the image **chimp.bmp** (from the DATA folder) attached.

Chapter 14: Multiprogramming

Objectives

- Understand multiprogramming
- Differentiate between threads and processes
- Know when threads benefit your program
- Learn the limitations of the GIL
- Create a threaded application
- Implement a queue object
- Use the multiprocessing module
- Develop a multiprocessing application

Multiprogramming

- Parallel processing
- Three main ways to achieve it
  - threading
  - multiple processes
  - asynchronous communication
- All three supported in standard library

Computer programs spend a lot of their time doing nothing. This occurs when the CPU is waiting for the relatively slow disk subsystem, network stack, or other hardware to fetch data.

Some applications can achieve more throughput by taking advantage of this slack time by seemingly doing more than one thing at a time. With a single-core computer, this doesn't really happen; with a multicore computer, an application really can be executing different instructions at the same time. This is called multiprogramming.

The three main ways to implement multiprogramming are threading, multiprocessing, and asynchronous communication:

Threading subdivides a single process into multiple subprocesses, or threads, each of which can be performing a different task. Threading in Python is good for IO-bound applications, but does not increase the efficiency of compute-bound applications.

Multiprocessing forks (spawns) new processes to do multiple tasks. Multiprocessing is good for both CPU-bound and IO-bound applications.

Asynchronous communication uses an event loop to poll multiple I/O channels rather than waiting for one to finish. Asynch communication is good for IO-bound applications.

The standard library supports all three.

What Are Threads?

- Like processes (but lighter weight)
- Process itself is one thread
- Process can create one more more additional threads
- Similar to creating new processes with fork()

Modern operating systems (OSs) use time-sharing to manage multiple programs which appear to the user to be running simultaneously. Assuming a standard machine with only one CPU, that simultaneity is only an illusion, since only one program can run at a time, but it is a very useful illusion. Each program that is running counts as a process. The OS maintains a process table, listing all current processes. Each process will be shown as currently being in either Run state or Sleep state.

A thread is like a process. A thread might even be a process, depending on the implementation. In fact, threads are sometimes called “lightweight” processes, because threads occupy much less memory, and take less time to create, than do processes.

A process can create any number of threads. This is similar to a process calling the fork() function. The process itself is a thread, and could be considered the "main" thread.

Just as processes can be interrupted at any time, so can threads.

The Python Thread Manager

- Python uses underlying OS's threads
- Alas, the GIL – Global Interpreter Lock
- Only one thread runs at a time
- Python interpreter controls end of thread's turn
- Cannot take advantage of multiple processors

Python "piggybacks" on top of the OS's underlying threads system. A Python thread is a real OS thread. If a Python program has three threads, for instance, there will be three entries in the OS's thread list.

However, Python imposes further structure on top of the OS threads. Most importantly, there is a global interpreter lock, the famous (or infamous) GIL. It is set up to ensure that (a) only one thread runs at a time, and (b) that the ending of a thread's turn is controlled by the Python interpreter rather than the external event of the hardware timer interrupt.

The fact that the GIL allows only one thread to execute Python bytecode at a time simplifies the Python implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines. The takeaway is that Python does not currently take advantage of multi-processor hardware.

NOTE *GIL* is pronounced "jill", according to Guido__

For a thorough discussion of the GIL and its implications, see http://www.dabeaz.com/python/UnderstandingGIL.pdf.

The threading Module

- Provides basic threading services
- Also provides locks
- Three ways to use threads
  - Instantiate **Thread** with a function
  - Subclass **Thread**
  - Use pool method from **multiprocessing** module

The threading module provides basic threading services for Python programs. The usual approach is to subclass threading.Thread and provide a run() method that does the thread's work.

Threads for the impatient

- No class needed (created "behind the scenes")
- For simple applications

For many threading tasks, all you need is a run() method and maybe some arguments to pass to it.

For simple tasks, you can just create an instance of Thread, passing in positional or keyword arguments.

Example

thr_noclass.py

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

import threading
import random
import time

def doit(num): ①
    time.sleep(random.randint(1,10))
    print("Hello from thread {}".format(num))

for i in range(10):
    t = threading.Thread(target=doit,args=(i,)) ②
    t.start() ③
```

① function to launch in each thread

② create thread

③ launch thread

thr_noclass.py

```
Hello from thread 4
Hello from thread 9
Hello from thread 7
Hello from thread 3
Hello from thread 0
Hello from thread 5
Hello from thread 6
Hello from thread 1
Hello from thread 2
Hello from thread 8
```

Creating a thread class

- Subclass Thread
- *Must* call base class's *init*()
- *Must* implement run()
- Can implement helper methods

A thread class is a class that starts a thread, and performs some task. Such a class can be repeatedly instantiated, with different parameters, and then started as needed.

The class can be as elaborate as the business logic requires. There are only two rules: the class must call the base class's *init*(), and it must implement a run() method. Other than that, the run() method can do pretty much anything it wants to.

The best way to invoke the base class *init*() is to use super().

The run() method is invoked when you call the start() method on the thread object. The start() method does not take any parameters, and thus run() has no parameters as well.

Any per-thread arguments can be passed into the constructor when the thread object is created.

Example

thr_simple.py

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

import threading
import random
import time

class SimpleThread(threading.Thread):
    def __init__(self,num):
        super().__init__()  ①
        self._threadnum =num

    def run(self): ②
        time.sleep(random.randint(1,10))
        print("Hello from thread {}".format(self._threadnum))

for i in range(10):
    t = SimpleThread(i) ③
    t.start() ④

print("Done.")
```

① call base class constructor — REQUIRED

② the function that does the work in the thread

③ create the thread

④ launch the thread

thr_simple.py

```
Done.
Hello from thread 7
Hello from thread 8
Hello from thread 1
Hello from thread 4
Hello from thread 5
Hello from thread 3
Hello from thread 6
Hello from thread 0
Hello from thread 9
Hello from thread 2
```

Example

thr_socket_server.py

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

import socket
import threading

class ClientHandler(threading.Thread):
    '''A thread to handle one client request'''

    def __init__(self, client_socket, client_address):
        super(ClientHandler, self).__init__()
        self._client_socket = client_socket
        self._client_address = client_address

    def run(self): ①
        request = self._client_socket.recv(1024) ②

        reply = request.upper()[::-1]  ③

        self._client_socket.sendall(reply)  ④
        self._client_socket.close() ⑤

def setup():
    '''Initialize the server socket'''
    serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM) ⑥
    serv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) ⑦
    # serv.setblocking(0)

    serv.bind((socket.gethostname(), 7777)) ⑧
    serv.listen(5)  ⑨

    return serv

def main():
    '''Main program.'''
    serv = setup()
    while True: ⑩
        (csock, addr) = serv.accept() ⑪
        handler = ClientHandler(csock, addr)  ⑫
        handler.start() ⑬

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

① the function that the thread runs

② read from the client (bytes, not str)

③ create reply to send to client

④ send the reply to the client

⑤ close the connection

⑥ close the socket

⑦ set some socket options

⑧ bind to local host and port 7777

⑨ start listening

⑩ loop until program is killed

⑪ sleep until client request arrives

⑫ create new thread to handle client

⑬ start the new thread

Variable sharing

- Variables declared *before thread starts* are shared
- Variables declared *after thread starts* are local
- Threads communicate via shared variables

A major difference between ordinary processes and threads how variables are shared.

Each thread has its own local variables, just as is the case for a process. However, variables that existed in the program before threads are spawned are shared by all threads. They are used for communication between the threads.

Access to global variables is controlled by locks.

