Custom Python Scripting for MMS

John Strickler

Version 1.0, March 2021

Table of Contents

About this course . . . 1
- Welcome! . . . 2
- Classroom etiquette . . . 3
- Course Outline . . . 4
- Student files . . . 5
- Extracting the student files . . . 6
- Examples . . . 7
- Lab Exercises . . . 8
- Appendices . . . 8

Chapter 1: Working with Files . . . 9
- Text file I/O . . . 10
- Opening a text file . . . 11
- The *with* block . . . 12
- Reading a text file . . . 13
- Writing to a text file . . . 18

Chapter 2: Binary Data . . . 23
- "Binary" (raw, or non-delimited) data . . . 24
- Binary vs Text data . . . 25
- Using Struct . . . 26
- Bitwise operations . . . 32

Chapter 3: Regular Expressions . . . 39
- Regular Expressions . . . 40
- RE Syntax Overview . . . 41
- Finding matches . . . 43
- RE Objects . . . 46
- Compilation Flags . . . 49
- Groups . . . 53
- Special Groups . . . 56
- Replacing text . . . 58
- Replacing with a callback . . . 60
- Splitting a string . . . 63

Chapter 4: iPython and Jupyter . . . 65
- About iPython . . . 66
- Features of iPython . . . 67
- The many faces of iPython . . . 68
- Starting iPython . . . 69

Getting Help . . . 70
Tab Completion . . . 71
Magic Commands . . . 72
Benchmarking . . . 73
External commands . . . 74
Jupyter notebooks . . . 75
Chapter 5: Introduction to NumPy . . . 77
Python's scientific stack . . . 78
NumPy overview . . . 79
Creating Arrays . . . 80
Creating ranges . . . 84
Working with arrays . . . 87
Shapes . . . 90
Slicing and indexing . . . 94
Indexing with Booleans . . . 97
Selecting rows based on conditions . . . 100
Stacking . . . 102
ufuncs and builtin operators . . . 104
Vectorizing functions . . . 105
Getting help . . . 110
Iterating . . . 111
Matrix Multiplication . . . 114
Data Types . . . 117
Reading and writing Data . . . 120
Saving and retrieving arrays . . . 125
Array creation shortcuts . . . 128
Chapter 6: Introduction to pandas . . . 133
About pandas . . . 134
pandas architecture . . . 135
Series . . . 136
DataFrames . . . 139
Index objects . . . 144
Basic Indexing . . . 148
Broadcasting . . . 152
Removing entries . . . 156
Data alignment . . . 158
Time Series . . . 160
Useful pandas methods . . . 164

Reading Data . . . 166
More pandas... . . . 169
Chapter 7: Introduction to Matplotlib . . . 171
About matplotlib . . . 172
matplotlib architecture . . . 173
Matplotlib Terminology . . . 174
Matplotlib Keeps State . . . 175
What Else Can You Do? . . . 176
Matplotlib Demo . . . 177
Chapter 8: Database Access . . . 179
The DB API . . . 180
Connecting to a Server . . . 181
Creating a Cursor . . . 183
Executing a Statement . . . 184
Fetching Data . . . 185
SQL Injection . . . 188
Parameterized Statements . . . 190
Dictionary Cursors . . . 197
Metadata . . . 201
Transactions . . . 204
Object-relational Mappers . . . 205
NoSQL . . . 206
Chapter 9: Network Programming . . . 213
Grabbing a web page . . . 214
Consuming Web services . . . 218
HTTP the easy way . . . 221
sending e-mail . . . 229
Email attachments . . . 232
Remote Access . . . 236
Copying files with Paramiko . . . 239
Chapter 10: Effective Scripts . . . 243
Using **glob** . . . 244
Using shlex.split() . . . 246
The subprocess module . . . 247
subprocess convenience functions . . . 248
Capturing stdout and stderr . . . 251
Permissions . . . 254
Using shutil . . . 256

Creating a useful command line script . . . 259
Creating filters . . . 260
Parsing the command line . . . 263
Simple Logging . . . 268
Formatting log entries . . . 270
Logging exception information . . . 272
Logging to other destinations . . . 274
Chapter 11: Sockets . . . 279
Sockets . . . 280
Socket options . . . 281
Server concepts . . . 282
Client concepts . . . 285
Application protocols . . . 287
Forking servers . . . 288
Chapter 12: Errors and Exception Handling . . . 293
Syntax errors . . . 294
Exceptions . . . 295
Handling exceptions with try . . . 296
Handling multiple exceptions . . . 297
Handling generic exceptions . . . 298
Ignoring exceptions . . . 299
Using else . . . 300
Cleaning up with finally . . . 302
Chapter 13: Introduction to Python Classes . . . 309
About object-oriented programming . . . 310
Defining classes . . . 311
Constructors . . . 313
Instance methods . . . 314
Properties . . . 318
Class methods and data . . . 322
Static Methods . . . 324
Private methods . . . 325
Inheritance . . . 326
Untangling the nomenclature . . . 329
Chapter 14: Multiprogramming . . . 333
Multiprogramming . . . 334
What Are Threads? . . . 335
The Python Thread Manager . . . 336

The threading Module . . . 337
Threads for the impatient . . . 338
Creating a thread class . . . 340
Variable sharing . . . 342
Using queues . . . 345
Debugging threaded Programs . . . 348
The multiprocessing module . . . 350
Using pools . . . 354
Alternatives to multiprogramming . . . 360
Appendix A: Python Bibliography . . . 363
Appendix B: String Formatting . . . 367
Overview . . . 367
Parameter Selectors . . . 368
Data types . . . 370
Field Widths . . . 373
Alignment . . . 376
Fill characters . . . 379
Signed numbers . . . 381
Parameter Attributes . . . 384
Formatting Dates . . . 386
Run-time formatting . . . 390
Miscellaneous tips and tricks . . . 393
Index . . . 395

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 classroom 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 killer rabbits to class. They might maim, dismember, or otherwise disturb your fellow students.

Course Outline

Day 1

Chapter 1 Working with files
Chapter 2 Binary data
Chapter 3 Regular expressions
Chapter 4 iPython and Jupyter
Chapter 5 Intro to Numpy
Chapter 6 Intro to Pandas
Chapter 7 Intro to Matplotlib
Chapter 8 Database access
Chapter 9 Network programming
Chapter 10 Effective scripts
Chapter 11 Sockets
Chapter 12 Errors and exception handling
Chapter 13 Introduction to Python classes
Chapter 14 Multiprogramming

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 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 **py3mmscyops**.

What's in the files?

py3mmscyops contains data and other files needed for the exercises
py3mmscyops/EXAMPLES contains the examples from the course manuals.
py3mmscyops/ANSWERS contains sample answers to the labs.

WARNING	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 **py3mmscyops.zip**. Extract all files to your desktop. This will create the folder **py3mmscyops**.

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

Copy or download **py3mmscyops.tar.gz** to your home directory. In your home directory, type

```
tar xzvf py3mmscyops.tar.gz
```

This will create the **py3mmscyops** directory under your home directory.

Examples

Nearly all examples from the course manual are provided in the EXAMPLES subdirectory.

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 Fred

```
['/Users/jstrick/curr/courses/python/examples3/cmd_line_args.py', 'Fred']
name is Fred
```

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 py3mmscyops/ANSWERS

Appendices

- Appendix A: Python Bibliography

Chapter 1: Working with Files

Objectives

- Reading a text file line-by-line
- Reading an entire text files
- Reading all lines of a text file into an array
- Writing to a text file

Text file I/O

- Create a file object with open
- Specify modes: read/write, text/binary
- Read or write from file object
- Close file object (or use **with** block)

Python provides a file object that is created by the built-in open() function. From this file object you can read or write data in several different ways. When opening a file, you specify the file name and the mode, which says whether you want to read, write, or append to the file, and whether you want text or binary (raw) processing.

NOTE This chapter is about working with generic files. For files in standard formats, such as XML, CSV, YAML, JSON, and many others, Python has format-specific modules to read them.

Opening a text file

- Specify the file name and the mode
- Returns a file object
- Mode can be read or write
- Specify "b" for binary (raw) mode
- Omit mode for reading

Open a text file with the open() command. Arguments are the file name, which may be specified as a relative or absolute path, and the mode. The mode consists of "r" for read, "w" for write, or "a" for append. To open a file in binary mode, add "b" to the mode, as in "rb", "wb", or "ab".

If you omit the mode, "r" is the default.

Example

```
ty = open("tyger.txt","r")  open for reading in text mode
ty = open("tyger.txt")      open for reading in text mode (default mode)
junk = open("junk.dat","rb")    open for reading in raw mode
stf = open("stuff.txt","w") open for writing in text mode
stf = open("stuff.txt","x") open for writing in text mode, fail if file exists
moju = open("morejunk.dat","wb")    open for writing in raw mode
config = open("spam.cfg","a")   open for append in text mode
```

TIP The **fileinput** module in the standard library makes it easy to loop over each line in all files specified on the command line, or STDIN if no files are specified. This avoids having to open and close each file.

The *with* block

- Provides "execution context"
- Automagically closes file object
- Not specific to file objects

Because it is easy to forget to close a file object, you can use a **with** block to open your file. This will automatically close the file object when the block is finished. The syntax is

```
with open(filename, mode) as fileobject:
    # process fileobject
```

Reading a text file

- Iterate through file with for/in

```
    for line in file_in
```

- Use methods of the file object

```
    file_in.readlines()   read all lines from file_in
    file_in.read()        read all of file_in
    file_in.read(n)       read n characters from file in text mode; n bytes from
file_in in binary mode
    file_in.readline()    read next line from file_in
```

The easiest way to read a file is by looping through the file object with a for/in loop. This is possible because the file object is an iterator, which means the object knows how to provide a sequence of values.

You can also read a text file one line or multiple lines at a time. **readline()** reads the next available line; **readlines()** reads all lines into a list.

read() will read the entire file; **read(n)** will read n bytes from the file (n *characters* if in text mode).

readline() will read the next line from the file.

Example

read_tyger.py

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

with open("../DATA/tyger.txt", "r") as tyger_in:  ①
    for line in tyger_in:  ②
        print(line, end='')  ③
```

① **tyger_in** is return value of **open(...)**

② **tyger_in** is a *generator*, returning one line at a time

③ the line already has a newline, so **print()** does not need one

read_tyger.py

```
           The Tyger

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?

In what distant deeps or skies
Burnt the fire of thine eyes?
On what wings dare he aspire?
What the hand dare seize the fire?

And what shoulder, & what art,
Could twist the sinews of thy heart?
And when thy heart began to beat,
What dread hand? & what dread feet?

What the hammer? what the chain?
In what furnace was thy brain?
What the anvil? what dread grasp
Dare its deadly terrors clasp?

When the stars threw down their spears
And water'd heaven with their tears,
Did he smile his work to see?
Did he who made the Lamb make thee?

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Dare frame thy fearful symmetry?

                    by William Blake
```

Example

reading_files.py

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

print("** About Spam **")
with open("../DATA/spam.txt") as spam_in:
    for line in spam_in:
        print(line.rstrip('\r\n')) ①

with open("../DATA/eggs.txt") as eggs_in:
    eggs = eggs_in.readlines()  ②

print("\n\n** About Eggs **")
print(eggs[0].rstrip())  ③
print(eggs[2].rstrip())
```

① rstrip('\n\r') removes \r or \n from end of string

② readlines() reads all lines into an array

③ [:-1] is another way to skip the newline (but .rstrip() does not remove spaces or tabs if they are significant)

reading_files.py

```
** About Spam **
SPAM may be famous now, but it wasn't always that way. Fact is, SPAM hails from some
rather humble beginnings.

Flash back to Austin, Minnesota, in 1937.

You're right. There isn't much here, except for an ambitious company called Hormel.

These good folks are about to hit upon an amazing little recipe: a spicy ham packaged in
a handy dandy 12-ounce can.

J. C. Hormel, then president, adds the crowning ingredient: He holds a contest to give
the product a name as distinctive as its taste.

SPAM soars. In fact, in that very first year of production, it grabs 18 percent of the
market.

Over 65 years years later, more than 6 billion cans of SPAM have been sold.


** About Eggs **
You can scramble, fry, poach and bake eggs or cook them in their shells.
Eggs are also the main ingredient in some dishes that came to the U.S.  from other
countries, such as a frittata, egg foo yung, quiche or souffle.
```

Writing to a text file

- Use write() or writelines()
- Add \n manually

To write to a text file, use the write() function to write a single string; or writelines() to write a list of strings.

writelines() will not add newline characters, so make sure the items in your list already have them.

Example

write_file.py

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

states = (
    'Virginia',
    'North Carolina',
    'Washington',
    'New York',
    'Florida',
    'Ohio',
)

with open("states.txt", "w") as states_out: ①
    for state in states:
        states_out.write(state + "\n")  ②
```

① "w" opens for writing, "a" for append

② write() does not add \n automatically

write_file.py

cat states.txt (or type states.txt under windows)

```
Virginia
North Carolina
Washington
New York
Florida
Ohio
```

"writelines" should have been called "writestrings"

Table 1. File Methods

Function	Description
f.close()	close file f
f.flush()	write out buffered data to file f
s = f.read(n) s = f.read()	read size bytes from file f into string s; if n is ⇐ 0, or omitted, reads entire file
s = f.readline() s = f.readline(n)	read one line from file f into string s. If n is specified, read no more than n characters
m = f.readlines()	read all lines from file f into list m
f.seek(n) f.seek(n,w)	position file f at offset n for next read or write; if argument w (whence) is omitted or 0, offset is from beginning; if 1, from current file position, if 2, from end of file
f.tell()	return current offset from beginning of file
f.write(s)	write string s to file f
f.writelines(m)	write list of strings m to file f; does not add line terminators

Chapter 1 Exercises

Exercise 1-1 (line_no.py)

Write a program to display each line of a file preceded by the line number. Allow your program to process one or more files specified on the command line. Be sure to reset the line number for each file.

> **TIP** Use enumerate().

Test with the following commands:

```
python line_no.py DATA/tyger.txt
python line_no.py DATA/parrot.txt DATA/tyger.txt
```

Test with other files, as desired

Exercise 1-2 (alt_lines.py)

Write a program to create two files, a.txt and b.txt from the file alt.txt. Lines that start with 'a' go in a.txt; the other lines (which all start with 'b') go in b.txt. Compare the original to the two new files.

Exercise 1-3 (count_alice.py, count_words.py)

A. Write a program to count how many lines of alice.txt contain the word "Alice". (There should be 392).

> **TIP** Use the **in** operator to test whether a line contains the word "Alice"

B. Modify count_alice.py to take the first command line parameter as a word to find, and the remaining parameters as filenames. For each file, print out the file name and the number of lines that contain the specified word. Test thoroughly

FOR ADVANCED STUDENTS (icount_words.py) Modify count_words.py to make the search case-insensitive.

Chapter 2: 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 2. 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 3. 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_in:
    chimp_bmp = chimp_in.read(s.size)  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
```

Bitwise operations

- Operators
  - & and
  - | or
  - ~ complement
  - ^ xor (exclusive or)
  - << left shift
  - >> right shift
- Integers only
- bin() displays in binary

Python has bitwise operations to compare individual bits in an integer. This is sometimes used for flags, or for packing more information into a byte. (Instead of using a 32-bit integer to store that something is true, you can use one bit.

The and and or operators work on two integers of the same size, and return a new integer with the bits modified according to the operator. The and operator can be used to clear bits. It is typically used with a *mask* composed of ones for all the bits you don't want to clear.

When comparing two numbers, the and operator, **&**, sets a result bit to 1 (one) if the corresponding bits in both numbers are both set to one. Otherwise, it sets the result bit to 0 (zero).

The or operator, |, sets a result bit to 1 (one) if *either* of the corresponding bits in both numbers are set to one. Otherwise, it sets the result bit to 0 (zero).

The complement operator, ~ reverses the values of bits — i.e., it changes ones to zeros and zeros to ones.

The xor operator, ^, sets a result bit to 1 (one) if the corresponding bits have *different* values.. Otherwise, it sets the result bit to 0 (zero).

The left shift operator, <<, moves bits to the left, a specified number of places.

The right shift operator, >>, works like left shift, but moves bits to the right.

Table 4. Struct bitwise operators

operator	**meaning**	**a**	**b**	**a** *op* **b**
&	and	1	1	1
		1	0	0
		0	1	0
		0	0	0
|	or	1	1	1
		1	0	1
		0	1	1
		0	0	0
^	xor (exclusive or)	1	1	0
		1	0	1
		0	1	1
		0	0	0
~	complement			
<<	left shift			
>>	right shift			

Example

bitwise_ops.py

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

a = 0b10101010  ①
b = 0b11110000

c = a & b  ②
print("  {:08b}".format(a))
print("& {:08b}".format(b))
print("  --------")
print("  {:08b}".format(c))
print()

c = a | b ③
print("  {:08b}".format(a))
print("| {:08b}".format(b))
print("  --------")
print("  {:08b}".format(c))
print()

c = a ^ b ④
print("  {:08b}".format(a))
print("^ {:08b}".format(b))
print("  --------")
print("  {:08b}".format(c))
print()

c = ~a ⑤
print(" ~ {:09b}".format(a))
print("   {:09b}".format(c))
print()

c = a >> 1 ⑥
print("{:08b} >> 1".format(a))
print("{:08b}".format(c))
print()

c = a >> 3 ⑦
print("{:08b} >> 3".format(a))
print("{:08b}".format(c))
print()

c = a << 1 ⑧
print("{:012b} << 1".format(a))
print("{:012b}".format(c))
```

```
print()

c = a << 3 ⑨
print("{:012b} << 3".format(a))
print("{:012b}".format(c))
print()
```

① a and b are integers

② bitwise AND

③ bitwise OR

④ bitwise XOR

⑤ complement (flip bit values)

⑥ shift right 1 bit

⑦ shift right 3 bits

⑧ shift left 1 bit

⑨ shift left 3 bits

bitwise_ops.py

```
  10101010
& 11110000
  --------
  10100000

  10101010
| 11110000
  --------
  11111010

  10101010
^ 11110000
  --------
  01011010

 ~ 010101010
   -10101011

10101010 >> 1
01010101

10101010 >> 3
00010101

000010101010 << 1
000101010100

000010101010 << 3
010101010000
```

Chapter 2 Exercises

Exercise 2-1 (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 2-2 (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 3: Regular Expressions

Objectives

- Creating regular expression objects
- Matching, searching, replacing, and splitting text
- Adding options to a pattern
- Replacing text with callbacks
- Specifying capture groups
- Using RE patterns without creating objects

Regular Expressions

- Specialized language for pattern matching
- Begun in UNIX; expanded by Perl
- Python adds some conveniences

> Regular expressions (or REs) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as Does this string match the pattern?", or Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string or to split it apart in various ways.
>
> — Python Regular Expression HOWTO

Regular expressions were first popularized thirty years ago as part of Unix text processing programs such as **vi**, **sed**, and **awk**. While they were improved incrementally over the years, it was not until the advent of Perl that they substantially changed from the originals. Perl added extensions of several different kinds – shortcuts for common sequences, look-ahead and look-behind assertions, non-greedy repeat counts, and a general syntax for embedding special constructs within the regular expression itself.

Python uses Perl-style regular expressions (AKA PCREs) and adds a few extensions of its own.

RE Syntax Overview

- Regular expressions contain branches
- Branches contain atoms
- Atoms may be quantified
- Branches and atoms may be anchored

A regular expression consists of one or more branches separated by the pipe symbol. The regular expression matches any text that is matched by any of the branches.

A branch is a left-to-right sequence of atoms. Each atom consists of either a one-character match or a parenthesized group. Each atom can have a quantifier (repeat count). The default repeat count is one.

A branch can be anchored to the beginning or end of the text. Any part of a branch can be anchored to the beginning or end of a word.

TIP There is frequently only one branch.

Two good web apps for working with Python regular expressions are
https://regex101.com/#python
http://www.pythex.org/

Table 5. Regular Expression Metacharacters

Pattern	Description
.	any character
[abc]	any character in set
[^abc]	any character not in set
\w,\W	any word, non-word char
\d,\D	any digit, non-digit
\s,\S	any space, non-space char
^,$	beginning, end of string
\b	beginning or end of word
\	escape a special character
*,+,?	0 or more, 1 or more, 0 or 1
{m}	exactly m occurrences
{m,}	at least m occurrences
{m,n}	m through n occurrences
a|b	match a or b
(?aiLmsux)	Set the A, I, L, M, S, U, or X flag for the RE (see below).
(?:…)	Non-grouping version of regular parentheses.
(?P<name>…)	The substring matched by the group is accessible by name.
(?P=name)	Matches the text matched earlier by the group named name.
(?#…)	A comment; ignored.
(?=…)	Matches if … matches next, but doesn’t consume the string.
(?!…)	Matches if … doesn’t match next.
(?⇐…)	Matches if preceded by … (must be fixed length).
(?<!…)	Matches if not preceded by … (must be fixed length).

Finding matches

- Module defines static functions
- Arguments: pattern, string

There are three primary methods for finding matches.

re.search(pattern, string)

Searches s and returns the first match. Returns a match object (**SRE_Match**) on success or **None** on failure. A match object is always evaluated as **True**, and so can be used in **if** statements and **while** loops. Call the **group()** method on a match object to get the matched text.

re.finditer(pattern, string)

Provides a match object for each match found. Normally used with a **for** loop.

re.findall(pattern, string)

Finds all matches and returns a list of matched strings.

Since regular expressions generally contain many backslashes, it is usual to specify the pattern with a raw string.

Other match methods

re.match() is like **re.search()**, but searches for the pattern at beginning of s. There is an implied ^ at the beginning of the pattern.

Likewise **re.fullmatch()** only succeeds if the pattern matches the entire string. ^ and $ around the pattern are implied.

Use the search() method unless you only want to match the beginning of the string.

Example

regex_finding_matches.py

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

import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'  ①

if re.search(pattern, s):  ②
    print("Found pattern.")
print()

m = re.search(pattern, s)  ③
print(m)
if m:
    print("Found:", m.group(0))  ④
print()

for m in re.finditer(pattern, s):  ⑤
    print(m.group())
print()

matches = re.findall(pattern, s)  ⑥
print("matches:", matches)
```

① store pattern in raw string

② search returns True on match

③ search actually returns match object

④ group(0) returns text that was matched by entire expression (or just m.group())

⑤ iterate over all matches in string:

⑥ return list of all matches

regex_finding_matches.py

```
Found pattern.

<re.Match object; span=(12, 17), match='M-302'>
Found: M-302

M-302
H-476
Q-51
A-110
H-332
Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

RE Objects

- **re** object contains a compiled regular expression
- Call methods on the object, with strings as parameters.

An **re** object is created by calling the compile() function, from the **re** module, with a pattern string. Once created, the object can be used for searching (matching), replacing, and splitting any string. The re.compile() function has an optional argument for flags which enable special features or fine-tune the match.

TIP	It is generally a good practice to create your re objects in a location near the top of your script, and then use them as necessary

Example

regex_objects.py

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

import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'[A-Z]-\d{2,3}')  ①

if rx_code.search(s):  ②
    print("Found pattern.")
print()

m = rx_code.search(s)
if m:
    print("Found:", m.group())
print()

for m in rx_code.finditer(s):
    print(m.group())
print()

matches = rx_code.findall(s)
print("matches:", matches)
```

① Create an re (regular expression) object

② Call search() method from the object

regex_objects.py

```
Found pattern.

Found: M-302

M-302
H-476
Q-51
A-110
H-332
Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

Compilation Flags

- Control match
- Add features

When compiling a pattern, you can specify various flags to control how the match occurs. The flags are aliases for numeric values, and can be combined by ORing them together Each flag has a short for and a long form.

re.I, re.IGNORECASE

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, [A-Z] will match lowercase letters, too, and Spam will match "Spam", "spam", or "spAM". This lower-casing doesn't take the current locale into account; it will if you also set the LOCALE flag.

re.L, re.LOCALE

Make \w, \W, \b, and \B, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write \w+ to match words, but \w only matches the character class [A-Za-z]; it won't match "é" or "ç". If your system is configured properly and a French locale is selected, certain C functions will tell the program that "é" should also be considered a letter. Setting the LOCALE flag enables \w+ to match French words as you'd expect.

re.M, re.MULTILINE

Usually ^ matches only at the beginning of the string, and $ matches only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, ^ matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the $ metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

re.S, re.DOTALL

Makes the "." special character match any character at all, including a newline; without this flag, "." will match anything except a newline.

re.X, re.VERBOSE

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. It also enables you to put comments within a RE that will be ignored by the engine; comments are marked by a "#" that's neither in a character class or preceded by an unescaped backslash. Use a triple-quoted string for your pattern to make best advantage of this flag.

Example

regex_flags.py

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

import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'

if re.search(pattern, s, re.IGNORECASE):  ①
    print("Found pattern.")
print()

m = re.search(pattern, s, re.I | re.M)  ②
if m:
    print("Found:", m.group())
print()

for m in re.finditer(pattern, s, re.I):
    print(m.group())
print()

matches = re.findall(pattern, s, re.I)
print("matches:", matches)
```

① make search case-insensitive

② short version of flag

regex_flags.py

```
Found pattern.

Found: M-302

M-302
r-99
H-476
Q-51
z-883
A-110
H-332
Y-45

matches: ['M-302', 'r-99', 'H-476', 'Q-51', 'z-883', 'A-110', 'H-332', 'Y-45']
```

Groups

- Marked with parentheses
- Capture whatever matched pattern within
- Access with match.group()

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a ":". This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value.

