

Custom Python Plus for Discover

John Strickler

Version 1.0, February 2020

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: Pythonic Programming	9
The Zen of Python	10
Tuples	11
Iterable unpacking	12
Unpacking function arguments	14
The sorted() function	18
Custom sort keys	19
Lambda functions	24
List comprehensions	26
Dictionary comprehensions	28
Set comprehensions	30
Iterables	31
Generator Expressions	33
Generator functions	35
String formatting	38
f-strings	40
Chapter 2: Functions, Modules and Packages	45
Functions	46
Function parameters	49
Default parameters	53
Python Function parameter behavior (from PEP 3102)	55
Name resolution (AKA Scope)	56
The global statement	59

Modules	60
Using import	61
How <i>import *</i> can be dangerous	65
Module search path	67
Executing modules as scripts	68
Packages	70
Configuring import with <code>__init__.py</code>	72
Documenting modules and packages	75
Python style	76
Chapter 3: Intermediate Classes	79
What is a class?	80
Defining Classes	81
Object Instances	82
Instance attributes	83
Instance Methods	84
Constructors	86
Getters and setters	87
Properties	88
Class Data	91
Class Methods	93
Inheritance	95
Using <code>super()</code>	96
Multiple Inheritance	101
Abstract base classes	104
Special Methods	107
Static Methods	113
Chapter 4: Context Managers	115
The with statement	116
What is a context manager?	117
Creating a context-aware class	118
About contextlib	123
<code>contextlib.closing</code>	124
<code>contextlib.contextmanager</code>	125
<code>contextlib.suppress</code>	127
Chapter 5: Generators and other Iterables	129

Iterables	130
Unpacking function parameters	131
Iterable unpacking	134
Extended iterable unpacking	137
What exactly is an iterable?	140
List comprehensions	141
Generators	144
Generator Expressions	145
Generator functions	148
Coroutines	151
Generator classes	154
Chapter 6: Container Classes	159
Objectives	159
Container classes	160
Builtin containers	161
Containers in the standard library	163
The array module	167
Emulating builtin types	170
Creating list-like containers	176
Creating dict-like containers	179
Free-form containers	181
Chapter 7: Advanced Data Handling	183
Objectives	183
Deep vs shallow copying	184
Default dictionary values	187
Counting with Counter	190
Named Tuples	191
Printing data structures	194
Zipped archives	197
Tar Archives	199
Serializing Data	202
Chapter 8: Metaprogramming	207
Objectives	207
Metaprogramming	208
globals() and locals()	209

The inspect module	212
Working with attributes	215
Adding instance methods	218
Decorators.....	221
Applying decorators.....	222
Trivial Decorator.....	225
Decorator functions.....	226
Decorator Classes	229
Decorator parameters	233
Creating classes at runtime	236
Monkey Patching.....	240
Do you need a Metaclass?.....	243
About metaclasses.....	244
Mechanics of a metaclass	246
Singleton with a metaclass	250
Chapter 9: Developer Tools.....	255
Objectives	255
Program development.....	256
Comments.....	257
pylint	258
Customizing pylint	259
Using pyreverse.....	260
The Python debugger.....	262
Starting debug mode	263
Stepping through a program	264
Setting breakpoints.....	265
Profiling.....	266
Benchmarking	268
Chapter 10: Multiprogramming.....	273
Objectives	273
Multiprogramming	274
What Are Threads?.....	275
The Python Thread Manager	276
The threading Module	277
Threads for the impatient	278

Creating a thread class	280
Variable sharing	283
Using queues	286
Debugging threaded Programs	289
The multiprocessing module	291
Using pools	295
Alternatives to multiprogramming	301
Chapter 11: Coroutines and <code>asyncio</code>	303
Asynchronous programming with <code>asyncio</code>	304
Key asynchronous words	305
Defining coroutines	306
The Event Loop	307
Futures	310
Tasks	312
Callbacks	316
How to run coroutines	317
Chapter 12: Effective Scripts	321
Using <code>glob</code>	322
Using <code>shlex.split()</code>	324
The subprocess module	325
subprocess convenience functions	326
Capturing stdout and stderr	329
Permissions	332
Using <code>shutil</code>	334
Creating a useful command line script	337
Creating filters	338
Parsing the command line	341
Simple Logging	346
Formatting log entries	348
Logging exception information	351
Logging to other destinations	353
Chapter 13: Unit Tests with <code>pytest</code>	357
Objectives	357
What is a unit test?	358
The <code>pytest</code> module	359

Creating tests	360
Running tests (basics)	361
Special assertions	362
Fixtures	364
User-defined fixtures	365
Builtin fixtures	367
Configuring fixtures	371
Parametrizing tests	374
Marking tests	377
Running tests (advanced)	379
Skipping and failing	381
Mocking data	384
pymock objects	385
Pytest and Unittest	392
Chapter 14: Serializing Data	395
Objectives	395
About XML	396
Normal Approaches to XML	397
Which module to use?	398
Getting Started With ElementTree	399
How ElementTree Works	400
Elements	401
Creating a New XML Document	404
Parsing An XML Document	407
Navigating the XML Document	408
Using XPath	412
About JSON	416
Reading JSON	417
Writing JSON	420
Customizing JSON	423
Reading and writing YAML	427
Reading CSV data	432
Nonstandard CSV	434
Using csv.DictReader	436
Writing CSV Data	438

Pickle	440
Chapter 15: Network Programming	445
Objectives	445
Grabbing a web page	446
Consuming Web services	450
HTTP the easy way	453
sending e-mail	460
Email attachments	463
Remote Access	467
Copying files with Paramiko	470
Chapter 16: Accessing git	475
What is source code control?	476
What is git?	477
git workflow	478
Using the command line	479
Using the <i>gitpython</i> module	480
Getting repo info	482
Creating a repo	486
Pushing to remote repos	488
Cloning repos	490
Working with branches	491
Tagging	493
Chapter 17: Using Docker from Python	497
What is Docker?	498
Creating and running a client	499
Listing containers	502
Working with images	505
Building an image	508
Working with Flask	513
Chapter 18: Diving Right In	515
Flask "Hello, world"	516
Creating a Flask app	518
Defining view functions	519
Configuring Routes	520
Deploying the app	521

Chapter 19: Creating Models	523
Creating models for a Flask Application	524
Connecting to a database	525
Creating a get_db() function	526
Using Flask-SQLAlchemy	527
Creating models	528
Initializing the database	529
Adding and accessing data	531
Chapter 20: Implementing REST with Flask	535
Objectives	535
What is REST?	536
Base REST constraints	537
Designing a REST API	538
Setting up RESTful routes	540
Choosing the return type	541
Using jsonify	543
Handling invalid requests	544
Receiving data from POST	545
Handling a DELETE request	546
Appendix A: Python Bibliography	549
Appendix B: Object Oriented Design	555
Object Oriented Design Principles	555
SOLID Design Principles	558
The Single Responsibility Principle (SRP)	559
Composite vs. Aggregate Types	562
The Open Closed Principle (OCP)	563
The Liskov Substitution Principle (LSP)	572
The Interface Segregation Principle (ISP)	577
The Dependency Inversion Principle (DIP)	582
Index	585

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** Pythonic Programming
- Chapter 2** Functions modules, and packages
- Chapter 3** Intermediate classes
- Chapter 4** Context managers

Day 2

- Chapter 5** Generators and other iterables
- Chapter 6** Container classes
- Chapter 7** Advanced data handling
- Chapter 8** Metaprogramming
- Chapter 9** Developer tools

Day 3

- Chapter 10** Multiprogramming
- Chapter 11** Coroutines and asyncio
- Chapter 12** Effective scripts
- Chapter 13** Unit testing with pytest

Day 4

- Chapter 14** Serializing data
- Chapter 15** Network programming
- Chapter 16** accessing git
- Chapter 17** using docker

Day 5

- Chapter 18** Flask: diving right in
- Chapter 19** Flask: creating models
- Chapter 20** Flask: rest api

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 **py3discplus**.

What's in the files?

py3discplus contains data and other files needed for the exercises

py3discplus/EXAMPLES contains the examples from the course manuals.

py3discplus/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 **py3discplus.zip**. Extract all files to your desktop. This will create the folder **py3discplus**.

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

Copy or download **py3discplus.tgz** to your home directory. In your home directory, type

```
tar xzvf py3discplus.tgz
```

This will create the **py3discplus** 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 py3discplus/ANSWERS

Appendices

- Appendix A: Python Bibliography
- Appendix B: Object-oriented Design

Chapter 1: Pythonic Programming

Objectives

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

The Zen of Python

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

— Tim Peters, from PEP 20

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

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

Tuples

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

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

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

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

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

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

TIP

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

Example

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

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

Iterable unpacking

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

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

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

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

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

Example

unpacking_people.py

```
#!/usr/bin/env python
#
people = [ ①
    ('Melinda', 'Gates', 'Gates Foundation'),
    ('Steve', 'Jobs', 'Apple'),
    ('Larry', 'Wall', 'Perl'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Bill', 'Gates', 'Microsoft'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linus', 'Torvalds', 'Linux'),
]
for first_name, last_name, org in people: ②
    print("{} {}{}".format(first_name, last_name))
```

① A list of 3-element tuples

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

unpacking_people.py

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

Unpacking function arguments

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

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

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

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

Example

unpacking_function_args.py

```
#!/usr/bin/env python
#
people = [ ①
    ('Joe', 'Schmoe', 'Burbank', 'CA'),
    ('Mary', 'Rattburger', 'Madison', 'WI'),
    ('Jose', 'Ramirez', 'Ames', 'IA'),
]
def person_record(first_name, last_name, city, state): ②
    print("{} {} lives in {}, {}".format(first_name, last_name, city, state))

for person in people: ③
    person_record(*person) ④
```

① list of 4-element tuples

② function that takes 4 parameters

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

④ *person unpacks the tuple into four individual parameters

unpacking_function_args.py

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

Example

shoe_sizes.py

```
#!/usr/bin/env python
#
BARLEYCORN = 1 / 3.0
CM_TO_INCH = 2.54
MENS_START_SIZE = 12
WOMENS_START_SIZE = 10.5

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

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

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

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

    print()

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

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

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

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

shoe_sizes.py

MEN'S		
Size	Inches	CM
6.0	10.00	25.40
6.5	10.17	25.82
7.0	10.33	26.25
7.5	10.50	26.67
8.0	10.67	27.09
8.5	10.83	27.52

...

The sorted() function

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

```
key=  
reverse=
```

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

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

Example

basic_sorting.py

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

"""Basic sorting example"""

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

sorted_fruit = sorted(fruit) ①

print(sorted_fruit)
```

① sorted() returns a list

basic_sorting.py

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

Custom sort keys

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

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

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

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

TIP

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

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

Example

custom_sort_keys.py

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

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

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

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

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

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

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

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

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

- ① Parameter is *one* element of iterable to be sorted
- ② Return value to sort on
- ③ Specify function with named parameter **key**
- ④ Key functions can return tuple of values to compare, in order
- ⑤ Numbers sort numerically by default
- ⑥ Sort numbers as strings

custom_sort_keys.py

Ignoring case:

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

By length, then name:

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

Numbers sorted numerically:

5 32 80 255 400 800 1000 5000

Numbers sorted as strings:

1000 255 32 400 5 5000 80 800

Example

sort_holmes.py

```
#!/usr/bin/env python
"""Sort titles, ignoring leading articles"""
import re

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

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

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

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

① compile regex to match leading articles

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

③ strip off article and convert title to lower case

④ sort using custom function

sort_holmes.py

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

Lambda functions

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

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

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

The basic syntax for creating a lambda function is

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

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

A lambda function could also be defined in the normal manner

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

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

Example

lambda_examples.py

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

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

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

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

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

lambda_examples.py

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

List comprehensions

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

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

A loop like this:

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

can be rewritten as

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

A conditional if may be added to filter values:

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

Example

listcomp.py

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

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

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

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

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

doubles = [v * 2 for v in values] ③

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

① Copy each fruit to upper case

② Select each fruit if it starts with 'a'

③ Copy each number, doubling it

listcomp.py

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

Dictionary comprehensions

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

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

Example

`dict_comprehension.py`

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

animals = ['OWL', 'Badger', 'bushbaby', 'Tiger', 'Wombat', 'GORILLA', 'AARDVARK']

d = {a.lower(): len(a) for a in animals} ①

print(d, '\n')

words = ['unicorn', 'stigmata', 'barley', 'bookkeeper']

d = {w:{c:w.count(c) for c in sorted(w)} for w in words} ②

for word, word_signature in d.items():
    print(word, word_signature)
```

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

② Use a nested dictionary comprehension to create a dictionary mapping words to dictionaries which map letters to their counts (could be useful for anagrams)

dict_comprehension.py

```
{'owl': 3, 'badger': 6, 'bushbaby': 8, 'tiger': 5, 'wombat': 6, 'gorilla': 7,  
'aardvark': 8}  
  
unicorn {'c': 1, 'i': 1, 'n': 2, 'o': 1, 'r': 1, 'u': 1}  
stigmata {'a': 2, 'g': 1, 'i': 1, 'm': 1, 's': 1, 't': 2}  
barley {'a': 1, 'b': 1, 'e': 1, 'l': 1, 'r': 1, 'y': 1}  
bookkeeper {'b': 1, 'e': 3, 'k': 2, 'o': 2, 'p': 1, 'r': 1}
```

Set comprehensions

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

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

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

Example

`set_comprehension.py`

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

import re

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

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

`set_comprehension.py`

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

Iterables

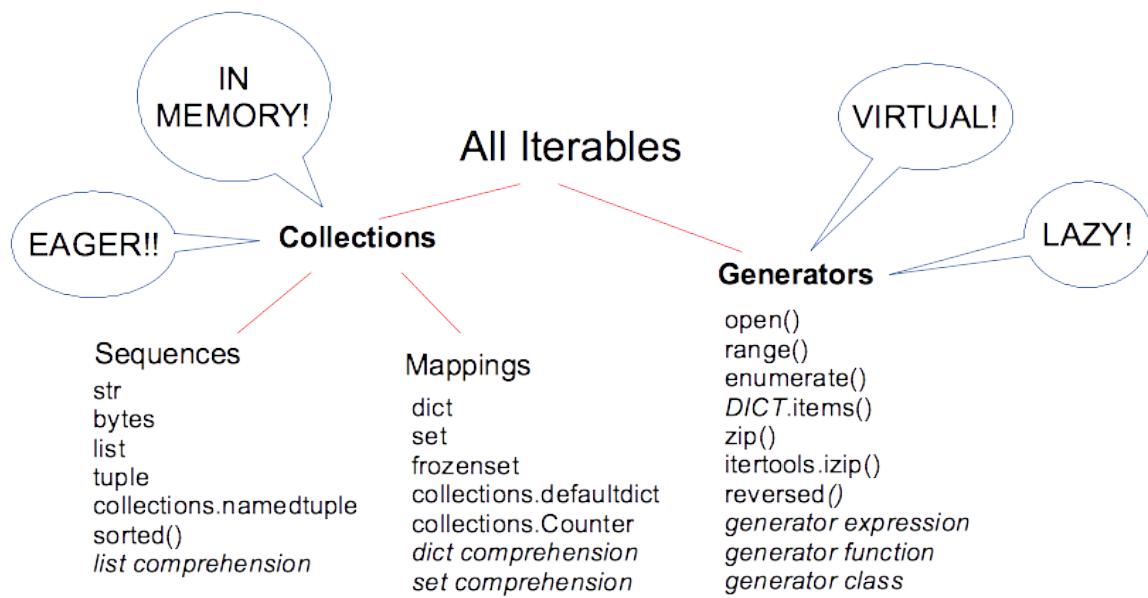
- Expression that can be looped over
- Can be collections *e.g.* list, tuple, str, bytes
- Can be generators *e.g.* range(), file objects, enumerate(), zip(), reversed()

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

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

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

Iterables



Generator Expressions

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

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

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

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

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

Example

gen_ex.py

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

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

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

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

① using list comprehension, entire list is stored in memory

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

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

gen_ex.py

```
285 285
```

```
30
```

Generator functions

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

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

Example

sieve_generator.py

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

def next_prime(limit):
    flags = set() ①

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j) ②
        yield i ③

np = next_prime(200) ④
for prime in np: ⑤
    print(prime, end=' ')
```

① initialize empty set (to be used for "is-prime" flags)

② add non-prime elements to set

③ execution stops here until next value is requested by for-in loop

④ next_prime() returns a generator object

⑤ iterate over **yielded** primes

sieve_generator.py

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

Example

line_trimmer.py

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

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

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

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

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

line_trimmer.py

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

String formatting

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

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

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

Formatting information can be added, preceded by a colon.

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

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

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

Example

`stringformat_ex.py`

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

from datetime import date

color = 'blue'
animal = 'iguana'

print('{} {}'.format(color, animal)) ①

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

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

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

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

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

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

③ {} placeholders can be manually numbered to reuse parameters

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

`stringformat_ex.py`

```
blue iguana
98.7
12345 3039 00030071 001100000111001
A 38
B 127
C 9
```

f-strings

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

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

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

In normal 3.x formatting:

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

f-strings let you do this:

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

Example

f_strings.py

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

import sys

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

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

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

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

    print(f"${result:,d}") ⑤

    city = 'Orlando'
    temp = 85

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

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

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

② > means right justify

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

④ d is decimal, o is octal, x is hex

⑤ , means add commas to numeric value

⑥ parameters can be selected by name instead of position

f_strings.py

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

Chapter 1 Exercises

Exercise 1-1 (pres_upper.py)

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

Exercise 1-2 (pres_by_dob.py)

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

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

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

Exercise 1-3 (pres_gen.py)

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

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

Chapter 2: Functions, Modules and Packages

Objectives

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

Functions

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

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

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

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

TIP

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

Example

function_basics.py

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

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

say_hello() ③

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

h = get_hello() ⑤
print(h)
print()

def sqrt(num): ⑥
    return num ** .5

m = sqrt(1234) ⑦
n = sqrt(2)

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

- ① Function takes no parameters
- ② If no **return** statement, return None
- ③ Call function (arguments, if any, in ())
- ④ Function returns value
- ⑤ Store return value in h
- ⑥ Function takes exactly one argument
- ⑦ Call function with one argument

function_basics.py

```
Hello, world
```

```
Hello, world
```

```
m is 35.128 n is 1.414
```

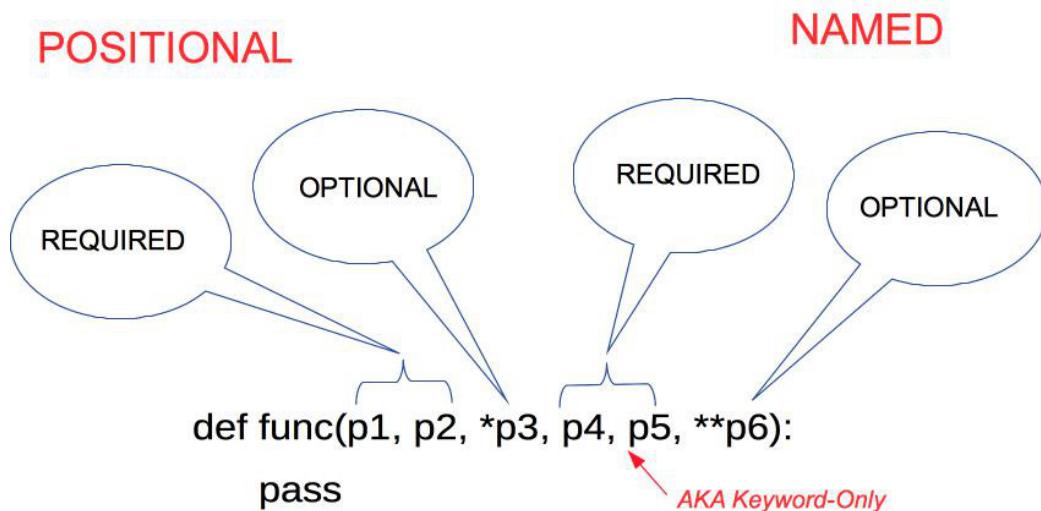
Function parameters

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

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

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

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



Example

function_parameters.py

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

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

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

def fun_two(n): ②
    return n ** 2

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

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

fun_three()
fun_three(10)
print()

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

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

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

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

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

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

- ① no parameters
- ② one required parameter
- ③ one required parameter with default value
- ④ one fixed, plus optional parameters
- ⑤ keyword-only parameters
- ⑥ keyword (named) parameters

function_parameters.py

```
fun_one(): Hello, world  
  
fun_two(5) is 25  
  
spam spam spam  
spam spam spam spam spam spam spam spam  
  
fun_four():  
n is apple  
opt is ()  
-----  
fun_four():  
n is apple  
opt is ('blueberry', 'peach', 'cherry')  
-----  
fun_five():  
spam is: 1  
eggs is: 2  
  
fun_five():  
spam is: 2  
eggs is: 2  
  
fun_five():  
spam is: 1  
eggs is: 0  
  
fun_five():  
spam is: 0  
eggs is: 2  
  
fun_five():  
spam is: 0  
eggs is: 0  
  
fun_six():  
name ==> Lancelot  
quest ==> Grail  
color ==> red
```

Default parameters

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

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

Example

default_parameters.py

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

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

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

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

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

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

② parameter supplied; default not used

③ parameter not supplied; default is used

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

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

default_parameters.py

```
Hello, Mom
```

```
Hello, world
```

```
Processing eggs as txt
```

```
Processing toast as csv
```

Python Function parameter behavior (from PEP 3102)

For each formal parameter, there is a slot which will be used to contain the value of the argument assigned to that parameter.

- Slots which have had values assigned to them are marked as 'filled'. Slots which have no value assigned to them yet are considered 'empty'.
- Initially, all slots are marked as empty.
- Positional arguments are assigned first, followed by keyword arguments.
- For each positional argument:
 - Attempt to bind the argument to the first unfilled parameter slot. If the slot is not a vararg slot, then mark the slot as 'filled'.
 - If the next unfilled slot is a vararg slot, and it does not have a name, then it is an error.
 - Otherwise, if the next unfilled slot is a vararg slot then all remaining non-keyword arguments are placed into the vararg slot.
- For each keyword argument:
 - If there is a parameter with the same name as the keyword, then the argument value is assigned to that parameter slot. However, if the parameter slot is already filled, then that is an error.
 - Otherwise, if there is a 'keyword dictionary' argument, the argument is added to the dictionary using the keyword name as the dictionary key, unless there is already an entry with that key, in which case it is an error.
 - Otherwise, if there is no keyword dictionary, and no matching named parameter, then it is an error.
- Finally:
 - If the vararg slot is not yet filled, assign an empty tuple as its value.
 - For each remaining empty slot: if there is a default value for that slot, then fill the slot with the default value. If there is no default value, then it is an error.
- In accordance with the current Python implementation, any errors encountered will be signaled by raising `TypeError`.

Name resolution (AKA Scope)

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

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

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

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

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

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

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

Example

scope_examples.py

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

x = 42 ①

def function_a():
    y = 5 ②

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

    return function_b

f = function_a() ⑧
f() ⑨
```

① global variable

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

③ local variable

④ local scope

⑤ nested (nonlocal) scope

⑥ global scope

⑦ builtin scope

⑧ calling function_a, which returns function_b

⑨ calling function_b

scope_examples.py

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

The global statement

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

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

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



Modules

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

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

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

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

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

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

Modules and packages should be documented with *docstrings*.