Example

thr_locking.py

```
#!/usr/bin/env python
import threading①
import random
import time

WORDS = 'apple banana mango peach papaya cherry lemon watermelon fig
elderberry'.split()

MAX_SLEEP_TIME = 3
WORD_LIST = []   ②
WORD_LIST_LOCK = threading.Lock()  ③
STDOUT_LOCK = threading.Lock() ③

class SimpleThread(threading.Thread):
    def __init__(self, num, word): ④
        super().__init__() ⑤
        self._word = word
        self._num = num

    def run(self): ⑥
        time.sleep(random.randint(1,MAX_SLEEP_TIME))
        with STDOUT_LOCK: ⑦
            print("Hello from thread {} ({})".format(self._num, self._word))

        with WORD_LIST_LOCK: ⑦
            WORD_LIST.append(self._word.upper())

all_threads = [] ⑧
for i, word in enumerate(WORDS, 1):
    t = SimpleThread(i, word) ⑨
    all_threads.append(t) ⑩
    t.start() ⑪

print("All threads launched...")

for t in all_threads:
    t.join() ⑫

print(WORD_LIST)
```

① see multiprocessing.dummy.Pool for the easier way

② the threads will append words to this list

③ generic locks

④ thread constructor

⑤ be sure to call parent constructor

⑥ function invoked by each thread

⑦ acquire lock and release when finished

⑧ make list ("pool") of threads (but see Pool later in chapter)

⑨ create thread

⑩ add thread to "pool"

⑪ launch thread

⑫ wait for thread to launch

thr_locking.py

```
All threads launched...
Hello from thread 1 (apple)
Hello from thread 4 (peach)
Hello from thread 2 (banana)
Hello from thread 8 (watermelon)
Hello from thread 3 (mango)
Hello from thread 6 (cherry)
Hello from thread 7 (lemon)
Hello from thread 10 (elderberry)
Hello from thread 5 (papaya)
Hello from thread 9 (fig)
['APPLE', 'PEACH', 'BANANA', 'WATERMELON', 'MANGO', 'CHERRY', 'LEMON', 'ELDERBERRY',
'PAPAYA', 'FIG']
```

Using queues

- Queue contains a list of objects
- Sequence is FIFO
- Worker threads can pull items from the queue
- Queue structure has builtin locks

Threaded applications often have some sort of work queue data structure. When a thread becomes free, it will pick up work to do from the queue. When a thread creates a task, it will add that task to the queue.

The queue must be guarded with locks. Python provides the Queue module to take care of all the lock creation, locking and unlocking, and so on, so that you don't have to bother with it.

=== Example

thr_queue.py

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

import queue
import random
from threading import Thread,Lock as tlock
import time

NUM_ITEMS = 25000
POOL_SIZE = 100

q = queue.Queue(0) ①

shared_list = []
shlist_lock = tlock() ②
stdout_lock = tlock() ②

import random

class RandomWord(object): ③
    def __init__(self):
        with open('../DATA/words.txt') as WORDS:
            self._words = [word.rstrip('\n\r') for word in WORDS.readlines()]
```

```
        self._num_words = len(self._words)

    def __call__(self):
        return self._words[random.randrange(0,self._num_words)]

class Worker(Thread): ④

    def __init__(self,name): ⑤
        Thread.__init__(self)
        self.name = name

    def run(self): ⑥
        while True:
            try:
                s1 = q.get(block=False) ⑦
                s2 = s1.upper() + '-' + s1.upper()
                with shlist_lock: ⑧
                    shared_list.append(s2)

            except queue.Empty: ⑨
                break

⑩
random_word = RandomWord()
for i in range(NUM_ITEMS):
    w = random_word()
    q.put(w)

start_time = time.ctime()

⑪
pool = []
for i in range(POOL_SIZE):
    name = "Worker {:c}".format(i+65)
    w = Worker(name)  ⑫
    w.start() ⑬
    pool.append(w)

for t in pool:
    t.join() ⑭

end_time = time.ctime()

print(shared_list[:20])
```

```
print(start_time)
print(end_time)
```

① initialize empty queue

② create locks

③ define callable class to generate words

④ worker thread

⑤ thread constructor

⑥ function invoked by thread

⑦ get next item from thread

⑧ acquire lock, then release when done

⑨ when queue is empty, it raises Empty exception

⑩ fill the queue

⑪ populate the threadpool

⑫ add thread to pool

⑬ launch the thread

⑭ wait for thread to finish

thr_queue.py

```
['DIGAMOUS-DIGAMOUS', 'IMPERATIVES-IMPERATIVES', 'SALAMANDER-SALAMANDER', 'EXULT-
EXULT', 'EQUIVOKES-EQUIVOKES', 'DEFENESTRATED-DEFENESTRATED', 'RESETTING-RESETTING',
'CANULATE-CANULATE', 'VAUNTFUL-VAUNTFUL', 'CANULATED-CANULATED', 'MISTAKER-
MISTAKER', 'RUBBISH-RUBBISH', 'TAMARAU-TAMARAU', 'TYPHUSES-TYPHUSES', 'SYNTAGMAS-
SYNTAGMAS', 'CEDARWOOD-CEDARWOOD', 'WISPLIKE-WISPLIKE', 'CONFUSION-CONFUSION',
'HOUSECLEAN-HOUSECLEAN', 'ALPHA-ALPHA']
Mon Mar 26 16:38:45 2018
Mon Mar 26 16:38:46 2018
```

Debugging threaded Programs

- Harder than non-threaded programs
- Context changes abruptly
- Use pdb.trace
- Set breakpoint programmatically

Debugging is always tough with parallel programs, including threads programs. It's especially difficult with pre-emptive threads; those accustomed to debugging non-threads programs find it rather jarring to see sudden changes of context while single-stepping through code. Tracking down the cause of deadlocks can be very hard. (Often just getting a threads program to end properly is a challenge.)

Another problem which sometimes occurs is that if you issue a "next" command in your debugging tool, you may end up inside the internal threads code. In such cases, use a "continue" command or something like that to extricate yourself.

Unfortunately, threads debugging is even more difficult in Python, at least with the basic PDB debugger.

One cannot, for instance, simply do something like this:

```
pdb.py buggyprog.py
```

This is because the child threads will not inherit the PDB process from the main thread. You can still run PDB in the latter, but will not be able to set breakpoints in threads.

What you can do, though, is invoke PDB from within the function which is run by the thread, by calling pdb.set trace() at one or more points within the code:

```
import pdb
pdb.set_trace()
```

In essence, those become breakpoints.

For example, we could add a PDB call at the beginning of a loop:

```
import pdb
while True:
    pdb.set_trace() # app will stop here and enter debugger
    k = c.recv(1)
    if k == '':
        break
```

You then run the program as usual, NOT through PDB, but then the program suddenly moves into debugging mode on its own. At that point, you can then step through the code using the n or s commands, query the values of variables, etc.

PDB's c ("continue") command still works. Can you still use the b command to set additional breakpoints? Yes, but it might be only on a one-time basis, depending on the context.

The multiprocessing module

- Drop-in replacement for the threading module
- Doesn't suffer from GIL issues
- Provides interprocess communication
- Provides process (and thread) pooling

The multiprocessing module can be used as a replacement for threading. It uses processes rather than threads to spread out the work to be done. While the entire module doesn't use the same API as threading, the multiprocessing.Process object is a drop-in replacement for a threading.Thread object. Both use run() as the overridable method that does the work, and both use start() to launch. The syntax is the same to create a process without using a class:

```
def myfunc(filename):
      pass

p = Process(target=myfunc, args=('/tmp/info.dat', ))
```

This solves the GIL issue, but the trade-off is that it's slightly more complicated for tasks (processes) to communicate. However, the module does the heavy lifting of creating pipes to share data.

The **Manager** class provided by multiprocessing allows you to create shared variables, as well as locks for them, which work across processes.

NOTE On windows, processes must be started in the "if __name__ == __main__" block, or they will not work.

Example

multi_processing.py

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

import random
from multiprocessing import Manager, Lock, Process, Queue, freeze_support
from queue import Empty
import time
```

```
NUM_ITEMS = 25000 ①
POOL_SIZE = 100

class RandomWord(object):  ②
    def __init__(self):
        with open('../DATA/words.txt') as WORDS:
            self._words = [word.rstrip('\n\r') for word in WORDS]
        self._num_words = len(self._words)

    def __call__(self): ③
        return self._words[random.randrange(0, self._num_words)]

class Worker(Process):  ④

    def __init__(self, name, q, result_lock, result):  ⑤
        Process.__init__(self)
        self.q = q
        self.result = result
        self.result_lock = result_lock
        self.name = name

    def run(self):  ⑥
        while True:
            try:
                s1 = self.q.get(block=False)  ⑦
                s2 = s1.upper()  ⑧
                with self.result_lock:
                    self.result.append(s2)  ⑨

            except Empty:  ⑩
                break

if __name__ == '__main__':
    q = Queue()  ⑪

    manager = Manager()  ⑫
    shared_result = manager.list()  ⑬
    result_lock = Lock()  ⑭

    random_word = RandomWord() ⑮
    for i in range(NUM_ITEMS):
        w = random_word()
        q.put(w)  ⑯
```

```
    start_time = time.ctime()

    pool = [] ⑰
    for i in range(POOL_SIZE): ⑱
        name = "Worker {:03d}".format(i)
        w = Worker(name, q, result_lock, shared_result)  ⑲
        #
        w.start() ⑳
        pool.append(w)

    for t in pool:
        t.join()

    end_time = time.ctime()

    print((shared_result[-50:]))  # print last 50 entries in shared result
    print(len(shared_result))
    print(start_time)
    print(end_time)
```

① set some constants

② callable class to provide random words

③ will be called when you call an instance of the class

④ worker class — inherits from Process

⑤ initialize worker process

⑥ do some work — will be called when process starts

⑦ get data from the queue

⑧ modify data

⑨ add to shared result

⑩ quit when there is no more data in the queue

⑪ create empty Queue object

⑫ create manager for shared data

⑬ create list-like object to be shared across all processes

⑭ create locks

⑮ create callable RandomWord instance

⑯ fill the queue

⑰ create empty list to hold processes

⑱ populate the process pool

⑲ create worker process

⑳ actually start the process — note: in Windows, should only call X.start() from main(), and may not work inside an IDE

add process to pool

wait for each queue to finish

multi_processing.py

```
['DEHYDROGENATIONS', 'DISORDERLINESS', 'UNDERWOOD', 'PROROGATES', 'PROCESSABLE',
'PRESENCE', 'VIOLATIONS', 'BOOTBLACKS', 'HYDROMORPHIC', 'REMIX', 'UNPLAITED',
'BANKINGS', 'PILAF', 'TANNING', 'CIMBALOM', 'HOOKA', 'TOROT', 'CITRONELLALS',
'DICKIE', 'REAFFORESTS', 'UPHEAVING', 'GRATULATE', 'TRUST', 'COMMENSURABILITY',
'COGENT', 'STEELWORKS', 'LABDANUMS', 'JETTED', 'DICTIONARIES', 'OVEREXUBERANT',
'SYNONYMY', 'CHOSE', 'OVERFOUL', 'TOWERING', 'CITHRENS', 'CONFITEORS', 'IMMUNE',
'CENSORS', 'DOUGHBOY', 'BROMATE', 'TENPINS', 'COUNTERTHRUSTS', 'SYMPATHY',
'ROCKABY', 'TEAKETTLE', 'DECONTAMINATING', 'OXYPHILIC', 'SCHEDULED', 'AMPHIPODS',
'LITERS']
25000
Mon Mar 26 16:38:46 2018
Mon Mar 26 16:38:50 2018
```

Using pools

- Provided by **multiprocessing**
- Both thread and process pools
- Simplifies multiprogramming tasks

For many multiprocessing tasks, you want to process a list (or other iterable) of data and do something with the results. This is easily accomplished with the Pool object provided by the **multiprocessing** module.

This object creates a pool of *n* processes. Call the **.map()** method with a function that will do the work, and an iterable of data. map() will return a list the same size as the list that was passed in, containing the results returned by the function for each item in the original list.

For a thread pool, import **Pool** from **multiprocessing.dummy**. It works exactly the same, but creates threads.

Example

proc_pool.py

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

import random
from multiprocessing import Pool

POOL_SIZE = 30 ①

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in] ②

random.shuffle(WORDS) ③

def my_task(word): ④
    return word.upper()

if __name__ == '__main__':
    ppool = Pool(POOL_SIZE) ⑤

    WORD_LIST = ppool.map(my_task, WORDS) ⑥

    print(WORD_LIST[:20]) ⑦

    print("Processed {} words.".format(len(WORD_LIST)))
```

① number of processes

② read word file into a list, stripping off \n

③ randomize word list

④ actual task

⑤ create pool of POOL_SIZE processes

⑥ pass wordlist to pool and get results; map assigns values from input list to processes as needed

⑦ print last 20 words

proc_pool.py

```
['ANENST', 'SULPHATED', 'GUIDEBOOK', 'IT', 'CLOUDS', 'ULTRAROMANTIC', 'COLUMNAL',
'FRAT', 'SMELTING', 'RESTING', 'HERSELF', 'VIDEODISCS', 'DONNYBROOKS', 'CASTOR',
'ANTAGONISM', 'UNDE', 'SUPERSENSITIVITIES', 'FLUSTEREDLY', 'MERCIES', 'DEPAINTED']
Processed 173476 words.
```

Example

thr_pool.py

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

import random
from multiprocessing.dummy import Pool

POOL_SIZE = 30 ①

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in] ②

random.shuffle(WORDS) ③

def my_task(word):  ④
    return word.upper()

tpool = Pool(POOL_SIZE) ⑤

WORD_LIST = tpool.map(my_task, WORDS) ⑥

print(WORD_LIST[:20])  ⑦

print("Processed {} words.".format(len(WORD_LIST)))
```

① Get the thread pool object

② Authentication information from site

③ URL of site

④ this is the function each thread will invoke

⑤ use requests() to fetch JSON data

⑥ convert JSON to Python structure (dict)

⑦ this list becomes one element of array returned by pool

⑧ create pool of 8 threads

⑨ use threads to process data using fetch_result() function

⑩ display results

thr_pool.py

```
['POLYCHOTOMIES', 'SCENARIST', 'BLITE', 'MICROPIPET', 'GRIEVOUSLY', 'PETUNTZES',
'HOMOGENISE', 'NEWSWRITING', 'SABOT', 'ISOGONIES', 'RICKSHAS', 'MILTIEST',
'EXECUTORS', 'GIMBALS', 'DECONSECRATIONS', 'MICROTUBULE', 'GOALPOSTS', 'REPUMP',
'GROUNDER', 'COLLUSIVE']
Processed 173476 words.
```