Groups are marked with parentheses, and 'capture' whatever matched the pattern inside the parentheses.

re.findall() returns a list of tuples, where each tuple contains the match for each group.

To access groups in more detail, use **finditer()** and call the **group()** method on each match object. The default group is 0, which is always the entire match. It can be retrieved with either ***match*.group(0)**, or just ***match*.group()**. ***match*.group(1)** returns text matched by the first set of parentheses, ***match*.group(2)** returns the text from the second set, etc.

In the same vein, ***match*.start()** or ***match*.start(0)** return the beginning 0-based offset of the entire match; ***match*.start(1)** returns the beginning offset of group 1, and so forth. The same is true for ***match*.end()** and ***match*.end(n)**.

***match*.span()** returns the the start and end offsets for the entire match. ***match*.span(1)** returns start and end offsets for group 1, and so forth.

Example

regex_group.py

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

import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'([A-Z])-(\d{2,3})'  ①

for m in re.finditer(pattern, s):
    print(m.group(0), m.group(1), m.group(2))  ②
    print(m.start(1), m.end(1), m.span())
print()

matches = re.findall(pattern, s)  ③
print("matches:", matches)
```

① parens delimit groups

② group 1 is first group, etc. (group 0 is entire match)

③ findall() returns list of tuples containing groups

regex_group.py

```
M-302 M 302
12 13 (12, 17)
H-476 H 476
102 103 (102, 107)
Q-51 Q 51
134 135 (134, 138)
A-110 A 110
398 399 (398, 403)
H-332 H 332
436 437 (436, 441)
Y-45 Y 45
470 471 (470, 474)

matches: [('M', '302'), ('H', '476'), ('Q', '51'), ('A', '110'), ('H', '332'), ('Y',
'45')]
```

Special Groups

- Non-capture groups are used just for grouping
- Named groups allow retrieval of sub-expressions by name rather than number
- Look-ahead and look-behind match, but do not capture

There are two variations on RE groups that are useful. If the first character inside the group is a question mark, then the parentheses contain some sort of extended pattern, designated by the next character after the question mark. The most basic is (?:pattern), which groups but does not capture.

A welcome addition in Python is the concept of named groups. Instead of remembering that the month is the 3rd group and the year is the 4th group, you can use the syntax (?P<name>pattern). You can then call match.group("name") to fetch the text match by that sub-expression; alternatively, you can call match.groupdict(), which returns a dictionary where the keys are the pattern names, and the values are the text matched by each pattern.

Another advanced concept is an assertion, either lookahead or lookbehind. A lookahead assertion uses the syntax (?=pattern). The string being matched must match the lookahead, but does not become part of the overall match.

For instance, "\d(?st|nd|rd|th)(?=street)" matches "1st", "2nd", etc., but only where they are followed by "street".

Example

regex_special.py

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

import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'(?P<letter>[A-Z])-(?P<number>\d{2,3})'  ①

for m in re.finditer(pattern, s):
    print(m.group('letter'), m.group('number'))  ②
```

① Use (?<NAME>...) to name groups

② Use m.group(NAME) to retrieve text

regex_special.py

```
M 302
H 476
Q 51
A 110
H 332
Y 45
```

Replacing text

- Use RE.sub(replacement,string[,count])
- RE.subn() returns tuple with string and count

To find and replace text using a regular expression, use the sub() method. It takes the replacement text and the string to search as arguments, and returns the modified string.

The third, optional argument is the maximum number of replacements to make.

Be sure to put the arguments in the proper order!

Example

regex_sub.py

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

import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub("[REDACTED]", s) ①
print(s2)
print()

s3, count = rx_code.subn("___", s) ②
print("Made {} replacements".format(count))
print(s3)
```

① replace pattern with string

② subn returns tuple with result string and replacement count

regex_sub.py

```
lorem ipsum [REDACTED] dolor sit amet, consectetur [REDACTED] adipiscing elit, sed do
 eiusmod tempor incididunt [REDACTED] ut labore et dolore magna [REDACTED] aliqua. Ut
enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo [REDACTED]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat [REDACTED] cupidatat non proident, sunt in [REDACTED] culpa qui
officia deserunt [REDACTED] mollit anim id est laborum

Made 8 replacements
lorem ipsum ___ dolor sit amet, consectetur ___ adipiscing elit, sed do
 eiusmod tempor incididunt ___ ut labore et dolore magna ___ aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo ___  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat ___ cupidatat non proident, sunt in ___ culpa qui
officia deserunt ___ mollit anim id est laborum
```

Replacing with a callback

- Replacement can be function
- Function expects match object, returns replacement text
- Use normally defined function or a lambda

In addition to using a string as the replacement, you can specify a function. This function will be called once for each match, with the match object as its only parameter.

Whatever string the function returns will be used as the replacement text. This lets you have complete control over the replacement.

Using a callback makes it simple to:

- preservecase in a replacement
- add text around the replacement
- look up the text in a dictionary or database to find replacement text

Example

regex_sub_callback.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
 eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 mollit anim id est laborum"""

#  my $rx_code = qr/(?P<letter>[A-Z])(?P<number>\d{2,3})/i;
#  if ($foo ~= /$rx_code/) { }

rx_code = re.compile(r'(?P<letter>[A-Z])(?P<number>\d{2,3})', re.I)


def update_code(m):  ①
    letter = m.group('letter').upper()
    number = int(m.group('number'))
    return '{}:{:04d}'.format(letter, number)  ②


s2 = rx_code.sub(update_code, s)  ③
print(s2)
```

① callback function is passed each match object

② function returns replacement text

③ sub takes callback function instead of replacement text

regex_sub_callback.py

```
lorem ipsum M:0302 dolor sit amet, consectetur R:0099 adipiscing elit, sed do
 eiusmod tempor incididunt H:0476 ut labore et dolore magna Q:0051 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z:0883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U:0901 eu fugiat nulla pariatur.
Excepteur sint occaecat A:0110 cupidatat non proident, sunt in H:0332 culpa qui
officia deserunt Y:0045 mollit anim id est laborum
```

Splitting a string

- Syntax: re.split(string[,max])

The re.split() method splits a string into pieces, returning the pieces as a list. The optional max argument limits the numbers of pieces.

Example

regex_split.py

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

import re

rx_wordsep = re.compile(r"[^a-z]+", re.I)  ①

s1 = '''There are 10 kinds of people in a Binary world, I hear" -- Geek talk'''

words = rx_wordsep.split(s1)  ②
print(words)
```

① When splitting, pattern matches what you **don't** want

② Retrieve text *separated* by your pattern

regex_split.py

```
['There', 'are', 'kinds', 'of', 'people', 'in', 'a', 'Binary', 'world', 'I', 'hear',
'Geek', 'talk']
```

Chapter 3 Exercises

Exercise 3-1 (pyfind.py)

Write a script which takes two or more arguments. The first argument is the pattern to search for; the remaining arguments are files to search. For each file, print out all lines which match the pattern.1

Exercise 3-2 (mark_big_words.py)

Copy parrot.txt to bigwords.txt adding asterisks around all words that are 8 or more characters long.

HINT: Use the \b anchor to indicate beginning and end of a word.

Exercise 3-3 (print_numbers.py)

Write a script to print out all lines in custinfo.dat which contain phone numbers.

Exercise 3-4 (word_freq.py)

Write a script that will read a text file and print out a list of all the words in the file, normalized to lower case, and with the number of times that word occurred in the file. Use the regular expression [^\w']+ for splitting each line into words.

Test with any of the text files in the DATA folder.

Chapter 4: iPython and Jupyter

Objectives

- Learn the basics of iPython
- Apply magics
- Use Jupyter notebooks

About iPython

- Enhanced interpreter for exploratory computing
- Efficient for data analysis (what-if? Scenarios)
- Great for plotting-tweaking-plotting
- Good for simple code development
- Not so good for enterprise development

iPython is an enhanced interpreter for Python. It provides a large number of "creature comforts" for the user, such as name completion and improved help features.

It’s great for ad-hoc queries, what-if scenarios, when you are not developing a full application.

Features of iPython

- Name completion (variables, modules, methods, etc.)
- Autoindent
- Inline plots and other figures
- Dynamic introspection (dir() on steroids)
- Search namespaces with wildcards
- Auto-parentheses ('sin 3' becomes 'sin(3)'
- Auto-quoting (',foo a b' becomes 'foo("a","b")'
- Enhanced help system
- Commands are numbered (and persistent) for recall
- 'Magic' commands for controlling iPython itself
- Aliasing system for interpreter commands
- Easy access to shell commands
- Background execution in separate thread
- Simplified (and lightweight) persistence
- Session logging (can be saved as scripts)
- Detailed tracebacks when errors occur
- Session restoring (playback log to specific state)
- Flexible configuration system
- Easy access to Python debugger
- Simple profiling
- Interactive parallel computing (if supported by hardware)

The many faces of iPython

- Console
- Colorized console (default)
- QTConsole (obsolete)
- Jupyter notebook

iPython can be run in several different modes. The most basic mode is console, which runs from the command prompt, without syntax colorizing.

If your terminal supports it (and most do), iPython can run as a colorized console, using different colors for variables, functions, strings, comments, and so forth.

Both the default console and the colorized console display plots in a separate popup window.

If QT is installed, iPython can run a GUI console, which looks and acts like a regular text console, but allows plots to be generated inline, and has enhanced context-sensitive help. This mode is deprecated, as it is simpler to just run a notebook.

The most flexible and powerful way to run iPython is in notebook mode. This mode starts a dedicated web server and begins a session using your default web browser. Other users can load notebooks from the notebook server, similarly to MatLab. The notebook part of iPython was split out and is now part of the Jupyter Project.

NOTE iPython also supports parallel computing, which will not be covered here.

Starting iPython

- Type ipython at the command line
- Huge number of options

To get started with iPython in console mode, just type ipython at the command line, or double-click the icon in Windows explorer.

In general, it works like the normal interpreter, but with many more features.

There is a huge number of options. To see them all, invoke iPython with the --help-all option:

```
ipython help-all
```

Getting Help

- ? basic help
- %quickref quick reference
- help standard Python help
- thing? help on thing

iPython provides help in several ways.

Typing **?** at the prompt will display an introduction to iPython and a feature overview.

For a quick reference card, type %quickref.

To start Python's normal help system, type help.

For help on any Python object, type `object?` or `?object`. This is similar to saying help("object") in the default interpreter, but is "smarter".

TIP For more help, add a second question mark. This does not work for all objects.

Tab Completion

- Press tab to complete
- keywords
- modules
- methods and attributes
- parameters to functions
- file and directory names

Pressing the <TAB> key will invoke autocomplete. If there is only one possible completion, it will be expanded. If there is more than one completion that will match, iPython will display a list of possible completions.

Autocomplete works on keywords, functions, classes, methods, and object attributes, as well as paths from your file system.

Magic Commands

- Start with % (line magic) or %% (cell magic)
- Simplify common tasks
- Use %lsmagic to list all magic commands

One of the enhancements in iPython is the set of "magic" commands. These are meta-commands that help you manipulate the iPython environment.

Normal magics apply to a single line. Cell magics apply to a cell (a group of lines).

For instance, `%history` will list previous commands.

TIP	Type lsmagic for a list of all magics

Benchmarking

- Use %timeit

iPython has a handy magic for benchmarking.

```
In [1]: color_values = { 'red':66, 'green':85, 'blue':77 }

In [2]: %timeit red_value = color_values['red']
10000000 loops, best of 3: 54.5 ns per loop

In [3]: %timeit red_value = color_values.get('red')
10000000 loops, best of 3: 115 ns per loop
```

%timeit will benchmark whatever code comes after it on the same line. %%timeit will benchmark contents of a notebook cell

External commands

- Precede command with !
- Can assign output to variable

Any shell command can be run by starting it with a !.

Windows

```
In [3]: !dir DATA\*.csv
 Volume in drive Z is Shared Folders
 Volume Serial Number is 0000-0064

 Directory of Z:\Desktop\py2forsci\DATA

02/20/2014  01:53 PM             5,511 airport_boardings.csv
02/20/2014  01:53 PM             2,182 energy_use_quad.csv
02/20/2014  01:53 PM             4,993 parasite_data.csv
               3 File(s)         12,686 bytes
               0 Dir(s)  352,625,324,032 bytes free

In [4]:
```

Non-Windows (Linux, OS X, etc)

```
In [2]: !ls -l DATA/*.csv
-rwxr-xr-x  1 jstrick  staff  5511 Jan 27 19:44 DATA/airport_boardings.csv
-rwxr-xr-x  1 jstrick  staff  2182 Jan 27 19:44 DATA/energy_use_quad.csv
-rwxr-xr-x  1 jstrick  staff  4642 Jan 27 19:44 DATA/parasite_data.csv

In [3]:
```

Jupyter notebooks

- Extension of iPython
- Puts the interpreter in a web browser
- Code is grouped into "cells"
- Cells can be edited, repeated, etc.

In 2015, the developers of iPython pulled the notebook feature out of iPython to make a separate product called Jupyter. It is still invoked via the ipython command, but now supports over 130 language kernels in addition to Python.

A Jupyter notebook is a journal-like python interpreter that lives in a browser window. Code is grouped into cells, which can contain multiple statements. Cells can be edited, repeated, rearranged, and otherwise manipulated.

A notebook (i.e, a set of cells, can be saved, and reopened). Notebooks can be shared among members of a team via the notebook server which is built into Jupyter.

Jupyter Demo

At this point please start the Jupyter notebook server and follow along with a demo of Jupyter notebooks as directed by the instructor

Open an Anaconda prompt and navigate to the top folder of the student files, then

```
cd NOTEBOOKS
jupyter notebook
```

Chapter 5: Introduction to NumPy

Objectives

- See the "big picture" of NumPy
- Create and manipulate arrays
- Learn different ways to initialize arrays
- Understand the NumPy data types available
- Work with shapes, dimensions, and ranks
- Broadcast changes across multiple array dimensions
- Extract multidimensional slices
- Perform matrix operations

```
// or maybe one of the standard datasets, like irises
```

Python’s scientific stack

- NumPy, SciPy, MatPlotLib (and many others)
- Python extensions, written in C/Fortran
- Support for math, numerical, and scientific operations

NumPy is part of what is sometimes called Python’s "scientific stack". Along with SciPy, Matplotlib, and other libraries, it provides a broad range of support for scientific and engineering tasks.

SciPy is a large group of mathematical algorithms, along with some convenience functions, for doing scientific and engineering calculations, including data science. SciPy routines accept and return NumPy arrays.

pandas ties some of the libraries together, and is frequently used interactively via **iPython** in a **Jupyter** notebook. Of course you can also create scripts using any of the scientific libraries.

NOTE	There is not an integrated *application* for all of the Python scientific libraries.

NumPy overview

- Install numpy module from numpy.scipy.org (included with Anaconda)
- Basic object is the array
- Up to 100x faster than normal Python math operations
- Functional-based (fewer loops)
- Dozens of utility functions

The basic object that NumPy provides is the array. Arrays can have as many dimensions as needed. Working with NumPy arrays can be 100 times faster than working with normal Python lists.

Operations are applied to arrays in a functional manner – instead of the programmer explicitly looping through elements of the array, the programmer specifies an expression or function to be applied, and the array object does all of the iteration internally.

There are many utility functions for accessing arrays, for creating arrays with specified values, and for performing standard numerical operations on arrays.

To get started, import the **numpy** module. It is conventional to import numpy as np. The examples in this chapter will follow that convention.

NumPy and the rest of the Python scientific stack is included with the Anaconda, Canopy, Python(x,y), and WinPython bundles. If you are not using one of these, install NumPy with

```
pip install numpy
```

NOTE all top-level NumPy routines are also available directly through the scipy package.

Creating Arrays

- Create with
  - array() function initialized with nested sequences
  - Other utilities (arange(), zeros(), ones(), empty()
- All elements are same type (default float)
- Useful properties: ndim, shape, size, dtype
- Can have any number of axes (dimensions)
- Each axis has a length

An array is the most basic object in NumPy. It is a table of numbers, indexed by positive integers. All of the elements of an array are of the same type.

An array can have any number of dimensions; these are referred to as axes. The number of axes is called the rank.

Arrays are rectangular, not ragged.

One way to create an array is with the `array()` function, which can be initialized from existing arrays.

The `zeros()` function expects a *shape* (tuple of axis lengths), and creates the corresponding array, with all values set to zero. The `ones()` function is the same, but initializes with ones.

The `full()` function expects a shape and a value. It creates the array, putting the specified value in every element.

The `empty()` function creates an array of specified shape initialized with random floats.

However, the most common way to crate an array is by loading data from a text or binary file.

When you print an array, NumPy displays it with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

NOTE the `ndarray()` object is initialized with the *shape*, not the *data*.

Example

np_create_arrays.py

```
import numpy as np

a = np.array([[1, 2.1, 3], [4, 5, 6], [7, 8, 9], [20, 30, 40]])  ①
print(a)
print("# dims", a.ndim)  ②
print("shape", a.shape)  ③
print()

a_zeros = np.zeros((3, 5), dtype=np.uint32)  ④
print(a_zeros)
print()

a_ones = np.ones((2, 3, 4, 5))  ⑤
print(a_ones)
print()

# with uninitialized values
a_empty = np.empty((3, 8))  ⑥
print(a_empty)

print(a.dtype)  ⑦

nan_array = np.full((5, 10), np.NaN)  ⑧
print(nan_array)
```

① create array from nested sequences

② get number of dimensions

③ get shape

④ create array of specified shape and datatype, initialized to zeroes

⑤ create array of specified shape, initialized to ones

⑥ create uninitialized array of specified shape

⑦ defaults to float64

⑧ create array of NaN values

np_create_arrays.py

```
[[ 1.   2.1  3. ]
 [ 4.   5.   6. ]
 [ 7.   8.   9. ]
 [20.  30.  40. ]]
# dims 2
shape (4, 3)

[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]

[[[[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]]


 [[[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]

  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]]]

[[ 3.10503618e+231  3.10503618e+231  2.24254314e-314  6.93715575e-310
   6.93715579e-310  6.93715579e-310  6.93716899e-310  6.93716902e-310]
 [ 6.93715579e-310  6.93715578e-310  3.10503618e+231 -1.29073996e-231
   1.69759663e-313  0.00000000e+000  0.00000000e+000  0.00000000e+000]
```

```
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000  0.00000000e+000
   0.00000000e+000  0.00000000e+000  3.10503618e+231 -2.00389715e+000]]
float64
[[nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan nan]]
```

Creating ranges

- Similar to builtin `range()`
- Returns a one-dimensional NumPy array
- Can use floating point values
- Can be reshaped

The `arange()` function takes a size, and returns a one-dimensional NumPy array. This array can then be reshaped as needed. The start, stop, and step parameters are similar to those of `range()`, or Python slices in general. Unlike the builtin Python `range()`, start, stop, and step can be floats.

The `linspace()` function creates a specified number of equally-spaced values. As with `numpy.arange()`, start and stop may be floats.

The resulting arrays can be reshaped into multidimensional arrays.

Example

np_create_ranges.py

```
#!/usr/bin/env python
import numpy as np

r1 = np.arange(50)  ①
print(r1)
print("size is", r1.size)  ②
print()

r2 = np.arange(5, 101, 5)  ③
print(r2)
print("size is", r2.size)
print()

r3 = np.arange(1.0, 5.0, .3333333)  ④
print(r3)
print("size is", r3.size)
print()

r4 = np.linspace(1.0, 2.0, 10)  ⑤
print(r4)
print("size is", r4.size)
print()
```

① create range of ints from 0 to 49

② size is 50

③ create range of ints from 5 to 100 counting by 5

④ start, stop, and step may be floats

⑤ 10 equal steps between 1.0 and 2.0

np_create_ranges.py

```
[ 0  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 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49]
size is 50

[  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
  95 100]
size is 20

[1.        1.3333333 1.6666666 1.9999999 2.3333332 2.6666665 2.9999998
 3.3333331 3.6666664 3.9999997 4.333333  4.6666663 4.9999996]
size is 13

[1.         1.11111111 1.22222222 1.33333333 1.44444444 1.55555556
 1.66666667 1.77777778 1.88888889 2.        ]
size is 10
```

Working with arrays

- Use normal math operators (+, -, /, and *)
- Use NumPy's builtin functions
- By default, apply to every element
- Can apply to single axis
- Operations on between two arrays applies operator to pairs of element

The array object is smart about applying functions and operators. A function applied to an array is applied to every element of the array. An operator applied to two arrays is applied to corresponding elements of the two arrays.

In-place operators (+=, *=, etc) efficiently modify the array itself, rather than returning a new array.

Example

np_basic_array_ops.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [
        [5, 10, 15],
        [2, 4, 6],
        [3, 6, 9, ],
    ]
)  ①

b = np.array(
    [
        [10, 85, 92],
        [77, 16, 14],
        [19, 52, 23],
    ]
)  ②
print("a")
print(a)
print()
print("b")
print(b)
print()

print("a * 10")
print(a * 10)  ③
print()

print("a + b")
print(a + b)  ④
print()

print("b + 3")
print(b + 3)  ⑤
print()

s1 = a.sum()  ⑥
s2 = b.sum()  ⑥
print("sum of a is {0}; sum of b is {1}".format(s1, s2))
print()

a += 1000  ⑦
print(a)
```

① create 2D array

② create another 2D array

③ multiply every element by 10 (not in place)

④ add every element of a to the corresponding element of b

⑤ add 3 to every element of b

⑥ calculate sum of all elements

⑦ add 1000 to every element of a (in place)

np_basic_array_ops.py

```
a
[[ 5 10 15]
 [ 2  4  6]
 [ 3  6  9]]

b
[[10 85 92]
 [77 16 14]
 [19 52 23]]

a * 10
[[ 50 100 150]
 [ 20  40  60]
 [ 30  60  90]]

a + b
[[ 15  95 107]
 [ 79  20  20]
 [ 22  58  32]]

b + 3
[[13 88 95]
 [80 19 17]
 [22 55 26]]

sum of a is 60; sum of b is 388

[[1005 1010 1015]
 [1002 1004 1006]
 [1003 1006 1009]]
```

Shapes

- Number of elements on each axis
- array.shape has shape tuple
- Assign to array.shape to change
- Convert to one dimension
  - array.ravel()
  - array.flatten()
- array.transpose() to flip the shape

Every array has a shape, which is the number of elements on each axis. For instance, an array might have the shape (3,5), which means that there are 3 rows and 5 columns.