Using import

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

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

Variation 1

`import module`

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

Variation 2

`from module import function, ...`

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

Variation 3

`from module import *`

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

NOTE

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

Example

samplelib.py

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

# sample Python module

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

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

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

use_samplelib1.py

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

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

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

② call function spam() in module samplelib

use_samplelib1.py

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

use_samplelib2.py

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

spam() ②
ham()
```

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

② module name not needed to call function spam()

use_samplelib2.py

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

use_samplelib3.py

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

spam() ②
ham()
```

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

② module name not needed to call function spam()

use_samplelib3.py

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

use_samplelib4.py

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

pig()
hog()
```

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

use_samplelib4.py

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

How `import *` can be dangerous

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

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

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

Example

`electrical.py`

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

default_amps = 10
default_voltage = 110
default_current = 'AC'

def amps():
    return default_amps

def voltage():
    return default_voltage

def current():
    return default_current
```

`navigation.py`

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

current_types = 'slow medium fast'.split()

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

why_import_star_is_bad.py

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

from electrical import * ①
from navigation import * ②

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

- ① import current *explicitly* from electrical
- ② import current *implicitly* from navigation
- ③ calls navigation.current(), not electrical.current()

why_import_star_is_bad.py

```
slow
110
10
```

how_to_avoid_import_star.py

```
#!/usr/bin/env python
import electrical as e ①
import navigation as n ②

print(e.current()) ③
print(n.current()) ④
```

how_to_avoid_import_star.py

```
AC
slow
```

Module search path

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

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

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

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

Windows

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

Linux/OS X

```
export PYTHONPATH="/home/bob/python"
```

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

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

Executing modules as scripts

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

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

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

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

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

Many modules do not need any initialization code.

Example

using_main.py

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

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

# constants (AKA global variables)

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

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

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

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

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

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

Packages

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

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

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

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

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

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

Packages can be nested.

Example

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

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

Configuring import with `__init__.py`

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

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

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

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

```
from PKG import *
```

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

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

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

Import statement	What it does
<code>from my_package import *</code>	Only imports module_a and module_b into main namespace.
<code>from my_package import module_c</code>	Imports module_c into the main namespace.

Documenting modules and packages

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

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

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

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

See [PEP 257](#) for a detailed guide on docstring conventions.

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

Python style

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

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

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

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

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

Chapter 2 Exercises

Exercise 2-1 (potus.py, potus_main.py)

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

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

Write a script to use the module.

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

Add the following functions to the module

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

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

Chapter 3: Intermediate Classes

Objectives

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

What is a class?

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

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

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

TIP

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

Defining Classes

- Syntax

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

- Specify base classes
- Use StudlyCaps for name

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

The simplest form of class definition looks like this:

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

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

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

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

Object Instances

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

```
obj = className(args...)
```

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

Instance attributes

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

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

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

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

Example

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

    def _beverage(self):    # private!
        pass

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

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

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

Instance Methods

- Called from objects
- Object is implicit parameter

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

Example

rabbit.py

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

class Rabbit:

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

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

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

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

① constructor, passed **self**

② instance method, passed **self**

③ pass parameters to constructor

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

rabbit.py

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

Constructors

- Named `__init__.py`
- Implicitly called when object is created
- `self` is object itself

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

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

Example

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

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

Getters and setters

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

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

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

NOTE

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

Example

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

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

    def get_name(self):
        return self._name

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

Properties

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

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

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

You can create getter, setter, and deleter properties.

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

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

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

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

Example

knight.py

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

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

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

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

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

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

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

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

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

    k.color = 'red' ⑤

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

- ① getter property decorator
- ② property implemented by name() method
- ③ setter property decorator
- ④ use property
- ⑤ set property

knight.py

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

Class Data

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

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

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

Example

class_data.py

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

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

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

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

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

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

① class data

② look up class data via instance

③ instance method uses class data

class_data.py

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

Class Methods

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

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

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

Example

`class_methods_and_data.py`

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

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

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

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

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

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

① 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
```

Inheritance

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

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

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

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

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

Using super()

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

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

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

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

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

```
class Foo(Bar):  
    def __init__(self):  
        super().__init__() # same as Bar.__init__(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", "Meowwww")
    felix = Animal("cat", "Felix", "Meowwww")

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

```
leo.make_sound()

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

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

- ① class data
- ② update class data from instance
- ③ zoo_size gets class object when called from instance or class

insect.py

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

from animal import Animal

class Insect(Animal):
    """
        An animal with 2 sets of wings and 3 pairs of legs
    """

    def __init__(self, species, name, sound, can_fly=True): ①
        super().__init__(species, name, sound) ②
        self._can_fly = can_fly

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

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

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

- ① constructor (AKA initializer)
- ② call base class constructor
- ③ "getter" property
- ④ defaults to can_fly being True
- ⑤ .name and .species inherited from base class (Animal)
- ⑥ .make_sound inherited from Animal

insect.py

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

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

Multiple Inheritance

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

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

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

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

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

TIP

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

Example

`multiple_inheritance.py`

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

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

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

class CanFly(): ③
    def fly(self):
        print("I'm flying")

class Dog(CanBark, AnimalBase): ④
    pass

class Sparrow(CanFly, AnimalBase): ⑤
    pass

d = Dog('Dennis')
d.get_id() ⑥
d.bark()
print()

s = Sparrow('Steve')
s.get_id()
s.fly() ⑦
print()

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

- ① create primary base class
- ② create additional (mixin) base class
- ③ inherit from primary base class plus mixin
- ④ all animals have id()
- ⑤ dogs can bark() (from mixin)
- ⑥ sparrows can fly() (from mixin)

multiple_inheritance.py

```
Dennis
woof-woof
```

```
Steve
I'm flying
```

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

Abstract base classes

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

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

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

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

NOTE [abc also provides decorators for abstract properties and abstract class methods.](#)

Example

abstract_base_classes.py

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

class Animal(metaclass=ABCMeta): ①

    @abstractmethod ②
    def speak(self):
        pass

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

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

class Duck(Animal): ③
    pass ⑤

d = Dog()
d.speak()

c = Cat()
c.speak()

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

- ① metaclasses control how classes are created; ABCMeta adds restrictions to classes that inherit from Animal
- ② when decorated with `@abstractmethod`, `speak()` becomes an abstract method
- ③ Inherit from abstract base class `Animal`
- ④ `speak()` **must** be implemented
- ⑤ Duck does not implement `speak()`
- ⑥ Duck throws a `TypeError` if instantiated

abstract_base_classes.py

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

Special Methods

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

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

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

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

TIP

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

Table 1. Special Methods and Variables

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

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

Method or Variables	Description
<code>_radd_(self, other)</code> <code>_rsub_(self, other)</code> <code>_rmul_(self, other)</code> <code>_rdiv_(self, other)</code> <code>_rtruediv_(self, other)</code> <code>_rfloordiv_(self, other)</code> <code>_rmod_(self, other)</code> <code>_rdivmod_(self, other)</code> <code>_rpow_(self, other)</code> <code>_rlshift_(self, other)</code> <code>_rrshift_(self, other)</code> <code>_rand_(self, other)</code> <code>_rxor_(self, other)</code> <code>_ror_(self, other)</code>	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)
<code>_iadd_(self, other)</code> <code>_isub_(self, other)</code> <code>_imul_(self, other)</code> <code>_idiv_(self, other)</code> <code>_itruediv_(self, other)</code> <code>_ifloordiv_(self, other)</code> <code>_imod_(self, other)</code> <code>_ipow_(self, other[, modulo])</code> <code>_ilshift_(self, other)</code> <code>_irshift_(self, other)</code> <code>_iand_(self, other)</code> <code>_ixor_(self, other)</code> <code>_ior_(self, other)</code>	Implement augmented (+=, -=, etc.) arithmetic operators
<code>_neg_(self)</code> <code>_pos_(self)</code> <code>_abs_(self)</code> <code>_invert_(self)</code>	Implement unary arithmetic operators -, +, abs(), and ~
<code>_oct_(self)</code> <code>_hex_(self)</code>	Implement oct() and hex() builtins
<code>_index_(self)</code>	Implement operator.index()
<code>_coerce_(self, other)</code>	Implement "mixed-mode" numeric arithmetic.

specialmethods.py

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

class Special():

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

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

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

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

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


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

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

```
print("w * 3", w * 3)
```

- ① all Special instances are strings
- ② define what happens when a Special instance is added to another Special object
- ③ define what happens when a Special instance is multiplied by a value
- ④ define what happens when str() called on a Special instance
- ⑤ define equality between two Special values
- ⑥ parameter to Special() is converted to a string
- ⑦ add two Special instances
- ⑧ multiply a Special instance by an integer

specialmethods.py

```
s + s spamsspam
s + t spameggs
t + t eggseggs
s * 10 spamsspamspamspamspamspamspamspamspam
t * 3 eggseggsseggs
str(s)=SPAM str(t)=EGGS
id(s)=4440103288 id(t)=4440103344 id(u)=4440103512
s == s True
s == t False
s == u True
v + v 55
v + w 522
w + w 2222
v * 10 5555555555
w * 3 222222
```

Static Methods

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

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

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

NOTE [Static methods are seldom needed.](#)

Chapter 3 Exercises

Exercise 3-1 (president.py, president_main.py)

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

Provide the following properties (types indicated after `->`):

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

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

```
from president import President

p = President(1)  # George Washington
print("George was born at {0}, {1} on {2}".format(
    p.birth_place, p.birth_state, p.birth_date
))
```

Chapter 4: Context Managers

Objectives

- Understand the concept of a context manager
- Create classes that work with the **with** statement
- Use tools from the **contextlib** module

The with statement

- Provides a context manager for a block
- Not all objects are context-aware

The **with** statement provides a **context manager** for a block. The syntax is

```
with statement:  
    pass
```

or

```
with statement as object:  
    pass
```

The first form is used when you don't need access to the context manager itself. An example of this is a lock manager from the `threading` module:

```
from threading import Lock  
my_lock = Lock()  
with my_lock:  
    # do something that needs exclusive access
```

The second form is used when you *do* need access to the context manager. The most common example of this is opening files:

```
with open('DATA/mary.txt') as mary_in:  
    # use mary_in
```

Not all Python objects are context managers; thus, not all objects are suitable for the **with** statement. The object must be context-aware. If the object is not, you will get a message like "AttributeError: `_enter_`".

What is a context manager?

- Acquires and releases some resource
- Object that automatically executes some code
 - On entry to block
 - On exit from block
- Only useful in a **with** block
- Implements setup/cleanup code

A context manager is an object that works with a **with** block. Something can be done when the block is entered, and something can be done when the block is exited.

Context managers are great when you want to automatically execute some code when an object is used. Common use cases are opening files, connecting to databases, and running external programs.

When the block is entered, a resource, such as a database connection, is acquired. When the block is exited, the resource is released.

The implementation of a context manager uses the **context protocol**, which described the required methods and how they are called.

Creating a context-aware class

- Create normal class
- Add required methods `_enter_` and `_exit_`.
- Return value of `_enter_` stored in context variable

To create a context-aware class, add the require methods `_enter_` and `_exit_`.

`_enter_` is called when the object is created (via the `with` statement). It is not passed any arguments other than `self`. The return value of `_enter_` is assigned to the context variable, if present. It is typical (but not required) for `_enter_` to return `self`.

`_exit_` is called when the block is exited. It is always called with three parameters. If an exception is raised in the block, it is passed the exception type, the value of the exception, and a traceback. If no exception is raised, all three parameters are `None`.

If `_exit_` returns a true value, then the exception raised is suppressed.

Example

context_manager_class_example.py

```
#!/usr/bin/env python
#
class OpenableThing():
    def __init__(self):
        print("Creating an OpenableThing")

    def __enter__(self): ①
        print("Entering block")
        return self

    def __exit__(self, exc_type, value, traceback): ②
        print("Leaving block")
        if exc_type is not None:
            print('***** EXCEPTION *****')
            print("exception type:", exc_type) # all None unless Exception occurs
            print("value:", value)
            print("traceback:", traceback)
            print('***** * *****')
        else:
            self.close()
        return True ③

    def close(self):
        print("Closing")

    def hello(self):
        print("Hello from an OpenableThing")

with OpenableThing() as ot: ④
    print("In the context...")
    ot.hello() ⑤

print('-' * 60)

with OpenableThing() as ot: ④
    print("In the context....")
    raise Exception("Oh nooooooo") ⑥
```

- ① called at beginning of block
- ② called at end of block
- ③ suppress exception raised in block
- ④ create Blorch object; obj.enter() is called
- ⑤ after last line, obj.exit() is called
- ⑥ exception is raised (but suppressed)

context_manager_class_example.py

```
Creating an OpenableThing
Entering block
In the context...
Hello from an OpenableThing
Leaving block
Closing
-----
Creating an OpenableThing
Entering block
In the context....
Leaving block
***** EXCEPTION *****
exception type: <class 'Exception'>
value: Oh nooooooo
traceback: <traceback object at 0x10c95da88>
***** *
```

Example

smtp_context.py

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

SMTP_HOST = 'smtpcorp.com'
SMTP_PORT = 2525
SMTP_USER = 'jstrickpython'
SMTP_PASSWORD = 'python(monty)'

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

MESSAGE = '''Subject: SMTP example
Hello hello?
Testing email from Python
'''

class SMTPOpener(): ②
    def __init__(self, host, port, username, password, debug=False):
        self._smtpserver = smtplib.SMTP(host, port)
        self._smtpserver.login(username, password)
        self._smtpserver.set_debuglevel(debug)

    def __enter__(self): ③
        return self._smtpserver

    def __exit__(self, exc_type, exc_value, tb): ④
        self._smtpserver.quit() ⑤

    with SMTPOpener(SMTP_HOST, SMTP_PORT, SMTP_USER, SMTP_PASSWORD, True) as so: ⑥
        so.sendmail(
            SENDER,
            RECIPIENTS,
            MESSAGE
        )
    ⑦
```

- ① create context-aware class
- ② *init()* accepts instance parameters
- ③ called at beginning of block; returns SMTP object
- ④ called at end of block
- ⑤ quit SMTP server object
- ⑥ create SMTPOpener object with a context block
- ⑦ after last line, `obj.exit()` is called, which in turn calls `quit()` on SMTP object

About **contextlib**

- Provides useful context-aware utilities
 - **closing**
 - **contextmanager**
 - **suppress**

The **contextlib** module is part of the standard library. Its primary purpose is to provide some utilities for creating and working with context managers.

It also provides abstract base classes for context managers.

contextlib.closing

- Wrapper for non-context-aware objects
- Adds context to any object that has a `.close()` method
- `.close()` is called even if error is raised

closing is a wrapper for objects that do not implement the context protocol (in other words, that don't have the required methods). A **closing** object takes the other object as a parameter, and returns a new object that works just like the original, except it has an `__exit__` method that calls `close()` on the original object.

Example

`context_closing.py`

```
#!/usr/bin/env python
from contextlib import closing

class Spam():
    def close(self):
        print("I'm outta here...")

with closing(Spam()) as c:
    pass
```

- ① define class containing `close()` method
- ② `close()` method may be called explicitly, or implicitly via `with`
- ③ `c.close()` is implicitly called

`context_closing.py`

```
I'm outta here...
```

contextlib.contextmanager

- Defines factory function for context managers
- Shortcut—does not require class
- Useful on objects that
 - are not context managers
 - don't have a `close()` method

`contextlib.contextmanager` is a decorator that can turn a function into a context manager, avoiding the extra work of creating a class. It is useful for objects that were not written with the context protocol, and which do not already have a `.close()` method.

In the decorated function, there must be a `yield` statement that yields a single object. This object is assigned to the context variable as though it were returned by `_enter_`. Code that comes after the `yield` statement is executed at the end of the block, as though it were in the `_exit_` method.

Example

`contextmanager_example.py`

```
#!/usr/bin/env python
from contextlib import contextmanager

@contextmanager
def spam(): ①
    yield 25 ②
    print("starting exit...") ③
    print("...finishing")

with spam() as s: ④
    print("s is", s)
    print("in the context block...")
```

① defines normal function (not a context-aware class)

② provide value for context variable (assigned by `with` statement)

③ code to be executed when block exits

④ `s` gets value `yield`'ed in function

contextmanager_example.py

```
s is 25
in the context block...
starting exit...
...finishing
```

contextlib.suppress

- Easy way to ignore exceptions
- Added in Python 3.4
- Handy for filesystem issues

The **suppress** manager makes it easy to ignore expected exceptions. A typical use case is creating a folder, when the folder might already exist. Just use **suppress** as the context manager (**with** statement) expression, passing in one or more exceptions that should be ignored.

NOTE

suppress is only available in 3.4 and later versions of Python

Example

suppressing_exceptions.py

```
#!/usr/bin/env python
from contextlib import suppress
import os

FILE_NAME = 'wombat_dossier.txt'

with suppress(FileNotFoundError): ①
    os.remove(FILE_NAME) ②
```

① register FileNotFoundError with context manager

② when FileNotFoundError is raised, ignore it

Chapter 4 Exercises

Exercise 4-1 (clear_list.py)

Write a context-aware class named ClearList that accepts a list, and sets all the elements to 0 on exit from the block.

Create a list of 25 integers using `list()` and `range()`. Use a `with` block to create an instance of the `ClearList` class, passing in the list of 25 integers.

Change all the elements to have the value 1234 (HINT: use a `for` loop and `len()`). Print out the list.

After the `with` block, print out the list to see that the values are all set to 0.k

Chapter 5: Generators and other Iterables

Objectives

- Unpack function arguments
- Unpack iterables with wildcards
- Use lambda functions for brevity
- Understand Python iterables
- Select and transform data with list comprehensions
- Create generators in 3 different ways

Iterables

- An iterable is an expression that can be looped through with for
- Some iterables are collections (list, tuple, str, bytes, dict, set)
- Many iterables are generators(enumerate(), dict.items(), open(), zip(), reversed(), etc)

for is one of the most powerful operators in Python. It is used for looping through a set of values. The set of values is provided by an iterable. Python has many built-in iterables – a file object, for instance, which allows iterating through the lines in a file.