Example

thr_pool_lcbo.py

```
#!/usr/bin/env python
from multiprocessing.dummy import Pool ①
from pprint import pprint
import requests

POOL_SIZE = 8
AUTH_TOKEN = 'CJAssociatesTraining' ②
AUTH_KEY=
'MDowYzMxMTg5Mi0yMzA5LTExZTUtODcxMC0wNzEwNDcxM2NkOTA6QVBxNklDQXU1M2RSNEkyUjBBOEpkZVN
QQVJUYXY2Q3liSzBy'

BASE_URL = 'http://lcboapi.com/products'  ③

search_terms = [ ④
    'stolichnaya', 'makers mark', 'woodford', 'wombat', 'molson', 'moosehead',
    'michelob', 'bacardi', 'old rotgut', 'four roses', 'moonshine', 'harvest',
    'captain morgan', 'tanqueray','green spot', 'chartreuse'
]

def fetch_data(search_term):  ⑤
    response = requests.get(
        BASE_URL,
        auth=(AUTH_KEY, AUTH_TOKEN),
        params={ 'q': search_term },
    )   ⑥
    raw_json = response.json() ⑦
    names = []
    if raw_json['result']:
        for result in raw_json['result']:
            names.append(result['name'])
    return names  ⑧

p = Pool(POOL_SIZE)  ⑨

results = p.map(fetch_data, search_terms)  ⑩

for search_term, result in zip(search_terms, results):  ⑩
    print("{}:".format(search_term.upper()))
    if result:
        pprint(result)
    else:
        print("** no results **")
```

① .dummy has Pool for threads

② credentials to access site

③ base url of site to access

④ terms to search for; each thread will search some of these terms

⑤ function invoked by each thread for each item in list passed to map()

⑥ make the request to the site

⑦ convert JSON to Python structure

⑧ return all names that matched one search term

⑨ create pool of POOL_SIZE threads

⑩ launch threads, collect results

⑪ iterate over results, mapping them to search terms

thr_pool_lcbo.py

```
STOLICHNAYA:
['Stolichnaya Vodka',
 'Stolichnaya Vodka',
 'Stolichnaya Vodka',
 'Stolichnaya Premium Vodka',
 'Stolichnaya Elit Vodka',
 'Stolichnaya Gold Vodka',
 'Stolichnaya Blueberi Vodka',
 'Stolichnaya Razberi']
MAKERS MARK:
["Maker's Mark Kentucky Bourbon",
 "Maker's Mark Kentucky Straight Bourbon",
 "Maker's Mark 46"]
WOODFORD:
["Woodford Reserve Distiller's Select",
 "Woodford Reserve Distiller's Select",
 'Woodford Reserve Double Oaked',
 'Woodford Reserve Straight Rye Whiskey',
 "Woodford Reserve Master's Collection Brandy Finish",
 'Woodford Reserve']
```

...

Alternatives to multiprogramming

- asyncio
- twisted

Threading and forking are not the only ways to have your program do more than one thing at a time. Another approach is asynchronous programming. This technique putting events (typically I/O events) in a list, or queue, and starting an event loop that processes the events one at a time. If the granularity of the event loop is small, this can be as efficient as multiprogramming.

Asynchronous programming is only useful for improving I/O throughput, such as networking clients and servers, or scouring a file system. Like threading (in Python), it will not help with raw computation speed.

The asyncio module in the standard library provides the means to write asynchronous clients and servers. The Twisted framework is a large and well-supported third-party module that provides support for many kinds of asynchronous communication. It has prebuilt objects for servers, clients, and protocols, as well as tools for authentication, translation, and many others. Find Twisted at twistedmaxtrix.com/trac.

Chapter 14 Exercises

Exercise 14-1 (pres_thread.py)

Using a thread pool (multiprocessing.dummy), calculate the age at inauguration of the presidents. To do this, read the presidents.txt file into an array of tuples, and then pass that array to the mapping function of the thread pool. The result of the map function will be the array of ages. You will need to convert the date fields into actual dates, and then subtract them.

Chapter 15: Scripting for System Administration

Objectives

- Launch external programs
- Check permissions on files
- Get system configuration information
- Store data offline
- Parse command line options
- Create Unix-style filters
- Configure application logging
- Parse command line options

Using glob

- Expands wildcards
- Windows and non-windows
- Useful with **subprocess** module

When executing external programs, sometimes you want to specify a list of files using a wildcard. The **glob** function in the **glob** module will do this. Pass one string containing a wildcard (such as `*.txt`) to glob(), and it returns a sorted list of the matching files. If no files match, it returns an empty list.

Example

glob_example.py

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

from glob import glob

files = glob('../DATA/*.txt') ①
print(files)
```

① expand file name wildcard into sorted list of matching names

glob_example.py

```
['../DATA/columns_of_numbers.txt', '../DATA/poe_sonnet.txt',
'../DATA/computer_people.txt', '../DATA/owl.txt', '../DATA/eggs.txt',
'../DATA/world_airport_codes.txt', '../DATA/stateinfo.txt', '../DATA/fruit2.txt',
'../DATA/us_airport_codes.txt', '../DATA/parrot.txt',
'../DATA/http_status_codes.txt', '../DATA/fruit1.txt', '../DATA/alice.txt',
'../DATA/spam.txt', '../DATA/world_median_ages.txt', '../DATA/phone_numbers.txt',
'../DATA/engineers.txt', '../DATA/tolkien.txt', '../DATA/tyger.txt',
'../DATA/states.txt', '../DATA/fruit.txt', '../DATA/areacodes.txt',
'../DATA/unabom.txt', '../DATA/presidents.txt', '../DATA/breakfast.txt',
'../DATA/Pride_and_Prejudice.txt', '../DATA/nsfw_words.txt', '../DATA/mary.txt',
'../DATA/2017FullMembersMontanaLegislators.txt', '../DATA/badger.txt',
'../DATA/README.txt', '../DATA/words.txt', '../DATA/primeministers.txt',
'../DATA/grail.txt', '../DATA/alt.txt', '../DATA/knights.txt',
'../DATA/world_airports_codes_raw.txt']
```

Using shlex.split()

- Splits string
- Preserves white space

If you have an external command you want to execute, you should split it into individual words. If your command has quoted whitespace, the normal **split()** method of a string won't work.

For this you can use **shlex.split()**, which preserves quoted whitespace within a string.

Example

shlex_split.py

```
#!/usr/bin/env python
# (c) 2018 CJ Associates
#
import shlex

cmd = 'herp derp "fuzzy bear" "wanga tanga" pop'  ①

print(cmd.split())  ②
print()

print(shlex.split(cmd))  ③
```

① Command line with quoted whitespace

② Normal split does the wrong thing

③ shlex.split() does the right thing

shlex_split.py

```
['herp', 'derp', '"fuzzy', 'bear"', '"wanga', 'tanga"', 'pop']

['herp', 'derp', 'fuzzy bear', 'wanga tanga', 'pop']
```

The subprocess module

- Spawns new processes
- works on Windows and non-Windows systems
- Convenience methods
  - **run()**
  - **call()**, **check_call()**

The **subprocess** module spawns and manages new processes. You can use this to run local non-Python programs, to log into remote systems, and generally to execute command lines.

subprocess implements a low-level class named Popen; However, the convenience methods **run()**, **check_call()**, and check_output()**, which are built on top of Popen(), are commonly used, as they have a simpler interface. You can capture *stdout** and **stderr**, separately. If you don't capture them, they will go to the console.

In all cases, you pass in an iterable containing the command split into individual words, including any file names. This is why this chapter starts with glob.glob() and shlex.split().

Table 21. CalledProcessError attributes

Attribute	Description
args	The arguments used to launch the process. This may be a list or a string.
returncode	Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully. A negative value -N indicates that the child was terminated by signal N (POSIX only).
stdout	Captured stdout from the child process. A bytes sequence, or a string if run() was called with an encoding or errors. None if stdout was not captured. If you ran the process with stderr=subprocess.STDOUT, stdout and stderr will be combined in this attribute, and stderr will be None. stderr

subprocess convenience functions

- run(), check_call() , check_output()
- Simpler to use than Popen

subprocess defines convenience functions, **call()**, **check_call()**, and **check_output()**.