The shape is stored as a tuple, in the shape attribute of an array. To change the shape of an array, assign to the shape attribute.

The `ravel()` and `flatten()` methods will flatten any array into a single dimension. `ravel()` returns a "view" of the original array, while `flatten()` returns a new array. If you modify the result of ravel(), it will modify the original data.

The `transpose()` method will flip shape (x,y) to shape (y,x). It is equivalent to `array.shape = list(reversed(array.shape))`.

Example

np_shapes.py

```
#!/usr/bin/env python
import numpy as np

a1 = np.arange(15)  ①
print("a1 shape", a1.shape)  ②
print()

print(a1)
print()

a1.shape = 3, 5  ③
print(a1)
print()

a1.shape = 5, 3  ④
print(a1)
print()

print(a1.flatten())  ⑤
print()

print(a1.transpose())  ⑥
print("------------------")

a2 = np.arange(40)  ⑦
a2.shape = 2, 5, 4  ⑧

print(a2)
print()
```

① create 1D array

② get shape

③ reshape to 3x5

④ reshape to 5x3

⑤ print array as 1D

⑥ print transposed array

⑦ create 1D array

⑧ reshape go 2x5x4

np_shapes.py

```
a1 shape (15,)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  3  6  9 12]
 [ 1  4  7 10 13]
 [ 2  5  8 11 14]]
------------------
[[[ 0  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]
  [32 33 34 35]
  [36 37 38 39]]]
```

Slicing and indexing

- Simple indexing similar to lists
- start, stop, step
- start is INclusive, stop is Exclusive
- : used for range for one axis
- ... means "as many : as needed"

NumPy arrays can be indexed and sliced like regular Python lists, but with some convenient extensions. Instead of `[x][y]`, NumPy arrays can be indexed with `[x,y]`. Within an axis, ranges can be specified with slice notation (start:stop:step) as usual.

For arrays with more than 2 dimensions, `⋯` can be used to mean "and all the other dimensions".

Example

np_indexing.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
)  ①

print(a)
print()

print('a[0] =>', a[0])  ②
print('a[0][0] =>', a[0][0])  ③
print('a[0,0] =>', a[0, 0])  ④
print('a[0,:3] =>', a[0, :3])  ⑤
print('a[0,::2] =>', a[0, ::2])  ⑥
print()
print('a[::2] =>', a[::2])  ⑦
print()
print('a[:3, -2:] =>', a[:3, -2:])  ⑧
```

① sample data

② first row

③ first element of first row

④ same, but numpy style

⑤ first 3 elements of first row

⑥ every second element of first row

⑦ every second row

⑧ every third element of every second row

np_indexing.py

```
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

a[0] => [70 31 21 76 19  5 54 66]
a[0][0] => 70
a[0,0] => 70
a[0,:3] => [70 31 21]
a[0,::2] => [70 21 19 54]

a[::2] => [[70 31 21 76 19  5 54 66]
 [11 84  7 10 31 50 11 98]
 [75 37 11 62 35 76 38  4]]

a[:3, -2:] => [[54 66]
 [65 73]
 [11 98]]
```

Indexing with Booleans

- Apply relational expression to array
- Result is array of Booleans
- Booleans can be used to index original array

If a relational expression (>, <, >=, ⇐) is applied to an array, the result is a new array containing Booleans reflecting whether the expression was true for each element. That is, for each element of the original array, the resulting array is set to True if the expression is true for that element, and False otherwise.

The resulting Boolean array can then be used as an index, to modify just the elements for which the expression was true.

Example

np_bool_indexing.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
)  ①

print('a =>', a, '\n')

i = a > 50  ②
print('i (a > 50) =>', i, '\n')

print('a[i] =>', a[i], '\n')  ③

print('a[a > 50] =>', a[a > 50], '\n')  ④

print('a[i].min(), a[i].max() =>', a[i].min(), a[i].max(), '\n')  ⑤

a[i] = 0  ⑥
print('a =>', a, '\n')

print("a[a < 15] += 10")
a[a < 15] += 10  ⑦
print(a, '\n')
```

① sample data

② create Boolean mask

③ print elements of a that are > 50 using mask

④ same, but without creating a separate mask

⑤ min and max values of result set with values less than 50

⑥ set elements with value > 50 to 0

⑦ add 10 to elements < 15

np_bool_indexing.py

```
a => [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

i (a > 50) => [[ True False False  True False False  True  True]
 [False False  True False False  True  True  True]
 [False  True False False False False False  True]
 [False False False False False  True False  True]
 [ True False False  True False  True False False]]

a[i] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[a > 50] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[i].min(), a[i].max() => 52 98

a => [[ 0 31 21  0 19  5  0  0]
 [23 29  0 12 27  0  0  0]
 [11  0  7 10 31 50 11  0]
 [25 13 43  1 31  0 41  0]
 [ 0 37 11  0 35  0 38  4]]

a[a < 15] += 10
[[10 31 21 10 19 15 10 10]
 [23 29 10 22 27 10 10 10]
 [21 10 17 20 31 50 21 10]
 [25 23 43 11 31 10 41 10]
 [10 37 21 10 35 10 38 14]]
```

Selecting rows based on conditions

- Index with boolean expressions
- Use **&**, not **and**

To select rows from an array, based on conditions, you can index the array with two or more Boolean expressions.

Since the Boolean expressions return arrays of True/False values, use the **&** bitwise AND operator (or | for OR).

Any number of conditions can be applied this way.

```
new_array = old_array[bool_expr1 & bool_expr2 ...]
```

Example

np_select_rows.py

```
#!/usr/bin/env python
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

print("first 5 rows of sample_data:")
print(sample_data[:5, :], '\n')

selected = sample_data[  ②
    (sample_data[:, 0] < 10) &  ③
    (sample_data[:, -1] > 35)
]

print("selected")
print(selected)
```

① Read some data into 2d array

② Index into the existing data

③ Combine two Boolean expressions with **&**

np_select_rows.py

```
first 5 rows of sample_data:
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 [29.  8. 40. 59. 10. 26.]
 [54.  9. 68.  4. 16. 21.]]

selected
[[ 8. 49.  2. 40. 50. 36.]
 [ 4. 49. 39. 50. 23. 39.]
 [ 6.  7. 40. 56. 31. 38.]
 [ 6.  1. 44. 55. 49. 36.]
 [ 5. 22. 45. 49. 10. 37.]]
```

Stacking

- Combining 2 arrays vertically or horizontally
- use `vstack()` or `hstack()`
- Arrays must have compatible shapes

You can combine two or more arrays vertically or horizontally with the `vstack()` or `hstack()` functions. These functions are also handy for adding rows or columns with the results of operations.

Example

np_stacking.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73]]
)  ①

b = np.array(
    [[11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90]]
)  ②

print('a =>\n', a)
print()
print('b =>\n', b)
print()
print('vstack((a,b)) =>\n', np.vstack((a, b)))  ③
print()

print('vstack((a,a[0] + a[1])) =>\n', np.vstack((a, a[0] + a[1])))  ④
print()

print('hstack((a,b)) =>\n', np.hstack((a, b)))  ⑤
```

① sample array a

② sample array b

③ stack arrays vertically (like pancakes)

④ add a row with sums of first two rows

⑤ stack arrays horizontally (like books on a shelf)

np_stacking.py

```
a =>
 [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]]

b =>
 [[11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,b)) =>
 [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,a[0] + a[1])) =>
 [[ 70  31  21  76  19   5  54  66]
 [ 23  29  71  12  27  74  65  73]
 [ 93  60  92  88  46  79 119 139]]

hstack((a,b)) =>
 [[70 31 21 76 19  5 54 66 11 84  7 10 31 50 11 98]
 [23 29 71 12 27 74 65 73 25 13 43  1 31 52 41 90]]
```

ufuncs and builtin operators

- Builtin functions for efficiency
- Map over array
- No **for** loops
- Use **vectorize()** for custom ufuncs

In normal Python, you are used to iterating over arrays, especially nested arrays, with a **for** loop. However, for large amounts of data, this is slow. The reason is that the interpreter must do type-checking and lookups for each item being looped over.

NumPy provides *vectorized* operations which are implemented by *ufuncs* — universal functions. ufuncs are implemented in C and work directly on NumPy arrays. When you use a normal math operator (+ - * /, etc) on a NumPy array, it calls the underlying ufunc. For instance, `array1 + array2` calls `np.add(array1, array2)`.

There are over 60 ufuncs built into NumPy. These normally return a NumPy array with the results of the operation. Some have options for putting the output into a different object.

The official docs for ufuncs are here: https://docs.scipy.org/doc/numpy/reference/ufuncs.html

You can scroll down to the list of available ufuncs.

Vectorizing functions

- Many functions "just work"
- `np.vectorize()` allows user-defined function to be broadcast.

ufuncs will automatically be broadcast across any array to which they are applied. For user-defined functions that don’t correctly broadcast, NumPy provides the **vectorize()** function. It takes a function which accepts one or more scalar values (float, integers, etc.) and returns a single scalar value.

Example

np_vectorize.py

```
#!/usr/bin/env python
import time
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
)

def set_default(value, limit, default): ②
    if value > limit:
        value = default

    return value

MAX_VALUE = 50      ③
DEFAULT_VALUE = -1  ④

print("Version 1: looping over arrays")
start = time.time() ⑤
try:
    version1_array = np.zeros(sample_data.shape, dtype=int)  ⑥
    for i, row in enumerate(sample_data):  ⑦
        for j, column in enumerate(row):
            version1_array[i, j] = set_default(sample_data[i, j], MAX_VALUE,
DEFAULT_VALUE)   ⑧
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.time()  ⑨
    elapsed = end - start  ⑩
    print(version1_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()


print("Version 2: broadcast without vectorize()")
start = time.time()
try:
    print("Without sp.vectorize:")
    version2_array = set_default(sample_data, MAX_VALUE, DEFAULT_VALUE) ⑪
except ValueError as err:
    print("Function failed:", err)
```

```
else:
    end = time.time()
    elapsed = end - start
    print(version2_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()

print("Version 3: broadcast with vectorize()")
set_default_vect = np.vectorize(set_default)  ⑫

start = time.time()
try:
    print("With sp.vectorize:")
    version3_array = set_default_vect(sample_data, MAX_VALUE, DEFAULT_VALUE) ⑬
except ValueError as err:
    print("Function failed:", err)
else:
    end = time.time()
    elapsed = end - start
    print(version3_array)
    print("took {:.5f} seconds".format(elapsed))
finally:
    print()
```

① Create some sample data

② Define function with more than one parameter

③ Define max value

④ Define default value

⑤ Get the current time as Unix timestamp (large integer)

⑥ Create array to hold results

⑦ Iterate over rows and columns of input array

⑧ Call function and put result in new array

⑨ Get current time

⑩ Get elapsed number of seconds and print them out

⑪ Pass array to function; it fails because it has more than one parameter

⑫ Convert function to vectorized version — creates function that takes one parameter and has the other two "embedded" in it

⑬ Call vectorized version with same parameters

np_vectorize.py

```
Version 1: looping over arrays
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00483 seconds

Version 2: broadcast without vectorize()
Without sp.vectorize:
Function failed: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()

Version 3: broadcast with vectorize()
With sp.vectorize:
[[-1 -1 -1 -1 50  4]
 [40 -1  9 -1 -1 17]
 [18 23  2 -1  1  9]
 ...
 [26 20 -1 46 38 23]
 [ 9  5 -1 23  2 26]
 [46 34 25  8 39 34]]
took 0.00089 seconds
```

Getting help

- Several help functions
  - `numpy.info()`
  - `numpy.lookfor()`
  - `numpy.source()`

NumPy has several functions for getting help. The first is `numpy.info()`, which provides a brief explanation of a function, class, module, or other object as well as some code examples.

If you're not sure what function you need, you can try `numpy.lookfor()`, which does a keyword search through the NumPy documentation.

These functions are convenient when using **iPython** or **Jupyter**.

Example

np_info.py

```
#!/usr/bin/env python
import numpy as np
import scipy.fftpack as ff


def main():
    np.info(ff.fft)  ①

    print('-' * 60)

    np.source(ff.fft)  ②




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

① Get help on the fft() function

② View the source of the fft() function

Iterating

- Similar to normal Python
- Iterates through first dimension
- Use array.flat to iterate through all elements
- Don’t do it unless you have to

Iterating through a NumPy array is similar to iterating through any Python list; iteration is across the first dimension. Slicing and indexing can be used.

To iterate across every element, use `array.flat`.

However, iterating over a NumPy array is generally much less efficient than using a *vectorized* approach — calling a *ufunc* or directly applying a math operator. Some tasks may require it, but you should avoid it if possible.

Example

np_iterating.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76],
     [23, 29, 71, 12]]
)  ①

print('a =>\n', a)
print()

print("for row in a: =>")
for row in a:  ②
    print("row:", row)
print()

print("for column in a.T:")
for column in a.T:  ③
    print("column:", column)
print()

print("for elem in a.flat: =>")
for elem in a.flat:  ④
    print("element:", elem)
```

① sample array

② iterate over rows

③ iterate over columns by transposing the array

④ iterate over all elements (row-major)

np_iterating.py

```
a =>
 [[70 31 21 76]
 [23 29 71 12]]

for row in a: =>
row: [70 31 21 76]
row: [23 29 71 12]

for column in a.T:
column: [70 23]
column: [31 29]
column: [21 71]
column: [76 12]

for elem in a.flat: =>
element: 70
element: 31
element: 21
element: 76
element: 23
element: 29
element: 71
element: 12
```

Matrix Multiplication

- Use normal ndarrays
- Most operations same as ndarray
- Use @ for multiplication

For traditional matrix operations, use a normal ndarray. Most operations are the same as for ndarrays. For matrix (diagonal) multiplication, use the @ (matrix multiplication) operator.

For transposing, use *array*.transpose(), or just *array*.T.

NOTE	There was formerly a Matrix type in NumPy, but it is deprecated since the addition of the @ operator in Python 3.5

Example

np_matrices.py

```
#!/usr/bin/env python
import numpy as np

m1 = np.array(
    [[2, 4, 6],
     [10, 20, 30]]
)  ①

m2 = np.array([[1, 15],
               [3, 25],
               [5, 35]])  ②

print('m1 =>\n', m1)
print()

print('m2 =>\n', m2)
print()

print('m1 * 10 =>\n', m1 * 10)  ③
print()

print('m1 @ m2 =>\n', m1 @ m2)  ④
print()
```

① sample 2x3 array

② sample 3x2 array

③ multiply every element of m1 times 10

④ matrix multiply m1 times m2 — diagonal product

np_matrices.py

```
m1 =>
 [[ 2  4  6]
 [10 20 30]]

m2 =>
 [[ 1 15]
 [ 3 25]
 [ 5 35]]

m1 * 10 =>
 [[ 20  40  60]
 [100 200 300]]

m1 @ m2 =>
 [[  44  340]
 [ 220 1700]]
```

Data Types

- Default is **float**
- Data type is inferred from initialization data
- Can be specified with `arange()`, `ones()`, `zeros()`, etc.

Numpy defines around 30 numeric data types. Integers can have different sizes and byte orders, and be either signed or unsigned. The data type is normally inferred from the initialization data. When using `arange()`, `ones()`, etc., to create arrays, the **dtype** parameter can be used to specify the data type.

The default data type is **np.float_**, which maps to the Python builtin type **float**.

The data type cannot be changed after an array is created.

See https://numpy.org/devdocs/user/basics.types.html for more details.

Example

np_data_types.py

```
#!/usr/bin/env python
import numpy as np

r1 = np.arange(45)  ①
r1.shape = (3, 3, 5)  ②
print('r1 datatype:', r1.dtype)
print('r1 =>\n', r1, '\n')

r2 = np.arange(45.)  ③
r2.shape = (3, 3, 5)
print('r2 datatype:', r2.dtype)
print('r2 =>\n', r2, '\n')

r3 = np.arange(45, dtype=np.int16)  ③
r3.shape = (3, 3, 5)
print('r3 datatype:', r3.dtype)
print('r3 =>\n', r3, '\n')
```

① create array — arange() defaults to int

② create array — passing float makes all elements float

③ create array — set datatype to short int

np_data_types.py

```
r1 datatype: int64
r1 =>
 [[[ 0  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 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]

r2 datatype: float64
r2 =>
 [[[ 0.  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. 32. 33. 34.]
  [35. 36. 37. 38. 39.]
  [40. 41. 42. 43. 44.]]]

r3 datatype: int16
r3 =>
 [[[ 0  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 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]
```

Reading and writing Data

- Read data from files into **ndarray**
- Text files
  - `loadtxt()`
  - `savetxt()`
  - `genfromtxt()`
- Binary (or text) files
  - `fromfile()`
  - `tofile()`

NumPy has several functions for reading data into an array.

`numpy.loadtxt()` reads a delimited text file. There are many options for fine-tuning the import.

`numpy.genfromtxt()` is similar to `numpy.loadtxt()`, but also adds support for handling missing data

Both functions allow skipping rows, user-defined per-column converters, setting the data type, and many others.

To save an array as a text file, use the `numpy.savetxt()` function. You can specify delimiters, header, footer, and formatting.

To read binary data, use `numpy.fromfile()`. It expects a file to contain all the same data type, i.e., ints or floats of a specified type. It will default to floats. `fromfile()` can also be used to read text files.

To save as binary data, you can use `numpy.tofile()`, but **tofile()** and **fromfile()** are not platform-independent. See the next section on **save()** and **load()** for platform-independent I/O.

Example

np_savetxt_loadtxt.py

```
#!/usr/bin/env python
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

print(sample_data)
print('-' * 60)

sample_data  /= 10  ②

float_file_name = 'save_data_float.txt'

np.savetxt(float_file_name, sample_data, delimiter=",", fmt="%5.2f")  ③

int_file_name = 'save_data_int.txt'

np.savetxt(int_file_name, sample_data, delimiter=",", fmt="%d")  ④

data = np.loadtxt(float_file_name, delimiter=",")  ⑤
print(data)
```

① Load data from space-delimited file

② Modify sample data

③ Write data to text file as floats, rounded to two decimal places, using commas as delimiter

④ Write data to text file as ints, using commas as delimiter

⑤ Read data back into **ndarray**

np_savetxt_loadtxt.py

```
[[63. 51. 59. 61. 50.  4.]
 [40. 66.  9. 64. 63. 17.]
 [18. 23.  2. 61.  1.  9.]
 ...
 [26. 20. 54. 46. 38. 23.]
 [ 9.  5. 59. 23.  2. 26.]
 [46. 34. 25.  8. 39. 34.]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```

Example

np_tofile_fromfile.py

```
#!/usr/bin/env python
import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=float
)

sample_data  /= 10  ②

print(sample_data)
print("-" * 60)

file_name = 'sample.dat'

sample_data.tofile(file_name)  ③

data = np.fromfile(file_name)  ④
data.shape = sample_data.shape  ⑤

print(data)
```

① Read in sample data

② Modify sample data

③ Write data to file (binary, but not portable)

④ Read binary data from file as one-dimensional array

⑤ Set shape to shape of original array

np_tofile_fromfile.py

```
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
------------------------------------------------------------
[[6.3 5.1 5.9 6.1 5.  0.4]
 [4.  6.6 0.9 6.4 6.3 1.7]
 [1.8 2.3 0.2 6.1 0.1 0.9]
 ...
 [2.6 2.  5.4 4.6 3.8 2.3]
 [0.9 0.5 5.9 2.3 0.2 2.6]
 [4.6 3.4 2.5 0.8 3.9 3.4]]
```

Saving and retrieving arrays

- Efficient binary format
- Save as NumPy data
  - Use `numpy.save()`
- Read into ndarray
  - Use `numpy.load()`

To save an array as a NumPy data file, use `numpy.save()`. This will write the data out to a specified file name, adding the extension '.npy'.