All collections (list, tuple, str, bytes) are iterables. They keep all their values in memory.

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

for

Unpacking function parameters

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

What do you do if you have a list (or other iterable) of three values, and you want to pass them to a method that expects three positional parameters? You could say value[0], value[1], value[2], etc., but there's a more Pythonic way:

Use * to unpack the iterable into individual items. The iterable must have the same number of values as the number of parameters in the function. However, you can combine individual arguments with unpacking:

```
foo(5, 10, *values)
```

In a similar way, use two asterisks to unpack a dictionary.

Example

param_unpacking.py

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

from datetime import date

dates = [
    (1968, 10, 11),
    (1968, 12, 21),
    (1969, 3, 3),
    (1969, 5, 18),
    (1969, 7, 16),
    (1969, 11, 14),
    (1970, 4, 11),
    (1971, 1, 31),
    (1971, 7, 26),
    (1972, 4, 16),
    (1927, 12, 7),
] ①

for dt in dates:
    d = date(*dt) ②
    print(d)

print()

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

sort_opts = {
    'key': lambda e: e.lower(),
    'reverse': True,
} ③

sorted_fruits = sorted(fruits, **sort_opts) ④
print(sorted_fruits)
```

- ① tuple of dates
- ② instead of date(dt[0], dt[1], dt[2])
- ③ config info in dictionary
- ④ dictionary converted to named parameters

param_unpacking.py

```
1968-10-11
1968-12-21
1969-03-03
1969-05-18
1969-07-16
1969-11-14
1970-04-11
1971-01-31
1971-07-26
1972-04-16
1927-12-07

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

Iterable unpacking

- Frequently used with for loops
- Can be used anywhere
- Commonly used with `enumerate()` and `DICT.items()`

It is convenient to unpack an iterable into a list of variables. It can be done with any iterable, and is frequently done with the loop variables of a for loop, rather than unpacking the value of each iteration separately.

```
var1, ... = iterable
```

TIP

Underscore ('_')can be used as a variable name for values you don't care about.

Example

iterable_unpacking.py

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

values = ['a', 'b', 'c']

x, y, z = values ①

print(x, y, z)
print()

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

for row in people:
    first_name, last_name, _ = row ② ③
    print(first_name, last_name)
print()

for first_name, last_name, _ in people: ④
    print(first_name, last_name)
print()
```

- ① unpack values (which is an iterable) into individual variables
- ② unpack **row** into variables
- ③ `_` is used as a "junk" variable that won't be used
- ④ a **for** loop unpacks if there is more than one variable

iterable_unpacking.py

```
a b c
```

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

```
Bill Gates
Steve Jobs
Paul Allen
Larry Ellison
Mark Zuckerberg
Sergey Brin
Larry Page
Linux Torvalds
```

Extended iterable unpacking

- Allows for one "wild card"
- Allows common "first, rest" unpacking

When unpacking iterables, sometimes you want to grab parts of the iterable as a group. This is provided by extended iterable unpacking.

One (and only one) variable in the result of unpacking can have a star prepended. This variable will receive all values from the iterable that do not go to other variables.

NOTE

[Extended variable unpacking is not available in Python 2.](#)

Example

extended_iterable_unpacking.py

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

values = ['a', 'b', 'c', 'd', 'e'] ①

x, y, *z = values ②
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

x, *y, z = values ②
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

*x, y, z = values ②
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

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

for *name, _ in people: ③
    print(name)
print()
```

① **values** has 6 elements

② * takes all extra elements from iterable

③ **name** gets all but the last field

extended_iterable_unpacking.py

```
x: a      y: b      z: ['c', 'd', 'e']

x: a      y: ['b', 'c', 'd']      z: e

x: ['a', 'b', 'c']      y: d      z: e

['Bill', 'Gates']
['Steve', 'Jobs']
['Paul', 'Allen']
['Larry', 'Ellison']
['Mark', 'Zuckerberg']
['Sergey', 'Brin']
['Larry', 'Page']
['Linux', 'Torvalds']
```

What exactly is an iterable?

- Object that provides an *iterator*
- Can be **collection** or **generator**

An *iterable* is an object that provides an *iterator* (via the special method `__iter__`). An iterator is an object that responds to the `next()` builtin function, via the special method `__next__()`. In other words, an iterator is an object which can be looped over with a `for` loop.

All generators are iterables. Most sequence and mapping types are also iterables. Generators are also iterators; `next()` can be used on them directly.

For some iterables (including most collections), you can not use `next()` on them directly; use `iter()` to get the iterator.

```
>>> r = range(1, 4)
>>> i = iter(r)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

NOTE

A `for` loop is really a `while` loop in disguise. It repeatedly calls `next()` on an iterator, and stops when `StopIteration` is raised.

List comprehensions

- Shortcut syntax for a **for** loop
- Selects and/or modifies values
- Can be faster

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

A loop like this:

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

can be rewritten as

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

A conditional if may be added:

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

A list comprehension can both select and modify values from an iterable and append them to a new list.

Common uses:

- convert values to strings
- pull fields out of a nested data structure
- select items and make them lower case
- normalize strings by cleaning up whitespace

Example

list_comprehensions.py

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

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

ufruits = [fruit.upper() for fruit in fruits]      ①
afruits = [fruit.title() for fruit in fruits if fruit.startswith('a')]]  ②

print("ufruits:", ufruits)
print("afruits:", afruits)
print()

values = [2, 42, 18, 39.7, 92, '14', "boom", ['a', 'b', 'c']]

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

print("doubles:", doubles, '\n')

nums = [x for x in values if isinstance(x, int)]  ④
print(nums, '\n')

dirty_strings = ['Gronk', 'PULABA', 'floog']

clean = [d.strip().lower() for d in dirty_strings]
for c in clean:
    print(">{}<".format(c), end=' ')
print("\n")

suits = 'Clubs', 'Diamonds', 'Hearts', 'Spades'
ranks = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

deck = [(rank, suit) for suit in suits for rank in ranks]  ⑤

for rank, suit in deck:
    print("{}-{}".format(rank, suit))
```

- ① Simple transformation of all elements
- ② Transformation of selected elements only
- ③ Any kind of data is OK
- ④ Select only integers from list
- ⑤ More than one **for** is OK

list_comprehensions.py

```
ufruits: ['WATERMELON', 'APPLE', 'MANGO', 'KIWI', 'APRICOT', 'LEMON', 'GUAVA']
afruits: ['Apple', 'Apricot']

doubles: [4, 84, 36, 79.4, 184, '1414', 'boomboom', ['a', 'b', 'c', 'a', 'b', 'c']]

[2, 42, 18, 92]

>gronk< >pulaba< >floog<

2-Clubs
3-Clubs
4-Clubs
5-Clubs
6-Clubs
7-Clubs
```

...

Generators

- Lazy iterables
- Do not contain data
- Provide values on demand
- Save memory

As part of the transition from Python 2 to Python 3, many operations that returned lists were changed to return **generators**

A generator is an object that provides values on demand (AKA "lazy"), rather than storing all the values (AKA "eager"). Generators are usually based on some other iterable. Another way of saying this is that a generator returns an iterator that returns a new value each time `next()` is called on it, until there are no more values.

The big advantage of generators is saving memory. They act as a *view* over a set of data.

Generators may only be used once. After the last value is provided, the generator must be recreated in order to start over.

Generators may not be indexed, and have no length. The only thing to do with a generator is to loop over it with **for**.

There are three ways to create generators in Python:

- generator expression
- generator function
- generator class

Generator Expressions

- Like list comprehensions, but create a generator object
- Use parentheses rather than brackets
- Saves memory

A generator expression is similar to a list comprehension, but it provides a generator instead of a list. That is, while a list comprehension returns a complete list, the generator expression returns one item at a time. The generator does not contain a copy of the source iterable, so it saves memory. In many cases generators are also faster than list comprehensions.

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

If a generator expression is passed as the only argument to a function, the extra parentheses are not needed.

```
my_func(float(i) for i in values)
```

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

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

Example

generator_expressions.py

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

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

ufruits = (fruit.upper() for fruit in fruits) ①
afruits = (fruit.title() for fruit in fruits if fruit.startswith('a')) 

print("ufruits:", ".join(ufruits))
print("afruits:", ".join(afruits))
print()

values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]
doubles = (v * 2 for v in values)

print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print("\n")

nums = (int(s) for s in values if isinstance(s, int))
for n in nums:
    print(n, end=' ')
print("\n")

dirty_strings = ['Gronk', 'PULABA', 'floog']

clean = (d.strip().lower() for d in dirty_strings)
for c in clean:
    print(">{}<".format(c), end=' ')
print("\n")

powers = ((i, i ** 2, i ** 3) for i in range(1, 11))
for num, square, cube in powers:
    print("{:2d} {:3d} {:4d}".format(num, square, cube))
print()
```

- ① These are all exactly like the list comprehension example, but return generators rather than lists

generator_expressions.py

```
uftuits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: Apple Apricot

doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']

2 42 18 92

>gronk< >pulaba< >floog<

1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

Generator functions

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

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

Example

`sieve_generator.py`

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

def next_prime(limit):
    flags = set() ①

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j) ②
        yield i ③

np = next_prime(200) ④
for prime in np: ⑤
    print(prime, end=' ')
```

① initialize empty set (to be used for "is-prime" flags)

② add non-prime elements to set

③ execution stops here until next value is requested by for-in loop

④ next_prime() returns a generator object

⑤ iterate over **yielded** primes

sieve_generator.py

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

Example

line_trimmer.py

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

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

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

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

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

line_trimmer.py

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

Coroutines

- **yield** is an expression (can be assigned to)
- uses **generator.send()**
- **yield** returns None by default

When writing generator functions, you can send a value *back* to the generator. This is accomplished by calling *generator.send()*, and in the generator it becomes the return value of the **yield** statement. This makes the generator into a *coroutine*. While generators and coroutines are similar, they are not the same thing. Generators tend to be *producers*, while coroutines tend to be *consumers*.

When assigning to the **yield** statement, you need to call *next()* once to "prime" the coroutine and poise it to receive the first input, and thus send the first output. There's an "off-by-one" feeling to this.

Coroutines are not designed for iteration. They can be used to set up pipelines (sequences of coroutines that each do one interesting thing to your data. **send()** puts data into the pipeline.

This can be useful for async-like coding.

When writing generator functions, you can send a value *back* to the generator. This is accomplished by calling *generator.send()*, and in the generator it becomes the return value of the **yield** statement. This makes the generator into a *coroutine*.

```
...
ham = yield spam
...
...
next(p)
p.send('foo')
print(next(p))
...
```

See <http://dabeaz.com/coroutines/> for an in-depth look at coroutines.

Example

coroutine_example.py

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

def coroutine(): ①
    in_value = ''
    while True:
        in_value = yield in_value.upper() ②
        print('in_value:', in_value)

c = coroutine() ③

print(c) ④
print(next(c)) ⑤
print()

for letter in "alpha", "beta", 'gamma':
    print("out_value:", c.send(letter)) ⑥
print()

def faux_range(): ⑦
    i = 0
    while i < 4:
        yield i
        i += 1

def spam():
    yield from faux_range() ⑧

s = spam() ⑨

for x in s: ⑩
    print(x)
```

① Define a coroutine function

② Yield gets the result of **send()** AND provides the next value

- ③ Create instance of coroutine
- ④ c is a coroutine object (also a generator)
- ⑤ **next()** will get the next value and prime the coroutine. You *must* call next before calling **send()**
- ⑥ Send a value to the coroutine
- ⑦ define a generator
- ⑧ Define generator; **yield from** delegates to another generator
- ⑨ Create instance of generator
- ⑩ Looping through a spam() instance effectively loops over faux_range()

coroutine_example.py

```
<generator object coroutine at 0x10f19d9a8>

in_value: alpha
out_value: ALPHA
in_value: beta
out_value: BETA
in_value: gamma
out_value: GAMMA

0
1
2
3
```

Generator classes

- Implement `__iter__` and `__next__`
- Emit values via **for** or `next()`
- Raise `StopIteration` exception when there are no more values to emit

The most flexible way to create a generator is by defining a generator class.

Such a class must implement two special methods:

`__iter__` must return an object that implements `__next__`. In most cases, this is the same class, so `__iter__` just returns `self`.

`__next__` returns the next value from the generator. This can be in a loop, or in sequential statements, or any combination.

When there are no more values to return, `__next__` should raise **StopIteration**.

Example

trimmedfile.py

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

class TrimmedFile:
    def __init__(self, file_name): ①
        self._file_in = open(file_name)

    def __iter__(self): ②
        return self

    def __next__(self): ③
        line = self._file_in.readline()
        if line == '':
            raise StopIteration ④
        else:
            return line.rstrip('\n\r') ⑤

if __name__ == '__main__':
    trimmed = TrimmedFile('../DATA/mary.txt') ⑥
    for line in trimmed:
        print(line)
```

① constructor is passed file name

② A generator must implement `__iter__()`, which must return an iterator. Typically it returns `self`, as the generator *is* the iterator

③ `__next__()` returns the next value of the generator

④ Raise **StopIteration** when there are no more values to generate

⑤ The actual work of this generator—remove the newline from the line

⑥ To use the generator, create an instance and iterator over it.

trimmedfile.py

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

Chapter 5 Exercises

Exercise 5-1 (pres_upper.py)

1. Use a list comprehension to read all the presidents' first and last names into a list of tuples
2. Use another list comprehension to make a new list with the names joined by spaces, and in upper case.
3. Loop through the upper case list and print out the names one per line.

Exercise 5-2 (pres_upper_gen.py)

Redo pres_upper.py but use two generator expressions rather than two list comprehensions.

Exercise 5-3 (pres_gen.py)

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

Exercise 5-4 (fauxrange.py)

Write a generator class named FauxRange that emulates the builtin range() function. Instances should take up to three parameters, and provide a range of integers (or, consider using floats — does that change the class?).

Chapter 6: Container Classes

Objectives

- Recap builtin container types
- Discover extra container types in standard library
- Save space with array objects
- Create custom variations of container types

Container classes

- Contain all elements in memory objects
- Elements are objects or key/object pairs
- Have a length
- Are indexable and iterable

Container classes are objects that can hold multiple objects. All of the objects contained are kept in memory. Containers can be indexed, and can be iterated over. All containers have a length, which is the number of elements.

Builtin containers

- list, tuple array-like
- dict dictionary
- set, frozenset set

Several container types are builtin.

There are two array-like containers – list and tuple. A list is a dynamic array, while a tuple is more like a struct or a record types. Both are array-like, meaning they have a length, can be indexed, and can be sliced.

The dict type is a dictionary of key/value pairs. In some languages, dictionaries are called hashes, or even more exotic names. Dictionaries are dynamic. Keys must be unique. The set type contains unique values. Elements may be added to and deleted from a set. A frozenset is a readonly set.

Example

builtin_containers.py

```
#!/usr/bin/env python
alist = ['alpha', 'beta', 'gamma', 'eta', 'zeta']
atuple = ('123 Elm Street', 'Toledo', 'Ohio')
adict = {'alpha': 5, 'beta': 10, 'gamma': 15}
aset = {'alpha', 'beta', 'gamma', 'eta', 'zeta'}

print(alist[0], atuple[0], adict['alpha'])
print(alist[-1], atuple[-1])
print(alist[:3], alist[3:], alist[2:5], alist[-2:])
print('alpha' in alist, 'Ohio' in atuple, 'gamma' in adict, 'zeta' in aset)
print(len(alist), len(atuple), len(adict), len(aset))
print(alist.count('alpha'), atuple.count('Ohio'))
```

builtin_containers.py

```
alpha 123 Elm Street 5
zeta Ohio
['alpha', 'beta', 'gamma'] ['eta', 'zeta'] ['gamma', 'eta', 'zeta'] ['eta', 'zeta']
True True True True
5 3 3 5
1 1
```

Containers in the standard library

- Many containers in the collections module
- Variations of lists, tuples, and dicts

The collections module provides several extra container types. In general, these are variations on lists, tuples, and dicts.

These types are:

- Counter – dictionary designed for counting
- defaultdict – dictionary with default value for missing keys
- deque – array optimized for access at ends only
- namedtuple – tuple with named elements
- OrderedDict – dictionary that remembers order elements were inserted

A **Counter** is a dict with a default value of 0, so values may be incremented without check to see whether the key exists. In addition, counter provides the n most common elements counted.

A **defaultdict** is a dict that provides a default value for missing keys. When the defaultdict is created, you pass in a function that provides that default value. If you need a constant value, such as 0 or "", you can use a lambda expression for the function.

A **deque** (pronounced "deck") is a double-ended queue. It is optimized for inserting and removing from the ends, and is much more efficient than a **list** for this purpose. The deque can be initialized with any iterable.

A **namedtuple** is a tuple with specified field names. In addition to accessing fields as *tuple[0]*, you can access them as *tuple.field_name*. Named tuples map very well to C **structs**.

An **OrderedDict** is a dictionary which preserves order. Any iteration over the dictionary's keys or values is guaranteed to be in the same order that the elements were added.

NOTE

Starting with Python 3.6, all dictionaries preserve order.

Example

stdlib_containers.py

```
#!/usr/bin/env python
from collections import Counter, defaultdict, deque, namedtuple, OrderedDict

# Counter
with open('../DATA/words.txt') as words_in:
    word_counter = Counter(line[0] for line in words_in) ①

print("a: {} q: {} x: {}".format(
    word_counter['a'], word_counter['q'], word_counter['x']) ②
))

print(word_counter.most_common(10)) ③
print('-' * 60)

# defaultdict
fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]

fruit_by_first = defaultdict(list) ④

for fruit in fruits:
    fruit_by_first[fruit[0]].append(fruit) ⑤

for letter, fruits in sorted(fruit_by_first.items()):
    print(letter, fruits)
print('-' * 60)

# deque
d = deque() ⑥
for c in 'abcdef':
    d.append(c) ⑦
print(d)
for c in 'ghijkl':
    d.appendleft(c) ⑧
print(d)
d.extend('mno') ⑨
print(d)
d.extendleft('pqr') ⑩
```

```
print(d)
print(d[9])
print(d.pop(), d.popleft()) ⑪
print(d)
print('-' * 60)

# namedtuple
President = namedtuple('President', 'first_name, last_name, party') ⑫
p = President('Theodore', 'Roosevelt', 'Republican') ⑬
print(p, len(p))
print(p[0], p[1], p[-1])
print(p.first_name, p.last_name, p.party) ⑭

p = President(last_name='Lincoln', party='Republican', first_name='Abraham')
print(p)
print(p.first_name, p.last_name)
print('-' * 60)

# OrderedDict
od = OrderedDict() ⑮
od['alpha'] = 10
od['beta'] = 20
od['gamma'] = 30
od['delta'] = 40
od['eta'] = 50

for key, value in od.items():
    print(key, value) ⑯
```

stdlib_containers.py

```
a: 10485 q: 827 x: 137
[('s', 19030), ('c', 16277), ('p', 14600), ('a', 10485), ('r', 10199), ('d', 10189),
('m', 9689), ('b', 9329), ('t', 8782), ('e', 7086)]
-----
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
-----
deque(['a', 'b', 'c', 'd', 'e', 'f'])
deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f'])
deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o'])
deque(['r', 'q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f',
'm', 'n', 'o'])
a
o r
deque(['q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm',
'n'])
-----
President(first_name='Theodore', last_name='Roosevelt', party='Republican') 3
Theodore Roosevelt Republican
Theodore Roosevelt Republican
President(first_name='Abraham', last_name='Lincoln', party='Republican')
Abraham Lincoln
-----
alpha 10
beta 20
gamma 30
delta 40
eta 50
```

The array module

- Efficient numeric arrays (fast than list)
- More compact than list
- Traditional arrays of the same type

The **array** module provides an array object which implements an efficient numeric array where each element is the same type. There are a number of different numeric types that can be used. The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the itemsize attribute.

This is efficient when you have a large number of numeric values to store, as it can take much less space than a normal list object, and is faster.

The numeric type flags are nearly the same as those use by the **struct** module.

The **ndarray** type in the **numpy** framework is very fast and efficient, and is typically used in science and engineering.

Example

array_examples.py

```
#!/usr/bin/env python
from sys import getsizeof
from array import array
from random import randint

values = [randint(1, 30000) for i in range(1000)] ①

print(f'Size of integer list: {getsizeof(values)}\n')

for datatype in 'i', 'h', 'L', 'Q', 'd':
    data_array = array(datatype, values) ②
    print(f'Size of {datatype} array: {getsizeof(data_array)} Contents:',
          data_array[:5], '...') ③
print()
```

array_examples.py