```
proc subprocess.run(cmd, ...)
```

Run command with arguments. Wait for command to complete, then return a **CompletedProcess** instance.

```
subprocess.check_call(cmd, ...)
```

Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute.

```
check_output(cmd, ...)
```

Run command with arguments and return its output as a byte string. If the exit code was non-zero it raises a CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute and output in the output attribute.

NOTE run() is only implemented in Python 3.5 and later.

Example

subprocess_conv.py

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

import sys
from subprocess import check_call, check_output, CalledProcessError
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

try:
    check_call(full_cmd)
except CalledProcessError as err:
    print("Command failed with return code", err.returncode)

print('-' * 60)


try:
    output = check_output(full_cmd)
    print("Output:", output.decode(), sep='\n')
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

subprocess_conv.py

```
-rwxr-xr-x  1 jstrick  staff    297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff   2198 Feb 14  2016 ../DATA/textfiles.zip
-rwxr-xr-x  1 jstrick  staff  73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff    834 Feb 14  2016 ../DATA/tyger.txt
------------------------------------------------------------
Output:
-rwxr-xr-x  1 jstrick  staff    297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff   2198 Feb 14  2016 ../DATA/textfiles.zip
-rwxr-xr-x  1 jstrick  staff  73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff    834 Feb 14  2016 ../DATA/tyger.txt

------------------------------------------------
```

NOTE showing Unix/Linux/Mac output – Windows will be similar

TIP The following commands are *internal* to CMD.EXE, and must be preceded by `cmd /c` or they will not work: ASSOC, BREAK, CALL ,CD/CHDIR, CLS, COLOR, COPY, DATE, DEL, DIR, DPATH, ECHO, ENDLOCAL, ERASE, EXIT, FOR, FTYPE, GOTO, IF, KEYS, MD/MKDIR, MKLINK (vista and above), MOVE, PATH, PAUSE, POPD, PROMPT, PUSHD, REM, REN/RENAME, RD/RMDIR, SET, SETLOCAL, SHIFT, START, TIME, TITLE, TYPE, VER, VERIFY, VOL

Capturing stdout and stderr

- Add stdout, stderr args
- Assign subprocess.PIPE

To capture stdout and stderr with the subprocess module, import **PIPE** from subprocess and assign it to the stdout and stderr parameters to run(), check_call(), or check_output(), as needed.

For check_output(), the return value is the standard output; for run(), you can access the **stdout** and **stderr** attributes of the CompletedProcess instance returned by run().

NOTE	output is returned as a bytes object; call decode() to turn it into a normal Python string.

Example

subprocess_capture.py

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

import sys
from subprocess import check_output, Popen, CalledProcessError, STDOUT, PIPE ①
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

②
try:
    output = check_output(full_cmd) ③
```

```
    print("Output:", output.decode(), sep='\n') ④
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)

⑤
try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=STDOUT) ⑥
    stdout, stderr = proc.communicate() ⑦
    print("Output:", stdout.decode()) ⑧
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)

try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=PIPE) ⑨
    stdout, stderr = proc.communicate() ⑩
    print("Output:", stdout.decode()) ⑪
    print("Error:", stderr.decode()) ⑪
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

① need to import PIPE and STDOUT

② capture only stdout

③ check_output() returns stdout

④ stdout is returned as bytes (decode to str)

⑤ capture stdout and stderr together

⑥ assign PIPE to stdout, so it is captured; assign STDOUT to stderr, so both are captured together

⑦ call communicate to get the input streams of the process; it returns two bytes objects representing stdout and stderr

⑧ decode the stdout object to a string

⑨ assign PIPE to stdout and PIPE to stderr, so both are captured individually

⑩ now stdout and stderr each have data

⑪ decode from bytes and output

subprocess_capture.py

```
Output:
-rwxr-xr-x  1 jstrick  staff    297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff   2198 Feb 14  2016 ../DATA/textfiles.zip
-rwxr-xr-x  1 jstrick  staff  73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff    834 Feb 14  2016 ../DATA/tyger.txt

--------------------------------------------------
Output: ls: spam.txt: No such file or directory
-rwxr-xr-x  1 jstrick  staff    297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff   2198 Feb 14  2016 ../DATA/textfiles.zip
-rwxr-xr-x  1 jstrick  staff  73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff    834 Feb 14  2016 ../DATA/tyger.txt

--------------------------------------------------
Output: -rwxr-xr-x  1 jstrick  staff    297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff   2198 Feb 14  2016 ../DATA/textfiles.zip
-rwxr-xr-x  1 jstrick  staff  73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff    834 Feb 14  2016 ../DATA/tyger.txt

Error: ls: spam.txt: No such file or directory

--------------------------------------------------
```

Permissions

- Simplest is os.access()
- Get mode from os.lstat()
- Use binary AND with permission constants

Each entry in a Unix filesystem has a inode. The inode contains low-level information for the file, directory, or other filesystem entity. Permissions are stored in the 'mode', which is a 16-bit unsigned integer. The first 4 bits indicate what kind of entry it is, and the last 12 bits are the permissions.

To see if a file or directory is readable, writable, or executable use os.access(). To test for specific permissions, use the os.lstat() method to return a tuple of inode data, and use the S_IMODE () method to get the mode information as a number. Then use predefined constants such as stat.S_IRUSR, stat.S_IWGRP, etc. to test for permissions.

Example

file_access.py

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

import sys
import os

if len(sys.argv) < 2:
    sys.stderr.write('Please specify a starting directory\n')
    sys.exit(1)

start_dir = sys.argv[1]

for base_name in os.listdir(start_dir): ①
    file_name = os.path.join(start_dir,base_name)
    if os.access(file_name,os.W_OK):  ②
        print(file_name,"is writable")
```

① os.listdir() lists the contents of a directory

② os.access() returns True if file has specified permissions (can be os.W_OK, os.R_OK, or os.X_OK, combined with | (OR))

file_access.py ../DATA

```
../DATA/presidents.csv is writable
../DATA/wetprf is writable
../DATA/presidents.html is writable
../DATA/presidents.xlsx is writable
../DATA/presidents.db is writable
../DATA/DurhamRestaurantData is writable
../DATA/testscores.dat is writable
../DATA/solar.json is writable
../DATA/nws_precip_pr_20061222.nc is writable
../DATA/columns_of_numbers.txt is writable
```

...

Using shutil

- Portable ways to copy, move, and delete files
- Create archives
- Misc utilities

The **shutil** module provides portable functions for copying, moving, renaming, and deleting files. There are several variations of each command, depending on whether you need to copy all the attributes of a file, for instance.

The module also provides an easy way to create a zip file or compressed **tar** archive of a folder.

In addition, there are some miscellaneous convenience routines.

Example

shutil_ex.py

```
#!/usr/bin/env python
# (c) 2018 CJ Associates
#
import shutil
import os

shutil.copy('../DATA/alice.txt', 'betsy.txt') ①

print("betsy.txt exists:", os.path.exists('betsy.txt'))

shutil.move('betsy.txt', 'fred.txt') ②
print("betsy.txt exists:", os.path.exists('betsy.txt'))
print("fred.txt exists:", os.path.exists('fred.txt'))

new_folder = 'remove_me'

os.mkdir(new_folder) ③
shutil.move('fred.txt', new_folder)

shutil.make_archive(new_folder, 'zip', new_folder) ④

print("{}.zip exists:".format(new_folder), os.path.exists(new_folder + '.zip'))

print("{} exists:".format(new_folder), os.path.exists(new_folder))

shutil.rmtree(new_folder) ⑤

print("{} exists:".format(new_folder), os.path.exists(new_folder))
```

① copy file

② rename file

③ create new folder

④ make a zip archive of new folder

⑤ recursively remove folder

shutil_ex.py

```
betsy.txt exists: True
betsy.txt exists: False
fred.txt exists: True
remove_me.zip exists: True
remove_me exists: True
remove_me exists: False
```

Creating a useful command line script

- More than just some lines of code
- Input + Business Logic + Output
- Process files for input, or STDIN
- Allow options for customizing execution
- Log results

A good system administration script is more than just some lines of code hacked together. It needs to gather data, apply the appropriate business logic, and, if necessary, output the results of the business logic to the desired destination.