To read the data back into a NumPy ndarray, use `numpy.load()`. Data are read and written in a way that preserves precision and endianness.

This the most efficient way to store numeric data for later retrieval, compared to **savetext()** and **loadtext()** or **tofile()** and **fromfile()**. Files written with `numpy.save()` are not human-readable.

Example

np_save_load.py

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


import numpy as np

sample_data = np.loadtxt(   ①
    "../DATA/columns_of_numbers.txt",
    skiprows=1,
    dtype=int
)

sample_data  *= 100  ②

print(sample_data)

file_name = 'sampledata'

np.save(file_name, sample_data)  ③

retrieved_data = np.load(file_name + '.npy')  ④

print('-' * 60)
print(retrieved_data)
```

① Read some sample data into an ndarray

② Modify the sample data (multiply every element by 100)

③ Write entire array out to NumPy-format data file (adds **.npy** extension)

④ Retrieve data from saved file

np_save_load.py

```
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
------------------------------------------------------------
[[6300 5100 5900 6100 5000  400]
 [4000 6600  900 6400 6300 1700]
 [1800 2300  200 6100  100  900]
 ...
 [2600 2000 5400 4600 3800 2300]
 [ 900  500 5900 2300  200 2600]
 [4600 3400 2500  800 3900 3400]]
```

Array creation shortcuts

- Use "magic" arrays r_, c_
- Either creates a new NumPy array
  - Index values determine resulting array
  - List of arrays creates a "stacked" array
  - List of values creates a 1D array
  - Slice notation creates a range of values
  - A complex step creates equally-spaced value

NumPy provides several shortcuts for working with arrays.

The **r**_ object can be used to magically build arrays via its index expression. It acts like a magic array, and "returns" (evaluates to) a normal NumPy ndarray object. (The **c**_ object is the same, but is column-oriented).

There are two main ways to use r_():

If the index expression contains a list of arrays, then the arrays are "stacked" along the first axis.

If the index contains slice notation, then it creates a one-dimensional array, similar to numpy.arange(). It uses start, stop, and step values. However, if step is an imaginary number (a literal that ends with 'j'), then it specifies the number of points wanted, more like numpy.linspace().

There can be more than one slice, as well as individual values, and ranges. They will all be concatenated into one array.

TIP If the first element in the index is a string containing one, two, or three integers separated by commas, then the first integer is the axis to stack the arrays along; the second is the minimum number of dimensions to force each entry into; the third allows you to control the shape of the resulting array.
This is especially useful for making a new array from selected parts of an existing array.

NOTE Most of the time you will be creating arrays by reading data — these are mostly useful for edge cases when you're creating some smaller specialized array.

Example

np_tricks.py

```
#!/usr/bin/env python
import numpy as np

a1 = np.r_[np.array([1, 2, 3]), 0, 0, np.array([4, 5, 6])]  ①
print(a1)
print()

a2 = np.r_[-1:1:6j, [0] * 3, 5, 6]  ②
print(a2)
print()

a = np.array([[0, 1, 2], [3, 4, 5]])  ③
a3 = np.r_['-1', a, a]  ④
print(a3)
print()

a4 = np.r_['0,2', [1, 2, 3], [4, 5, 6]]  ④
print(a4)
print()

a5 = np.r_['0,2,0', [1, 2, 3], [4, 5, 6]]  ⑤
print(a5)
print()

a6 = np.r_['1,2,0', [1, 2, 3], [4, 5, 6]]  ⑥
print(a6)
print()

m = np.r_['r', [1, 2, 3], [4, 5, 6]]
print(m)
print(type(m))
```

① build array from a sequence of array-like things

② faux slice with complex step implements linspace <3>

np_tricks.py

```
[1 2 3 0 0 4 5 6]

[-1.  -0.6 -0.2  0.2  0.6  1.   0.   0.   0.   5.   6. ]

[[0 1 2 0 1 2]
 [3 4 5 3 4 5]]

[[1 2 3]
 [4 5 6]]

[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]

[[1 4]
 [2 5]
 [3 6]]

[[1 2 3 4 5 6]]
<class 'numpy.matrix'>
```

Chapter 5 Exercises

Exercise 5-1 (big_arrays.py)

Starting with the file big_arrays.py, convert the Python list values into a NumPy array.

Make a copy of the array named values_x_3 with all values multiplied by 3.

Print out values_x_3

Exercise 5-2 (create_range.py)

Using arange(), create an array of 35 elements.

Reshape the arrray to be 5 x 7 and print it out.

Reshape the array to be 7 x 5 and print it out.

Exercise 5-3 (create_linear_space.py)

Using linspace(), create an array of 500 elements evenly spaced between 100 and 200.

Reshape the array into 5 x 10 x 10.

Multiply every element by .5

Print the result.

Chapter 6: Introduction to pandas

Objectives

- Understand what the pandas module provides
- Load data from CSV and other files
- Access data tables
- Extract rows and columns using conditions
- Calculate statistics for rows or columns

About pandas

- Reads data from file, database, or other sources
- Deals with real-life issues such as invalid data
- Powerful selecting and indexing tools
- Builtin statistical functions
- Munge, clean, analyze, and model data
- Works with numpy and matplotlib

pandas is a package designed to make it easy to get, organize, and analyze large datasets. Its strengths lie in its ability to read from many different data sources, and to deal with real-life issues, such as missing, incomplete, or invalid data.

pandas also contains functions for calculating means, sums and other kinds of analysis.

For selecting desired data, pandas has many ways to select and filter rows and columns.

It is easy to integrate pandas with NumPy, Matplotlib, and other scientific packages.

While pandas can handle three (or higher) dimensional data, it is generally used with two-dimensional (row/column) data, which can be visualized like a spreadsheet.

pandas provides powerful split-apply-combine operations — **groupby** enables transformations, aggregations, and easy-access to plotting functions. It is easy to emulate R's plyr package via pandas.

NOTE pandas gets its name from *pan*el *da*ta *s*ystem

pandas architecture

- Two main structures: Series and DataFrame
- Series – one-dimensional
- DataFrame – two-dimensional

The two main data structures in pandas are the **Series** and the **DataFrame**. A series is a one-dimensional indexed list of values, something like an ordered dictionary. A DataFrame is is a two-dimensional grid, with both row and column indices (like the rows and columns of a spreadsheet, but more flexible).

You can specify the indices, or pandas will use successive integers. Each row or column of a DataFrame is a Series.

NOTE pandas used to support the **Panel** type, which is more more or less a collection of DataFrames, but Panel has been deprecated in favor of hierarchical indexing.

Series

- Indexed list of values
- Similar to a dictionary, but ordered
- Can get sum(), mean(), etc.
- Use index to get individual values
- Indices are not positional

A Series is an indexed sequence of values. Each item in the sequence has an index. The default index is a set of increasing integer values, but any set of values can be used.

For example, you can create a series with the values 5, 10, and 15 as follows:

```
s1 = pd.Series([5,10,15])
```

This will create a Series indexed by [0, 1, 2]. To provide index values, add a second list:

```
s2 = pd.Series([5,10,15], ['a','b','c'])
```

This specifies the indices as 'a', 'b', and 'c'.

You can also create a Series from a dictionary. pandas will put the index values in order:

```
s3 = pd.Series({'b':10, 'a':5, 'c':15})
```

There are many methods that can be called on a Series, and Series can be indexed in many flexible ways.

Example

pandas_series.py

```
#!/usr/bin/env python
import numpy as np
import pandas as pd

NUM_VALUES = 10
index = [chr(i) for i in range(97, 97 + NUM_VALUES)]  ①
print("index:", index, '\n')

s1 = pd.Series(np.linspace(1, 5, NUM_VALUES), index=index)  ②
s2 = pd.Series(np.linspace(1, 5, NUM_VALUES))  ③

print("s1:", s1, "\n")
print("s2:", s2, "\n")

print("selecting elements")
print(s1[['h', 'b']], "\n")  ④

print(s1[['a', 'b', 'c']], "\n")  ④

print("slice of elements")
print(s1['b':'d'], "\n")  ⑤

print("sum(), mean(), min(), max():")
print(s1.sum(), s1.mean(), s1.min(), s1.max(), "\n")  ⑥

print("cumsum(), cumprod():")
print(s1.cumsum(), s1.cumprod(), "\n")  ⑥

print('a' in s1)  ⑦
print('m' in s1)  ⑦
print()

s3 = s1 * 10  ⑧
print("s3 (which is s1 * 10)")
print(s3, "\n")

s1['e'] *= 5

print("boolean mask where s3 > 25:")
print(s3 > 25, "\n")  ⑨

print("assign -1 where mask is true")
s3[s3 < 25] = -1  ⑩
print(s3, "\n")
```

```
s4 = pd.Series([-0.204708, 0.478943, -0.519439])  ⑪
print("s4.max(), .min(), etc.")
print(s4.max(), s4.min(), s4.max() - s4.min(), '\n')  ⑫

s = pd.Series([5, 10, 15], ['a', 'b', 'c'])  ⑬
print("creating series with index")
print(s)
```

① make list of 'a', 'b', 'c', ...

② create series with specified index

③ create series with auto-generated index (0, 1, 2, 3, ...)

④ select items from series

⑤ select slice of elements

⑥ get stats on series

⑦ test for existence of label

⑧ create new series with every element of s1 multiplied by 10

⑨ create boolean mask from series

⑩ set element to -1 where mask is True

⑪ create new series

⑫ print stats

⑬ create new series with index

DataFrames

- Two-dimensional grid of values
- Row and column labels (indices)
- Rich set of methods
- Powerful indexing

A DataFrame is the workhorse of pandas. It represents a two-dimensional grid of values, containing indexed rows and columns, something like a spreadsheet.

There are many ways to create a DataFrame. They can be modified to add or remove rows/columns. Missing or invalid data can be eliminated or normalized.

DataFrames can be initialized from many kinds of data. See the table on the next page for a list of possibilities.

NOTE The panda DataFrame is modeled after R's data.frame

Table 6. DataFrame Initializers

Initializer	Description
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are union-ed together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are union-ed to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

NOTE This utility method is used in some of the example scripts:

printheader.py

```
#!/usr/bin/env python
HEADER_CHAR = '='

def print_header(comment, header_width=50):
    ''' Print comment and separator '''
    header_line = HEADER_CHAR * header_width
    print(header_line)
    print(comment.center(header_width-2).center(header_width, HEADER_CHAR))
    print(header_line)

if __name__ == '__main__':
    print_header("this is a test")
```

Example

pandas_simple_dataframe.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header


cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  ①
indices = ['a', 'b', 'c', 'd', 'e', 'f']  ②

values = [  ③
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('cols')
print(cols, '\n')

print_header('indices')
print(indices, '\n')

print_header('values')
print(values, '\n')

df = pd.DataFrame(values, index=indices, columns=cols)  ④
print_header('DataFrame df')
print(df, '\n')

print_header("df['gamma']")
print(df['gamma'])  ⑤
```

① column names

② row names

③ sample data

④ create dataframe with row and column names

⑤ select column 'gamma'

pandas_simple_dataframe.py

```
================================================
=                      cols                    =
================================================
['alpha', 'beta', 'gamma', 'delta', 'epsilon']

================================================
=                    indices                   =
================================================
['a', 'b', 'c', 'd', 'e', 'f']

================================================
=                     values                   =
================================================
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340], [400,
410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]

================================================
=                  DataFrame df                =
================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

================================================
=                  df['gamma']                 =
================================================
a    120
b    220
c    320
d    420
e    520
f    620
Name: gamma, dtype: int64
```

Index objects

- Used to index Series or DataFrames
- index = pandas.core.frame.Index(sequence)
- Can be named

An *index object* is a kind of ordered set that is used to access rows or columns in a dataset. As shown earlier, indexes can be specified as lists or other sequences when creating a Series or DataFrame.

You can create an index object and then create a Series or a DataFrame using the index object. Index objects can be named, either something obvious like 'rows' or columns', or more appropriate to the specific type of of data being indexed.

Remember that index objects act like sets, so the main operations on them are unions, intersections, or differences.

TIP You can replace an existing index on a DataFrame with the set_index() method.

Example

pandas_index_objects.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

index1 = pd.Index(['a', 'b', 'c'], name='letters')  ①
index2 = pd.Index(['b', 'a', 'c'])
index3 = pd.Index(['b', 'c', 'd'])
index4 = pd.Index(['red', 'blue', 'green'], name='colors')

print_header("index1, index2, index3", 70)  ②
print(index1)
print(index2)
print(index3)
print()

print_header("index2 & index3", 70)
# these are the same
print(index2 & index3)  ③
print(index2.intersection(index3))  ③
print()
```

```
print_header("index2 | index3", 70)
# these are the same
print(index2 | index3)  ④
print(index2.union(index3))
print()

print_header("index1.difference(index3)", 70)
print(index1.difference(index3))  ⑤
print()

print_header("Series([10,20,30], index=index1)", 70)
series1 = pd.Series([10, 20, 30], index=index1)  ⑥
print(series1)
print()

print_header("DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index1, columns=index4)", 70)
dataframe1 = pd.DataFrame([(1, 2, 3), (4, 5, 6), (7, 8, 9)], index=index1,
columns=index4)
print(dataframe1)
print()

print_header("DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index4, columns=index1)", 70)
dataframe2 = pd.DataFrame([(1, 2, 3), (4, 5, 6), (7, 8, 9)], index=index4,
columns=index1)
print(dataframe2)
print()
```

① create some indexes

② display indexes

③ get intersection of indexes

④ get union of indexes

⑤ get difference of indexes

⑥ use index with series (can also be used with dataframe)

pandas_index_objects.py

```
========================================================================
=                       index1, index2, index3                         =
========================================================================
Index(['a', 'b', 'c'], dtype='object', name='letters')
Index(['b', 'a', 'c'], dtype='object')
Index(['b', 'c', 'd'], dtype='object')

========================================================================
=                           index2 & index3                            =
========================================================================
Index(['b', 'c'], dtype='object')
Index(['b', 'c'], dtype='object')

========================================================================
=                           index2 | index3                            =
========================================================================
Index(['a', 'b', 'c', 'd'], dtype='object')
Index(['a', 'b', 'c', 'd'], dtype='object')

========================================================================
=                      index1.difference(index3)                       =
========================================================================
Index(['a'], dtype='object')

========================================================================
=                    Series([10,20,30], index=index1)                  =
========================================================================
letters
a    10
b    20
c    30
dtype: int64

========================================================================
= DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index1, columns=index4) =
========================================================================
colors   red  blue  green
letters
a          1     2      3
b          4     5      6
c          7     8      9

========================================================================
= DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index4, columns=index1) =
========================================================================
letters  a  b  c
```

```
colors
red      1  2  3
blue     4  5  6
green    7  8  9
```

Basic Indexing

- Similar to normal Python or numpy
- Slices select rows

One of the real strengths of pandas is the ability to easily select desired rows and columns. This can be done with simple subscripting, like normal Python, or extended subscripting, similar to numpy. In addition, pandas has special methods and attributes for selecting data.

For selecting columns, use the column name as the subscript value. This selects the entire column. To select multiple columns, use a sequence (list, tuple, etc.) of column names.

For selecting rows, use slice notation. This may not map to similar tasks in normal python. That is, dataframe[x:y] selects rows x through y, but dataframe[x] selects column x.

Example

pandas_selecting.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  ①
index = ['a', 'b', 'c', 'd', 'e', 'f']  ②

values = [  ③
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=cols)  ④
print_header('DataFrame df')
print(df, '\n')

print_header("df['alpha']")
print(df['alpha'], '\n')  ⑤

print_header("df.beta")
print(df.beta, '\n')  ⑥

print_header("df['b':'e']")
print(df['b':'e'], '\n')  ⑦

print_header("df[['alpha','epsilon','beta']]")
print(df[['alpha', 'epsilon', 'beta']])  ⑧
print()

print_header("df[['alpha','epsilon','beta']]['b':'e']")
print(df[['alpha', 'epsilon', 'beta']]['b':'e'])  ⑨
print()
```

① column labels

② row labels

③ sample data

④ create dataframe with data, row labels, and column labels

⑤ select column 'alpha'

⑥ same, but alternate syntax (only works if column name is letters, digits, and underscores)

⑦ select rows 'b' through 'e' using slice of row labels

⑧ select columns — note index is an iterable

⑨ select columns AND slice rows

pandas_selecting.py

```
================================================
=                  DataFrame df                =
================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

================================================
=                  df['alpha']                 =
================================================
a    100
b    200
c    300
d    400
e    500
f    600
Name: alpha, dtype: int64

================================================
=                    df.beta                   =
================================================
a    110
b    210
c    310
d    410
e    510
f    610
Name: beta, dtype: int64
```

```
==================================================
=                  df['b':'e']                   =
==================================================
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540

==================================================
=         df[['alpha','epsilon','beta']]         =
==================================================
   alpha  epsilon  beta
a    100      140   110
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
f    600      640   610

==================================================
=    df[['alpha','epsilon','beta']]['b':'e']     =
==================================================
   alpha  epsilon  beta
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
```

Broadcasting

- Operation is applied across rows and columns
- Can be restricted to selected rows/columns
- Sometimes called vectorization
- Use apply() for more complex operations

If you multiply a dataframe by some number, the operation is broadcast, or vectorized, across all values. This is true for all basic math operations.

The operation can be restricted to selected columns.

For more complex operations, the apply() method will apply a function that selects elements. You can use the name of an existing function, or supply a lambda (anonymous) function.

Example

pandas_broadcasting.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']  ①
index = pd.date_range('2013-01-01 00:00:00', periods=6, freq='D')  ②

print(index, "\n")

values = [  ③
    [100, 110, 120, 930, 140],
    [250, 210, 120, 130, 840],
    [300, 310, 520, 430, 340],
    [275, 410, 420, 330, 777],
    [300, 510, 120, 730, 540],
    [150, 610, 320, 690, 640],
]

df = pd.DataFrame(values, index, cols)  ④
print_header("Basic DataFrame:")
print(df)
print()

print_header("Triple each value")
print(df * 3)
print()  ⑤

print_header("Multiply column gamma by 1.5")
df['gamma'] *= 1.5  ⑥
print(df)
print()
```

① column labels

② date range to be used as row indexes

③ sample data

④ create dataframe from data

⑤ multiply every value by 3

⑥ multiply values in column 'gamma' by 1.

pandas_broadcasting.py

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

==================================================
=                Basic DataFrame:                =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110    120    930      140
2013-01-02    250   210    120    130      840
2013-01-03    300   310    520    430      340
2013-01-04    275   410    420    330      777
2013-01-05    300   510    120    730      540
2013-01-06    150   610    320    690      640

==================================================
=               Triple each value                =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    300   330    360   2790      420
2013-01-02    750   630    360    390     2520
2013-01-03    900   930   1560   1290     1020
2013-01-04    825  1230   1260    990     2331
2013-01-05    900  1530    360   2190     1620
2013-01-06    450  1830    960   2070     1920

==================================================
=          Multiply column gamma by 1.5          =
==================================================
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110  180.0    930      140
2013-01-02    250   210  180.0    130      840
2013-01-03    300   310  780.0    430      340
2013-01-04    275   410  630.0    330      777
2013-01-05    300   510  180.0    730      540
2013-01-06    150   610  480.0    690      640
```

Removing entries

- Remove rows or columns
- Use drop() method

To remove columns or rows, use the drop() method, with the appropriate labels. Use axis=1 to drop columns, or axis=0 to drop rows.