```
Size of integer list: 9024  
Size of i array: 4064  Contents: array('i', [14266, 4183, 17228, 23396, 22737]) ...  
Size of h array: 2064  Contents: array('h', [14266, 4183, 17228, 23396, 22737]) ...  
Size of L array: 8064  Contents: array('L', [14266, 4183, 17228, 23396, 22737]) ...  
Size of Q array: 8064  Contents: array('Q', [14266, 4183, 17228, 23396, 22737]) ...  
Size of d array: 8064  Contents: array('d', [14266.0, 4183.0, 17228.0, 23396.0,  
22737.0]) ...
```

Table 2. array object type codes

Format	C Type	Python Type	Standard size	Notes
b	signed char	integer	1	(1),(3)
B	unsigned char	integer	1	(3)
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)

NOTE

The 'q' and 'Q' type codes are available only if the platform C compiler used to build Python supports C long long, or, on Windows, __int64.

Emulating builtin types

- Inherit from builtin type
- Override special methods as needed
- Use super() to invoke base class's special methods

To emulate builtin types, you can inherit from builtin classes and override special methods as needed to get the desired behavior; other special methods will work in the normal way.

When overriding the special methods, you usually want to invoke the base class's special methods. Use the super() builtin function; the syntax is:

```
super().{dunder}specialmethod{dunder}(...)
```

To start from scratch, you can inherit from abstract base classes defined in the **collections.abc** module. These can be used as mixins, to make sure you're implemented the necessary methods for various container types. In other words, Sized plus Iterable will create a list-like object.

Example

`container_abc.py`

```
#!/usr/bin/env python
from collections.abc import Sized, Iterator ①

class BadContainer(Sized): ②
    pass

class GoodContainer(Sized):
    def __len__(self): ③
        return 42

try:
    bad = BadContainer() ④
```

```
except TypeError as err:  
    print(err) ⑤  
else:  
    print(bad)  
  
print()  
  
try:  
    good = GoodContainer() ⑥  
except TypeError as err:  
    print(err)  
else:  
    print(good) ⑦  
    print(len(good)) ⑧  
  
print()  
  
class MyIterator(Iterator): ⑨  
    data = 'a', 'b', 'c'  
    index = 0  
  
    def __next__(self): ⑩  
        if self.index >= len(self.data):  
            raise StopIteration  
        else:  
            return_val = self.data[self.index]  
            self.index += 1  
            return return_val  
  
m = MyIterator() ⑪  
for i in m: ⑫  
    print(i)  
print()  
  
print(hasattr(m, '__iter__')) ⑬
```

container_abc.py

```
Can't instantiate abstract class BadContainer with abstract methods __len__
```

```
<__main__.GoodContainer object at 0x1034a4630>
```

```
42
```

```
a
```

```
b
```

```
c
```

```
True
```

Table 3. Special Methods and Variables

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

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

Method or Variables	Description
<code>_radd_(self, other)</code> <code>_rsub_(self, other)</code> <code>_rmul_(self, other)</code> <code>_rdiv_(self, other)</code> <code>_rtruediv_(self, other)</code> <code>_rfloordiv_(self, other)</code> <code>_rmod_(self, other)</code> <code>_rdivmod_(self, other)</code> <code>_rpow_(self, other)</code> <code>_rlshift_(self, other)</code> <code>_rrshift_(self, other)</code> <code>_rand_(self, other)</code> <code>_rxor_(self, other)</code> <code>_ror_(self, other)</code>	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)
<code>iadd_(self, other)</code> <code>isub_(self, other)</code> <code>imul_(self, other)</code> <code>idiv_(self, other)</code> <code>itruediv_(self, other)</code> <code>ifloordiv_(self, other)</code> <code>imod_(self, other)</code> <code>ipow_(self, other[, modulo])</code> <code>ilshift_(self, other)</code> <code>irshift_(self, other)</code> <code>iand_(self, other)</code> <code>ixor_(self, other)</code> <code>ior_(self, other)</code>	Implement augmented (+=, -=, etc.) arithmetic operators
<code>neg_(self)</code> <code>pos_(self)</code> <code>abs_(self)</code> <code>invert_(self)</code>	Implement unary arithmetic operators -, +, abs(), and ~
<code>oct_(self)</code> <code>hex_(self)</code>	Implement oct() and hex() builtins
<code>index_(self)</code>	Implement operator.index()
<code>coerce_(self, other)</code>	Implement "mixed-mode" numeric arithmetic.

Creating list-like containers

- Inherit from list
- Override special methods as needed
- Commonly overridden methods
 - `__getitem__`
 - `__setitem__`
 - `__append__`

To create a list-like container, inherit from `list`. Commonly overridden methods include `__getitem__` and `__setitem__`.

Example

`multiindexlist.py`

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

class MultiIndexList(list): ①

    def __getitem__(self, item): ②
        if isinstance(item, tuple): ③
            if len(item) == 0:
                raise ValueError("Tuple must be non-empty")
            else:
                tmp_list = []
                for index in item:
                    tmp_list.append(
                        super().__getitem__(index) ④
                    )
                return tmp_list
        else:
            return super().__getitem__(item) ⑤

if __name__ == '__main__':
    m = MultiIndexList(
        'banana peach nectarine fig kiwi lemon lime'.split()
```

```
) ⑥
m.append('apple') ⑦
m.append('mango')
print(m)

print(m[0])
print(m[1])
print(m[5, 2, 0])
print(m[:4])
print(len(m))
print(m[5, ])
print(m[:2, -2:])
print()
print(m)
m.extend(['durian', 'kumquat'])
print(m)
print()
for fruit in m:
    print(fruit)
print(len(fruit))
```

multiindexlist.py

```
['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango']
banana
peach
['lemon', 'nectarine', 'banana']
['banana', 'peach', 'nectarine', 'fig']
9
['lemon']
[['banana', 'peach'], ['apple', 'mango']]

['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango']
['banana', 'peach', 'nectarine', 'fig', 'kiwi', 'lemon', 'lime', 'apple', 'mango',
'durian', 'kumquat']

banana
peach
nectarine
fig
kiwi
lemon
lime
apple
mango
durian
kumquat
7
```

Creating dict-like containers

- Inherit from dict
- Implement special methods
- Commonly overridden methods
 - `__getitem__`
 - `__haskey__`
 - `__setitem__`

To create custom dictionaries, inherit from dict and implement special methods as needed. Commonly overridden special methods include `__getitem__()`, `__setitem__()`, and `__haskey__()`.

Example

stringkeydict.py

```

#!/usr/bin/env python

class StringKeyDict(dict): ①
    def __setitem__(self, key, value): ②
        if isinstance(key, str): ③
            super().__setitem__(key, value) ④
        else:
            raise TypeError("Keys must be strings not {}s".format(
                type(key).__name__
            ))
    if __name__ == '__main__':
        d = StringKeyDict(a=10, b=20) ⑥
        for k, v in [('c', 30), ('d', 40), (1, 50), (('a', 1), 60), (5.6, 201)]:
            try:
                print("Setting {} to {}".format(k, v), end=' ')
                d[k] = v ⑦
            except TypeError as err:
                print(err) ⑧
            else:
                print('SUCCESS')
        print()
        print(d)

```

stringkeydict.py

```

Setting c to 30 SUCCESS
Setting d to 40 SUCCESS
Setting 1 to 50 Keys must be strings not ints
Setting ('a', 1) to 60 Keys must be strings not tuples
Setting 5.6 to 201 Keys must be strings not floats

```

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

Free-form containers

- Easy to create hybrid containers
- Objects may implement any special methods
- Create list+dict, ordered set
- Be creative

There's really no limit to the types of objects you can create. You can make hybrids that act like lists and dictionaries at the same time. Just implement the special methods required for this behavior.

A well-known example of this is the **Element** class in the `lxml.etree` module. An Element is a list of its children, and at the same time is a dictionary of its XML attributes:

```
e = Element(...)  
child_element = e[0]  
attr_value = e.get("attribute")
```

Chapter 6 Exercises

Exercise 6-1 (maxlist.py)

Create a new type, MaxList, that will only grow to a certain size. It should raise an IndexError if the user attempts to add an item beyond the limit.

HINT: you will need to override the append() and extend() methods; to be thorough, you would need to override __init__() as well, so that the initializer doesn't have more than the maximum number of items., and pop(), and maybe some others.

For the ambitious (ringlist.py): Modify MaxList so that once it reaches maximum size, appending to the list also removes the first element, so the list stays constant size.

Exercise 6-1 (strdict.py)

Create a new type, NormalStringDict, that only allows strings as keys *and* values. Values are normalized by making them lower case and remove all whitespace.

Chapter 7: Advanced Data Handling

Objectives

- Set default dictionary values
- Count items with the Counter object
- Define named tuples
- Prettyprint data structures
- Create and extract from compressed archives
- Save Python structures to the hard drive

Deep vs shallow copying

- Normal assignments create aliases
- New objects are shallow copies
- Use the **copy.deepcopy** module for deep copies *

Consider the following code:

```
colors = ['red', 'blue', 'green']
c1 = colors
```

The assignment to variable **c1** does not create a new object; it is another name that is *bound* to the same list object as the variable **colors**.

To create a new object, you can either use the list constructor `list()`, or use a slice which contains all elements:

```
c2 = list(colors)
c3 = colors[::]
```

In both cases, **c2** and **c3** are each distinct objects.

However, the elements of **c2** and **c3** are not copied, but are still bound to the same objects as the elements of **colors**.

For example:

```
data1 = [ [1, 2, 3], [4, 5, 6] ]
d1 = list(data)
d1[0].append(99)
print(data1)
```

This will show that the first element of **data1** contains the value 99, because **data1[0]** and **d1[0]** are both bound to the same object. To do a "deep" (recursive) copy, use the **deepcopy** function in the **copy** module:

Example

deep_copy.py

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

data = [
    [1, 2, 3],
    [4, 5, 6],
]

d1 = data ①
d2 = list(data) ②
d3 = copy.deepcopy(data) ③

d1.append("d1") ④
d1[0].append(50) ⑤

d2.append("d2") ⑥
d2[0].append(99) ⑦

d3.append("d3") ⑧
d3[0].append(500) ⑨

print("data:", data, id(data))
print("d1:", d1, id(d1))
print("d2:", d2, id(d2))
print("d3:", d3, id(d3))
print()

print("id(d1[0]):", id(d1[0]))
print("id(d2[0]):", id(d2[0]))
print("id(d3[0]):", id(d3[0]))
```

① Bind d1 to same object as data

② Make shallow copy of data and store in d2

③ Make deep copy of data and store in d3

④ Append to d1 (same as appending to data)

⑤ Append to first element of d1 (same first element as data)

⑥ Append to d2 (does not affect data)

- ⑦ Append to first element of d2 (same first element as data and d1)
- ⑧ Append to d3 (does not affect data)
- ⑨ Append to first element of d3 (does not affect data, d1, or d2)

deep_copy.py

```
data: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 4391055752
d1: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 4391055752
d2: [[1, 2, 3, 50, 99], [4, 5, 6], 'd2'] 4391124936
d3: [[1, 2, 3, 500], [4, 5, 6], 'd3'] 4391124680

id(d1[0]): 4390775368
id(d2[0]): 4390775368
id(d3[0]): 4391124616
```

Default dictionary values

- Use defaultdict from the collections module
- Specify function that provides default value
- Good for counting, datasets

Normally, when you use an invalid key with a dictionary, it raises a `KeyError`. The `defaultdict` class in the `collections` module allows you to provide a default value, so there will never be a `KeyError`. You will provide a function that returns the default value. A `lambda` function can be used for the function.

Example

defaultdict_ex.py

```
#!/usr/bin/env python
"""
@author: jstrick
Created on Wed Mar 13 20:45:42 2013
"""

from collections import defaultdict

dd = defaultdict(lambda: 0) ①

dd['spam'] = 10 ②
dd['eggs'] = 22

print(dd['spam']) ③
print(dd['eggs'])
print(dd['foo']) ④

print('-' * 60)

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape" ]

fruit_info = defaultdict(list)

for fruit in fruits:
    first_letter = fruit[0]
    fruit_info[first_letter].append(fruit)

for letter, fruits in sorted(fruit_info.items()):
    print(letter, fruits)
```

① create default dict with function that returns 0

② assign some values to the dict

③ print values

④ missing key 'foo' invokes function and returns 0

defaultdict_ex.py

```
10
22
0
-----
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
```

Counting with Counter

- Use `collections.Counter`
- Values default to 0
- Just increment

For ease in counting, collections provides a `Counter` object. This is essentially a `defaultdict` whose default value is zero. Thus, you can just increment the value for any key, whether it's been seen before or no.

Example

`count_with_counter.py`

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

from collections import Counter

counts = Counter() ①

with open("../DATA/breakfast.txt") as breakfast_in:
    for line in breakfast_in:
        item = line.rstrip('\n\r') ②
        counts[item] += 1 ③

for item, count in counts.items(): ④
    print(item, count)
```

① create `Counter` object (which defaults to 0 for missing keys)

② remove EOL from line

③ increment count for current item

④ iterate over results

`count_with_counter.py`

```
spam 10
eggs 3
crumpets 1
```

Named Tuples

- From collections module
- Like tuple, but each element has a name
- tuple.name in addition to tuple[n]

A named tuple is a tuple where each element has a name, and can be accessed via the name in addition to the normal access by index. Thus, if p were a named tuple representing a point, you could say p.x and p.y, or you could say p[0] and p[1].

A named tuple is created with the namedtuple class in the collections module. You are essentially creating a new data type. Pass the name of the tuple followed by a string containing the individual field names separated by spaces. namedtuple() return a class which with you can create instances of the tuple.

You can convert a named tuple into a dictionary with the _asdict() method.

A named tuple is the closest thing Python has to a C struct.

Example

`named_tuple_ex.py`

```
#!/usr/bin/env python
"""
@author: jstrick
Created on Thu Mar 14 09:07:57 2013

"""

from collections import namedtuple

Knight = namedtuple('Knight', 'name title color quest comment') ①

k = Knight('Bob', 'Sir', 'green', 'whirled peas', 'Who am i?') ②

print(k.title, k.name) ③
print(k[1], k[0]) ④
print()

knights = {} ⑤
with open('../DATA/knights.txt') as knights_in:
    for line in knights_in:
        flds = line.rstrip().split(':')
        knights[flds[0]] = Knight(*flds) ⑥

for knight, info in knights.items(): ⑦
    print('{0} {1}: {2}'.format(info.title, knight, info.color))
```

- ① create named tuple class with specified fields (could also provide fieldnames as iterable)
- ② create named tuple instance (must specify all fields)
- ③ can access fields by name...
- ④ ...or index
- ⑤ initialize dict for knight data
- ⑥ add knight tuple to dictionary where key is knight name
- ⑦ iterate over dictionary; info is knight tuple

named_tuple_ex.py

```
Sir Bob  
Sir Bob  
  
King Arthur: blue  
Sir Galahad: red  
Sir Lancelot: blue  
Sir Robin: yellow  
Sir Bedevere: red, no blue!  
Sir Gawain: blue
```

Printing data structures

- Default representation of data structures is ugly
- pprint makes structures human friendly
- Use `pprint.pprint()`
- Useful for debugging

When debugging data structures, the `print` command is not so helpful. A complex data structure is just printed out all jammed together, one element after another, and is hard to read.

The `pprint` (pretty print) module will analyze a structure and print it out with indenting, to make it much easier to read.

You can customize the output with some named parameters: `indent` (default 1) specifies how many spaces to indent nested structures; `width` (default 80) constrains the width of the output; `depth` (default unlimited) says how many levels to print – levels beyond depth are show with '...'.

Example

pretty_printing.py

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

@author: jstrick
Created on Wed Mar 13 21:13:22 2013

"""

from pprint import pprint

struct = { ①
    'part1': [
        ['a', 'b', 'c'], ['d', 'e', 'f']
    ],
    'part2': {
        'red': 55,
        'blue': [8, 98, -3],
        'purple': ['Chicago', 'New York', 'L.A.'],
    },
    'part3': ['g', 'h', 'i'],
}

print('Without pprint:')
print(struct) ②
print()

print('With pprint:')
pprint(struct) ③
print()

print('With pprint (depth=2):')
pprint(struct, depth=2) ④
print()
```

① nested data structure

② print normally

③ pretty-print

④ only print top two levels of structure

pretty_printing.py

Without pprint:

```
{'part1': [['a', 'b', 'c'], ['d', 'e', 'f']], 'part2': {'red': 55, 'blue': [8, 98, -3], 'purple': ['Chicago', 'New York', 'L.A.']},
'part3': ['g', 'h', 'i']}
```

With pprint:

```
{'part1': [['a', 'b', 'c'], ['d', 'e', 'f']],
'part2': {'blue': [8, 98, -3],
'purple': ['Chicago', 'New York', 'L.A.'],
'red': 55},
'part3': ['g', 'h', 'i']}
```

With pprint (depth=2):

```
{'part1': [..., ...],
'part2': {'blue': [...], 'purple': [...], 'red': 55},
'part3': ['g', 'h', 'i']}
```

Zipped archives

- import zipfile for zipped files
- Get a list of files
- Extract files

The zipfile module allows you to read and write to zipped archives. In either case you first create a zipfile object; specifying a mode of "w" if you want to create an archive, and a mode of "r" (or nothing) if you want to read an existing zip file.

Modules for gzipped and bzipped files: gzip, bz2

Example

`zipfile_ex.py`

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

from zipfile import ZipFile, ZIP_DEFLATED
import os.path

# reading & extracting
rzip = ZipFile("../DATA/textfiles.zip") ①
print(rzip.namelist()) ②
ty = rzip.read('tyger.txt').decode() ③
print(ty[:50])
rzip.extract('parrot.txt') ④

# creating a zip file
wzip = ZipFile("example.zip", mode="w", compression=ZIP_DEFLATED) ⑤
for base in "parrot tyger knights alice poe_sonnet spam".split():
    filename = os.path.join("../DATA", base + '.txt')
    print("adding {} as {}".format(filename, base + '.txt'))
    wzip.write(filename, base + '.txt') ⑥
```

① Open zip file for reading

② Print list of members in zip file

③ Read (raw binary) data from member and convert from bytes to string

④ Extract member

⑤ Create new zip file

⑥ Add member to zip file

zipfile_ex.py

```
['fruit.txt', 'parrot.txt', 'tyger.txt', 'spam.txt']  
The Tyger
```

```
Tyger! Tyger! burning bright  
adding ../DATA/parrot.txt as parrot.txt  
adding ../DATA/tyger.txt as tyger.txt  
adding ../DATA/knights.txt as knights.txt  
adding ../DATA/alice.txt as alice.txt  
adding ../DATA/poe_sonnet.txt as poe_sonnet.txt  
adding ../DATA/spam.txt as spam.txt
```

Tar Archives

- Use the `tarfile` module
- Check for existence of tar archives
- Read and extract members from tar archives

To work with a tar archive, use the `tarfile` module. It can analyze a file to see whether it's a valid tar file, extract files from the archive, add files to the archive, and other tar chores.

The `is_tarfile()` function will check a tar file and return `True` if it is a valid tar file. This will work even if it is gzip bzip2 compressed.

For other actions, use the `open()` function to create a `TarFile` object. From this object you can list, add, or extract members, depending on how the tar file was opened.

A `TarFile` object is iterable as a list of `TarInfo` objects, which contain the details about each member. Use `getmembers()` to get a list of members as `TarInfo` objects. Use `extract()` to extract a file to disk. The first argument to `extract()` is the member name; the named parameter `path` specifies the destination directory (default `'.'`).

Use `extractall()` to extract all members to the current directory, or a specified path. A list of members may be specified; it must be a subset of the list returned by `getmembers()`.

To create a tar file, use `tarfile.open()` with a mode of `w` for an uncompressed archive. Use modes `w:gz` or `w:bz2` to the mode for a compressed archive. Use the `add()` method to add a file.