Python has two tools in the standard library to help create professional command line scripts. One of these is the argparse module, for parsing options and parameters on the script's command line. The other is fileinput, which simplifies processing a list of files specified on the command line.

We will also look at the logging module, which can be used in any application to output to a variety of log destinations, including a plain file, syslog on Unix-like systems or the NTLog service on Windows, or even email.

Creating filters

- Filter reads files or STDIN and writes to STDOUT

> Common on Unix systems Well-known filters: awk, sed, grep, head, tail, cat Reads command line arguments as files, otherwise STDIN use fileinput.input()

A common kind of script iterates over all lines in all files specified on the command line. The algorithm is

```
for filename in sys.argv[1:]:
    with open(filename) as F:
       for line in F:
            # process line
```

Many Unix utilities are written to work this way – sed, grep, awk, head, tail, sort, and many more. They are called filters, because they filter their input in some way and output the modified text. Such filters read STDIN if no files are specified, so that they can be piped into.

The fileinput.input() class provides a shortcut for this kind of file processing. It implicitly loops through sys.argv[1:], opening and closing each file as needed, and then loops through the lines of each file. If sys.argv[1:] is empty, it reads sys.stdin. If a filename in the list is '-', it also reads sys.stdin.

fileinput works on Windows as well as Unix and Unix-like platforms.

To loop through a different list of files, pass an iterable object as the argument to fileinput.input().

There are several methods that you can call from fileinput to get the name of the current file, e.g.

Table 22. fileinput Methods

Method	Description
filename()	Name of current file being readable
lineno()	Cumulative line number from all files read so far
filelineno()	Line number of current file
isfirstline()	True if current line is first line of a file
isstdin()	True if current file is sys.stdin
close()	Close fileinput

Example

file_input.py

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

import fileinput

for line in fileinput.input(): ①
    if 'bird' in line:
        print('{}: {}'.format(fileinput.filename(), line), end=' ') ②
```

① fileinput.input() is a generator of all lines in all files in sys.argv[1:]

② fileinput.filename() has the name of the current file

file_input.py ../DATA/parrot.txt ../DATA/alice.txt

```
 ../DATA/parrot.txt: At that point, the guy is so mad that he throws the bird into
the
 ../DATA/parrot.txt: For the first few seconds there is a terrible din. The bird
kicks
 ../DATA/parrot.txt: bird may be hurt. After a couple of minutes of silence, he's so
 ../DATA/parrot.txt: The bird calmly climbs onto the man's out-stretched arm and
says,
 ../DATA/alice.txt: with the birds and animals that had fallen into it:  there were
a
 ../DATA/alice.txt: bank--the birds with draggled feathers, the animals with their
 ../DATA/alice.txt: some of the other birds tittered audibly.
 ../DATA/alice.txt: and confusion, as the large birds complained that they could not
```

Parsing the command line

- Parse and analyze sys.argv
- use argparse
- parses entire command line
- very flexible
- validates options and arguments

Many command line scripts need to accept options and arguments. In general, options control the behavior of the script, while arguments provide input. Arguments are frequently file names, but can be anything. All arguments are available in Python via sys.argv

There are at least three modules in the standard library to parse command line options. The oldest module is getopt (earlier than v1.3), then optparse (introduced 2.3, now deprecated), and now, argparse is the latest and greatest. (Note: argparse is only available in 2.7 and 3.0+).

To get started with argparse, create an ArgumentParser object. Then, for each option or argument, call the parser's add_argument() method.

The add_argument() method accepts the name of the option (e.g. '-count') or the argument (e.g. 'filename'), plus named parameters to configure the option.

Once all arguments have been described, call the parser's parse_args() method. (By default, it will process sys.argv, but you can pass in any list or tuple instead.) parse_args() returns an object containing the arguments. You can access the arguments using either the name of the argument or the name specified with dest.

One useful feature of argparse is that it will convert command line arguments for you to the type specified by the type parameter. You can write your own function to do the conversion, as well.

Another feature is that argparse will automatically create a help option, -h, for your application, using the help strings provided with each option or parameter.

argparse parses the entire command line, not just arguments

Table 23. add_argument() named parameters

parameter	description
dest	Name of attribute (defaults to argument name)
nargs	Number of arguments Default: one argument, returns string '*': 0 or more arguments, returns list '+': 1 or more arguments, returns list '?': 0 or 1 arguments, returns list N: exactly N arguments, returns list
const	Value for options that do not take a user-specified value
default	Value if option not specified
type	type which the command-line arguments should be converted ; one of 'string', 'int', 'float', 'complex' or a function that accepts a single string argument and returns the desired object. (Default: 'string')
choices	A list of valid choices for the option
required	Set to true for required options
metavar	A name to use in the help string (default: same as dest)
help	Help text for option or argument

Example

parsing_args.py

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

import sys
import re
import fileinput
import argparse
from glob import glob  ①
from itertools import chain  ②

arg_parser = argparse.ArgumentParser(description="Emulate grep with python") ③

arg_parser.add_argument(
    '-i',
    dest='ignore_case', action='store_true',
    help='ignore case'
) ④

arg_parser.add_argument(
    'pattern', help='Pattern to find (required)'
) ⑤

arg_parser.add_argument(
    'filenames',nargs='*',
    help='filename(s) (if no files specified, read STDIN)'
) ⑥

args = arg_parser.parse_args() ⑦

print('-' * 40)
print(args)
print('-' * 40)

regex = re.compile(args.pattern, re.I if args.ignore_case else 0) ⑧

filename_gen = (glob(f) for f in args.filenames) ⑨
filenames = chain.from_iterable(filename_gen) ⑩

for line in fileinput.input(filenames): ⑪
    if regex.search(line):
        print(line.rstrip())
```

① needed on Windows to parse filename wildcards

② needed on Windows to flatten list of filename lists

③ create argument parser

④ add option to the parser; dest is name of option attribute

⑤ add required argument to the parser

⑥ add optional arguments to the parser

⑦ actually parse the arguments

⑧ compile the pattern for searching; set re.IGNORECASE if -i option

⑨ for each filename argument, expand any wildcards; this returns list of lists

⑩ flatten list of lists into a single list of files to process (note: both filename_gen and filenames are generators; these two lines are only needed on Windows — non-Windows systems automatically expand wildcards)