Example

pandas_drop.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('values:')
print(values, '\n\n')

df = pd.DataFrame(values, index=index, columns=cols)  ①
print_header('DataFrame df')
print(df, '\n')

df2 = df.drop(['beta', 'delta'], axis=1)  ②
print_header("After dropping beta and delta:")
print(df2, '\n')

print_header("After dropping rows b, c, and e")
df3 = df.drop(['b', 'c', 'e'])  ③
print(df3)
```

① create dataframe

② drop columns beta and delta (axes: 0=rows, 1=columns)

③ drop rows b, c, and e

pandas_drop.py

```
================================================
=                   values:                    =
================================================
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340], [400,
410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]


================================================
=                 DataFrame df                 =
================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640

================================================
=         After dropping beta and delta:       =
================================================
   alpha  gamma  epsilon
a    100    120      140
b    200    220      240
c    300    320      340
d    400    420      440
e    500    520      540
f    600    620      640

================================================
=        After dropping rows b, c, and e       =
================================================
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
d    400   410    420    430      440
f    600   610    620    630      640
```

Data alignment

- pandas will auto-align data by rows and columns
- Non-overlapping data will be set as NaN

When two dataframes are combined, columns and indices are aligned.

The result is the union of matching rows and columns. Where data doesn't exist in one or the other dataframe, it is set to NaN.

A default value can be specified for the overlapping cells when combining dataframes with methods such as add() or mul().

Use the fill_value parameter to set a default for missing values.

Example

pandas_alignment.py

```
#!/usr/bin/env python
import numpy as np
import pandas as pd
from printheader import print_header  ①

dataset1 = np.arange(9.).reshape((3, 3))  ②

df1 = pd.DataFrame(  ③
    dataset1,
    columns=['apple', 'banana', 'mango'],
    index=['orange', 'purple', 'blue']
)

dataset2 = np.arange(12.).reshape((4, 3))  ②

df2 = pd.DataFrame(  ③
    dataset2,
    columns=['apple', 'banana', 'kiwi'],
    index=['orange', 'purple', 'blue', 'brown']
)

print_header('df1')
print(df1)  ④
print()

print_header('df2')
print(df2)  ④
print()

print_header('df1 + df2')
print(df1 + df2)  ⑤

print_header('df1.add(df2, fill_value=0)')
print(df1.add(df2, fill_value=0))  ⑥
```

Time Series

- Use time_series()
- Specify start/end time/date, number of periods, time units
- Useful as index to other data
- freq=time_unit
- periods=number_of_periods

pandas provides a function **time_series()** to generate a list of timestamps. You can specify the start/end times as dates or dates/times, and the type of time units. Alternatively, you can specify a start date/time and the number of periods to create.

The frequency strings can have multiples – 5H means every 5 hours, 3S means every 3 seconds, etc.

Table 7. Units for time_series() freq flag

Unit	Represents
M	Month
D	Day
H	Hour
T	Minute
S	Second

Example

pandas_time_slices.py

```
#!/usr/bin/env python
import pandas as pd
import numpy as np

hourly = pd.date_range('1/1/2013 00:00:00', '1/3/2013 23:59:59', freq='H')  ①
print("Number of periods: ", len(hourly))

seconds = pd.date_range('1/1/2013 12:00:00', freq='S', periods=(60 * 60 * 18))  ②
print("Number of periods: ", len(seconds))
print("Last second: ", seconds[-1])

monthly = pd.date_range('1/1/2013', '12/31/2013', freq='M')  ③
print("Number of periods: {0} Seventh element: {1}".format(
    len(monthly),
    monthly[6]
))

NUM_DATA_POINTS = 1441  ④

dates = pd.date_range('4/1/2013 00:00:00', periods=NUM_DATA_POINTS, freq='T')  ⑤

data = np.random.random(NUM_DATA_POINTS)  ⑥

series = pd.Series(data, index=dates)  ⑦

time_slice = series['4/1/2013 10:00:00':'4/1/2013 10:30:00']  ⑧
print(time_slice)  # 31 values
```

① make time series — every hour for 3 days

② make time series — every second for 18 hours

③ every month for 1 year

④ number of minutes in a day

⑤ create range from starting point with specified number of points — one day's worth of minutes

⑥ a day's worth of data

⑦ series indexed by minutes

⑧ select the half hour of data from 10:00 to 10:30

pandas_time_slices.py

```
Number of periods:  72
Number of periods:  64800
Last second:  2013-01-02 05:59:59
Number of periods: 12 Seventh element: 2013-07-31 00:00:00
2013-04-01 10:00:00    0.779324
2013-04-01 10:01:00    0.399369
2013-04-01 10:02:00    0.988013
2013-04-01 10:03:00    0.361195
2013-04-01 10:04:00    0.368400
2013-04-01 10:05:00    0.279782
2013-04-01 10:06:00    0.757594
2013-04-01 10:07:00    0.516288
2013-04-01 10:08:00    0.808836
2013-04-01 10:09:00    0.874611
2013-04-01 10:10:00    0.657730
2013-04-01 10:11:00    0.405349
2013-04-01 10:12:00    0.659032
2013-04-01 10:13:00    0.954761
2013-04-01 10:14:00    0.496885
2013-04-01 10:15:00    0.660708
2013-04-01 10:16:00    0.281537
2013-04-01 10:17:00    0.421557
2013-04-01 10:18:00    0.258586
2013-04-01 10:19:00    0.369397
2013-04-01 10:20:00    0.341791
2013-04-01 10:21:00    0.926929
2013-04-01 10:22:00    0.995008
2013-04-01 10:23:00    0.690462
2013-04-01 10:24:00    0.698885
2013-04-01 10:25:00    0.500728
2013-04-01 10:26:00    0.309860
2013-04-01 10:27:00    0.806451
2013-04-01 10:28:00    0.631646
2013-04-01 10:29:00    0.529562
2013-04-01 10:30:00    0.389145
Freq: T, dtype: float64
```

Useful pandas methods

Table 8. Methods and attributes for fetching DataFrame/Series data

Method	Description
DF.columns()	Get or set column labels
DF.shape() S.shape()	Get or set shape (length of each axis)
DF.head(n) DF.tail(n)	Return n items (default 5) from beginning or end
DF.describe() S.describe()	Display statistics for dataframe
DF.info()	Display column attributes
DF.values S.values	Get the actual values from a data structure
DF.loc[row_indexer[1], col_indexer]	Multi-axis indexing by label (not by position)
DF.iloc[row_indexer[2], col_indexer]	Multi-axis indexing by position (not by labels)

[1] Indexers can be label, slice of labels, or iterable of labels.
[2] Indexers can be numeric index (0-based), slice of indexes, or iterable of indexes.

Table 9. Methods for Computations/Descriptive Stats (called from pandas)

Method	Returns
abs()	absolute values
corr()	pairwise correlations
count()	number of values
cov()	Pairwise covariance
cumsum()	cumulative sums
cumprod()	cumulative products
cummin(), cummax()	cumulative minimum, maximum
kurt()	unbiased kurtosis
median()	median
min(), max()	minimum, maximum values
prod()	products
quantile()	values at given quantile
skew()	unbiased skewness
std()	standard deviation
var()	variance

NOTE these methods return Series or DataFrames, as appropriate, and can be computed over rows (axis=0) or columns (axis=1). They generally skip NA/null values.

Reading Data

- Supports many data formats
- Reads headings to create column indexes
- Auto-creates indexes as needed
- Can used specified column as row index

pandas support many different input formats. It will reading file headings and use them to create column indexes. By default, it will use integers for row indices, but you can specify a column to use as the index.

The read_... functions have many options for controlling and parsing input. For instance, if large integers in the file contain commas, the thousands options let you set the separator as comma (in the US), so it will ignore them.

read_csv() is the most frequently used function, and has many options. It can also be used to read generic flat-file formats. **read_table** is similar to **read_csv()**, but doesn't assume CSV format.

There are corresponding "to_...." functions for many of the read functions. `to_csv()` and `to_ndarray()` are particular useful.

These are not the only functions for reading into

See https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html?highlight=output#io-html for details on the I/O functions.

NOTE	See Jupyter notebook **pandas_Input_Demo** for examples of reading most types of input.

Table 10. pandas I/O functions

Format	Input function	Output function
CSV	read_csv()	to_csv()
Delimited file (generic)	read_table()	to_csv()
Excel worksheet	read_excel()	to_excel()
File with fixed-width fields	read_fwf()	
Google BigQuery	read_gbq()	to_gbq()
HDF5	read_hdf()	to_hdf()
HTML table	read_html()	to_html()
JSON	read_json()	to_json()
OS clipboard data	read_clipboard()	to_clipboard()
Parquet	read_parquet()	to_parquet()
pickle	read_pickle()	to_pickle()
SAS	read_sas()	
SQL query	read_sql()	to_sql()

NOTE All **read_...()** functions return a new **DataFrame**, except **read_html()**, which returns a list of **DataFrames**

Example

pandas_read_csv.py

```
#!/usr/bin/env python
import pandas as pd
from printheader import print_header

# data from
# http://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/publications/
# national_transportation_statistics/html/table_01_44.html

airports_df = pd.read_csv('../DATA/airport_boardings.csv', thousands=',', index_col=1)
①

print_header("HEAD OF DATAFRAME")

print(airports_df.head(), "\n")

print_header("SELECTED COLUMNS WITH FILTERED ROWS")
columns_wanted = ['2001 Total', 'Airport']
sort_col = '2001 Total'
max_val = 20000000
selector = airports_df['2001 Total'] > max_val
selected = airports_df[selector][columns_wanted]
print(selected)

print_header("COLUMN TOTALS")
print(airports_df[['2001 Total', '2010 Total']].sum(), "\n")

# print_header("'CODE' COLUMN SET AS INDEX")
# airports_df.set_index('Code')
# print(airports_df)

print_header("FIRST FIVE ROWS")
print(airports_df.iloc[:5])
```

① Read CSV file into dataframe; parse numbers containing commas,use first column as row index.

More pandas...

At this point, please load the following Jupyter notebooks for more pandas exploration:

- pandas_Demo.ipynb
- pandas_Input_Demo.ipynb
- pandas_Selection_Demo.ipynb

NOTE	The instructor will explain how to start the Jupyter server.

Chapter 6 Exercises

Exercise 6-1 (simple_dataframe.py)

Create a DataFrame with columns named 'Test1', 'Test2', up to 'Test6'. Use default row indexes. Fill the DataFrame with random values.

- Print only columns 'Test3' and 'Test5'.
- Print the dataframe with every value multiplied by 3.6

Exercise 6-2 (parasites.py))

The file parasite_data.csv, in the DATA folder, has some results from analysis on some intestinal parasites (not that it matters for this exercise...). Read parasite_data.csv into a DataFrame. Print out all rows where the Shannon Diversity is >= 1.0.

Chapter 7: Introduction to Matplotlib

Objectives

- Understand what matplotlib can do
- Create many kinds of plots
- Label axes, plots, and design callouts

About matplotlib

- matplotlib is a package for making 2D plots
- Emulates MATLAB®, but not a drop-in replacement
- matplotlib's philosophy: create simple plots simply
- Plots are publication quality
- Plots can be rendered in GUI applications

This chapter's discussion of matplotlib will use the iPython notebook named **MatplotlibExamples.ipynb**. Please start the iPython notebook server and load this notebook, as directed by the instructor.

matplotlib architecture

- pylab/pyplot front end plotting functions
- API create/manage figures, text, plots
- backends device-independent renderers

matplotlib consists of roughly three parts: pylab/pyplot, the API, and and the backends.

pyplot is a set of functions which allow the user to quickly create plots. Pyplot functions are named after similar functions in MATLAB.

The API is a large set of classes that do all the work of creating and manipulating plots, lines, text, figures, and other graphic elements. The API can be called directly for more complex requirements.

pylab combines pyplot with numpy. This makes pylab emulate MATLAB more closely, and thus is good for interactive use, e.g., with iPython. On the other hand, pyplot alone is very convenient for scripting. The main advantage of pylab is that it imports methods from both pyplot and pylab.

There are many backends which render the in-memory representation, created by the API, to a video display or hard-copy format. For example, backends include PS for Postscript, SVG for scalable vector graphics, and PDF.

The normal import is

```
import matplotlib.pyplot as plt
```

Matplotlib Terminology

- Figure
- Axis
- Subplot

A Figure is one "picture". It has a border ("frame"), and other attributes. A Figure can be saved to a file.

A Plot is one set of values graphed onto the Figure. A Figure can contain more than one Plot.

Axes and Subplot are similar; the difference is how they get placed on the figure. Subplots allow multiple plots to be placed in a rectangular grid. Axes allow multiple plots to placed at any location, including within other plots, or overlapping.

matplotlib uses default objects for all of these, which are sufficient for simple plots. You can explicitly create any or all of these objects to fine-tune a graph. Most of the time, for simple plots, you can accept the defaults and get great-looking figures.

Matplotlib Keeps State

- Primary method is matplotlib.pyplot()
- The current figure can have more than one plot
- Calling show() displays the current figure

matplotlib.pyplot is the workhorse of figure drawing. It is usually aliased to "plt".

While Matplotlib is object oriented, and you can manually create figures, axes, subplots, etc., pyplot() will create a figure object for you automatically, and commands called from pyplot() (usually through the **plt** alias) will work on that object.

Calling **plt.plot()** plots one set of data on the current figure. Calling it again adds another plot to the same figure.

plt.show() displays the figure, although iPython may display each separate plot, depending on the current settings.

You can pass one or two datasets to plot(). If there are two datasets, they need to be the same length, and represent the x and y data.

What Else Can You Do?

- Multiple plots
- Control ticks on any axis
- Scatter plots
- Polar axes
- 3D Plots
- Quiver plots
- Pie Charts

There are many other types of drawings that matplotlib can create. Also, there are many more style details that can be tweaked. See http://matplotlib.org/gallery.html for dozens of sample plots and their source.

There are many extensions (AKA toolkits) for Matplotlib, including Seaborne, CartoPy, at Natgrid.

Matplotlib Demo

At this point, please open the notebook **MatPlotLibExamples.ipynb** for an instructor-led tour of MPL features.

Chapter 7 Exercises

Exercise 7-1 (energy_use_plot.py)

Using the file energy_use_quad.csv in the DATA folder, use matplotlib to plot the data for "Transportation", "Industrial", and "Residential and Commercial". Don’t plot the "as a percent...".

You can do this in iPython, or as a standalone script. If you create a standalone script, save the figure to a file, so you can view it.

Use pandas to read the data. The columns are, in Python terms:

```
['Desc',"1960","1965","1970","1975","1980","1985","1990","1991","1992","1993","1994","199
5","1996","1997","1998","1999","2000","2001","2002","2003","2004","2005","2006","2007","2
008","2009","2010","2011"]
```

TIP See the script pandas_energy.py in the EXAMPLES folder to see how to load the data.

Chapter 8: 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 11. 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 12. 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 not 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

```
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
Joseph Robinette Biden
```

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, plus a few extras.

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 13. 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()  # [('apple', .49), ('kiwi', .38)]

    try:
        conn.executemany(INSERT, fruit_data)  ⑦
    except sqlite3.DatabaseError as err:
        print(err)
        conn.rollback()
    else:
        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    8.06
cherry        12.84
apricot       10.77
date          12.10
apple         10.77
lemon         12.36
kiwi           8.40
orange         8.75
lime           9.86
watermelon     6.62
guava          9.16
papaya         6.21
fig            5.67
pear           8.46
banana         7.88
tamarind       9.27
persimmon      6.65
elderberry    11.52
peach         14.57
blueberry     12.48
lychee         9.08
grape         11.77
```

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 14. 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.DictCursor)`
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 attribute is a tuple with metadata about the columns in the query. It contains one tuple for each column in the query, containing 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.

Example

NOTE

Many 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. __

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 cursor_row in cursor.fetchall():
        row_dict = dict(zip(column_names, cursor_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

```
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
Joseph Robinette Biden
```

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, errors

FIELD_NAMES = (
    'termnumber lastname firstname '
    'birthdate '
    'deathdate birthplace birthstate '
    'termstartdate '
    'termenddate '
    'party'
).split()  ①

mc = MongoClient()  ②

try:
    mc.drop_database("presidents")  ③
except errors.PyMongoError as err:
    print(err)

db = mc["presidents"]  ④

coll = db.presidents  ⑤

with open('../DATA/presidents.txt') as presidents_in:  ⑥
    for line in presidents_in:
        flds = line[:-1].split(':')
        kvpairs = zip(FIELD_NAMES, flds)
        record_dict = dict(kvpairs)
        coll.insert_one(record_dict)  ⑦

print(db.list_collection_names())  ⑧
print()

abe = coll.find_one({'termnumber': '16'})  ⑨
print(abe, '\n')

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'}):  ⑬
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))

print('-' * 50)
print("removing Millard Fillmore")
result = coll.delete_one({'lastname': 'Fillmore'})  ⑭
print(result)
result = coll.delete_one({'lastname': 'Roosevelt'})  ⑭
print(result)
print('-' * 50)


result = coll.delete_one({'lastname': 'Bush'})
print(dir(result))
print()

result = coll.count_documents({})  ⑮
print(result)


for president in coll.find():  ⑪
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

animals = db.animals

print(animals, '\n')

animals.insert_one({'name': 'wombat', 'country': 'Australia'})
animals.insert_one({'name': 'ocelot', 'country': 'Mexico'})
animals.insert_one({'name': 'honey badger', 'country': 'Iran'})

for doc in animals.find():
    print(doc['name'])
```

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

```
William Howard            Taft
Woodrow                   Wilson
Warren Gamaliel           Harding
Calvin                    Coolidge
Herbert Clark             Hoover
Franklin Delano           Roosevelt
Harry S.                  Truman
Dwight David              Eisenhower
John Fitzgerald           Kennedy
Lyndon Baines             Johnson
Richard Milhous           Nixon
Gerald Rudolph            Ford
James Earl 'Jimmy'        Carter
Ronald Wilson             Reagan
William Jefferson 'Bill'  Clinton
George Walker             Bush
Barack Hussein            Obama
Donald John               Trump
Joseph Robinette          Biden
--------------------------------------------------
Collection(Database(MongoClient(host=['localhost:27017'], document_class=dict,
tz_aware=False, connect=True), 'presidents'), 'animals')

wombat
ocelot
honey badger
```

Chapter 8 Exercises

Exercise 8-1 (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 located in the DATA folder of the student files.

The data has the following layout

Table 15. 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
termstart	date	YES	NULL
termend	date	YES	NULL
birthplace	varchar(128)	YES	NULL
birthstate	varchar(32)	YES	NULL
birthdate	date	YES	NULL
deathdate	date	YES	NULL
party	varchar(32)	YES	NULL

Refactor the **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.

NOTE	If you created a president.py module as part of an earlier lab, use that. Otherwise, use the supplied president.py module in the top folder of the student files.

Exercise 8-2 (add_pres_sqlite.py)

Add the next president to the presidents database. Just make up the data — let's keep this non-political. Don't use any real-life people.

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 9: 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

```
Connection: close
Content-Length: 50750
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Via: 1.1 vegur, 1.1 varnish, 1.1 varnish
Accept-Ranges: bytes
Date: Tue, 02 Mar 2021 13:16:47 GMT
Age: 1864
X-Served-By: cache-bwi5134-BWI, cache-fty21335-FTY
X-Cache: HIT, HIT
X-Cache-Hits: 1, 1
X-Timer: S1614691007.116671,VS0,VE1
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 Merriam-Webster dictionary API. It requires a word on the command line, which will be looked up in the online dictionary.
TIP	List of public RESTful APIs: http://www.programmableweb.com/apis/directory/1?protocol=REST

Example

web_content_consumer_urllib.py

```
#!/usr/bin/env python
"""
Fetch a word definition from Merriam-Webster's API
"""
import sys
from urllib.request import Request, urlopen
import json
# from pprint import pprint

DATA_TYPE = 'application/json'

API_KEY = 'b619b55d-faa3-442b-a119-dd906adc79c8'

URL_TEMPLATE =
'https://www.dictionaryapi.com/api/v3/references/collegiate/json/{}?key={}'  ①

def main(args):
    if len(args) < 1:
        print("Please specify a word to look up")
        sys.exit(1)

    search_term = args[0].replace(' ', '+')

    url = URL_TEMPLATE.format(search_term, API_KEY)  ②

    do_query(url)

def do_query(url):
    print("URL:", url)
    request = Request(url)
    response = urlopen(request)  ③
    raw_json_string = response.read().decode()  ④
    data = json.loads(raw_json_string)  ⑤
    # print("RAW DATA:")
    # pprint(data)
    for entry in data:  ⑥
        if isinstance(entry, dict):
            meta = entry.get("meta") ⑦
            if meta:
                part_of_speech = '({})'.format(entry.get('fl'))
                word_id = meta.get("id")
                print("{} {}".format(word_id.upper(), part_of_speech))
            if "shortdef" in entry:
                print('\n'.join(entry['shortdef']))
```

```
            print()
        else:
            print(entry)
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 iterate over each entry in results

⑦ retrieve items from results (JSON convert to lists and dicts)

web_content_consumer_urllib.py dewars

```
URL: https://www.dictionaryapi.com/api/v3/references/collegiate/json/wombat?key=b619b55d-
faa3-442b-a119-dd906adc79c8
WOMBAT (noun)
any of several stocky burrowing Australian marsupials (genera Vombatus and Lasiorhinus of
the family Vombatidae) resembling small bears
```

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
    if response.headers.get('content-type') == 'application/pdf':
        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

```
import sys
import requests

BASE_URL = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/'  ①

API_KEY = 'b619b55d-faa3-442b-a119-dd906adc79c8' ②


def main(args):
    if len(args) < 1:
        print("Please specify a search term")
        sys.exit(1)

    response = requests.get(
        BASE_URL + args[0],
        params={'key': API_KEY},
    )  ③

    if response.status_code == requests.codes.OK:
        data = response.json()  ④
        for entry in data: ⑤
            if isinstance(entry, dict):
                meta = entry.get("meta")
                if meta:
                    part_of_speech = '({})'.format(entry.get('fl'))
                    word_id = meta.get("id")
                    print("{} {}".format(word_id.upper(), part_of_speech))
                if "shortdef" in entry:
                    print('\n'.join(entry['shortdef']))
                print()
            else:
                print(entry)

    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 wombat

```
WOMBAT (noun)
any of several stocky burrowing Australian marsupials (genera Vombatus and Lasiorhinus of
the family Vombatidae) resembling small bears
```

TIP	See details of requests API at http://docs.python-requests.org/en/v3.0.0/api/#main-interface

Table 16. 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
from getpass import getpass  ①
import smtplib  ②
from email.message import EmailMessage  ③
from datetime import datetime

TIMESTAMP = datetime.now().ctime()  ④

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']
MESSAGE_SUBJECT = 'Python SMTP example'

MESSAGE_BODY = """
Hello at {}.