Example

tarfile_ex.py

```

#!/usr/bin/env python
import tarfile
import os

for tar_file in ('pres.tar', 'NOT_A.tar', 'potus.tar.gz'): ①
    filename = os.path.join('../DATA', tar_file)
    is_valid = tarfile.is_tarfile(filename) ②
    text = 'IS' if is_valid else 'IS NOT'
    print("{} {} a tarfile".format(filename, text))
print()

with tarfile.open('../DATA/pres.tar') as tarfile_in: ③
    for member in tarfile_in: ④
        print(member.name, member.size) ⑤
    print()

with tarfile.open('../DATA/pres.tar') as tarfile_in:
    tarfile_in.extract('presidents.txt', path='..TEMP') ⑥

with tarfile.open('../DATA/potus.tar.gz') as tarfile_in:
    tarfile_in.extract('presidents.csv', path='..TEMP') ⑥

with tarfile.open('..TEMP/text_files.tar', 'w') as tarfile_out: ⑦
    tarfile_out.add('../DATA/parrot.txt') ⑧
    tarfile_out.add('../DATA/alice.txt') ⑧

with tarfile.open('..TEMP/more_text_files.tar.gz', 'w:gz') as tar_gz_write: ⑨
    tar_gz_write.add('../DATA/parrot.txt') ⑧
    tar_gz_write.add('../DATA/alice.txt') ⑧

```

① iterate over sample files

② check to see if file is a tarfile

③ open tar file

④ iterate over members

⑤ access member data

⑥ extract member to local file

- ⑦ open new tar archive for writing
- ⑧ add member
- ⑨ open new tar archive for writing; archive is compressed with gzip

tarfile_ex.py

```
../DATA/pres.tar IS a tarfile
../DATA/NOT_A.tar IS NOT a tarfile
../DATA/potus.tar.gz IS a tarfile

presidents.csv 2467
presidents.txt 4473
```

Serializing Data

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Serializing data means taking a data structure and transforming it so it can be written to a file or other destination, and later read back into the same data structure.

Python uses the pickle module for data serialization.

To create pickled data, use either `pickle.dump()` or `pickle.dumps()`. Both functions take a data structure as the first argument. `dumps()` returns the pickled data as a string. `dump()` writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use `pickle.load()`, which takes a file-like object that has been open for writing, or `pickle.loads()` which reads from a string. Both functions return the original data structure that had been pickled.

NOTE pickle files must be opened in binary mode.

Example

pickling.py

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

@author: jstrick
Created on Sat Mar 16 00:47:05 2013

"""

import pickle
from pprint import pprint

①
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

data = [ ②
    colors,
    airports,
]

with open('../TEMP/pickled_data.pic', 'wb') as pic_out: ③
    pickle.dump(data, pic_out) ④

with open('../TEMP/pickled_data.pic', 'rb') as pic_in: ⑤
    pickled_data = pickle.load(pic_in) ⑥

pprint(pickled_data) ⑦
```

① some data structures

② list of data structures

③ open pickle file for writing in binary mode

④ serialize data structures to pickle file

- ⑤ open pickle file for reading in binary mode
- ⑥ de-serialize pickle file back into data structures
- ⑦ view data structures

pickling.py

```
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'}]
```

Chapter 7 Exercises

Exercise 7-1 (count_ext.py)

Write a script which will count number of files with each extension in a file tree. It should take the initial directory as a command line argument, and then display the total number of files with each distinct file extension that it finds. Files with no extension should be skipped. Use a Counter object to do the counting.

Exercise 7-2 (save_potus_info.py)

Write a script which creates a named tuple President, with fields lastname, firstname, birthplace, birthstate, and party. Read the data from Presidents.csv into an array of 44 President tuples.

Write the array out to a file named potus.pic. (use the Pickle module).

Exercise 7-3 (read_potus_info.py)

Write a script to open potus.pic, and restore the data back into an array.

Then loop through the array and print out each president's first name, last name, and party.

Exercise 7-4 (make_zip.py)

Write a script which creates a zip file containing save_potus_info.py, read_potus_info.py, and potus.pic.

Chapter 8: Metaprogramming

Objectives

- Learn what metaprogramming means
- Access local and global variables by name
- Inspect the details of any object
- Use attribute functions to manipulate an object
- Design decorators for classes and functions
- Define classes with the type() function
- Create metaclasses

Metaprogramming

- Writing code that writes (or at least modifies) code
- Can simplify some kinds of programs
- Not as hard as you think!
- Considered deep magic in other languages

Metaprogramming is writing code that generates or modifies other code. It includes fetching, changing, or deleting attributes, and writing functions that return functions (AKA factories).

Metaprogramming is easier in Python than many other languages. Python provides explicit access to objects, even the parts that are hidden or restricted in other languages.

For instance, you can easily replace one method with another in a Python class, or even in an object instance. In Java, this would be deep magic requiring many lines of code.

globals() and locals()

- Contain all variables in a namespace
- `globals()` returns all global objects
- `locals()` returns all local variables

The **globals()** builtin function returns a dictionary of all global objects. The keys are the object names, and the values are the objects values. The dictionary is "live"—changes to the dictionary affect global variables.

The **locals()** builtin returns a dictionary of all objects in local scope.

Example

globals_locals.py

```
#!/usr/bin/env python
from pprint import pprint ①

spam = 42 ②
ham = 'Smithfield'

def eggs(fruit): ③
    name = 'Lancelot' ④
    idiom = 'swashbuckling' ④
    print("Globals:")
    pprint(globals()) ⑤
    print()
    print("Locals:")
    pprint(locals()) ⑥

eggs('mango')
```

- ① import prettyprint function
- ② global variable
- ③ function parameters are local
- ④ local variable
- ⑤ globals() returns dict of all globals
- ⑥ locals() returns dict of all locals

globals_locals.py

```
Globals:  
{'__annotations__': {},  
 '__builtins__': <module 'builtins' (built-in)>,  
 '__cached__': None,  
 '__doc__': None,  
 '__file__': '/Users/jstrick/curr/courses/python/examples3/globals_locals.py',  
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x101d976d8>,  
 '__name__': '__main__',  
 '__package__': None,  
 '__spec__': None,  
 'eggs': <function eggs at 0x101dd31e0>,  
 'ham': 'Smithfield',  
 'pprint': <function pprint at 0x101f72f28>,  
 'spam': 42}  
  
Locals:  
{'fruit': 'mango', 'idiom': 'swashbuckling', 'name': 'Lancelot'}
```

The inspect module

- Simplifies access to metadata
- Provides user-friendly functions for testing metadata

The **inspect** module provides user-friendly functions for accessing Python metadata.

Example

inspect_ex.py

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

import inspect

class Spam: ①
    pass

def Ham(p1, p2='a', *p3, p4, p5='b', **p6): ②
    print(p1, p2, p3, p4, p5, p6)

for thing in (inspect, Spam, Ham):
    print("{}: Module? {}. Function? {}. Class? {}".format(
        thing.__name__,
        inspect.ismodule(thing), ③
        inspect.isfunction(thing), ④
        inspect.isclass(thing), ⑤
    ))
    print()

    print("Function spec for Ham:", inspect.getfullargspec(Ham)) ⑥
    print()

    print("Current frame:", inspect.getframeinfo(inspect.currentframe())) ⑦
```

- ① define a class
- ② define a function
- ③ test for module
- ④ test for function
- ⑤ test for class
- ⑥ get argument specifications for a function
- ⑦ get frame (function call stack) info

inspect_ex.py

```
inspect: Module? True. Function? False. Class? False
Spam: Module? False. Function? False. Class? True
Ham: Module? False. Function? True. Class? False

Function spec for Ham: FullArgSpec(args=['p1', 'p2'], varargs='p3', varkw='p6',
defaults=('a',), kwonlyargs=['p4', 'p5'], kwonlydefaults={'p5': 'b'},
annotations={})

Current frame:
Traceback(filename='/Users/jstrick/curr/courses/python/examples3/inspect_ex.py',
lineno=27, function='<module>', code_context=['print("Current frame:",',
inspect.getframeinfo(inspect.currentframe())) # <7>\n'], index=0)
```

Table 4. inspect module convenience functions

Function(s)	Description
ismodule(), isclass(), ismethod(), isfunction(), isgeneratorfunction(), isgenerator(), istraceback(), isframe(), iscode(), isbuiltin(), isroutine()	check object types
getmembers()	get members of an object that satisfy a given condition
getfile(), getsourcefile(), getsource()	find an object's source code
getdoc(), getcomments()	get documentation on an object
getmodule()	determine the module that an object came from
getclasstree()	arrange classes so as to represent their hierarchy
getargspec(), getargvalues()	get info about function arguments
formatargspec(), formatargvalues()	format an argument spec
getouterframes(), getinnerframes()	get info about frames
currentframe()	get the current stack frame
stack(), trace()	get info about frames on the stack or in a traceback

Working with attributes

- Objects are dictionaries of attributes
- Special functions can be used to access attributes
- Attributes specified as strings
- Syntax

```
getattr(object, attribute [,defaultvalue] )
hasattr(object, attribute)
setattr(object, attribute, value)
delattr(object, attribute)
```

All Python objects are essentially dictionaries of attributes. There are four special builtin functions for managing attributes. These may be used to programmatically access attributes when you have the name as a string.

getattr() returns the value of a specified attribute, or raises an error if the object does not have that attribute. `getattr(a, 'spam')` is the same as `a.spam`. An optional third argument to `getattr()` provides a default value for nonexistent attributes (and does not raise an error).

hasattr() returns the value of a specified attribute, or `None` if the object does not have that attribute.

setattr() an attribute to a specified value.

delattr() deletes an attribute and its corresponding value.

Example

attributes.py

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

class Spam():

    def eggs(self, msg): ①
        print("eggs!", msg)

    s = Spam()

    s.eggs("fried")

    print("hasattr()", hasattr(s, 'eggs')) ②

    e = getattr(s, 'eggs') ③
    e("scrambled")

    def toast(self, msg):
        print("toast!", msg)

    setattr(Spam, 'eggs', toast) ④

    s.eggs("buttered!")

    delattr(Spam, 'eggs') ⑤

    try:
        s.eggs("shirred")
    except AttributeError as err: ⑥
        print(err)
```

① create attribute

② check whether attribute exists

③ retrieve attribute

④ set (or overwrite) attribute

- ⑤ remove attribute
- ⑥ missing attribute raises error

attributes.py

```
eggs! fried
hasattr() True
eggs! scrambled
toast! buttered!
'Spam' object has no attribute 'eggs'
```

Adding instance methods

- Use `setattr()`
- Add instance method to class
- Add instance method to instance

Using `setattr()`, it is easy to add instance methods to classes. Just add a function object to the class. Because it is part of the class itself, it will automatically be bound to the instance. Remember that an instance method expects `self` as the first parameter. In fact, this is the meaning of a bound instance—it is "bound" to the instance, and therefore when called, it is passed the instance as the first parameter.

Once added, the method may be called from any existing *or new* instance of the class.

To add an instance method to an *instance* takes a little more effort. Because it's not being added to the class, it is not automatically bound. The function needs to know what instance it should be bound to. This can be accomplished with the `types.MethodType` function.

Pass the function and the instance to the `MethodType()` function.

Example

adding_instance_methods.py

```
#!/usr/bin/env python
from types import MethodType

class Dog(): ①
    pass

d1 = Dog() ②

def bark(self): ③
    print("Woof! woof!")

setattr(Dog, "bark", bark) ④

d2 = Dog() ⑤

d1.bark() ⑥
d2.bark()

def wag(self): ⑦
    print("Wagging...")

setattr(d1, "wag", MethodType(wag, d1)) ⑧

d1.wag() ⑨
try:
    d2.wag() ⑩
except AttributeError as err:
    print(err)
```

- ① Define Dog type
- ② Create instance of Dog
- ③ Define (unbound) function
- ④ Add function to class (which binds it as an instance method)
- ⑤ Define another instance of Dog
- ⑥ New function can be called from either instance
- ⑦ Create another unbound function
- ⑧ Add function to instance after passing it through MethodType()
- ⑨ Call instance method
- ⑩ Instance method not available - only bound to d1

adding_instance_methods.py

```
Woof! woof!
Woof! woof!
Wagging...
'Dog' object has no attribute 'wag'
```

Decorators

- Classic design pattern
- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take parameters (but not required to)
- `functools.wraps()` preserves function's properties

In Python, many decorators are provided by the standard library, such as `property()` or `classmethod()`.

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the `@app.route()` decorator in Flask maps a URL to a view function.

As another example, `unittest` provides decorators to skip tests. A very common decorator is `@property`, which converts a class method into a property object.

A decorator can be any callable, which means it can be a normal function, a class method, or a class which implements the `__call__()` method (AKA callable class).

A simple decorator expects the item being decorated as its parameter, and returns a replacement. Typically, the replacement is a new function, but there is no restriction on what is returned. If the decorator itself needs parameters, then the decorator returns a wrapper function that expects the item being decorated, and then returns the replacement.

Applying decorators

- Use @ symbol
- Applied to *next* item only
- Multiple decorators OK

The @ sign is used to apply a decorator to a function or class. A decorator only applies to the next definition in the script.

The most important thing to know about the decorators is the following syntax:

```
@spam  
def ham():  
    pass
```

is exactly the same as

```
ham = spam(ham)
```

and

```
@spam(a, b, c)  
def ham():  
    pass
```

is exactly the same as

```
ham = spam(a, b, c)(ham)
```

Once you understand this, then creating decorators is just a matter of writing functions and having them return the appropriate thing.

Table 5. Decorators in the standard library

Decorator	Description
@abc.abstractmethod	Indicate abstract method (must be implemented).
@abc.abstractproperty	Indicate abstract property (must be implemented).
@asyncio.coroutine	Mark generator-based coroutine.
@atexit.register	Register function to be executed when interpreter (script) exits.
@classmethod	Indicate class method (receives class object, not instance object)
@contextlib.contextmanager	Define factory function for with statement context managers (no need to create <code>_enter_()</code> and <code>_exit_()</code> methods)
@functools.lru_cache	Wrap a function with a memoizing callable
@functools.singledispatch	Transform function into a single-dispatch generic function.
@functools.total_ordering	Supply all other comparison methods if class defines at least one.
@functools.wraps	Invoke <code>update_wrapper()</code> so decorator's replacement function keeps original function's name and other properties.
@property	Indicate a class property.
@staticmethod	Indicate static method (passed neither instance nor class object).
@types.coroutine	Transform generator function into a coroutine function.
@unittest.mock.patch	Patch target with a new object. When the function/with statement exits patch is undone.
@unittest.mock.patch.dict	Patch dictionary (or dictionary-like object), then restore to original state after test.

Decorator	Description
@unittest.mock.patch.multiple	Perform multiple patches in one call.
@unittest.mock.patch.object	Patch object attribute with mock object.
@unittest.skip()	Skip test unconditionally
@unittest.skipIf()	Skip test if condition is true
@unittest.skipUnless()	Skip test unless condition is true
@unittest.expectedFailure()	Mark Test as expected failure
@unittest.removeHandler()	Remove Control-C handler

Trivial Decorator

- Decorator can return anything
- Not very useful, usually

A decorator does not have to be elaborate. It can return anything, though typically decorators return the same type of object they are decorating.

In this example, the decorator returns the integer value 42. This is not particularly useful, but illustrates that the decorator always replaces the object being decorated with *something*.

Example

`deco_trivial.py`

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

def void(old_function):
    return 42 ①

name = "Guido"
x = void(name)

@void ②
def hello():
    print("Hello, world")

print(hello, type(hello)) ③
print(x, type(x))
```

① replace function with 42

② decorate hello() function

③ hello is now the integer 42, not a function

`deco_trivial.py`

```
42 <class 'int'>
42 <class 'int'>
```

Decorator functions

- Provide a wrapper around a function
- Purposes
 - Add functionality
 - Register
 - ??? (open-ended)
- Optional parameters

A decorator function acts as a wrapper around some object (usually function or class). It allows you to add features to a function without changing the function itself. For instance, the `@property`, `@classmethod`, and `@staticmethod` decorators are used in classes.

A decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code.

The new function should be defined with generic arguments (`*args`, `**kwargs`) so it can handle the original function's arguments.

The `wraps` decorator from the `functools` module in the standard library should be used with the function that returns the replacement function. This makes sure the replacement function keeps the same properties (especially the name) as the original (target) function. Otherwise, the replacement function keeps all of its own attributes.

Example

deco_debug.py

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

from functools import wraps

def debugger(old_func): ①

    @wraps(old_func) ②
    def new_func(*args, **kwargs): ③
        print("*" * 40) ④
        print("** function", old_func.__name__, "**") ④

        if args: ④
            print("\targs are ", args)
        if kwargs: ④
            print("\tkwargs are ", kwargs)

        print("*" * 40) ④

    return old_func(*args, **kwargs) ⑤

    return new_func ⑥

@debugger ⑦
def hello(greeting, whom='world'):
    print("{} , {} ".format(greeting, whom))

hello('hello', 'world') ⑧
print()

hello('hi', 'Earth')
print()

hello('greetings')
```

① decorator function — expects decorated (original) function as a parameter

② @wraps preserves name of original function after decoration

- ③ replacement function; takes generic parameters
- ④ new functionality added by decorator
- ⑤ call the original function
- ⑥ return the new function object
- ⑦ apply the decorator to a function
- ⑧ call new function

deco_debug.py

```
*****
** function hello **
  args are ('hello', 'world')
*****
hello, world

*****
** function hello **
  args are ('hi', 'Earth')
*****
hi, Earth

*****
** function hello **
  args are ('greetings',)
*****
greetings, world
```

Decorator Classes

- Same purpose as decorator functions
- Two ways to implement
 - No parameters
 - Expects parameters
- Decorator can keep state

A class can also be used to implement a decorator. The advantage of using a class for a decorator is that a class can keep state, so that the replacement function can update information stored at the class level.

Implementation depends on whether the decorator needs parameters.

If the decorator does *not* need parameters, the class must implement two methods: `__init__()` is passed the original function, and can perform any setup needed. The `__call__` method *replaces* the original function. In other word, after the function is decorated, calling the function is the same as calling `CLASS.__call__()`.

If the decorator *does* need parameters, `__init__.py` is passed the parameters, and `__call__()` is passed the original function, and must *return* the replacement function.

A good use for a decorator class is to log how many times a function has been called, or even keep track of the arguments it is called with (see example for this).

Example

deco_debug_class.py

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

class debugger(): ①

    function_calls = []

    def __init__(self, func): ②
        self._func = func

    def __call__(self, *args, **kwargs): ③

        # print("*" * 40) ④
        # print("function {}".format(self._func.__name__)) ④
        # print("\targs are ", args) ④
        # print("\tkwargs are ", kwargs) ④
        #
        # print("*" * 40) ④

        self.function_calls.append( ⑤
            (self._func.__name__, args, kwargs)
        )

        result = self._func(*args, **kwargs) ⑥
        return result ⑦

    @classmethod
    def get_calls(cls): ⑧
        return cls.function_calls

@debugger ⑨
def hello(greeting, whom="world"):
    print("{}{}, {}".format(greeting, whom))

@debugger ⑨
def bark(bark_word, *, repeat=2):
    print("{}! ".format(bark_word) * repeat)

hello('hello', 'world') ⑩
print()
```

```
hello('hi', 'Earth')
print()

hello('greetings')

bark("woof", repeat=3)
bark("yip", repeat=4)
bark("arf")

hello('hey', 'girl')

print('-' * 60)

for i, info in enumerate(debugger.get_calls(), 1): ⑪
    print("{:2d}. {:10s} {!s:20s} {!s:20s}".format(i, info[0], info[1], info[2]))
```

- ① class implementing decorator
- ② original function passed into decorator's constructor
- ③ *call()* is replacement function
- ④ add useful features to original function
- ⑤ add function name and arguments to saved list
- ⑥ call the original function
- ⑦ return result of calling original function
- ⑧ define method to get saved function call information
- ⑨ apply debugger to function
- ⑩ call replacement function
- ⑪ display function call info from class

deco_debug_class.py

```
hello, world  
hi, Earth  
  
greetings, world  
woof! woof! woof!  
yip! yip! yip! yip!  
arf! arf!  
hey, girl  
-----  
1. hello      ('hello', 'world')    {}  
2. hello      ('hi', 'Earth')     {}  
3. hello      ('greetings',)     {}  
4. bark       ('woof',)          {'repeat': 3}  
5. bark       ('yip',)           {'repeat': 4}  
6. bark       ('arf',)           {}  
7. hello      ('hey', 'girl')    {}
```

Decorator parameters

- Decorator functions require two nested functions
- Method `__call__()` returns replacement function in classes

A decorator can be passed parameters. This requires a little extra work.

For decorators implemented as functions, the decorator itself is passed the parameters; it contains a nested function that is passed the decorated function (the target), and it returns the replacement function.

For decorators implemented as classes, `init` is passed the parameters, `__call__()` is passed the decorated function (the target), and `__call__` returns the replacement function.

There are many combinations of decorators (8 total, to be exact). This is because decorators can be implemented as either functions or classes, they may take parameters, or not, and they can decorate either functions or classes. For an example of all 8 approaches, see the file **decorama.py** in the EXAMPLES folder.