⑪ loop over list of file names and read them one line at a time

parsing_args.py

```
usage: parsing_args.py [-h] [-i] pattern [filenames [filenames ...]]
parsing_args.py: error: the following arguments are required: pattern, filenames
```

parsing_args.py -i '\bbil' ../DATA/alice.txt ../DATA/presidents.txt

```
----------------------------------------
Namespace(filenames=['../DATA/alice.txt', '../DATA/presidents.txt'],
ignore_case=True, pattern='\\bbil')
----------------------------------------
                The Rabbit Sends in a Little Bill
Bill's got the other--Bill! fetch it here, lad!--Here, put 'em up
Here, Bill! catch hold of this rope--Will the roof bear?--Mind
crash)--`Now, who did that?--It was Bill, I fancy--Who's to go
then!--Bill's to go down--Here, Bill! the master says you're to
  `Oh! So Bill's got to come down the chimney, has he?' said
Alice to herself.  `Shy, they seem to put everything upon Bill!
I wouldn't be in Bill's place for a good deal:  this fireplace is
above her:  then, saying to herself `This is Bill,' she gave one
Bill!' then the Rabbit's voice along--`Catch him, you by the
  Last came a little feeble, squeaking voice, (`That's Bill,'
The poor little Lizard, Bill, was in the middle, being held up by
end of the bill, "French, music, AND WASHING--extra."'
Bill, the Lizard) could not make out at all what had become of
Lizard as she spoke.  (The unfortunate little Bill had left off
42:Clinton:William Jefferson 'Bill':1946-08-19:NONE:Hope:Arkansas:1993-01-20:2001-
01-20:Democratic
```

parsing_args.py -h

```
usage: parsing_args.py [-h] [-i] pattern [filenames [filenames ...]]

Emulate grep with python

positional arguments:
  pattern     Pattern to find (required)
  filenames   filename(s) (if no files specified, read STDIN)

optional arguments:
  -h, --help  show this help message and exit
  -i          ignore case
```

Simple Logging

- Specify file name
- Configure the minimum logging level
- Messages added at different levels
- Call methods on logging

For simple logging, just configure the log file name and minimum logging level with the basicConfig() method. Then call one of the per-level methods, such as logging.debug or logging.error, to output a log message for that level. If the message is at or above the minimal level, it will be added to the log file.

The file will continue to grow, and must be manually removed or truncated. If the file does not exist, it will be created.

The logger module provides 5 levels of logging messages, from DEBUG to CRITICAL. When you set up a logger, you specify the minimum level of messages to be logged. If you set up the logger with the minimum level set to ERROR, then only messages at ERROR and CRITICAL levels will be logged. Setting the minimum level to DEBUG allows all messages to be logged.

Table 24. Logging Levels

Level Value	CRITICAL FATAL
50	ERROR
40 WARN	WARNING
30	INFO
20	DEBUG
10	UNSET

Example

logging_simple.py

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

import logging

logging.basicConfig(
    filename='../TEMP/simple.log',
    level=logging.WARN,
) ①

logging.warning('This is a warning') ②
logging.debug('This message is for debugging') ③
logging.error('This is an ERROR') ④
logging.critical('This is ***CRITICAL***')
logging.info('The capital of North Dakota is Bismark')
```

① setup logging; minimal level is WARN

② message will be output

③ message will NOT be output

④ message will be output

⑤ message will be output

⑥ message will not be output

simple.log

```
WARNING:root:This is a warning
ERROR:root:This is an ERROR
CRITICAL:root:This is ***CRITICAL***
```

Formatting log entries

- Add format=format to basicConfig() parameters
- Format is a string containing directives and (optionally) other text
- Use directives in the form of %(item)type
- Other text is left as-is

To format log entries, provide a format parameter to the basicConfig() method. This format will be a string contain special directives (i.e. Placeholders) and, optionally, other text. The directives are replaced with logging information; other data is left as-is.

Directives are in the form %(item)type, where item is the data field, and type is the data type.

Example

logging_formatted.py

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

import logging

logging.basicConfig(
    format='%(asctime)s %(levelname)s %(message)s', ①
    filename='../TEMP/formatted.log',
    level=logging.INFO,
)

logging.info("this is information")
logging.warning("this is a warning")
logging.info("this is information")
logging.fatal("this is fatal")
```

① set the format for log entries

formatted.log

```
2018-03-26 16:38:54,184 INFO this is information
2018-03-26 16:38:54,184 WARNING this is a warning
2018-03-26 16:38:54,185 INFO this is information
2018-03-26 16:38:54,185 CRITICAL this is fatal
```

Table 25. Log entry formatting directives

Directive	Description
%(name)s	Name of the logger (logging channel)
%(levelno)s	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL)
%(levelname)s	Text logging level for the message ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
%(pathname)s	Full pathname of the source file where the logging call was issued (if available)
%(filename)s	Filename portion of pathname
%(module)s	Module (name portion of filename)
%(lineno)d	Source line number where the logging call was issued (if available)
%(funcName)s	Function name
%(created)f	Time when the LogRecord was created (time.time() return value)
%(asctime)s	Textual time when the LogRecord was created
%(msecs)d	Millisecond portion of the creation time
%(relativeCreated)d	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded (typically at application startup time)
%(thread)d	Thread ID (if available)
%(threadName)s	Thread name (if available)
%(process)d	Process ID (if available)
%(message)s	The result of record.getMessage(), computed just as the record is emitted

Logging exception information

- Use logging.exception()
- Adds exception info to message
- Only in **except** blocks

The logging.exception() function will add exception information to the log message. It should only be called in an **except** block.

Example

logging_exception.py

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

import logging

logging.basicConfig(
    filename='../TEMP/exception.log',
    level=logging.WARNING,
) ①

for i in range(3):
    try:
        result = i/0
    except ZeroDivisionError:
        logging.exception('Logging with exception info') ②
```

① configure logging

② minimum level

③ add exception info to the log

exception.log

```
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
```

Logging to other destinations

- Use specialized handlers to write to other destinations
- Multiple handlers can be added to one logger
  - NTEventLogHandler for Windows event log
  - SysLogHandler for syslog
  - SMTPHandler for logging via email

The logging module provides some preconfigured log handlers to send log messages to destinations other than a file.

Each handler has custom configuration appropriate to the destination. Multiple handlers can be added to the same logger, so a log message will go to a file and to email, for instance, and each handler can have its own minimum level. Thus, all messages could go to the message file, but only CRITICAL messages would go to email.

Be sure to read the documentation for the particular log handler you want to use

NOTE

On Windows, you must run the example script (logging.altdest.py) as administrator. You can find `Command Prompt (admin)` on the main Windows 8/10 menu. You can also right-click on `Command Prompt` from the Windows 7 menu and choose "Run as administrator".

Example

logging_altdest.py

```
#!/usr/bin/env python
import sys
import logging
import logging.handlers

logger = logging.getLogger('ThisApplication') ①
logger.setLevel(logging.DEBUG)  ②

if sys.platform == 'win32':
    eventlog_handler = logging.handlers.NTEventLogHandler("Python Log Test") ③
    logger.addHandler(eventlog_handler) ④
else:
    syslog_handler = logging.handlers.SysLogHandler() ⑤
    logger.addHandler(syslog_handler) ⑥

# note -- use your own SMTP server...
email_handler = logging.handlers.SMTPHandler(
    ('smtpcorp.com',8025),
    'LOGGER@pythonclass.com',
    ['jstrick@mindspring.com'],
    'ThisApplication Log Entry',
    ('jstrickpython', 'python(monty)'),
)  ⑦