Testing email from Python
""".format(TIMESTAMP)

SMTP_USER = 'pythonclass'
SMTP_PASSWORD = getpass("Enter SMTP server password:")  ⑤

smtpserver = smtplib.SMTP("smtp2go.com", 2525)  ⑥
smtpserver.login(SMTP_USER, SMTP_PASSWORD)  ⑦

msg = EmailMessage()  ⑧
msg.set_content(MESSAGE_BODY)  ⑨
msg['Subject'] = MESSAGE_SUBJECT  ⑩
msg['from'] = SENDER  ⑪
msg['to'] = RECIPIENTS  ⑫

try:
    smtpserver.send_message(msg)  ⑬
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtpserver.quit()  ⑭
```

① module for hiding password

② module for sending email

③ module for creating message

④ get a time string for the current date/time

⑤ get password (not echoed to screen)

⑥ connect to SMTP server

⑦ log into SMTP server

⑧ create empty email message

⑨ add the message body

⑩ add the message subject

⑪ add the sender address

⑫ add a list of recipients

⑬ send the message

⑭ 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
from datetime import datetime
from imghdr import what  ①
from email.message import EmailMessage ②
from getpass import getpass ③


SMTP_SERVER = "smtp2go.com"  ④
SMTP_PORT = 2525

SMTP_USER = 'pythonclass'

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']


def main():
    smtp_server = create_smtp_server()
    now = datetime.now()
    msg = create_message(
        SENDER,
        RECIPIENTS,
        'Here is your attachment',
        'Testing email attachments from python class at {}\n\n'.format(now),
    )
    add_text_attachment('../DATA/parrot.txt', msg)
    add_image_attachment('../DATA/felix_auto.jpeg', msg)
    send_message(smtp_server, msg)


def create_message(sender, recipients, subject, body):
    msg = EmailMessage()  ⑤
    msg.set_content(body)  ⑥
    msg['From'] = sender
    msg['To'] = recipients
    msg['Subject'] = subject
    return msg


def add_text_attachment(file_name, message):
    with open(file_name) as file_in:  ⑦
        attachment_data = file_in.read()
    message.add_attachment(attachment_data)  ⑧
```

```
def add_image_attachment(file_name, message):
    with open(file_name, 'rb') as file_in:  ⑨
        attachment_data = file_in.read()
    image_type = what(None, h=attachment_data)  ⑩
    message.add_attachment(attachment_data, maintype='image', subtype=image_type)  ⑪


def create_smtp_server():
    password = getpass("Enter SMTP server password:")  ⑫
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)  ⑬
    smtpserver.login(SMTP_USER, password)  ⑭

    return smtpserver


def send_message(server, message):
    try:
        server.send_message(message)  ⑮
    finally:
        server.quit()


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

① module to determine image type

② module for creating email message

③ module for reading password privately

④ global variables for external information (IRL should be from environment — command line, config file, etc.)

⑤ create instance of EmailMessage to hold message

⑥ set content (message text) and various headers

⑦ read data for text attachment

⑧ add text attachment to message

⑨ read data for binary attachment

⑩ get type of binary data

⑪ add binary attachment to message, including type and subtype (e.g., "image/jpg)"

⑫ get password from user (don’t hardcode sensitive data in script)

⑬ create SMTP server connection

⑭ log into SMTP connection

⑮ send message

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 0
drwx------+  3 python  staff   96 Feb 11 06:19 Desktop
drwx------+  3 python  staff   96 Feb 11 06:19 Documents
drwx------+  3 python  staff   96 Feb 11 06:19 Downloads
drwx------+ 26 python  staff  832 Feb 11 06:19 Library
drwx------+  3 python  staff   96 Feb 11 06:19 Movies
drwx------+  3 python  staff   96 Feb 11 06:19 Music
drwx------+  3 python  staff   96 Feb 11 06:19 Pictures
drwxr-xr-x+  4 python  staff  128 Feb 11 06:19 Public
drwxr-xr-x   3 python  staff   96 Feb 18 11:11 text_files

------------------------------------------------------------
STDOUT:
-rw-r--r--  1 root  wheel  6946 Jun  5  2020 /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')  ⑤

    sftp.mkdir(REMOTE_DIR)  ⑥
    # sftp.put(local-file)
    # sftp.put(local-file, remote-file)
    sftp.put('../DATA/alice.txt', 'text_files/betsy.txt')  ⑦
    sftp.put('../DATA/alice.txt', 'alice.txt')
    sftp.put('../DATA/alice.txt', 'text_files')
    sftp.get(remote_file, 'eileen.txt')  ⑧

with paramiko.SSHClient() as ssh:  ⑨
    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)
```

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 path for remote file

⑥ create a folder on the remote host

⑦ copy a file to the remote host

⑧ copy a file from the remote host

⑨ use SSHClient to confirm operations (not needed, just for illustration)

paramiko_copy_files.py

```
drwx------   1 503      20             96 11 Feb 06:19 Music
-rw-------   1 503      20              3 11 Feb 06:19 .CFUserTextEncoding
drwx------   1 503      20             96 11 Feb 06:19 Pictures
drwxr-xr-x   1 503      20             96 18 Feb 11:11 text_files
drwx------   1 503      20             96 11 Feb 06:19 Desktop
drwx------   1 503      20            832 11 Feb 06:19 Library
drwxr-xr-x   1 503      20            128 11 Feb 06:19 Public
drwx------   1 503      20             96 11 Feb 06:19 Movies
drwx------   1 503      20             96 11 Feb 06:19 Documents
drwx------   1 503      20             96 11 Feb 06:19 Downloads
------------------------------------------------------------
```

Chapter 9 Exercises

Exercise 9-1 (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 9-2 (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 occurrences 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 9-3 (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 10: Effective Scripts

Objectives

- Launch external programs
- Check permissions on files
- Get system configuration information
- Store data offline
- Create Unix-style filters
- Parse command line options
- Configure application logging

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, '\n')

no_files = glob('../JUNK/*.avi')
print(no_files, '\n')
```

① expand file name wildcard into sorted list of matching names

glob_example.py

```
['../DATA/presidents_plus_biden.txt', '../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/littlewomen.txt', '../DATA/spam.txt', '../DATA/world_median_ages.txt',
'../DATA/phone_numbers.txt', '../DATA/sales_by_month.txt', '../DATA/engineers.txt',
'../DATA/underrated.txt', '../DATA/tolkien.txt', '../DATA/tyger.txt',
'../DATA/example_data.txt', '../DATA/states.txt', '../DATA/kjv.txt', '../DATA/fruit.txt',
'../DATA/areacodes.txt', '../DATA/float_values.txt', '../DATA/unabom.txt',
'../DATA/chaos.txt', '../DATA/noisewords.txt', '../DATA/presidents.txt',
'../DATA/bible.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', '../DATA/correspondence.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
#
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 17. 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

```
-rw-r--r--  1 jstrick  staff  3178541 Nov  2 09:27 ../DATA/tate_data.zip
-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
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-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:
-rw-r--r--  1 jstrick  staff  3178541 Nov  2 09:27 ../DATA/tate_data.zip
-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
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-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 (Windows only) 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:
-rw-r--r--  1 jstrick  staff  3178541 Nov  2 09:27 ../DATA/tate_data.zip
-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
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-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: -rw-r--r--  1 jstrick  staff     3178541 Nov  2 09:27 ../DATA/tate_data.zip
-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
-rw-r--r--  1 jstrick  staff      106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff      284160 Jul 26  2017 ../DATA/titanic3.xls
-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
-rw-r--r--  1 jstrick  students       22 Nov 24 09:05 spam.txt

--------------------------------------------------
Output: -rw-r--r--  1 jstrick  staff     3178541 Nov  2 09:27 ../DATA/tate_data.zip
-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
-rw-r--r--  1 jstrick  staff      106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff      284160 Jul 26  2017 ../DATA/titanic3.xls
-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
-rw-r--r--  1 jstrick  students       22 Nov 24 09:05 spam.txt

Error:
--------------------------------------------------
```

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:
    start_dir = "."
else:
    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/uri-schemes-1.csv is writable
../DATA/presidents.html is writable
../DATA/presidents.xlsx is writable
../DATA/presidents_plus_biden.txt is writable
../DATA/baby_names is writable
../DATA/presidents.db is writable
../DATA/testscores.dat is writable
../DATA/solar.json 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
#
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 18. 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 19. 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 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 20. Logging Levels

Level	Value
CRITICAL FATAL	50
ERROR	40
WARN WARNING	30
INFO	20
DEBUG	10
UNSET	0

Example

logging_simple.py

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

import logging

logging.basicConfig(
    filename='../TEMP/simple.log',
    level=logging.WARNING,
) 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***') ⑤
logging.info('The capital of North Dakota is Bismark') ⑥
```

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='%(name)s %(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.critical("this is critical")
```

① set the format for log entries

formatted.log

```
root 2021-03-02 08:16:50,011 INFO this is information
root 2021-03-02 08:16:50,011 WARNING this is a warning
root 2021-03-02 08:16:50,011 INFO this is information
root 2021-03-02 08:16:50,011 CRITICAL this is critical
```

Table 21. 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( ①
    filename='../TEMP/exception.log',
    level=logging.WARNING,  ②
)

for i in range(3):
    try:
        result = i/0
    except ZeroDivisionError:
        logging.exception('Logging with exception info') ③
```

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 10 Exercises

Exercise 10-1 (copy_files.py)

Write a script to find all text files (only the files that end in ".txt") 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 11: Sockets

Objectives

- Understanding socket concepts
- Creating a server socket
- Creating a client socket
- Creating a forking server application

Sockets

- One end of network connection
- Analogous to file handles
- One objects, two types: client and server
- Many different protocols

A socket is used to write a network client or server. A socket is one end of a network connection. When a socket is created, it can be used to read data from or write data to the other end.

The socket module is part of the Python standard library. It provides the socket class and some utility methods.

A socket is analogous to a file handle. It was designed that way long ago by the programmers of Berkeley Unix, to follow Unix's "everything looks like a file" rule, which keeps I/O simple.

There are two ways to use sockets – client and server. The same class, **socket.socket** is used for both. What makes a socket act like a client or server depends on how you configure it.

Sockets can be implemented for different kinds of protocols, but are usually used for TCP/IP, also known as the Internet protocol.

Socket options

- Socket domain (networking scheme)
  - AF_INET
  - AF_UNIX
- Socket type (transport protocol)
  - SOCK_STREAM
  - SOCK_DGRAM

There are two 'families' of network underpinnings that can be used by a socket. **AF_INET** is TCP/IP, the protocol used by nearly all network applications in use today. The other is AF_UNIX, which is an internal Unix protocol, and which can be used to communicate among processes on a Unix-like machine. AF_UNIX is unavailable on Windows machines.

The AF_INET protocol provides two socket types, which refers to the transport protocols used. **SOCK_STREAM** is for **TCP** connections, and is used by most applications. **SOCK_DGRAM** provides a **UDP** connection, which is faster, but doesn’t provide the reliable transport of TCP.

In the "old days", UDP was used for NFS (file sharing) and X (the Unix GUI), because it was significantly faster. It is still used for X, but NFS has been re-implemented to use TCP.

Server concepts

- Create socket
- Bind to server's IP + port/protocol
- Send request
- Receive response

A server starts up, binds to its own IP and a **transport protocol**, and then waits for a client request with the **accept()** method. When the request comes in, accept() returns a tuple with a new socket and the address of the client. The new socket is used to communicate with the client, and is closed when the client session ends. The accept() method is usually called in a loop. Each time it's called, it waits for the next client request.

Example

socket_server.py

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

import socket


def main():
    serv = setup()
    while True:
        (csock, addr) = serv.accept()  ①
        handle_client(csock)


def setup():
    serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  ②
    serv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  ③

    serv.bind((socket.gethostname(), 7777))  ④

    serv.listen(5)  ⑤

    return serv


def handle_client(cli_sock):
    request = cli_sock.recv(1024)  ⑥

    reply = request.upper()[::-1]  ⑦

    cli_sock.sendall(reply)  ⑧
    cli_sock.close()  ⑨


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

① blocks until client connects

② create server socket

③ reuse a port even if it is busy (usually no consequences from this)

④ listen on port 7777

⑤ mark socket as accepting connections; limit listen queue to 5 connections

⑥ read data (in bytes) from client

⑦ reverse client data and make it upper case

⑧ send reply to client

⑨ close client connection

Client concepts

- Create socket
- Connect to server's IP + port/protocol
- Send request
- Receive response

A client program creates a socket that connects to a server (service) by specifying three values: IP address, port number, and transport protocol.

The client program begins the network conversation by sending a request to the server. It then reads the response from the server. It may go back and forth more than once, but many application protocols involve only a single interaction with the server.

Example

socket_client.py

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

import sys
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  ①

s.connect((socket.gethostname(), 7777))  ②

if len(sys.argv) > 1:
    msg = sys.argv[1]
else:
    msg = "default message"

s.sendall(msg.encode())  ③

reply = s.recv(4096)  ④

print("Server said >{}<".format(reply.strip().decode()))  ⑤

s.close()  ⑥
```

① create a new TCP/IP socket

② connect to server via (host, port) tuple

③ send message (must be bytes, not str)

④ receive reply (as bytes) from server

⑤ decode message from bytes to str and strip any trailing whitespace

⑥ close connection

Application protocols

- Shared between client and server
- Typically request/response
- Messages can be
  - Fixed length
  - Header plus variable length
  - Delimited

There is no set standard for communicating between a client and a server. Instead, there are dozen of standards, defined for dozens of application domains. A standard for exchanging data between client and server (or between two peer nodes) is called an application protocol.

Some of the most commonly used application protocols are SMTP, HTTP, and SSH.

An application protocol can be binary or text-based, and can be fixed-length or variable length.

SMTP and HTTP are variable length, text-based protocols; the messages for these protocols resemble lines in a text file.

Other services (SSH, FTP, SNMP) use binary protocols, which contain fixed length headers whose datastreams can be decomposed into integers, bit flags, and so forth. The header generally describes the length of the data portion of the message.

Forking servers

- Create new process to handle client
- Main process waits for next client
- Less efficient for Windows

In real-life network applications, a new client request may come in before the server has finished with the previous request. This can result in a delay or failure to connect.

There are several ways of improving server throughput. One way is to create a forking server, which creates a new process to handle each client request.

TIP This is not an efficient technique for Windows – use threads instead.

Example

socket_server_forking.py

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

import socket
import os


def setup():
    serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    serv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # serv.setblocking(0)

    serv.bind((socket.gethostname(), 7777))

    serv.listen(5)

    return serv


def main():
    serv = setup()
    while True:
        (csock, addr) = serv.accept()
        handle_client(csock)


def handle_client(cli_sock):
    pid = os.fork()  ①
    if pid:  ②
        return
    request = cli_sock.recv(1024)  ③

    reply = request.upper()[::-1]  # upper & reversed

    cli_sock.sendall(reply)
    cli_sock.close()


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

① fork a new process; pid set to actual PID in parent, set to 0 in child

② if pid is non-0, then in parent, don’t do anything else

③ pid was 0, code is in child, handle client

Exercises

Exercise 1 (pres_server.py, pres_client.py)

Write a simple internet server for presidential data . It should listen on port 1776 (TCP) for client requests, and respond appropriately.

Write a client to get info from your server. It will connect to port 1776 (TCP) on the server machine, send a president index and field, read the response, and print it to the screen.

For the data, use any of your previous modules that provide a President class. If you haven't already written a President class, you can read the data from **presidents.txt** in the **DATA** folder.

Presidential Protocol

Requests

Requests to the server will be in the form:

```
TERM REQUEST\n
```

where TERM is the numeric index of the president (1-45), as a string, and REQUEST is any one of the following: FIRSTNAME LASTNAME BIRTHDATE DEATHDATE BIRTHPLACE BIRTHSTATE TERMSTART TERMEND PARTY

Responses

Responses will be newline ("\n") terminated strings containing the requested data.

For instance, to get the birth date for George Washington, send the following string:

```
1 BIRTHDATE\n
```

The reply should be the string

```
1732-02-22\n
```

If an invalid index is sent, return the string "ERROR: INDEX". If an invalid field name is sent, return the string "ERROR: FIELDNAME"

NOTE	How could you make this protocol more efficient (or elegant)?

Chapter 12: Errors and Exception Handling

Objectives

- Understanding syntax errors
- Handling exceptions with try-except-else-finally
- Learning the standard exception objects

Syntax errors

- Generated by the parser
- Cannot be trapped

Syntax errors are generated by the Python parser, and cause execution to stop (your script exits). They display the file name and line number where the error occurred, as well as an indication of where in the line the error occurred.

Because they are generated as soon as they are encountered, syntax errors may not be handled.

Example

```
  File "<stdin>", line 1
    for x in bargle
                  ^
SyntaxError: invalid syntax
```

TIP When running in interactive mode, the filename is <stdin>.

Exceptions

- Generated when runtime errors occur
- Usually fatal if not handled

Even if code is syntactically correct, errors can occur. A common run-time error is to attempt to open a non-existent file. Such errors are called exceptions, and cause the interpreter to stop with an error message.

Python has a hierarchy of builtin exceptions; handling an exception higher in the tree will handle any children of that exception.

TIP Custom exceptions can be created by sub-classing the Exception object.

Example

exception_unhandled.py

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

x = 5
y = "cheese"

z = x + y  ①
```

① Adding a string to an int raises **TypeError**

exception_unhandled.py

```
Traceback (most recent call last):
  File "exception_unhandled.py", line 6, in <module>
    z = x + y  ①
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Handling exceptions with try

- Use try/except clauses
- Specify expected exception

To handle an exception, put the code which might generate an exception in a try block. After the try block, you must specify a except block with the expected exception. If an exception is raised in the try block, execution stops and the interpreter looks for the exception in the except block. If found, it executes the except block and execution continues; otherwise, the exception is treated as fatal and the interpreter exits.

Example

exception_simple.py

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

try:  ①
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")

except TypeError as err:    ②
    print("Naughty programmer! ", err)

print("After try-except")  ③
```

① Execute code that might have a problem

② Catch the expected error; assign error object to **err**

③ Get here whether or not exception occurred

exception_simple.py

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
After try-except
```

Handling multiple exceptions

- Use a tuple of exception names, but with single argument

If your try clause might generate more than one kind of exception, you can specify a tuple of exception types, then the variable which will hold the exception object.

Example

exception_multiple.py

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except (IOError, TypeError) as err:  ①
    print("Naughty programmer! ", err)
```

① Use a tuple of 2 or more exception types

exception_multiple.py

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
```

Handling generic exceptions

- Use **Exception**
- Specify except with no exception list
- Clean up any uncaught exceptions

As a shortcut, you can specify **Exception** or an empty exception list. This will handle any exception that occurs in the try block.

Example

exception_generic.py

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except Exception as err: ①
    print("Naughty programmer! ", err)
```

① Will catch *any* exception

exception_generic.py

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
```

Ignoring exceptions

- Use the **pass** statement

Use the **pass** statement to do nothing when an exception occurs

Because the except clause must contain some code, the pass statement fulfills the syntax without doing anything.

Example

exception_ignore.py

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except(TypeError, IOError): ①
    pass
```

① Catch exceptions, and do nothing

exception_ignore.py

```
_no output_
```

This is probably a bad idea…

Using else

- executed if no exceptions were raised
- not required
- can make code easier to read

The last except block can be followed by an else block. The code in the else block is executed only if there were no exceptions raised in the try block. Exceptions in the else block are not handled by the preceding except blocks.

The else lets you make sure that some code related to the try clause (and before the finally clause) is only run if there's no exception, without trapping the exception specified in the except clause.

```
try:
    something_that_can_throw_ioerror()
except IOError as e:
    handle_the_IO_exception()
else:
# we don't want to catch this IOError if it's raised
    something_else_that_throws_ioerror()
finally:
    something_we_always_need_to_do()
```

Example

exception_else.py

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

numpairs = [(5, 1), (1, 5), (5, 0), (0, 5)]

total = 0

for x, y in numpairs:
    try:
        quotient = x / y
    except Exception as err:
        print("uh-oh, when y = {}, {}".format(y, err))
    else:
        total += quotient  ①
print(total)
```

① Only if no exceptions were raised

exception_else.py

```
uh-oh, when y = 0, division by zero
5.2
```

Cleaning up with finally

- Executed whether or not exception occurs
- Code executed whether or not exception raised
- Code runs even if **exit()** called
- For cleanup

A **finally** block can be used in addition to, or instead of, an **except** block. The code in a **finally** block is executed whether or not an exception occurs. The **finally** block is executed after the **try**, **except**, and **else** blocks.

What makes **finally** different from just putting statements after try-except-else is that the **finally** block will execute even if there is a **return()** or **exit()** in the **except** block.

The purpose of a **finally** block is to clean up any resources left over from the **try** block. Examples include closing network connections and removing temporary files.

Example

exception_finally.py

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

try:
    x = 5
    y = 37
    z = x + y
    print("z is", z)
except TypeError as err:    ①
    print("Caught exception:", err)
finally:
    print("Don't care whether we had an exception")  ②

print()

try:
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")
except TypeError as err:
    print("Caught exception:", err)
finally:
    print("Still don't care whether we had an exception")
```

① Catch **TypeError**

② Print whether **TypeError** is caught or not

exception_finally.py

```
z is 42
Don't care whether we had an exception

Caught exception: unsupported operand type(s) for +: 'int' and 'str'
Still don't care whether we had an exception
```

The Standard Exception Hierarchy (Python 3.7)

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
```

```
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
```

Chapter 12 Exercises

Exercise 12-1 (c2f_loop_safe.py)

Rewrite c2f_loop.py to handle the error that occurs if the user enters non-numeric data. The script should print a message and keep going if an error occurs.

Exercise 12-2 (c2f_batch_safe.py)

Rewrite c2f_batch.py to handle the ValueError that occurs if sys.argv[1] is not a valid number.

Chapter 13: Introduction to Python Classes

Objectives

- Understanding the big picture of OO programming
- Defining a class and its constructor
- Creating object methods
- Adding attributes and properties to a class
- Using inheritance for code reuse
- Adding class data and methods

About object-oriented programming

- Definitions of objects
- Can be used directly as well
- Objects contain data and methods

Python is an object-oriented language. It supports the creation of classes, which define *objects* (AKA *instances*).

Objects contain both data and the methods (functions) that operate on the data. Each object created has its own personal data, called *instance data*. It can also have data that is shared with all the objects created from its class, called *class data*.

Each class defines a *constructor* , which initializes and returns an object.

Objects may inherit attributes (data and methods) from other objects.

Methods are not polymorphic; i.e., you can't define multiple versions of a method, with different signatures, and have the corresponding method selected at runtime. However, because Python has dynamic typing, this is seldom needed.

Defining classes

- Use the **class** statement
- **Syntax**

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

To create a class, declare the class with the **class** statement. Any base classes may be specified in parentheses, but are not required.

Classes are conventionally named with Camel Case (i.e., all words, including the first, are capitalized). This is also known as CapWords, StudlyCaps, etc. Modules conventionally have lower-case names. Thus, it is usual to have module **rocketengine** containing class **RocketEngine**.

A method is a function defined in a class. All methods, including the constructor, are passed the object itself. This is conventionally named "self", and while this is not mandatory, most Python programmers expect it.