Example

deco_params.py

```
#!/usr/bin/env python
#
from functools import wraps ①

def multiply(multiplier): ②

    def deco(old_func): ③

        @wraps(old_func) ④
        def new_func(*args, **kwargs): ⑤
            result = old_func(*args, **kwargs) ⑥
            return result * multiplier ⑦

        return new_func ⑧

    return deco ⑨

@multiply(4)
def spam():
    return 5

@multiply(10)
def ham():
    return 8

a = spam()
b = ham()
print(a, b)
```

① wrapper to preserve properties of original function

② actual decorator — receives decorator parameters

③ "inner decorator" — receives function being decorated

④ retain name, etc. of original function

- ⑤ replacement function — this is called instead of original
- ⑥ call original function and get return value
- ⑦ multiple result of original function by multiplier
- ⑧ deco() returns new_function
- ⑨ multiply returns deco

deco_params.py

```
20 80
```

Creating classes at runtime

- Use the `type()` function
- Provide dictionary of attributes

A class can be created programmatically, without the use of the `class` statement. The syntax is

```
type("name", (base_class, ...), {attributes})
```

The first argument is the name of the class, the second is a tuple of base classes (use `object` if you are not inheriting from a specific class), and the third is a dictionary of the class's attributes.

NOTE Instead of `type`, any other *metaclass* can be used.

Example

creating_classes.py

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

def function_1(self): ①
    print("Hello from f1()")

def function_2(self): ①
    print("Hello from f2()")

NewClass = type("NewClass", (), { ②
    'hello1': function_1,
    'hello2': function_2,
    'color': 'red',
    'state': 'Ohio',
})

n1 = NewClass() ③

n1.hello1() ④
n1.hello2()
print(n1.color) ⑤
print()

SubClass = type("SubClass", (NewClass,), {'fruit': 'banana'}) ⑥
s1 = SubClass() ⑦
s1.hello1() ⑧
print(s1.color) ⑨
print(s1.fruit)
```

① create method (not inside a class — could be a lambda)

② create class using type() — parameters are class name, base classes, dictionary of attributes

③ create instance of new class

④ call instance method

⑤ access class data

⑥ create subclass of first class

- ⑦ create instance of subclass
- ⑧ call method on subclass
- ⑨ access class data

creating_classes.py

```
Hello from f1()  
Hello from f2()  
red
```

```
Hello from f1()  
red  
banana
```

Monkey Patching

- Modify existing class or object
- Useful for enabling/disabling behavior
- Can cause problems

"Monkey patching" refers to technique of changing the behavior of an object by adding, replacing, or deleting attributes from outside the object's class definition.

It can be used for:

- Replacing methods, attributes, or functions
- Modifying a third-party object for which you do not have access
- Adding behavior to objects in memory

If you are not careful when creating monkey patches, some hard-to-debug problems can arise

- If the object being patched changes after a software upgrade, the monkey patch can fail in unexpected ways.
- Conflicts may occur if two different modules monkey-patch the same object.
- Users of a monkey-patched object may not realize which behavior is original and which comes from the monkey patch.

Monkey patching defeats object encapsulation, and so should be used sparingly.

Decorators are a convenient way to monkey-patch a class. The decorator can just add a method to the decorated class.

Example

meta_monkey.py

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

class Spam(): ①

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

    def eggs(self): ②
        print("Good morning, {}".format(self._name))

    def scrambled(self): ⑤
        print("Hello, {}. Enjoy your scrambled eggs".format(self._name))

setattr(Spam, "eggs", scrambled) ⑥

s.eggs() ⑦
```

- ① create normal class
- ② add normal method
- ③ create instance of class
- ④ call method
- ⑤ define new method outside of class
- ⑥ monkey patch the class with the new method
- ⑦ call the monkey-patched method from the instance

meta_monkey.py

Good morning, Mrs. Higgenbotham. Here are your delicious fried eggs.
Hello, Mrs. Higgenbotham. Enjoy your scrambled eggs

Do you need a Metaclass?

- YAGNI (You ain't gonna need it) (probably)
- Deep magic
- Used in frameworks such as Django

Before we cover the details of metaclasses, a disclaimer: you will probably never need to use a metaclass. When you think you might need a metaclass, consider using inheritance or a class decorator. However, metaclasses may be a more elegant approach to certain kinds of tasks, such as registering classes when they are defined.

There are two use cases where metaclasses are always an appropriate solution, because they must be done before the class is created:

- Modifying the class name
- Modifying the the list of base classes.

Several popular frameworks use metaclasses, Django in particular. In Django they are used for models, forms, form fields, form widgets, and admin media.

Remember that metaclasses can be a more elegant way to accomplish things that can also be done with inheritance, composition, decorators, and other techniques that are less "magic".

About metaclasses

- Metaclass:Class::Class:Object

Just as a class is used to create an instance, a *metaclass* is used to create a class.

The primary reason for a metaclass is to provide extra functionality at *class creation* time, not *instance creation* time. Just as a class can share state and actions across many instances, a metaclass can share (or provide) data and state across many *classes*.

The metaclass might modify the list of base classes, or register the class for later retrieval.

The builtin metaclass that Python provides is **type**.

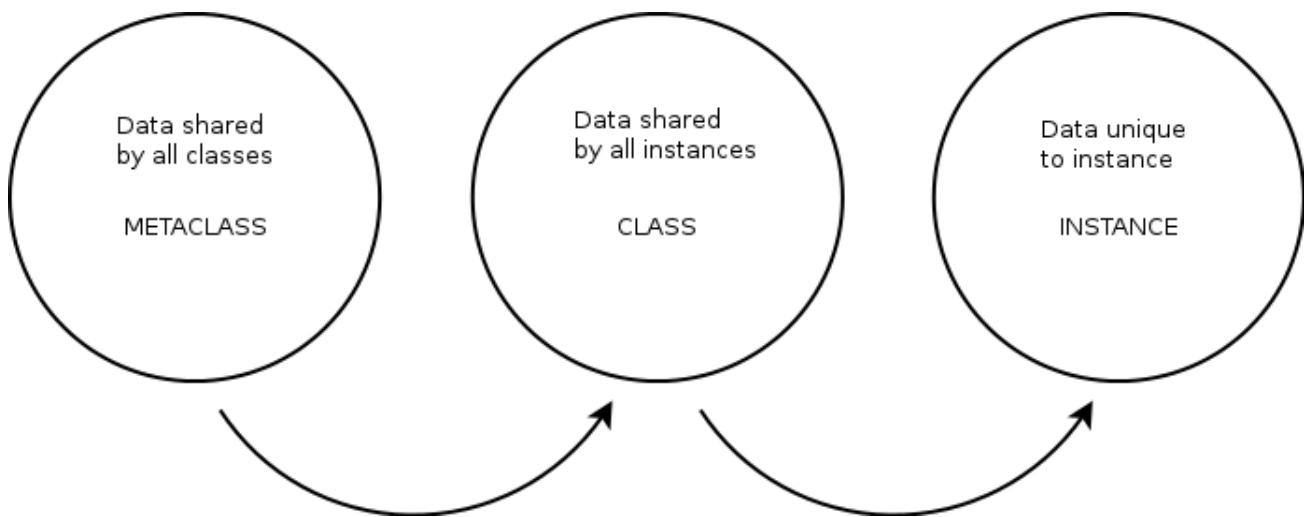
As we saw earlier ,you can create a class from a metaclass by passing in the new class's name, a tuple of base classes (which can be empty), and a dictionary of class attributes (which also can be empty).

```
class Spam(Ham):  
    id = 1
```

is exactly equivalent to

```
Spam = type('Spam', (Ham,), {"id": 1})
```

Replacing "type" with the name of any other metaclass works the same.



Mechanics of a metaclass

- Like normal class
- *Should* implement `__new__`
- *Can* implement
 - `__init__`
 - `__prepare__`
 - `__call__`

To create a metaclass, define a normal class. Most metaclasses implement the `__new__` method. This method is called with the type, name, base classes, and attribute dictionary (if any) of the new class. It should return a new class, typically using `super().__new__()`, which is very similar to how normal classes create instances. This is one place you can modify the class being created. You can add or change attributes, methods, or properties.

For instance, the Django framework uses metaclasses for Models. When you create an instance of a Model, the metaclass code automatically creates methods for the fields in the model. This is called "declarative programming", and is also used in SQLAlchemy's declarative model, in a way pretty similar to Django.

When you execute the following code:

```
class SomeClass(metaclass=SomeMeta):  
    pass
```

`META(name, bases, attrs)` is executed, where `META` is the metaclass (normally `type()`). Then,

1. The `__prepare__` method of the metaclass is called
2. The `__new__` method of the metaclass is called
3. The `__init__` method of the metaclass is called.

Next, after the following code runs:

```
obj = SomeClass()
```

`SomeMeta.call()` is called. It returns whatever `SomeMeta.__new__()` returned.

Example

metaclass_generic.py

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

class Meta(type):

    def __prepare__(class_name, bases):
        """
        "Prepare" the new class. Here you can update the base classes.

        :param name: Name of new class as a string
        :param bases: Tuple of base classes
        :return: Dictionary that initializes the namespace for the new class (must
be a dict)
        """
        print("in metaclass (class={}) __prepare__().format(class_name), end=' ==> '
')
        print("params: name={}, bases={}".format(class_name, bases))
        return {'animal': 'wombat', 'id': 100}

    def __new__(metatype, name, bases, attrs):
        """
        Create the new class. Called after __prepare__(). Note this is only called
when classes

        :param metatype: The metaclass itself
        :param name: The name of the class being created
        :param bases: bases of class being created (may be empty)
        :param attrs: Initial attributes of the class being created
        :return:
        """
        print("in metaclass (class={}) __new__().format(name), end=' ==> ')
        print("params: type={} name={} bases={} attrs={}".format(metatype, name,
bases, attrs))
        return super().__new__(metatype, name, bases, attrs)

    def __init__(cls, *args):
        """
        :param cls: The class being created (compare with 'self' in normal class)
        :param args: Any arguments to the class
        """

```

```
    print("in metaclass (class={}) __init__().format(cls.__name__), end=' ==>')
')
print("params: cls={}, args={}".format(cls, args))

super().__init__(cls)

def __call__(self, *args, **kwargs):
"""
    Function called when the metaclass is called, as in NewClass = Meta(...)

:param args:
:param args:
:param kwargs:
:return:
"""

print("in metaclass (class={})__call__().format(self.__name__))"

class MyBase():
    pass

print('=' * 60)

class A(MyBase, metaclass=Meta):
    id = 5

    def __init__(self):
        print("In class A __init__()")

print('=' * 60)

class B(MyBase, metaclass=Meta):
    animal = 'wombat'

    def __init__(self):
        print("In class B __init__()")

print('-' * 60)
m1 = A()
print('-' * 60)
m2 = B()
print('-' * 60)
m3 = A()
```

```

print('-' * 60)
m4 = B()
print('-' * 60)
print("animal: {} id: {}".format(A.animal, B.id))

```

metaclass_generic.py

```

=====
in metaclass (class=A) __prepare__() ==> params: name=A, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=A) __new__() ==> params: type=<class '__main__.Meta'> name=A
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 5, '__module__':
'__main__', '__qualname__': 'A', '__init__': <function A.__init__ at 0x100e5f0d0>}
in metaclass (class=A) __init__() ==> params: cls=<class '__main__.A'>, args=('A',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 5, '__module__':
'__main__', '__qualname__': 'A', '__init__': <function A.__init__ at 0x100e5f0d0>})
=====

in metaclass (class=B) __prepare__() ==> params: name=B, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=B) __new__() ==> params: type=<class '__main__.Meta'> name=B
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 100,
['__module__': '__main__', '__qualname__': 'B', '__init__': <function B.__init__ at
0x100e5f158>]
in metaclass (class=B) __init__() ==> params: cls=<class '__main__.B'>, args=('B',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 100, '__module__':
'__main__', '__qualname__': 'B', '__init__': <function B.__init__ at 0x100e5f158>})
=====

in metaclass (class=A)__call__()

-----
in metaclass (class=B)__call__()

-----
in metaclass (class=A)__call__()

-----
in metaclass (class=B)__call__()

-----
animal: wombat id: 100

```

__prepare__(){empty __new__(){empty __init__(){empty __call__(){empty

Singleton with a metaclass

- Classic example
- Simple to implement
- Works with inheritance

One of the classic use cases for a metaclass in Python is to create a *singleton* class. A singleton is a class that only has one actual instance, no matter how many times it is instantiated. Singletons are used for loggers, config data, and database connections, for instance.

To create a single, implement a metaclass by defining a class that inherits from `type`. The class should have a class-level dictionary to store each class's instance. When a new instance of a class is created, check to see if that class already has an instance. If it does not, call `__call__` to create the new instance, and add the instance to the dictionary.

In either case, then return the instance where the key is the class object.

Example

metaclass_singleton.py

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

class Singleton(type): ①
    _instances = {} ②

    def __new__(typ, *junk):
        # print("__new__()")
        return super().__new__(typ, *junk)

    def __call__(cls, *args, **kwargs): ③
        # print("__call__()")
        if cls not in cls._instances: ④
            cls._instances[cls] = super().__call__(*args, **kwargs) ⑤

        return cls._instances[cls] ⑥

class ThingA(metaclass=Singleton): ⑦
    def __init__(self, value):
        self.value = value

class ThingB(metaclass=Singleton): ⑦
    def __init__(self, value):
        self.value = value

ta1 = ThingA(1) ⑧
ta2 = ThingA(2)
ta3 = ThingA(3)

tb1 = ThingB(4)
tb2 = ThingB(5)
tb3 = ThingB(6)

for thing in ta1, ta2, ta3, tb1, tb2, tb3:
    print(type(thing).__name__, id(thing), thing.value) ⑨
```

① use type as base class of a metaclass

- ② dictionary to keep track of instances
- ③ *call* is passed the new class plus its parameters
- ④ check to see if the new class has already been instantiated
- ⑤ if not, create the (single) class instance and add to dictionary
- ⑥ return the (single) class instance
- ⑦ Define two different classes which use Singleton
- ⑧ Create instances of ThingA and ThingB
- ⑨ Print the type, name, and ID of each thing — only one instance is ever created for each class

metaclass_singleton.py

```
ThingA 4465711200 1
ThingA 4465711200 1
ThingA 4465711200 1
ThingB 4465711256 4
ThingB 4465711256 4
ThingB 4465711256 4
```

Chapter 8 Exercises

Exercise 8-1 (pres_attr.py)

Instantiate the President class. Get the first name, last name, and party attributes using getattr().

Exercise 8-2 (pres_monkey.py, pres_monkey_amb.py)

Monkey-patch the President class to add a method get_full_name which returns a single string consisting of the first name and the last name, separated by a space.

TIP Instead of a method, make full_name a property.

Exercise 8-3 (sillystring.py)

Without using the **class** statement, create a class named SillyString, which is initialized with any string. Include an instance method called every_other which returns every other character of the string.

Instantiate your string and print the result of calling the **every_other()** method. Your test code should look like this:

```
ss = SillyString('this is a test')
print(ss.every_other())
```

It should output

```
ti sats
```

Exercise 8-4 (doubledeco.py)

Write a decorator to double the return value of any function. If a function returns 5, after decoration it should return 10. If it returns "spam", after decoration it should return "spamsspam", etc.

Exercise 8-5 (word_actions.py)

Write a decorator, implemented as a class, to register functions that will process a list of words. The decorated functions will take one parameter—a string—and return the modified string.

The decorator itself takes two parameters—minimum length and maximum length. The class will store the min/max lengths as the key, and the functions as values, as class data.

The class will also provide a method named **process_words**, which will open **DATA/words.txt** and read it line by line. Each line contains a word.

For every registered function, if the length of the current word is within the min/max lengths, call all the functions whose key is that min/max pair.

In other words, if the registry key is (5, 8), and the value is [func1, func2], when the current word is within range, call func1(*w*) and func2(*w*), where *w* is the current word.

Example of class usage:

```
word_select = WordSelect() # create callable instance

@word_select(16, 18) # register function for length 16-18, inclusive
def make_upper(s):
    return s.upper()

word_select.process_words() # loop over words, call functions if selected
```

Suggested functions to decorate:

- make the word upper-case
- put stars before or around the word
- reverse the word

Remember all the decorated functions take one argument, which is one of the strings in the word list, and return the modified word.

Chapter 9: Developer Tools

Objectives

- Run pylint to check source code
- Debug scripts
- Find speed bottlenecks in code
- Compare algorithms to see which is faster

Program development

- More than just coding
 - Design first
 - Consistent style
 - Comments
 - Debugging
 - Testing
 - Documentation

Comments

- Keep comments up-to-date
- Use complete sentences
- Block comments describe a section of code
- Inline comments describe a line
- Don't state the obvious

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement; they should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1      # Increment x
```

Only use an inline comment if the reason for the statement is not obvious:

```
x = x + 1      # Add one so range() does the right thing
```

The above was adapted from PEP 8

pylint

- Checks many aspects of code
- Finds mistakes
- Rates your code for standards compliance
- Don't worry if your code has a lower rating!
- Can be highly customized

pylint is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells (as defined in Martin Fowler's Refactoring book)

from the pylint documentation

pylint can be very helpful in identifying errors and pointing out where your code does not follow standard coding conventions. It was developed by Python coders at Logilab <http://www.logilab.fr>.

It has very verbose output, which can be modified via command line options.

pylint can be customized to reflect local coding conventions. To use pylint, just say pylint filename, or pylint directory. Most Python IDEs have pylint, or the equivalent, built in.

Other tools for analyzing Python code: * pyflakes * pychecker

Customizing pylint

- Use `pylint --generate-rcfile`
- Redirect to file
- Edit as needed
- Knowledge of regular expressions useful
- Name file `\~/pylintrc` on Linux/Unix/OS X
- Use `-rcfile` file to specify custom file on Windows

To customize pylint, run pylint with only the `-generate-rcfile` option. This will output a well-commented configuration file to STDOUT, so redirect it to a file.

Edit the file as needed. The comments describe what each part does. You can change the allowed names of variables, functions, classes, and pretty much everything else. You can even change the rating algorithm.

Windows

Put the file in a convenient location (name it something like `pylintrc`). Invoke pylint with the `-rcfile` option to specify the location of the file.

pylint will also find a file named `pylintrc` in the current directory, without needing the `-rcfile` option.

Non-Windows systems

On Unix-like systems (Unix, Mac OS, Linux, etc.), `/etc/pylintrc` and `\~/pylintrc` will be automatically loaded, in that order.

See docs.pylint.org for more details.

Using pyreverse

- Source analyzer
- Reverse engineers Python code
- Part of pylint
- Generates UML diagrams

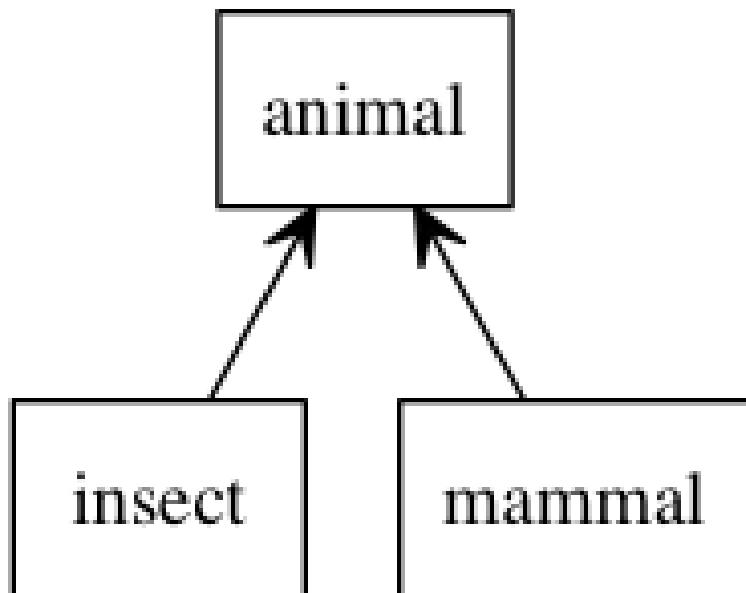
pyreverse is a Python source code analyzer. It reads a script, and the modules it depends on, and generates UML diagrams. It is installed as part of the pylint package.

There are many options to control what it analyzes and what kind of output it produces.

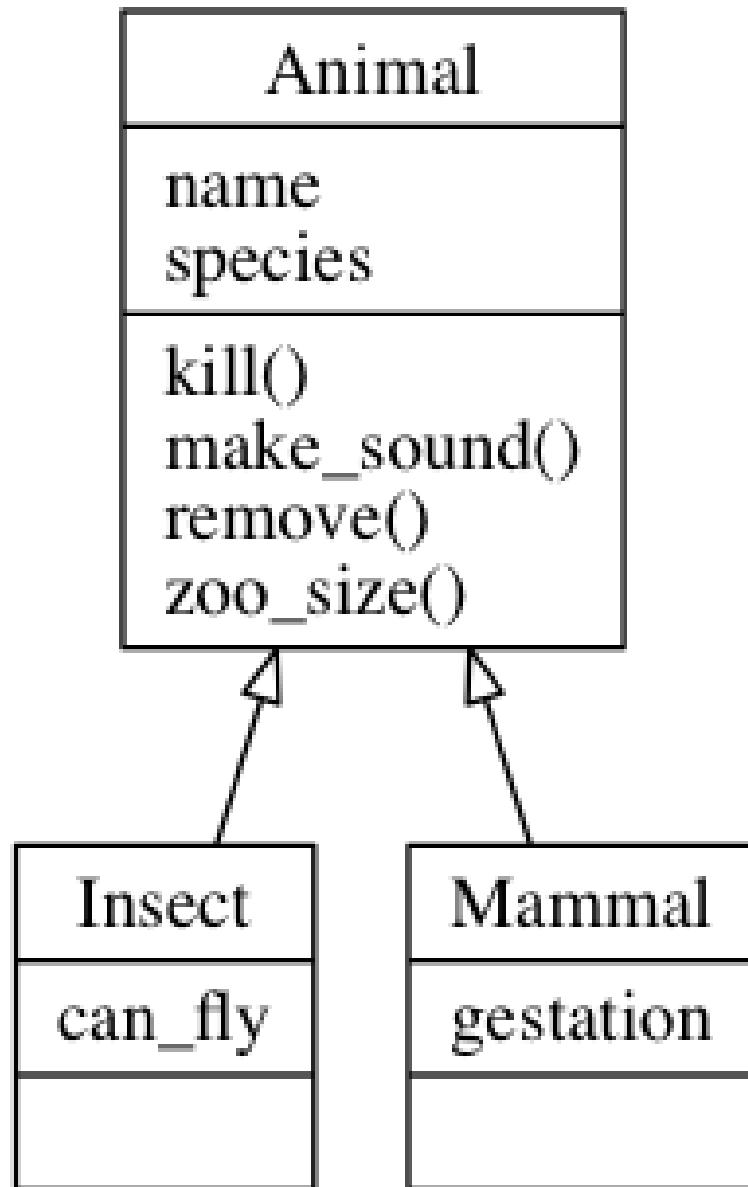
Example

```
pyreverse -o png -p MyProject -A animal.py mammal.py insect.py
```

`packages_MyProject.png`



classes_MyProject.png

**NOTE**

[pyreverse](#) requires the (non-Python) Graphviz utility to be installed.

The Python debugger

- Implemented via pdb module
- Supports breakpoints and single stepping
- Based on gdb

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The pdb module provides debugging facilities for Python.

The usual way to use pdb is from the command line:

```
python -m pdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

Starting debug mode

- Syntax

```
python -m pdb script
```

or

```
import pdb
pdb.run('function')
```

pdb is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import some_module
>>> pdb.run('some_module.function_to_text()')
> <string>(0)?()
(Pdb) c    # (c)ontinue
> <string>(1)?()
(Pdb) c    # (c)ontinue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

To get help, type **h** at the debugger prompt.