logger.addHandler(email_handler) ⑧

logger.debug('this is debug') ⑨
logger.critical('this is critical') ⑨
logger.warning('this is a warning') ⑨
```

① get logger for application

② minimum log level

③ create NT event log handler

④ install NT event handler

⑤ create syslog handler

⑥ install syslog handler

⑦

create email handler

⑧ install email handler

⑨ goes to all handlers

Chapter 15 Exercises

Exercise 15-1 (copy_files.py)

Write a script to find all text files in the DATA folder of the student files and copy them to C:\TEMP (Windows) or /tmp (non-windows). On Windows, create the C:\TEMP folder if it does not already exist.

Add logging to the script, and log each filename at level INFO.

> **TIP** use shutil.copy() to copy the files.

Chapter 16: Serializing data

Objectives

- Have a good understanding of the XML format
- Know which modules are available to process XML
- Use lxml ElementTree to create a new XML file
- Parse an existing XML file with ElementTree
- Using XPath for searching XML nodes
- Load JSON data from strings or files
- Write JSON data to strings or files
- Read and write CSV data

About XML

- Variant of SGML
- All data contained within tags

An XML document consists of a single element, which contains sub-elements, which can have further sub-elements inside them. Elements are indicated by tags in the text. Tags are always inside angle brackets < >. Elements can either contain content, or they can be empty.

Tags can contain attributes, indicated by attribute="value". Tags can either appear in begin/end pairs, in which the end tag starts with a slash, or as a single tag, in which case the tag ends with a slash. Attributes must be surrounded by double quotes. All tag names and attribute names should be lower case.

Normal Approaches to XML

- SAX
  - One scan through file
  - Good for large files
  - Uses callbacks on XML parsing events
- DOM
  - Builds a document tree
  - Python supports both through many libraries

There are two approaches normally used in working with XML.

The first is called SAX, which stands for for Simple API for XML . It processes an XML file as a single stream, and so is appropriate for large XML files.

SAX processing consists of attaching callback functions to SAX events. These events are created when the XML reader encounters all the various components of an XML file – begin tags, end tags, data, and so forth.

The second approach is called the DOM, (Document Object Model), which parses an XML document into a tree that's fully resident in memory.

A top-level Document instance is the root of the tree, and has a single child which is the top-level Element instance; this Element has child nodes representing the content and any sub-elements, which may in turn have further children and so forth. Classes such as Text, Comment, CDATASection, EntityReference, provide access to XML structure and data. Nodes have methods for accessing the parent and child nodes, accessing element and attribute values, insert and delete nodes, and converting the tree back into XML.

The DOM is often useful for modifying XML documents, because you can create a DOM, modify it by adding new nodes and moving sub-trees around, and then produce a new XML document as output.

While the DOM specification doesn't require that the entire tree be resident in memory at one time, many of the Python DOM implementations keep the whole tree in RAM, which can limit the size of the file being processed.

Which module to use?

- Bewildering array of XML modules
- Some are SAX, some are DOM
- Use xml.etree.ElementTree

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, **xml.etree.ElementTree** is fast, provides both SAX and DOM style parsing, and unlike many of the other modules, has a Pythonic interface.

If available, use **lxml.etree**, which is a superset of ElementTree with some nice extra features, such as pretty-printing.

Table 26. XML Modules in the Python 3 Standard Library

Module	Description
xml.parsers.expat	Fast XML parsing using Expat
xml.dom	The Document Object Model API
xml.dom.minidom	Lightweight DOM implementation
xml.dom.pulldom	Support for building partial DOM trees
xml.sax	Support for SAX2 parsers
xml.sax.handler	Base classes for SAX handlers
xml.sax.saxutils	SAX Utilities
xml.sax.xmlreader	Interface for XML parsers
xml.etree.ElementTree	Pythonic interface to XML

Getting Started With ElementTree

- Import xml.etree.ElementTree (or lxml.etree) as ET for convenience
- Parse XML or create empty ElementTree

ElementTree is part of the Python standard library; lxml is included with the Anaconda distribution.

Since putting "xml.etree.ElementTree" in front of its methods requires a lot of extra typing , it is typical to alias xml.etree.ElementTree to just ET when importing it: import xml.etree.ElementTree as ET

You can check the version of ElementTree via the VERSION attribute:

```
import xml.etree.ElementTree as ET
print(ET.VERSION)
```

How ElementTree Works

- ElementTree contains root Element
- Document is tree of Elements

In ElementTree, an XML document consists of a nested tree of Element objects. Each Element corresponds to an XML tag.

An ElementTree object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use ElementTree.parse(); this creates the ElementTree wrapper and the tree of Elements. You can then navigate to, or search for, Elements within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

```
element = root.find('sometag')
for subelement in element:
    print(subelement.tag)
print(element.get('someattribute'))
```

Elements

- Element has
  - Tag name
  - Attributes (implemented as a dictionary)
  - Text
  - Tail
  - Child elements (implemented as a list) (if any)
- SubElement creates child of Element

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** class is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the get() method).

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is any text following the element, before the next element.

The **SubElement** class is a convenient way to add children to an existing Element.

TIP Only the tag property of an Element is required; other properties are optional.

Table 27. Element properties and methods

Property	Description
append(element)	Add a subelement element to end of subelements
attrib	Dictionary of element's attributes
clear()	Remove all subelements
find(path)	Find first subelement matching path
findall(path)	Find all subelements matching path
findtext(path)	Shortcut for find(path).text
get(attr)	Get an attribute; Shortcut for attrib.get()
getiterator()	Returns an iterator over all descendants
getiterator(path)	Returns an iterator over all descendants matching path
insert(pos,element)	Insert subelement element at position pos
items()	Get all attribute values; Shortcut for attrib.items()
keys()	Get all attribute names; Shortcut for attrib.keys()
remove(element)	Remove subelement element
set(attrib,value)	Set an attribute value; shortcut for attr[attrib] = value
tag	The element's tag
tail	Text following the element
text	Text contained within the element

Table 28. ElementTree properties and methods

Property	Description
find(path)	Finds the first toplevel element with given tag; shortcut for getroot().find(path).
findall(path)	Finds all toplevel elements with the given tag; shortcut for getroot().findall(path).
findtext(path)	Finds element text for first toplevel element with given tag; shortcut for getroot().findtext(path).
getiterator(path)	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
getroot()	Return the root node of the document
parse(filename) parse(fileobj)	Parse an XML source (filename or file-like object)
write(filename,encoding)	Writes XML document to filename, using encoding (Default us-ascii).

Creating a New XML Document

- Create root element
- Add descendants via SubElement
- Use keyword arguments for attributes
- Add text after element created
- Create ElementTree for import/export

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once the root element is created, use SubElement to add elements to the root element, and then nested Elements as needed. SubElement returns the new element, so you can assign the contents of the tag to the **text** attribute.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text. From the ElementTree object, call write.

To output an XML string from your elements, call ET.tostring(), passing the root of the element tree as a parameter. It will return a bytes object (pure ASCII), so use .decode() to convert it to a normal Python string.

For an example of creating an XML document from a data file, see **xml_create_knights.py** in the EXAMPLES folder

Example

xml_create_movies.py

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

# from xml.etree import ElementTree as ET
import lxml.etree as ET


movie_data = [
    ('Jaws', 'Spielberg, Stephen'),
    ('Vertigo', 'Alfred Hitchcock'),
    ('Blazing Saddles', 'Brooks, Mel'),
    ('Princess Bride', 'Reiner, Rob'),
    ('Avatar', 'Cameron, James'),
]

movies = ET.Element('movies')

for name, director in movie_data:
    movie = ET.SubElement(movies, 'movie', name=name)
    ET.SubElement(movie, 'director').text = director

print(ET.tostring(movies, pretty_print=True).decode())

doc = ET.ElementTree(movies)

doc.write('movies.xml')
```

xml_create_movies.py

```
<movies>
  <movie name="Jaws">
    <director>Spielberg, Stephen</director>
  </movie>
  <movie name="Vertigo">
    <director>Alfred Hitchcock</director>
  </movie>
  <movie name="Blazing Saddles">
    <director>Brooks, Mel</director>
  </movie>
  <movie name="Princess Bride">
    <director>Reiner, Rob</director>
  </movie>
  <movie name="Avatar">
    <director>Cameron, James</director>
  </movie>
</movies>
```

Parsing An XML Document

- Use ElementTree.parse()
- returns an ElementTree object
- Use get... or find... methods to select element

Use the parse() method to parse an existing XML document. It returns an ElementTree object, from which you can find the root, or any other element within the document.

To get the root element, use the getroot() method.

Example