The basic layout is this:

```
class ClassName(baseclass):
    classvar = value

    def __init__:(self,...):
        self._attrib = instancevalue;
        ClassName.attrib = classvalue;

    def method1:(self,...):
        self._attrib = instancevalue

    def method2:(self,...):
        x = self.method1()
```

Example

simple_class.py

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

class Simple():  ①
    def __init__(self, message_text):  ②
        self._message_text = message_text  ③

    def text(self):  ④
        return self._message_text


if __name__ == "__main__":
    msg1 = Simple('hello')  ⑤
    print(msg1.text())  ⑥

    msg2 = Simple('hi there')  ⑦
    print(msg2.text())
```

① default base class is **object**

② constructor

③ message text stored in instance object

④ instance method

⑤ instantiate an instance of Simple

⑥ call instance method

⑦ create 2nd instance of Simple

simple_class.py

```
hello
hi there
```

Constructors

- Constructor is named **__init__**
- AKA initializer
- Passed *self* plus any parameters

A class's constructor (also known as the initializer) is named __init__. It receives the object being created, and any parameters passed into the initializer in the code as part of instantiation.

As with any Python function, the constructor's parameters can be fixed, optional, keyword-only, or keyword.

It is also normal to name data elements (variables) of a class with a leading underscore to indicate (in a non-mandatory way) that the variable is *private*. Access to private variables should be provided via public access methods (AKA getters) or properties.

Instance methods

- Expect the object as first parameter
- Object conventionally named *self*
- Otherwise like normal Python functions
- Use *self* to access instance attributes or methods
- Use class name to access class data

Instance methods are defined like normal functions, but like constructors, the object that the method is called from is passed in as the first parameter. As with the constructor, the parameter should be named *self*.

Example

animal.py

```
#!/usr/bin/env python
class Animal():
    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

animal.py

```
Leo is a African lion -- Roarrrrrrr
Garfield is a cat -- Meowwwww
Felix is a cat -- Meowwwww
```

Properties

- Properties are managed attributes
- Create with @property decorator
- Create getter, setter, deleter, docstring
- Specify getter only for read-only property

An object can have properties, or managed attributes. When a property is evaluated, its corresponding getter method is invoked; when a property is assigned to, its corresponding setter method is invoked.

Properties can be created with the @property decorator and its derivatives. @property applied to a method causes it to be a "getter" method for a property with the same name as the method.

Using @name.setter on a method with the same name as the property creates a setter method, and @name .deleter on a method with the same name creates a deleter method.

Why properties? Consider that you had a

Example

properties.py

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

class Person():

    def __init__(self, firstname=None, lastname=None):
        self._first_name = None
        self._last_name = None
        self.first_name = firstname  ①
        self.last_name = lastname

    @property
    def first_name(self):  ②
        return self._first_name

    @first_name.setter  ③
    def first_name(self, value):  ④
        if value is None or value.isalpha():
            self._first_name = value
        else:
            raise ValueError("First name may only contain letters")

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if value is None or value.isalpha():
            self._last_name = value
        else:
            raise ValueError("Last name may only contain letters")


if __name__ == '__main__':
    person1 = Person('Ferneater', 'Eulalia')

    person2 = Person()
    person2.last_name = 'Pepperpot'  ⑤
    person2.first_name = 'Hortense'

    print("{} {}".format(person1.first_name, person1.last_name))
    print("{} {}".format(person2.first_name, person2.last_name))
```

```
    try:
        person3 = Person("R2D2")
    except ValueError as err:
        print(err)
    else:
        print("{} {}".format(person3.first_name, person3.last_name))
```

① calls property

② getter property

③ decorator comes from getter property

④ setter property

⑤ access property

properties.py

```
Ferneater Eulalia
Hortense Pepperpot
First name may only contain letters
```

Class methods and data

- Defined in the class, but outside of methods
- Defined as attribute of class name (similar to self)
- Define class methods with @classmethod
- Class methods get the class object as 1st parameter

Most classes need to store some data that is common to all objects created in the class. This is generally called class data.

Class attributes can be created by using the class name directly, or via class methods.

A class method is created by using the @classmethod decorator. Class methods are implicitly passed the class object.

Class methods can be called from the class object or from an instance of the class; in either case the method is passed the class object.

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 class data (not duplicated in instances)

② instance method

③ the **@classmethod** decorator makes a function receive the class object, not the instance object

④ *get_location() is a *class* method

⑤ class methods can access class data via **cls**

⑥ call class method from class

⑦ call class method from instance

class_methods_and_data.py

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

Static Methods

- Define with @staticmethod

A static method is a utility method that is included in the API of a class, but does not require either an instance or a class object. Static methods are not passed any implicit parameters.

Many classes do not need any static methods.

Define static methods with the @staticmethod decorator.

Example

```
class Spam():

    @staticmethod
    def format_as_title(s):   # no implicit parameters
        return s.strip().title()
```

Private methods

- Called by other methods in the class
- Not visible to users of the class
- Conventionally named with leading underscore

Private methods are those that are called only within the class. They are not part of the API – they are not visible to users of the class. Private methods may be instance, class, or static methods.

Name private methods with a leading underscore. This does not protect it from use, but gives programmers a hint that it's for internal use only.

Inheritance

- Specify base classes after class name
- Multiple inheritance OK
- Depth-first, left-to-right search for methods not in derived class

Classes may inherit methods and data. Specify a parenthesized list of base classes after the class name in the class definition.

If a method or attribute is not found in the derived class, it is first sought in the first base class in the list. If not found, it is sought in the base class of that class, if any, and so on. This is usually called a depth-first search.

The derived class inherits all attributes of the base class. If the base class initializer takes the same arguments as the derived class, then no extra coding is needed. Otherwise, to explicitly call the initializer in the base class, use super().__init__(args).

The simplest derived class would be:

```
class Mammal(Animal):
    pass
```

A Mammal object will have all the attributes and methods of an Animal object.

Example

mammal.py

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

from animal import Animal


class Mammal(Animal):  ①
    def __init__(self, species, name, sound, gestation):
        super(Mammal, self).__init__(species, name, sound)
        self._gestation = gestation

    @property
    def gestation(self):  ②
        """Length of gestation period in days"""
        return self._gestation


if __name__ == "__main__":
    mammal1 = Mammal("African lion", "Bob", "Roarrrr", 120)
    print(mammal1.name, "is a", mammal1.species, "--", end=' ')
    mammal1.make_sound()

    print("Number of animals", mammal1.zoo_size())

    mammal2 = Mammal("Fruit bat", "Freddie", "Squeak!!", 180)
    print(mammal2.name, "is a", mammal2.species, "--", end=' ')
    mammal2.make_sound()

    print("Number of animals", mammal2.zoo_size())
    print("Number of animals", Mammal.zoo_size())

    mammal1.kill()
    print("Number of animals", Mammal.zoo_size())

    print("Gestation period of the", mammal1.species, "is", mammal1.gestation, "days")
    print("Gestation period of the", mammal2.species, "is", mammal2.gestation, "days")
```

① inherit from Animal

② add property to existing Animal properties

mammal.py

```
Bob is a African lion -- Roarrrr
Number of animals 1
Freddie is a Fruit bat -- Squeak!!
Number of animals 2
Number of animals 2
Number of animals 1
Gestation period of the African lion is 120 days
Gestation period of the Fruit bat is 180 days
```

Untangling the nomenclature

There are many terms to distinguish the various parts of a Python program. This chart is an attempt to help you sort out what is what:

Table 22. Objected-oriented Nomenclature

attribute	A variable or method that is part of a class or object
base class	A class from which other classes inherit
child class	Same as derived class
class	A Python module from which objects may be created
class method	A function that expects the class object as its first parameter. Such a function can be called from either the class itself or an instance of the class. Created with @classmethod decorator.
derived class	A class which inherits from some other class
function	An executable subprogram.
instance method	A function that expects the instance object, conventionally named self, as its first parameter. See "method".
method	A function defined inside a class.
module	A file containing python code, and which is designed to be imported into Python scripts or other modules.
package	A folder containing one or more modules. Packages may be imported. There must be a file named __init__.py in the package folder.
parent class	Same as base class
property	A managed attribute (variable) of an instance of a class
script	A Python program. A script is an executable file containing Python commands.
static method	A function in a class that does not automatically receive any parameters; typically used for private utility functions. Created with @staticmethod decorator.
superclass	Same as base class

Chapter 13 Exercises

Exercise 13-1 (knight.py, knight_info.py)

Part 1:

Create a module which defines a class named **Knight**.

The initializer for the class should expect the knight's name as a parameter. Get the information from the file **knights.txt** to initialize the object.

The object should have these (read-only) properties:

name
title
favorite_color
quest
comment

Example

```
from knight import Knight
k = Knight('Arthur')
print k.favorite_color
```

Part 2:

Create an application to use the **Knight** class created in part one. For each knight specified on the command line, create a **knight** object and print out the knight's name, favorite color, quest, and comment. Precede the name with the knight's title.

Example output:

```
Arthur Bedevere
Name: King Arthur
Favorite Color: blue
Quest: The Grail
Comment: King of the Britons

Name: Sir Bedevere
Favorite Color: red, no, blue!
Quest: The Grail
Comment: AARRRRRRRGGGGHH
```

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, 3))
    print("Hello from thread {}".format(num))


for i in range(10):
    t = threading.Thread(target=doit, args=(i,))  ②
    t.start()  ③

print("Done.")  ④
```

① function to launch in each thread

② create thread

③ launch thread

④ "Done" is printed immediately — the threads are "in the background"

thr_noclass.py

```
Done.
Hello from thread 3
Hello from thread 0
Hello from thread 5
Hello from thread 6
Hello from thread 9
Hello from thread 1
Hello from thread 2
Hello from thread 7
Hello from thread 4
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

from threading import Thread
import random
import time


class SimpleThread(Thread):
    def __init__(self, num):
        super().__init__()  ①
        self._threadnum = num

    def run(self):  ②
        time.sleep(random.randint(1, 3))
        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 5
Hello from thread 8
Hello from thread 2
Hello from thread 6
Hello from thread 1
Hello from thread 9
Hello from thread 0
Hello from thread 3
Hello from thread 4
Hello from thread 7
```

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 threading  ①
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, random_word in enumerate(WORDS, 1):
    t = SimpleThread(i, random_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 start thread

⑫ wait for thread to finish

thr_locking.py

```
All threads launched...
Hello from thread 1 (apple)
Hello from thread 7 (lemon)
Hello from thread 9 (fig)
Hello from thread 2 (banana)
Hello from thread 4 (peach)
Hello from thread 5 (papaya)
Hello from thread 6 (cherry)
Hello from thread 10 (elderberry)
Hello from thread 3 (mango)
Hello from thread 8 (watermelon)
['APPLE', 'LEMON', 'FIG', 'BANANA', 'PEACH', 'PAPAYA', 'CHERRY', 'ELDERBERRY', 'MANGO',
'WATERMELON']
```

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 random
import queue
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


class RandomWord():  ③
    def __init__(self):
        with open('../DATA/words.txt') as words_in:
            self._words = [word.rstrip('\n\r') for word in words_in.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):
    worker_name = "Worker {:c}".format(i + 65)
    w = Worker(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

```
['DISGUISED-DISGUISED', 'UNTACKS-UNTACKS', 'PROSELYTING-PROSELYTING', 'YESHIVOT-
YESHIVOT', 'PEPPERINESSES-PEPPERINESSES', 'FAINTS-FAINTS', 'FLOTAGES-FLOTAGES',
'KEESHOND-KEESHOND', 'NONPHILOSOPHICAL-NONPHILOSOPHICAL', 'JUGGERNAUT-JUGGERNAUT',
'PERSONNELS-PERSONNELS', 'BIOCHEMIST-BIOCHEMIST', 'IGNOMINY-IGNOMINY', 'SEIZIN-SEIZIN',
'QOPH-QOPH', 'FITTABLE-FITTABLE', 'PROLETARIANISED-PROLETARIANISED', 'INDECOROUSNESSES-
INDECOROUSNESSES', 'UNDERACTS-UNDERACTS', 'WHITEFISHES-WHITEFISHES']
Tue Mar  2 08:16:59 2021
Tue Mar  2 08:17:00 2021
```

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 sys
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():  ②
    def __init__(self):
        with open('../DATA/words.txt') as words_in:
            self._words = [word.rstrip('\n\r') for word in words_in]
        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, queue, lock, result):  ⑤
        Process.__init__(self)
        self.queue = queue
        self.result = result
        self.lock = lock
        self.name = name

    def run(self):  ⑥
        while True:
            try:
                word = self.queue.get(block=False)  ⑦
                word = word.upper()  ⑧
                with self.lock:
                    self.result.append(word)  ⑨

            except Empty:  ⑩
                break


if __name__ == '__main__':
    if sys.platform == 'win32':
        freeze_support()

    word_queue = 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()
        word_queue.put(w)  ⑯

    start_time = time.ctime()
```

```
    pool = []  ⑰
    for i in range(POOL_SIZE):  ⑱
        worker_name = "Worker {:03d}".format(i)
        w = Worker(worker_name, word_queue, 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(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

print last 50 entries in shared result

multi_processing.py

```
['NAVALLY', 'PLASTID', 'RIVERSIDE', 'RESTORED', 'VAKIL', 'BAGGER', 'POLYLYSINES',
'PINENE', 'LYSOGENISING', 'RIDICULOUSLY', 'CERIPH', 'SHAITAN', 'UNBAR', 'BIFIDLY',
'FUMIGATE', 'DISOWNMENT', 'ELATERS', 'INSPECTORSHIP', 'TOLANES', 'DINK', 'ARISEN',
'ACTIVISTIC', 'FEUDALISTS', 'JACKLEGS', 'COUNTERCOMPLAINTS', 'CHURNINGS', 'TUBATE',
'ANAPHYLACTOID', 'TRUNCATED', 'COUTURIER', 'ANAEROBIOSES', 'INARCH', 'MARKETER',
'FICKLER', 'DUKE', 'TAMARI', 'COUNTERMELODY', 'COMMENSAL', 'JUG', 'HYPERCOMPLEX',
'JACKAROOS', 'PHYLLOIDS', 'WOODSTOVE', 'ASTRONAUTIC', 'COLESLAWS', 'SANGARS', 'IGNEOUS',
'DEFAULTED', 'HONDLES', 'OVERPRINTING']
25000
Tue Mar  2 08:17:00 2021
Tue Mar  2 08:17:02 2021
```

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

```
['INTERROGATIVELY', 'PRONATORS', 'KLUDGE', 'HEDGEPIGS', 'AMPERAGE', 'PANHANDLER',
'EJECTOR', 'GENIAL', 'GRAYNESSES', 'MAGNETOHYDRODYNAMICS', 'PREDISCOVERIES',
'PROPRANOLOL', 'WAGERS', 'OUTPOPULATING', 'WUTHERS', 'RADIOCARBON', 'PERSPICUITY',
'GROUCH', 'PHOTOINDUCTIONS', 'MAPPINGS']
Processed 173466 words.
```

Example

thr_pool.py

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

import random
from multiprocessing.dummy import Pool ①

POOL_SIZE = 30 ②

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in] ③

random.shuffle(WORDS) ④

def my_task(word):  ⑤
    return word.upper()

tpool = Pool(POOL_SIZE) ⑥

WORD_LIST = tpool.map(my_task, WORDS) ⑦

print(WORD_LIST[:20])  ⑧

print("Processed {} words.".format(len(WORD_LIST)))
```

① get the thread pool object

② set # of threads to create

③ get list of 175K words

④ shuffle the word list <5>

thr_pool.py

```
['HOMOLOGATES', 'CORPOSANTS', 'CLIQUISHLY', 'SCREENLANDS', 'GAINFULNESSES', 'IMITABLE',
'ASSIMILATIVE', 'PHARMACOPOEIAL', 'LEECHING', 'UNMOLESTED', 'CONFOUNDING',
'PELLETIZATION', 'THEREFORE', 'MERETRICIOUS', 'RESPRAY', 'ABIOGENISTS', 'LINKSMAN',
'QUADRUPEDS', 'ARBORVITAES', 'MIDI']
Processed 173466 words.
```

Example

thr_pool_mw.py

```
#!/usr/bin/env python
from multiprocessing.dummy import Pool  ①
from pprint import pprint
import requests

POOL_SIZE = 4

BASE_URL = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/'  ②

API_KEY = 'b619b55d-faa3-442b-a119-dd906adc79c8' ③

search_terms = [  ④
    'wombat',
    'frog', 'muntin', 'automobile', 'green', 'connect',
    'vial', 'battery', 'computer', 'sing', 'park',
    'ladle', 'ram', 'dog', 'scalpel'
]


def fetch_data(term):  ⑤
    try:
        response = requests.get(
            BASE_URL + term,
            params={'key': API_KEY},
        )  ⑥
    except requests.HTTPError as err:
        print(err)
        return []
    else:
        data = response.json()  ⑦
        parts_of_speech = []
        for entry in data: ⑧
            if isinstance(entry, dict):
                meta = entry.get("meta")
                if meta:
                    part_of_speech = entry.get("fl")
                    if part_of_speech:
                        parts_of_speech.append(part_of_speech)
        return sorted(set(parts_of_speech))  ⑨


p = Pool(POOL_SIZE)  ⑩

results = p.map(fetch_data, search_terms)  ⑪
```

```
for search_term, result in zip(search_terms, results):  ⑫
    print("{}:".format(search_term.upper()))
    if result:
        print(result)
    else:
        print("** no results **")
```

① .dummy has Pool for threads

② base url of site to access

③ credentials to access site

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 loop over entries matching search terms

⑨ return list of parsed entries matching search term

⑩ create pool of POOL_SIZE threads

⑪ launch threads, collect results

⑫ iterate over results, mapping them to search terms

...

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

For each exercise, ask the questions: Should this be multi-threaded or multi-processed? Distributed or local?

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.

Exercise 14-2 (folder_scanner.py)

Write a program that takes in a directory name on the command line, then traverses all the files in that directory tree and prints out a count of:

- how many total files
- how many total lines (count '\n')
- how many bytes (len() of file contents)

HINT: Use either a thread or a process pool in combination with **os.walk()**.

FOR ADVANCED STUDENTS

Exercise 14-3 (web_spider.py)

Write a website-spider. Given a domain name, it should crawl the page at that domain, and any other URLs from that page with the same domain name. Limit the number of parallel requests to the web server to no more than 4.

Exercise 14-4 (sum_tuple.py)

Write a function that will take in two large arrays of integers and a target. It should return an array of tuple pairs, each pair being one number from each input array, that sum to the target value.

Appendix A: Python Bibliography

Title	Author	Publisher
Data Science		
Building machine learning systems with Python	William Richert, Luis Pedro Coelho	Packt Publishing
High Performance Python	Mischa Gorlelick and Ian Ozsvald	O'Reilly Media
Introduction to Machine Learning with Python	Sarah Guido	O'Reilly & Assoc.
iPython Interactive Computing and Visualization Cookbook	Cyril Rossant	Packt Publishing
Learning iPython for Interactive Computing and Visualization	Cyril Rossant	Packt Publishing
Learning Pandas	Michael Heydt	Packt Publishing
Learning scikit-learn: Machine Learning in Python	Raúl Garreta, Guillermo Moncecchi	Packt Publishing
Mastering Machine Learning with Scikit-learn	Gavin Hackeling	Packt Publishing
Matplotlib for Python Developers	Sandro Tosi	Packt Publishing
Numpy Beginner's Guide	Ivan Idris	Packt Publishing
Numpy Cookbook	Ivan Idris	Packt Publishing
Practical Data Science Cookbook	Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta	Packt Publishing
Python Text Processing with NLTK 2.0 Cookbook	Jacob Perkins	Packt Publishing
Scikit-learn cookbook	Trent Hauck	Packt Publishing
Python Data Visualization Cookbook	Igor Milovanovic	Packt Publishing
Python for Data Analysis	Wes McKinney	O'Reilly & Assoc.
Design Patterns		
Design Patterns: Elements of Reusable Object-Oriented Software	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides	Addison-Wesley Professional

Title	Author	Publisher
Head First Design Patterns	Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra	O'Reilly Media
Learning Python Design Patterns	Gennadiy Zlobin	Packt Publishing
Mastering Python Design Patterns	Sakis Kasampalis	Packt Publishing
General Python development		
Expert Python Programming	Tarek Ziadé	Packt Publishing
Fluent Python	Luciano Ramalho	O'Reilly & Assoc.
Learning Python, 2nd Ed.	Mark Lutz, David Asher	O'Reilly & Assoc.
Mastering Object-oriented Python	Stephen F. Lott	Packt Publishing
Programming Python, 2nd Ed.	Mark Lutz	O'Reilly & Assoc.
Python 3 Object Oriented Programming	Dusty Phillips	Packt Publishing
Python Cookbook, 3nd. Ed.	David Beazley, Brian K. Jones	O'Reilly & Assoc.
Python Essential Reference, 4th. Ed.	David M. Beazley	Addison-Wesley Professional
Python in a Nutshell	Alex Martelli	O'Reilly & Assoc.
Python Programming on Win32	Mark Hammond, Andy Robinson	O'Reilly & Assoc.
The Python Standard Library By Example	Doug Hellmann	Addison-Wesley Professional
Misc		
Python Geospatial Development	Erik Westra	Packt Publishing
Python High Performance Programming	Gabriele Lanaro	Packt Publishing
Networking		
Python Network Programming Cookbook	Dr. M. O. Faruque Sarker	Packt Publishing
Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers	T J O'Connor	Syngress
Web Scraping with Python	Ryan Mitchell	O'Reilly & Assoc.
Testing		

Title	Author	Publisher
Python Testing Cookbook	Greg L. Turnquist	Packt Publishing
Learning Python Testing	Daniel Arbuckle	Packt Publishing
Learning Selenium Testing Tools, 3rd Ed.	Raghavendra Prasad MG	Packt Publishing
Web Development		
Building Web Applications with Flask	Italo Maia	Packt Publishing
Django 1.0 Website Development	Ayman Hourieh	Packt Publishing
Django 1.1 Testing and Development	Karen M. Tracey	Packt Publishing
Django By Example	Antonio Melé	Packt Publishing
Django Design Patterns and Best Practices	Arun Ravindran	Packt Publishing
Django Essentials	Samuel Dauzon	Packt Publishing
Django Project Blueprints	Asad Jibran Ahmed	Packt Publishing
Flask Blueprints	Joel Perras	Packt Publishing
Flask by Example	Gareth Dwyer	Packt Publishing
Flask Framework Cookbook	Shalabh Aggarwal	Packt Publishing
Flask Web Development	Miguel Grinberg	O'Reilly & Assoc.
Full Stack Python (e-book only)	Matt Makai	Gumroad (or free download)
Full Stack Python Guide to Deployments (e-book only)	Matt Makai	Gumroad (or free download)
High Performance Django	Peter Baumgartner, Yann Malet	Lincoln Loop
Instant Flask Web Development	Ron DuPlain	Packt Publishing
Learning Flask Framework	Matt Copperwaite, Charles O Leifer	Packt Publishing
Mastering Flask	Jack Stouffer	Packt Publishing
Two Scoops of Django: Best Practices for Django 1.11	Daniel Roy Greenfeld, Audrey Roy Greenfeld	Two Scoops Press
Web Development with Django Cookbook	Aidas Bendoraitis	Packt Publishing

Appendix B: String Formatting

Overview

- Strings have a format() method
- Allows values to be inserted in strings
- Values can be formatted
- Add a field as placeholders for variable
- Field syntax: `{SELECTOR:FORMATTING}`
- Selector can be index or keyword
- Formatting controls alignment, width, padding, etc.