Stepping through a program

- **s** single-step, stepping into functions
- **n** single-step, stepping over functions
- **r** return from function
- **c** run to next breakpoint or end

The debugger provides several commands for stepping through a program. Use **s** to step through one line at a time, stepping into functions.

Use **n** to step over functions; use **r** to return from a function; use **c** to continue to next breakpoint or end of program.

Pressing Enter repeats most commands; if the previous command was list, the debugger lists the next set of lines.

Setting breakpoints

- Syntax

```
b list all breakpoints  
b linenumber (, condition)  
b file:linenumber (, condition)  
b function name (, condition)
```

Breakpoints can be set with the `b` command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The `tbreak` command creates a one-time breakpoint that is deleted after it is hit the first time.

Profiling

- Use the **profile** module from the command line
- Shows where program spends the most time
- Output can be tweaked via options

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

To use the profiler, execute the following at the command line:

```
python -m profile scriptname.py
```

This will output a simple report to STDOUT. You can also specify an output file with the `-o` option, and the sort order with the `-s` option. See the docs for more information.

TIP

The [pycallgraph](#) module (not in the standard library) will create a graphical representation of an application's profile, indicating visually where the application is spending the most time.

Example

```
python -m profile count_with_dict.py
...script output...
    19 function calls in 0.000 seconds

Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      14    0.000    0.000    0.000    0.000 :0(get)
       1    0.000    0.000    0.000    0.000 :0(items)
       1    0.000    0.000    0.000    0.000 :0(open)
       1    0.000    0.000    0.000    0.000 :0(setprofile)
       1    0.000    0.000    0.000    0.000 count_with_dict.py:3(<module>)
       1    0.000    0.000    0.000    0.000 profile:0(<code object <module> at
0xb74c36e0, file "count_with_dict.py", line 3>)
       0    0.000        0.000            profile:0(profiler)
```

Benchmarking

- Use the `timeit` module
- Create a `timer` object with specified # of repetitions

Use the `timeit` module to benchmark two or more code snippets. To time code, create a `Timer` object, which takes two strings of code. The first is the code to test; the second is setup code, that is only run once per timer .

Call the `timeit()` method with the number of times to call the test code, or call the `repeat()` method which repeats `timeit()` a specified number of times.

You can also use the `timeit` module from the command line. Use the `-s` option to specify startup code:

```
python -m timeit -s "startup code..." "code..."
```

Example

bm_range_vs_while.py

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

import timeit

setup_code = 'values = []' ①

test_code_one = '''
for i in range(10000):
    values.append(i)
''' ②
test_code_two = '''
i = 0
while i < 10000:
    values.append(i)
    i += 1
''' ②

t1 = timeit.Timer(test_code_one, setup_code) ③
t2 = timeit.Timer(test_code_two, setup_code) ③

print("test one:")
print(t1.timeit(1000)) ④
print()

print("test two:")
print(t2.timeit(1000)) ④
print()
```

- ① setup code is only executed once
- ② code fragment executed many times
- ③ Timer object creates time-able code
- ④ timeit() runs code fragment N times

bm_range_vs_while.py

```
test one:  
0.948215389
```

```
test two:  
1.285203622
```

Chapter 9 Exercises

Exercise 9-1

Pick several of your scripts (from class, or from real life) and run pylint on them.

Exercise 9-2

Use any available debugger to step through any of the scripts you have written so far.

Chapter 10: 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 or 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
```

① function to launch in each thread

② create thread

③ launch thread

thr_noclass.py

```
Hello from thread 3  
Hello from thread 4  
Hello from thread 0  
Hello from thread 6  
Hello from thread 9  
Hello from thread 5  
Hello from thread 7  
Hello from thread 1  
Hello from thread 8  
Hello from thread 2
```

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 6  
Hello from thread 5  
Hello from thread 2  
Hello from thread 7  
Hello from thread 0  
Hello from thread 8  
Hello from thread 9  
Hello from thread 1  
Hello from thread 3  
Hello from thread 4
```

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 3 (mango)
Hello from thread 5 (papaya)
Hello from thread 7 (lemon)
Hello from thread 9 (fig)
Hello from thread 2 (banana)
Hello from thread 4 (peach)
Hello from thread 8 (watermelon)
Hello from thread 6 (cherry)
Hello from thread 10 (elderberry)
['APPLE', 'MANGO', 'PAPAYA', 'LEMON', 'FIG', 'BANANA', 'PEACH', 'WATERMELON',
'CHERRY', 'ELDERBERRY']
```

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

```
[ 'CONFIGURATIVE-CONFIGURATIVE', 'HIGHBINDER-HIGHBINDER', 'PLUMIEST-PLUMIEST',  
'CROSSRUFFED-CROSSRUFFED', 'PONTIFICATE-PONTIFICATE', 'PRICKY-PRICKY', 'ETHNICS-  
ETHNICS', 'ILLEGIBILITY-ILLEGIBILITY', 'QUADDING-QUADDING', 'MULTINUCLATE-  
MULTINUCLATE', 'SCRUBBABLE-SCRUBBABLE', 'SCRAIGHING-SCRAIGHING', 'INSTIGATIVE-  
INSTIGATIVE', 'MISSOUT-MISSOUT', 'ONTOLOGICALLY-ONTOLOGICALLY', 'PARTICULARIZES-  
PARTICULARIZES', 'PULSANT-PULSANT', 'COLEOPTEROUS-COLEOPTEROUS', 'DECLAMATIONS-  
DECLAMATIONS', 'FUMITORIES-FUMITORIES' ]
```

Thu Feb 20 06:37:34 2020

Thu Feb 20 06:37:34 2020

Debugging threaded Programs

- Harder than non-threaded programs
- Context changes abruptly
- Use `pdb.set_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() ⑯

random_word = RandomWord() ⑰
for i in range(NUM_ITEMS):
    w = random_word()
    word_queue.put(w) ⑱

start_time = time.ctime()

pool = [] ⑲
for i in range(POOL_SIZE): ⑳
    worker_name = "Worker {:03d}".format(i)
    w = Worker(worker_name, word_queue, result_lock, shared_result) ㉑
    #
    w.start() ㉒
    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

```
[ 'FORERUNS', 'PETIOLED', 'EPINASTIES', 'SLAVOCRACIES', 'RHEOLOGIST', 'LANGUISHMENT',
  'QUIBBLER', 'MASTERFUL', 'OUTFAST', 'AWOLS', 'SHIVERING', 'UNRIPER', 'HISTRIONICS',
  'QUERULOUSNESSES', 'ABSQUATULATED', 'ATROPHYING', 'MAULING', 'TALLOWY',
  'REINVESTIGATE', 'COCKNEYNS', 'WAGONAGE', 'TRIMMED', 'MARASMIC', 'PERIPATUS',
  'FAWNINGLY', 'FILL', 'RUBRICAL', 'ZOANTHARIAN', 'REEDIER', 'PARTICIPATION',
  'CHAMISOS', 'INCONSOLABLE', 'SHRIMPIEST', 'STRIPPERS', 'FIDDLEBACKS', 'OUTHEARD',
  'APPARELLING', 'COMPLEMENT', 'FINDS', 'PREAPPROVED', 'TELOPHASES', 'DUALIZED',
  'REGINA', 'NONSTATISTICAL', 'STALAGMITES', 'INTHRONES', 'MALIGNING', 'PATRONAGE',
  'EXPRESSIVITY', 'BEDIGHT']
```

25000

Thu Feb 20 06:37:34 2020

Thu Feb 20 06:37:37 2020

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

```
['EPITASES', 'UNARTFUL', 'KIFS', 'RIVERFRONTS', 'MICROMETEOROLOGY', 'AIMLESS',  
 'SCATTERBRAINS', 'REWEDDING', 'BINDINGLY', 'YUANS', 'BAYONETING', 'ICONOCLASTS',  
 'REDEYE', 'TRANSPOSING', 'SCHIZZIEST', 'ENDEMICS', 'WELDED', 'SLEEKEN', 'DEFECT',  
 'DROMONDS']
```

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

```
['TRANSVALUATIONS', 'GERMANIUMS', 'MHO', 'DESCRIER', 'SUBINDUSTRIES',
'SPECTROMETRIC', 'DIVESTITURES', 'MISROUTING', 'OBSOLESCENCE', 'DEFLATIONS',
'IMMUTABLY', 'INFLEXIBILITIES', 'DISHERITED', 'ASTROPHYSICIST', 'WIMPS',
'RITZINESSES', 'DRAWING', 'DORONICUM', 'SKYLINE', 'BRUSHWOOD']
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 puts 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 twistedmatrix.com/trac.

Chapter 10 Exercises

For each exercise, ask the questions: Should this be multi-threaded or multi-processed? Distributed or local?

Exercise 10-1 (`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 10-2 (`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 files,
- how many lines,
- how many words, and
- how many bytes.

FOR ADVANCED STUDENTS

Exercise 10-3 (`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 10-4 (`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.

Chapter 11: Coroutines and asyncio

Objectives

- Understand Asynchronous Programming
- Learn when to use `async` and `await`
- Easily parallelize I/O with coroutines

Asynchronous programming with `asyncio`

- Provides easy concurrency primitives
- Doesn't block for I/O
- Makes programs complete faster

`asyncio` adds easier concurrent programming to Python. Like threads, asynchronous code can take advantage of the time while I/O processes are blocking. While the GIL does this transparently in multi-threaded programs, using `asyncio` primitives allows the programmer a better understanding and control of program flow.

In synchronous (normal) programming, an I/O call (reading from a file, a web site, printing, etc.) will wait until the call returns, known as **blocking**. In asynchronous programming, when the code is waiting for I/O or some other long-running task, it will manually return control to an event loop so other code can run.

Why not just use threads? Asynchronous programming, while complex to understand, can greatly simplify concurrent code.

Key asynchronous words

- Event loop
- Coroutine
- Task
- Future
- `async`
- `await`

An **event loop** is what controls the asynchronous functions. When a function needs something, it makes an I/O call. If the data are not ready, it returns control to the loop so that some other function can run. The event loop is like a miniature multi-threaded pool.

Coroutine is short for "cooperative routine". Coroutines are functions that can cooperate by sharing the event loop. A coroutine is a more general form of a function. A normal function "blocks", in that it starts at the beginning, and runs until it returns. A coroutine can "pause" its execution at various points and yield control to some *other* piece of code, rather than running to completion. Generators in Python are a stepping stone between a basic function and a full-blown coroutine.

A **Future** is an object that will eventually (in the future) have a result. Futures are generally not explicit in application code; they are for lower-level library code.

A **Task** is a Future wrapping a coroutine, specifically. The event loop does not run coroutines directly—it runs tasks.

Defining coroutines

A *Coroutine* is a function defined with the **async** modifier keyword.

```
import asyncio

async def lazy_print(msg):
    print(msg)

if __name__ == '__main__':
    asyncio.run(lazy_print('Hello World'))
```

Any coroutine cannot be run by calling it directly. Instead, it must be run through the **asyncio** framework. The **run()** function will handle running the coroutine, it fires up the event loop and places the code as a **Task** inside the event loop.

When a coroutine is called, it doesn't call the function, but creates and returns a *Coroutine object*.

Compare this with creating a generator function. When a function contains **yield**, calling the function creates and returns a generator object; it does not execute the function. Calling **next()** on the object is what actually runs code. Similarly, using a coroutine object in an **await** expression actually runs the code.

The coroutine must be invoked by one of the event loop's methods.

TIP Use **asyncio.iscoroutine()** to check whether an object is a coroutine.

TIP Use **asyncio.iscoroutinefunction()** to check whether a function is a coroutine.

The Event Loop

All coroutines must be executed within an *event loop*. By default, such an event loop is exposed in the `asyncio` module via `.get_event_loop()`. While it is possible to create custom loop behavior or driver code, it is generally easiest to use the builtin event loop. Only one event loop can be running in a given thread.

```
import asyncio

loop = asyncio.get_event_loop()

loop.run_until_complete(lazy_print('Hello World'))
```

Once the loop is created, coroutines may be added to it to run at some later time. There are a number of such methods to facilitate this.

`.create_task(coro)`

Add the coroutine to the loop, to be run later.

`.run_forever()`

Run the event loop.

`.run_until_complete(coro)`

Run the event loop immediately, suspending when the coroutine is complete.

`.wait(coros)`

Create one coroutine that iterates through the argument list of coroutines, effectively combining them into a single coroutine object.

In addition to `loop.create_task()`, there is a module-level `asyncio.create_task()` function that can be used to create and schedule tasks on the current thread's event loop.

Effective Coroutines

Truly effective coroutines are able to "give up" their ownership of a currently running thread and allow some other portion of work to progress. They do this by prefixing a function call with `await`. While that `awaited` function executes in the background, a different coroutine in the event loop may make progress. When the `awaited` function has finished, the coroutine will start again.

As such, this waiting is only effective when the function invoked is not CPU-bound. If the call is a `sleep`, or reading from a network socket, or publishing an event to an event queue, that call will be I/O-bound, no matter how fast the CPU is. These are prime targets for the `await` invocation. Only a coroutine declared with `async` may use `await` to control program flow.

```
import asyncio

async def serial_sleeper():
    print(1)
    await asyncio.sleep(2)
    await lazy_print('In Sleeper')
    await asyncio.sleep(1)
    print(2)

if __name__ == '__main__':
    asyncio.run(serial_sleeper())
```

Note that using `await` does not cause the current thread to execute out-of-order, it merely *allows* other asynchronous coroutines to start executing. The previous code will print 1, sleep for two seconds, print '`In Sleeper`', sleep for one second, then print 2. It executes these in that order because no other coroutines are scheduled to run in the event loop.

A function declared as `async` **must** be `awaited`. Merely calling the function returns a special object, similar to how a generator call returns the generator object. That special object, a **future**, must be `awaited`. Failing to do so results in a warning, letting the developer know that there is a programming error.

Scheduling multiple coroutines demonstrates much more clearly how asynchronous methods are scheduled by the event loop.

```
import asyncio
from random import randrange

async def sleeper(n):
    while True:
        timeout = randrange(10)
        print(f'[{n}] Sleeping for {timeout}s')
        await asyncio.sleep(timeout)
        print(f'[{n}] Done sleeping for {timeout}s')

if __name__ == '__main__':
    loop = asyncio.get_event_loop()

    loop.create_task(sleeper(1))
    loop.create_task(sleeper(2))
    loop.run_forever()
```

Tasks are added to the loop to be run, and the loop switches between the tasks when possible. While not being very interesting when it comes to mere sleep calls, the I/O multiplexing possibilities are limitless.

Awaitable objects

An object is *awaitable* if it can be used with the `await` keyword; that is, if it can be called asynchronously. This generally means a coroutine, Future, or Task.

There are a number of interfaces that work with awaitables, the specific instance is generally not relevant (and, due to duck typing, should not be relevant in the majority of cases).

The vast majority of awaitables will be either coroutines or event loop-level Task objects.

Futures

A `Future` is an object that will eventually hold the result of some code. That result may be an exception thrown by said code. It is possible to query the `Future` object to discover its current status, the result, or the exception.

`done()`

Indicates whether the `Future` has finished

`result()`

Returns the result or raises the exception from the code

`exception()`

Returns the exception from the code, or `None`

`add_done_callback()`

Add a callback to be executed when the `Future` is done

`cancel()`

Cancels the `Future`'s execution; starts callbacks

`cancelled()`

Indicates whether the `Future` was cancelled

Unfortunately, there are two incompatible types of `Future` objects in Python: `concurrent.futures.Future` (thread-safe) and `asyncio.futures.Future` (not thread-safe). In general, these objects should not be created directly, but by various helper functions.

A `Future` is an awaitable object, so other coroutines and asynchronous code can `await` them for a result, exception, or cancellation.

Futures are generally managed and manipulated by lower-level library code, rather than application code.

```
import asyncio
from random import randrange

async def monitor(fut):
    while not fut.done():
        print(f'Waiting for {fut}')
        await asyncio.sleep(1)
    print(f'{fut} complete')

async def long_process(name):
    print(f'Beginning {name}')
    timeout = randrange(2, 10)
    # Simulate a long asynchronous work item
    await asyncio.sleep(timeout)
    print(f'Ending {name}')

if __name__ == '__main__':
    loop = asyncio.get_event_loop()

    # Not preferred, see next section
    m1 = monitor(asyncio.ensure_future(long_process(1)))
    m2 = monitor(asyncio.ensure_future(long_process(2)))

    # Not preferred, see next section
    asyncio.ensure_future(m1)
    asyncio.ensure_future(m2)

    loop.run_forever()
```

The `ensure_future()` function will make sure that the object passed to it conforms to the `Future` interface, and automatically add it to an event loop. Once again, this is using the low-level `Future` object. Most code interaction, if any, should be done with `Task` objects.

Tasks

A `Task` is a subclass of `Future` that specifically wraps a coroutine. Because a `Task` wraps a coroutine, it has additional methods to extract information about its execution that a `Future` object does not have:

`get_stack`

Returns a list of stack frames for the current `Task`

`print_stack`

Prints stack frames for the current `Task`

A `Future` object should never be created by end-user code, only `Task` objects. The `create_task` method of the event loop can be used to wrap a coroutine in a task and add it to the loop.

```
import asyncio
from random import randrange

async def monitor(fut):
    while not fut.done():
        print(f'Waiting for {fut}')
        await asyncio.sleep(1)
    print(f'{fut} complete')

async def long_process(name):
    print(f'Beginning {name}')
    timeout = randrange(2, 10)
    # Simulate a long asynchronous work item
    await asyncio.sleep(timeout)
    print(f'Ending {name}')

if __name__ == '__main__':
    loop = asyncio.get_event_loop()

    m1 = monitor(loop.create_task(long_process(1)))
    m2 = monitor(loop.create_task(long_process(2)))
    loop.create_task(m1)
    loop.create_task(m2)

    loop.run_forever()
```

Without the loop having its own signal handler, it is not possible for normal signal handlers to affect the event loop. Each task managed by the event loop is captured in a `Task` object, which can be queried by the `all_tasks()` function.

When it comes to managing awaitable objects, it can be useful to wait for a group of them to complete. It is possible to group multiple asynchronous pieces of work using the `gather()` function in `asyncio`. This function creates a new awaitable object that simply `awaits` all the passed awaitables concurrently.