Python provides a powerful and flexible way to format data. The string method format() takes one or more parameters, which are inserted into the string via placeholders.

The placeholders, called fields, consist of a pair of braces enclosing parameter selectors and formatting directives.

The selector can be followed by a set of formatting directives, which always start with a colon. The simplest directives specify the type of variable to be formatted. For instance, {1:d} says to format the second parameter as an integer; {0:.2f} says to format the first parameter as a float, rounded to two decimal points.

The formatting part can consist of the following components, which will be explained in detail in the following pages:

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

Parameter Selectors

- Null for auto-numbering
- Can be numbers or keywords
- Start at 0 for numbers

Selectors refer to which parameter will be used in a placeholder.

Null (empty) selectors — the most common — will be treated as though they were filled in with numbers from left to right, beginning with 0. Null selectors cannot be mixed with numbered or named selectors — either all of the selectors or none of the selectors must be null.

Non-null selectors can be either numeric indices or keywords (strings). Thus, {0} will be replaced with the first parameter, {4} will be replaced with the fifth parameter, and so on. If using keywords, then {name} will be replaced by the value of keyword 'name', and {age} will be replaced by keyword 'age'.

Parameters do not have to be in the same order in which they occur in the string, although they typically are. The same parameter can be used in multiple fields.

If positional and keyword parameters are both used, the keyword parameters must come after all positional parameters.

Example

fmt_params.py

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

person = 'Bob'
age = 22

print("{0} is {1} years old.".format(person, age))  ①
print("{0}, {0}, {0} your boat".format('row'))  ②
print("The {1}-year-old is {0}".format(person, age))  ③
print("{name} is {age} years old.".format(name=person, age=age))  ④
print()
print("{} is {} years old.".format(person, age))  ⑤
print("{name} is {} and his favorite color is {}".format(22, 'blue', name='Bob'))  ⑥
```

① Placeholders can be numbered

② Placeholders can be reused

③ They do not have to be in order (but usually are)

④ Selectors can be named

⑤ Empty selectors are autonumbered (but all selectors must either be empty or explicitly numbered)

⑥ Named and numbered selectors can be mixed

fmt_params.py

```
Bob is 22 years old.
row, row, row your boat
The 22-year-old is Bob
Bob is 22 years old.

Bob is 22 years old.
Bob is 22 and his favorite color is blue
```

Data types

- Fields can specify data type
- Controls formatting
- Raises error for invalid types

The type part of the format directive tells the formatter how to convert the value. Builtin types have default formats – 's' for strings, 'd' for integers, 'f' for float.

Some data types can be specified as either upper or lower case. This controls the output of letters. E.g, {:x} would format the number 48879 as 'beef', but {:X} would format it as 'BEEF'.

The type must generally match the type of the parameter. An integer cannot be formatted with type 's'. Integers can be formatted as floats, but not the other way around. Only integers may be formatted as binary, octal, or hexadecimal.

Example

fmt_types.py

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

person = 'Bob'
value = 488
bigvalue = 3735928559
result = 234.5617282027

print('{:s}'.format(person))    ①
print('{name:s}'.format(name=person))    ②
print('{:d}'.format(value))    ③
print('{:b}'.format(value))    ④
print('{:o}'.format(value))    ⑤
print('{:x}'.format(value))    ⑥
print('{:X}'.format(bigvalue))    ⑦
print('{:f}'.format(result))    ⑧
print('{:.2f}'.format(result))    ⑨
```

① String

② String

③ Integer (displayed as decimal)

④ Integer (displayed as binary)

⑤ Integer (displayed as octal)

⑥ Integer (displayed as hex)

⑦ Integer (displayed as hex with uppercase digits)

⑧ Float (defaults to 6 places after the decimal point)

⑨ Float rounded to 2 decimal places

fmt_types.py

```
Bob
Bob
488
111101000
750
1e8
DEADBEEF
234.561728
234.56
```

Table 23. Formatting Types

b	Binary – converts number to base 2
c	Character – converts to corresponding character, like chr()
d	Decimal – outputs number in base 10
e, E	Exponent notation. 'e' prints the number in scientific notation using the letter 'e' to indicate the exponent. 'E' is the same, except it uses the letter 'E'
f,F	Floating point. 'F' and 'f' are the same.
g	General format. For a given precision p >= 1, rounds the number to p significant digits and then formats the result in fixed-point or scientific notation, depending on magnitude. This is the default for numbers
G	Same as g, but upper-cases 'e', 'nan', and 'inf"
n	Same as d, but uses locale setting for number separators
o	Octal – converts number to base 8
s	String format. This is the default type for strings
x, X	Hexadecimal – convert number to base 16; A-F match case of 'x' or 'X'
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

Field Widths

- Specified as {0:width.precision}
- Width is really minimum width
- Precision is either maximum width or # decimal points

Fields can specify a minimum width by putting a number before the type. If the parameter is shorted than the field, it will be padded with spaces, on the left for numbers, and on the right for strings.

The precision is specified by a period followed by an integer. For strings, precision means the maximum width. Strings longer than the maximum will be truncated. For floating point numbers, precision means the number of decimal places displayed, which will be padded with zeros as needed.

Width and precision are both optional. The default width for all fields is 0; the default precision for floating point numbers is 6.

It is invalid to specify precision for an integer.

Example

fmt_width.py

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

name = 'Ann Elk'
value = 10000
airspeed = 22.347
# note: [] are used to show blank space, and are not part of the formatting
print('[{:s}]'.format(name))     ①
print('[{:10s}]'.format(name))   ②
print('[{:3s}]'.format(name))    ③
print('[{:3.3s}]'.format(name))  ④
print()
print('[{:8d}]'.format(value))       ⑤
print('[{:8f}]'.format(value))       ⑥
print('[{:8f}]'.format(airspeed))    ⑦
print('[{:.2f}]'.format(airspeed))   ⑧
print('[{:8.3f}]'.format(airspeed))  ⑨
```

① Default format — no padding

② Left justify, 10 characters wide

③ Left justify, 3 characters wide, displays entire string

④ Left justify, 3 characters wide, truncates string to max width

⑤ Right justify, decimal, 8 characters wide (all numbers are right-justified by default)

⑥ Right justify int as float, 8 characters wide

⑦ Right justify float as float, 8 characters wide

⑧ Right justify, float, 3 decimal places, no maximum width

⑨ Right justify, float, 3 decimal places, maximum width 8

fmt_width.py

```
[Ann Elk]
[Ann Elk   ]
[Ann Elk]
[Ann]

[   10000]
[10000.000000]
[22.347000]
[22.35]
[  22.347]
```

Alignment

- Alignment within field can be left, right, or centered
  - < left align
  - > right align
  - ^ center
  - = right align but put padding after sign

You can align the data to be formatted. It can be left-aligned (the default), right-aligned, or centered. If formatting signed numbers, the minus sign can be placed on the left side.

Example

fmt_align.py

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

name = 'Ann'
value = 12345
nvalue = -12345

①
print('[{0:10s}]'.format(name))    ②
print('[{0:<10s}]'.format(name))   ③
print('[{0:>10s}]'.format(name))   ④
print('[{0:^10s}]'.format(name))   ⑤
print()
print('[{0:10d}] [{1:10d}]'.format(value, nvalue))    ⑥
print('[{0:>10d}] [{1:>10d}]'.format(value, nvalue))  ⑦
print('[{0:<10d}] [{1:<10d}]'.format(value, nvalue))  ⑧
print('[{0:^10d}] [{1:^10d}]'.format(value, nvalue))  ⑨
print('[{0:=10d}] [{1:=10d}]'.format(value, nvalue))  ⑩
```

① note: all of the following print in a field 10 characters widedd

② Default (left) alignment

③ Explicit left alignment

④ Right alignment

⑤ Centered

⑥ Default (right) alignment

⑦ Explicit right alignment

⑧ Left alignment

⑨ Centered

⑩ Right alignment, but pad *after* sign

fmt_align.py

```
[Ann        ]
[Ann        ]
[        Ann]
[   Ann     ]

[      12345] [    -12345]
[      12345] [    -12345]
[12345      ] [-12345    ]
[  12345    ] [  -12345  ]
[      12345] [-    12345]
```

Fill characters

- Padding character must precede alignment character
- Default is one space
- Can be any character except }

By default, if a field width is specified and the data does not fill the field, it is padded with spaces. A character preceding the alignment character will be used as the fill character.

Example

fmt_fill.py

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

name = 'Ann'
value = 123

print('[{:>10s}]'.format(name))    ①
print('[{:.>10s}]'.format(name))   ②
print('[{:->10s}]'.format(name))    ③
print('[{:.10s}]'.format(name))    ④
print()
print('[{:10d}]'.format(value))    ⑤
print('[{:010d}]'.format(value))   ⑥
print('[{:_>10d}]'.format(value))  ⑦
print('[{:+>10d}]'.format(value))  ⑧
```

① Right justify string, pad with space (default)

② Right justify string, pad with '.'

③ Right justify string, pad with '-'

④ Left justify string, pad with '.'

⑤ Right justify number, pad with space (default

⑥ Right justify number, pad with zeroes

⑦ Right justfy, pad with '_' ('>' required)

⑧ Right justfy, pad with '+' ('>' required)

fmt_fill.py

```
[       Ann]
[.......Ann]
[-------Ann]
[Ann]

[       123]
[0000000123]
[_______123]
[+++++++123]
```

Signed numbers

- Can pad with any character except '{}'
- Sign can be '+', '-', or space
- Only appropriate for numeric types

The sign character follows the alignment character, and can be plus, minus, or space.

A plus sign means always display + or – preceding non-zero numbers.

A minus sign means only display a sign for negative numbers.

A space means display a – for negative numbers and a space for positive numbers.

Example

fmt_signed.py

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

values = 123, -321, 14, -2, 0

for value in values:
    print("default: |{:d}|".format(value))  ①
print()

for value in values:
    print("   plus: |{:+d}|".format(value))  ②
print()

for value in values:
    print("  minus: |{:-d}|".format(value)) ③
print()

for value in values:
    print("  space: |{: d}|".format(value)) ④
print()
```

① default (pipe symbols just to show white space)

② plus sign puts '+' on positive numbers (and zero) and '-' on negative

③ minus sign only puts '-' on negative numbers

④ space puts '-' on negative numbers and space on others

fmt_signed.py

```
default: |123|
default: |-321|
default: |14|
default: |-2|
default: |0|

   plus: |+123|
   plus: |-321|
   plus: |+14|
   plus: |-2|
   plus: |+0|

  minus: |123|
  minus: |-321|
  minus: |14|
  minus: |-2|
  minus: |0|

  space: | 123|
  space: |-321|
  space: | 14|
  space: |-2|
  space: | 0|
```

Parameter Attributes

- Specify elements or properties in template
- No need to repeat parameters
- Works with sequences, mappings, and objects

When specifying container variables as parameters, you can select elements in the format rather than in the parameter list. For sequences or dictionaries, index on the selector with []. For object attributes, access the attribute from the selector with . (period).

Example

fmt_attrib.py

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

from datetime import date

fruits = 'apple', 'banana', 'mango'
values = [5, 18, 27, 6]
dday = date(1944, 6, 6)
pythons = {'Idle': 'Eric', 'Cleese': 'John', 'Gilliam': 'Terry',
           'Chapman': 'Graham', 'Palin': 'Michael', 'Jones': 'Terry'}

print('{0[0]} {0[2]}'.format(fruits))  ①
print('{f[0]} {f[2]}'.format(f=fruits))  ②
print()
print('{0[0]} {0[2]}'.format(values))  ③
print()
print('{0[Palin]} {0[Cleese]}'.format(pythons))  ④
print('{names[Palin]} {names[Cleese]}'.format(names=pythons))  ⑤
print()
print('{0.month}-{0.day}-{0.year}'.format(dday))  ⑥
```

① select from tuple

② named parameter + select from tuple

③ Select from list

④ select from dict

⑤ named parameter + select from dict

⑥ select attributes from date

fmt_attrib.py

```
apple mango
apple mango

5 27

Michael John
Michael John

6-6-1944
```

Formatting Dates

- Special formats for dates
- Pull appropriate values from date/time objects

To format dates, use special date formats. These are placed, like all formatting codes, after a colon. For instance, {0:%B %d, %Y} will format a parameter (which must be a datetime.datetime or datetime.date) as "Month DD, YYYY".

Example

fmt_dates.py

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

from datetime import datetime

event = datetime(2016, 1, 2, 3, 4, 5)

print(event)  ①
print()

print("Date is {0:%m}/{0:%d}/{0:%y}".format(event))  ②
print("Date is {:%m/%d/%y}".format(event))  ③
print("Date is {:%A, %B %d, %Y}".format(event))  ④
```

① Default string version of date

② Use three placeholders for month, day, year

③ Format month, day, year with a single placeholder

④ Another single placeholder format

fmt_dates.py

```
2016-01-02 03:04:05

Date is 01/02/16
Date is 01/02/16
Date is Saturday, January 02, 2016
```

Table 24. Date Formats

Directive	Meaning	See note
%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 of the year (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 of the year (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
%Z	Time zone name (empty string if the object is naive).	
%%	A literal '%' character.	

1. When used with the strptime() method, the %f directive accepts from one to six digits and zero pads on the right. %f is an extension to the set of format characters in the C standard (but

implemented separately in datetime objects, and therefore always available).

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; according to the Posix standard this accounts for leap seconds and the (very rare) double leap seconds. The time module may produce and does accept leap seconds since it is based on the Posix standard, but the datetime module does not accept leap seconds instrptime() input nor will it produce them in strftime() output.
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'.

Run-time formatting

- Use parameters to specify alignment, precision, width, and type
- Use {} placeholders for runtime values for the above

To specify formatting values at runtime, use a {} placeholder for the value, and insert the desired value in the parameter list. These placeholders are numbered along with the normal placeholders.

Example

fmt_runtime.py

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

FIRST_NAME = 'Fred'
LAST_NAME = 'Flintstone'
AGE = 35

print("{0} {1}".format(FIRST_NAME, LAST_NAME))

WIDTH = 12
print("{0:{width}s} {1:{width}s}".format(  ①
    FIRST_NAME,
    LAST_NAME,
    width=WIDTH,
))

PAD = '-'
WIDTH = 20
ALIGNMENTS = ('<', '>', '^')

for alignment in ALIGNMENTS:
    print("{0:{pad}{align}{width}s} {1:{pad}{align}{width}s}".format(  ②
        FIRST_NAME,
        LAST_NAME,
        width=WIDTH,
        pad=PAD,
        align=alignment,
    ))
```

① value of WIDTH used in format spec

② values of PAD, WIDTH, ALIGNMENTS used in format spec

fmt_runtime.py

```
Fred Flintstone
Fred         Flintstone
Fred---------------- Flintstone----------
----------------Fred ----------Flintstone
--------Fred-------- -----Flintstone-----
```

Miscellaneous tips and tricks

- Adding commas to large numbers {n:,}
- Auto-converting parameters to strings (!s)
- Non-decimal prefixes

- Adding commas to large numbers {n:,}

You can add a comma to the format to add commas to numbers greater than 999.

Using a format type of !s will call str() on the parameter and force it to be a string.

Using a # (pound sign) will cause binary, octal, or hex output to be preceded by '0b', '0o', or '0x'. This is only valid with type codes b, o, and x.

Example

fmt_misc.py

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

'''Demonstrate misc formatting'''

big_number = 2303902390239

print("Big number: {:,d}".format(big_number))  ①
print()

value = 27

print("Binary: {:#010b}".format(value))  ②
print("Octal:  {:#010o}".format(value))  ③
print("Hex:    {:#010x}".format(value))  ④
print()
```

① Add commas for readability

② Binary format with leading 0b

③ Octal format with leading 0o

④ Hexadecimal format with leading 0x

fmt_misc.py

```
Big number: 2,303,902,390,239

Binary: 0b00011011
Octal:  0o00000033
Hex:    0x0000001b
```

Index

@

@name.setter, 318
@property, 318
@staticmethod, 324

A

accept(), 282
AF_INET, 281
API, 180
application protocol, 287
arange(), 84
argparse, 263
array.flat, 111
asynchronous communication, 334
asyncio, 360
autocommit, 204

B

Berkeley Unix, 280
binary mode, 214

C

CapWords, 311
Cassandra, 206
class
    constructor, 313
    definition, 311
    instance methods, 314
    private method, 325
    static method, 324
class data, 310, 322
class method, 322
class object, 322
class statement, 311
classes, 310
client, 280
command line scripts, 259
commit, 204
connect(), 181
constructor, 310, 313
context manager, 181
creating Unix-style filters, 260
cursor, 183
cursor.description, 201
cx_oracle, 180
c_ object, 128

D

data types, 117
database object, 183
database programming, 180
database server, 181
DataFrame, 135, 139, 144
DB API, 180
derived class, 326
dictionary cursor, 197
    emulating, 202
Django, 205
Django ORM, 205
dtype, 117

E

email
    attachments, 232
    sending, 229
email.mime, 232
empty(), 80
exceptions, 295
    else, 300
    finally, 302
    generic, 298
    ignoring, 299
    list, 306
    multiple, 297
execute(, 184, 190
executemany(, 190
executing SQL statements, 184

F

fetch methods, 185
file handle, 280
file(), 19
Firebird (and Interbase, 180
forking server, 288

format string, 26
FTP, 287
full(), 80

G

GET, 221
GIL, 336
glob, 244
grabbing a web page, 214

H

HTTP, 287
HTTP verbs, 221

I

IBM DB2, 180
In-place operators, 87
index object, 144
Informix, 180
informixdb, 180
ingmod, 180
Ingres, 180
inheritance, 326
initializer, 310
instance data, 310
instance methods, 314
IP address, 285
iPython
  %timeit, 73
  benchmarking, 73
  getting help, 70
  magic commands, 72
  notebook, 68
  parallel computing, 68
  quick reference, 70
  tab completion, 71
Iterating, 111

J

Jupyter Project, 68

K

KInterbasDB, 180

L

linspace(), 84
logging
  alternate destinations, 274
  exceptions, 272
  formatted, 270
  simple, 268
lsmagic, 72

M

matplotlib.pyplot, 175
MatPlotLibExamples.ipynb, 177
metadata, 201
Microsoft SQL Server, 180
MongoDB, 206
multiprocessing, 334, 354
  Manager, 350
multiprocessing module, 350
multiprocessing.dummy, 354
multiprocessing.dummy.Pool, 354
multiprocessing.Pool, 354
multiprogramming, 334
  alternatives to, 360
MySQL, 180

N

ndarray
  iterating, 111
non-query statement, 190
non-relational, 206
NoSQL, 206
NumPy
  getting help, 110
numpy.info(), 110
numpy.lookfor(), 110

O

object-oriented language, 310
object-oriented programming, 310
Object-relational mapper, 205
objects, 310
ODBC, 180
ones(), 80
Oracle, 180

ORM, 205

P

pandas, 134
  broadcasting, 152
  DataFrame
    initialize, 139
  Dataframe, 135
  dataframe alignment, 159
  drop(), 157
  I/O functions, 167
  index object, 144
  indexing, 148
  reading data, 166
  read_csv(), 166
  selecting, 151
  Series, 135
  time series, 160
Panel, 135
parameterized SQL statements, 190
paramiko, 236
parsing the command line, 264
Perl, 40
permissions, 254
  checking, 254
placeholder, 190
plt.plot(), 175
plt.show(, 175
polymorphic, 310
Popen, 247
port number, 285
POST, 221
PostgreSQL, 180
preconfigured log handlers, 274
private data, 313
private methods, 325
properties, 318
psycopg2, 180
PUT, 221
PyDB2, 180
pymssql, 180
pymysql, 180
pyodbc, 180

R

raw data, 24
re.compile(), 46
re.findall(), 43
re.finditer(), 43
re.search(), 43
read(), 19
readline(), 19
readlines(), 19
read_csv(), 166
read_table, 166
Redis, 206
Regular Expression Metacharacters
  table, 42
regular expressions, 40
  about, 40
  atoms, 41
  branches, 41
  compilation flags, 49-50
  finding matches, 43
  grouping, 53
  re objects, 46
  replacing text, 58
  replacing text with callback, 60
  special groups, 56
  splitting text, 63
  syntax overview, 41
remote access, 236
requests, 221
  methods
    keyword parameters, 228
rollback, 204
r_ object, 128

S

SAP DB, 180
sapdbapi, 180
SciPy, 78
self, 314
sendmail(, 229
Series, 135-136
server, 280
SFTP, 239
shape, 90

shlex.split(), 246
shutil, 256
SMTP, 287
smtplib, 229
SNMP, 287
socket, 280
SOCK_DGRAM, 281
SOCK_STREAM, 281
SQL code, 183
SQL data integrity, 204
SQL injection, 188
SQL queries, 184
SQL statements, 184
SQLAlchemy, 205
SQLite, 180
sqlite3, 180
SSH, 287
ssh protocol, 236
standard exception hierarchy, 306
static method, 324
string formatting
  alignment, 376
  data types, 370
  dates, 386
  field widths, 373
  fill characters, 379
  misc, 393
  parameter attributes, 384
  run-time, 390
  selectors, 368
  signed numbers, 381
Struct, 26
struct module, 26
Struct.pack(), 26
Struct.unpack(), 26
StudlyCaps, 311
subprocess, 247-248
  capturing stdout/stderr, 251
  check_call(), 248
  check_output(), 248
  run(), 248
Sybase, 180
syntax errors, 294

T

TCP, 281
TCP/IP, 280
thread, 335
thread class
  creating, 340
threading, 334
threading module, 337
threading.Thread, 337
threads
  debugging, 349
  locks, 342
  queue, 345
  simple, 338
  variable sharing, 342
time_series(), 160
transactions, 204
transport protocol, 282, 285
Twisted, 360

U

UDP, 281
ufuncs, 104
urllib.parse.urlencode(, 218
urllib.request, 214, 218
urllib.request.Request, 218
urlopen(), 214
using try/except, 296

V

vectorize(), 105
vectorized, 104

W

web services
  consuming, 218
write(), 19
writelines(), 19

Z

zeros(), 80