```
import asyncio
from random import randrange

async def long_process(name):
    print(f'Beginning {name}')
    timeout = randrange(2, 10)
    # Simulate a long asynchronous work item
    await asyncio.sleep(timeout)
    print(f'Ending {name}')
    return timeout * 1000

async def roll_up(times):
    jobs = [long_process(n) for n in range(times)]
    results = await asyncio.gather(*jobs)
    print(f'Slept a total of {sum(results)}ms across {times} tasks')

if __name__ == '__main__':
    asyncio.run(roll_up(10))
```

A `Task` object can be cancelled, and its results inspected. Doing so raises a `CancelledError` exception, which can be suppressed during normal cleanup. The `gather()` function also takes an optional keyword parameter `return_exceptions`, which will return `Exception` objects instead of raising them, which can be useful when dealing with multiple tasks.

Similar to `gather` is the `wait` function. The difference is that the `asyncio.wait()` function is capable of returning early, using its `return_when` keyword.

```
import asyncio
from random import randrange

async def maybe_throw(n):
    throws = randrange(10) < 1
    timeout = randrange(2, 10)

    print(f'[{n}] will throw in {timeout}s: {throws}')
    await asyncio.sleep(timeout)

    if throws:
        raise Exception('Throwing!')
    else:
        return True

async def upto_first_exception(times):
    jobs = [maybe_throw(n) for n in range(times)]

    done, pending = await asyncio.wait(jobs, return_when=asyncio.FIRST_EXCEPTION)

    print(f'Ran {len(done)} jobs, leaving {len(pending)}')

if __name__ == '__main__':
    asyncio.run(upto_first_exception(10))
```

Part of its utility is the ability to capture exceptions of awaitables as results rather than allowing the exception to continue bubbling up.

Exceptions

Dealing with `s` in asynchronous code is unintuitive at first. An unhandled exception can easily lead to a deadlock, yet the exception may be raised far from the code when it is scheduled. Further, the `KeyboardInterrupt` exception can produce very ugly tracebacks.

One difficult portion is making sure that all buffers are flushed, cleaned, and logged in an asynchronous application. Fortunately, the event loop itself has its own signal handlers. A specific signal can be used to control a graceful shutdown of the loop and therefore the application.

```
import asyncio
import signal

async def shutdown(loop):
    tasks = [t for t in asyncio.all_tasks(loop)
             if t is not asyncio.current_task(loop)]
    for task in tasks:
        task.cancel()

    # Allowing exceptions to bubble will cause the program to
    # hang forever, as nothing handles the exceptions nor stops
    # the loop
    await asyncio.gather(*tasks, return_exceptions=True)
    loop.stop()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()

    loop.add_signal_handler(signal.SIGINT,
                           asyncio.create_task, shutdown(loop))

    loop.create_task(sleeper(1))
    loop.create_task(sleeper(2))

    loop.run_forever()
```

Callbacks

It is possible to schedule functions to run at specific times, using the callback capability of the `asyncio` module. Note that callback functions are *not* coroutines! They execute synchronously!

```
import asyncio

loop = asyncio.get_event_loop()

loop.call_later(6, print, 0, 'BOOM')
loop.call_later(5, print, 1)
loop.call_later(4, print, 2)
loop.call_later(3, print, 3)
loop.call_soon(print, 'Starting countdown!')

loop.run_forever()
```

A callback may be invoked after a specific delay with `call_later` or as soon as possible with `call_soon`, or even at a specific time with `call_at`. The function in question will be invoked with the specified parameters.

Generally, global callbacks can and should be avoided in favor of awaitables. Individual `Future` objects might have some few callbacks associated with them.

How to run coroutines

- Ways to schedule
 - `loop.create_task(coro)`
 - `asyncio.create_task(coro)`
- Wrap task list with `asyncio.wait()`
 - Pass result to `loop.run_until_complete()`

There are multiple ways to run coroutines; generally involving methods from the `asyncio` module.

The simplest way is to use the `create_task()` method of the event loop to create one or more tasks, then pass a task list to `asyncio.wait()` to schedule them. Finally, call `loop.run_until_complete()` with the object returned by `asyncio.wait()`.

Example

`async_simple.py`

```
import asyncio

async def say(what, when): ①
    await asyncio.sleep(when) ②
    print(what) ③

loop = asyncio.get_event_loop() ④
loop.run_until_complete(say('hello, async world', 1)) ⑤
loop.close() ⑥
```

① create a coroutine by using the `async` keyword

② call a sleep function to simulate doing actual work

③ print a message; in real life probably return a value or update a datastore

④ get the event loop

⑤ run the coroutine

⑥ close the loop

async_simple.py

```
hello, async world
```

Chapter 11 Exercises

Exercise 11-1

Using callbacks, print a countdown from 10 to '**Blast off!**'

Exercise 11-2

Using `asyncio` and `aiofiles`, 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:

1. how many files,
2. how many lines,
3. how many words, and
4. how many bytes.

Exercise 11-3

Using `asyncio` and the `aiohttp` module, 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.

HINT: See `async_lcbo.py` for some examples of how to fetch JSON from a website.

Chapter 12: 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/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 6. 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 <code>run()</code> was called with an encoding or errors. None if <code>stdout</code> was not captured. If you ran the process with <code>stderr=subprocess.STDOUT</code> , <code>stdout</code> and <code>stderr</code> will be combined in this attribute, and <code>stderr</code> will be None. <code>stderr</code>

subprocess convenience functions

- `run()`, `check_call()`, `check_output()`
- Simpler to use than `Popen`

`subprocess` defines convenience functions, `call()`, `check_call()`, and `check_output()`.

```
proc subprocess.run(cmd, ...)
```

Run command with arguments. Wait for command to complete, then return a `CompletedProcess` instance.

```
subprocess.check_call(cmd, ...)
```

Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute.

```
check_output(cmd, ...)
```

Run command with arguments and return its output as a byte string. If the exit code was non-zero it raises a `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute and output in the `output` attribute.

NOTE `run()` is only implemented in Python 3.5 and later.

Example

subprocess_conv.py

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

import sys
from subprocess import check_call, check_output, CalledProcessError
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

try:
    check_call(full_cmd)
except CalledProcessError as err:
    print("Command failed with return code", err.returncode)

print('-' * 60)

try:
    output = check_output(full_cmd)
    print("Output:", output.decode(), sep='\n')
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

subprocess_conv.py

```
-rwxr-xr-x 1 jstrick staff    297 Nov 17 2016 ./DATA/testscores.dat
-rwxr-xr-x 1 jstrick staff    2198 Feb 14 2016 ./DATA/textfiles.zip
-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:

```
-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

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()) ⑫
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

- ① need to import PIPE and STDOUT
- ② capture only stdout
- ③ check_output() returns stdout
- ④ stdout is returned as bytes (decode to str)
- ⑤ capture stdout and stderr together
- ⑥ assign PIPE to stdout, so it is captured; assign STDOUT to stderr, so both are captured together
- ⑦ call communicate to get the input streams of the process; it returns two bytes objects representing stdout and stderr
- ⑧ decode the stdout object to a string
- ⑨ assign PIPE to stdout and PIPE to stderr, so both are captured individually
- ⑩ now stdout and stderr each have data

⑪ decode from bytes and output*subprocess_capture.py***Output:**

```
-rwxr-xr-x 1 jstrick staff      297 Nov 17 2016 ./DATA/testscores.dat
-rwxr-xr-x 1 jstrick staff      2198 Feb 14 2016 ./DATA/textfiles.zip
-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: -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 Feb 18 13:22 spam.txt
```

```
-----
Output: -rwxr-xr-x 1 jstrick staff      297 Nov 17 2016 ./DATA/testscores.dat
-rwxr-xr-x 1 jstrick staff      2198 Feb 14 2016 ./DATA/textfiles.zip
-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 Feb 18 13:22 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:
    sys.stderr.write('Please specify a starting directory\n')
    sys.exit(1)

start_dir = sys.argv[1]

for base_name in os.listdir(start_dir): ①
    file_name = os.path.join(start_dir, base_name)
    if os.access(file_name, os.W_OK): ②
        print(file_name, "is writable")
```

① `os.listdir()` lists the contents of a directory

② `os.access()` returns True if file has specified permissions (can be `os.W_OK`, `os.R_OK`, or `os.X_OK`, combined with | (OR))

file_access.py ..//DATA

```
..//DATA/presidents.csv is writable
..//DATA/wetprf is writable
..//DATA/presidents.html is writable
..//DATA/presidents.xlsx is writable
..//DATA/baby_names is writable
..//DATA/presidents.db is writable
..//DATA/testscores.dat is writable
..//DATA/solar.json is writable
..//DATA/nws_precip_pr_20061222.nc is writable
..//DATA/columns_of_numbers.txt is writable
```

...

Using `shutil`

- Portable ways to copy, move, and delete files
- Create archives
- Misc utilities

The **shutil** module provides portable functions for copying, moving, renaming, and deleting files. There are several variations of each command, depending on whether you need to copy all the attributes of a file, for instance.

The module also provides an easy way to create a zip file or compressed **tar** archive of a folder.

In addition, there are some miscellaneous convenience routines.

Example

shutil_ex.py

```
#!/usr/bin/env python
#
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 7. 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 8. 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 9. Logging Levels

Level	Value
CRITICAL	50
FATAL	
ERROR	40
WARN	30
WARNING	
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='%(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

```
2020-02-20 06:37:39,962 INFO this is information
2020-02-20 06:37:39,962 WARNING this is a warning
2020-02-20 06:37:39,962 INFO this is information
2020-02-20 06:37:39,962 CRITICAL this is critical
```

Table 10. 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 12 Exercises

Exercise 12-1 (`copy_files.py`)

Write a script to find all text files in the DATA folder of the student files and copy them to C:\TEMP (Windows) or /tmp (non-windows). On Windows, create the C:\TEMP folder if it does not already exist.

Add logging to the script, and log each filename at level INFO.

TIP | [use `shutil.copy\(\)` to copy the files.](#)

Chapter 13: Unit Tests with pytest

Objectives

- Understand the purpose of unit tests
- Design and implement unit tests with pytest
- Run tests in different ways
- Use builtin fixtures
- Create and use custom fixtures
- Mark tests for running in groups
- Learn how to mock data for tests

What is a unit test?

- Tests *unit* of code in isolation
- Ensures repeatable results
- Asserts expected behavior

A *unit test* is a test which asserts that an isolated piece of code (one function, method, class, or module) has some expected behavior. It is a way of making sure that code provides repeatable results.

There are four main components of a unit testing system:

1. Unit tests – individual assertions that an expected condition has been met
2. Test cases – collections of related unit tests
3. Fixtures — provide data to set up tests in order to get repeatable results
4. Test runners – utilities to execute the tests in one or more test cases

Unit tests should each test one aspect of your code, and each test should be independent of all other tests, including the order in which tests are run.

Each test asserts that some condition is true.

Unit tests may collected into a **test case**, which is a related group of unit tests. With pytest, a test case can be either a module or a class. This is optional in pytest.

Fixtures provide repeatable, known input to a test.

The final component is a **Test runner**, which executes one, some, or all tests and reports on the results. There are many different test runners for pytest. The builtin runner is very flexible.

The pytest module

- Provides
 - test runner
 - fixtures
 - special assertions
 - extra tools
- Not based on xUnit¹

The **pytest** module provides tools for creating, running, and managing unit tests.

Each test supplies one or more **assertions**. An assertion confirms that some condition is true.

Here's how **pytest** implements the main components of unit testing:

unit test

A normal Python function that uses the **assert** statement to assert some condition is true

test case

A class or a module than contains unit tests (tests can be grouped with *markers*).

fixture

A special parameter of a unit test function that provides test resources (fixtures can be nested)

test runner

A text-based test runner is built in, and there are many third-party test runners

pytest is more flexible than classic xUnit implementations. For example, fixtures can be associated with any number of individual tests, or with a test class. Test cases need not be classes.

¹ The builtin unit testing module, **unittest**, is based on **xUnit** patterns, as implemented in Java and other languages.

Creating tests

- Create test functions
- Use builtin **assert**
- Confirm something is true
- Optional message

To create a test, create a function whose name begins with "test". These should normally be in a separate script, whose name begins with "test_" or ends with "_test". For the simplest cases, tests do not even need to import **pytest**.

Each test function should use the builtin **assert** statement one or more times to confirm that the test passes. If the assertion fails, the test fails.

pytest will print an appropriate message by introspecting the expression, or you can add your own message after the expression, separated by a comma

It is a good idea to make test names verbose. This will help when running tests in verbose mode, so you can see what tests are passing (or failing).

```
assert result == 'spam'  
assert 2 == 3, "Two is not equal to three!"
```

Example

`pytests/test_simple.py`

```
#!/usr/bin/env python  
  
def test_two_plus_two_equals_four(): ①  
    assert 2 + 2 == 4    # ②
```

① tests should begin with "test" (or will not be found automatically)

② if **assert** statement succeeds, the test passes

Running tests (basics)

- Needs a test runner
- **pytest** provides **pytest** script

To actually run tests, you need a *test runner*. A test runner is software that runs one or more tests and reports the results.

pytest provides a script (also named **pytest**) to run tests.

You can run a single test, a test case, a module, or all tests in a folder and all its subfolders.

`pytest test_...py`

to run the tests in a particular module, and

`pytest -v test_...py`

to add verbose output.

By default, pytest captures anything written to stdout/stderr. If you want to see the output of `print()` statements in your tests, add the `-s` option, which turns off output capture.

`pytest -s ...`

NOTE

In older versions of pytest, the test runner script was named **py.test**. Newer versions support that name, but the developers recommend only using **pytest**.

TIP

PyCharm automatically detects a script containing test cases. When you run the script the first time, PyCharm will ask whether you want to run it normally or use its builtin test runner. Use **Edit Configurations** to modify how the script is run. Note: in PyCharm's settings, you can select the default test runner to be **pytest**, **Unittest**, or other test runners.

Special assertions

- Special cases
 - `pytest.raises()`
 - `pytest.approx()`

There are two special cases not easily handled by `assert`.

`pytest.raises`

For testing whether an exception is raised, use `pytest.raises()`. This should be used with the `with` statement:

```
with pytest.raises(ValueError):  
    w = Wombat('blah')
```

The assertion will succeed if the code inside the `with` block raises the specified error.

`pytest.approx`

For testing whether two floating point numbers are *close enough* to each other, use `pytest.approx()`:

```
assert result == pytest.approx(1.55)
```

The default tolerance is 1e-6 (one part in a million). You can specify the relative or absolute tolerance to any degree. Infinity and NaN are special cases. NaN is normally not equal to anything, even itself, but you can specify `nanok=True` as an argument to `approx()`.

NOTE

See <https://docs.pytest.org/en/latest/reference.html#pytest-approx> for more information on `pytest.approx()`

Example

pytests/test_special_assertions.py

```
#!/usr/bin/env python
import pytest
import math

FILE_NAME = 'IDONOTEXIST.txt'

def test_missing_filename():
    with pytest.raises(FileNotFoundError): ①
        open(FILE_NAME) ②

def test_list():
    print()
    assert (.1 + .2) == pytest.approx(.3) ③

def test_approximate_pi():
    assert 22 / 7 == pytest.approx(math.pi, .001) ④
```

① assert FileNotFoundError is raised inside block

② will fail test if file is not found

③ fail unless values are within 0.000001 of each other (actual result is 0.3000000000000004)

④ Default tolerance is 0.000001; smaller (or larger) tolerance can be specified

Fixtures

- Provide resources for tests
- Implement as functions
- Scope
 - Per test
 - Per class
 - Per module
- Source of fixtures
 - Builtin
 - User-defined

When writing tests for a particular object, many tests might require an instance of the object. This instance might be created with a particular set of arguments.

What happens if twenty different tests instantiate a particular object, and the object's API changes? Now you have to make changes in twenty different places.

To avoid duplicating code across many tests, pytest supports *fixtures*, which are functions that provide information to tests. The same fixture can be used by many tests, which lets you keep the fixture creation in a single place.

A fixture provides items needed by a test, such as data, functions, or class instances.

Fixtures can be either builtin or custom.

TIP

Use `py.test --fixtures` to list all available builtin and user-defined fixtures.

User-defined fixtures

- Decorate with `pytest.fixture`
- Return value to be used in test
- Fixtures may be nested

To create a fixture, decorate a function with `pytest.fixture`. Whatever the function returns is the value of the fixture.

To use the fixture, pass it to the test function as a parameter. The return value of the fixture will be available as a local variable in the test.

Fixtures can take other fixtures as parameters as well, so they can be nested to any level.

Put fixtures into their own module so they can be shared across multiple test scripts.

TIP

Add docstrings to your fixtures and the docstrings will be displayed via `pytest --fixtures`

Example

pytests/test_simple_fixture.py

```
#!/usr/bin/env python
from collections import namedtuple
import pytest

Person = namedtuple('Person', 'first_name last_name') ①

FIRST_NAME = "Guido"
LAST_NAME = "Von Rossum"

@pytest.fixture ②
def person():
    """
    Return a 'Person' named tuple with fields 'first_name' and 'last_name'
    """
    return Person(FIRST_NAME, LAST_NAME) ③

def test_first_name(person): ④
    assert person.first_name == FIRST_NAME

def test_last_name(person): ④
    assert person.last_name == LAST_NAME
```

① create object to test

② mark **person** as a fixture

③ return value of fixture

④ pass fixture as test parameter

Builtin fixtures

- Variety of common fixtures
- Provide
 - Temp files and dirs
 - Logging
 - STDOUT/STDERR capture
 - Monkeypatching tools

Pytest provides a large number of builtin fixtures for common testing requirements.

Using a builtin fixture is like using user-defined fixtures. Just specify the fixture name as a parameter to the test. No imports are needed for this.

See <https://docs.pytest.org/en/latest/reference.html#fixtures> for details on builtin fixtures.

Example

pytests/test_builtin_fixtures.py

```
COUNTER_KEY = 'test_cache/counter'

def test_cache(cache): ①
    value = cache.get(COUNTER_KEY, 0)
    print("Counter before:", value)
    cache.set(COUNTER_KEY, value + 1) ②
    value = cache.get(COUNTER_KEY, 0) ②
    print("Counter after:", value)
    assert True ③

def hello():
    print("Hello, pytesting world")

def test_capsys(capsys):
    hello() ④
    out, err = capsys.readouterr() ⑤
    print("STDOUT:", out)

def bhello():
    print(b"Hello, binary pytesting world\n")

def test_captionsbinary(capsys):
    bhello() ⑥
    out, err = capsys.readouterr() ⑦
    print("BINARY STDOUT:", out)

def test_temp_dir1(tmpdir):
    print("TEMP DIR:", str(tmpdir)) ⑧

def test_temp_dir2(tmpdir):
    print("TEMP DIR:", str(tmpdir))

def test_temp_dir3(tmpdir):
    print("TEMP DIR:", str(tmpdir))
```

- ① cache persists values between test runs
- ② cache fixture is similar to dictionary, but with `.set()` and `.get()` methods
- ③ Make test successful
- ④ Call function that writes text to STDOUT
- ⑤ Get captured output
- ⑥ Call function that writes binary text to STDOUT
- ⑦ Get captured output
- ⑧ `tmpdir` fixture provides unique temporary folder name

Table 11. Pytest Builtin Fixtures

Fixture	Brief Description
cache	Return cache object to persist state between testing sessions.
capsys	Enable capturing of writes (text mode) to <code>sys.stdout</code> and <code>sys.stderr</code>
capsysbinary	Enable capturing of writes (binary mode) to <code>sys.stdout</code> and <code>sys.stderr</code>
capfd	Enable capturing of writes (text mode) to file descriptors 1 and 2
capfdbinary	Enable capturing of writes (binary mode) to file descriptors 1 and 2
doctest_namespace	Return <code>dict</code> that will be injected into namespace of doctests
pytestconfig	Session-scoped fixture that returns <code>_pytest.config.Config</code> object.
record_property	Add extra properties to the calling test.
record_xml_attribute	Add extra xml attributes to the tag for the calling test.
caplog	Access and control log capturing.
monkeypatch	Return <code>monkeypatch</code> fixture providing monkeypatching tools
recwarn	Return <code>WarningsRecorder</code> instance that records all warnings emitted by test functions.
tmp_path	Return <code>pathlib.Path</code> instance with unique temp directory
tmp_path_factory	Return a <code>_pytest.tmpdir.TempPathFactory</code> instance for the test session.
tmpdir	Return <code>py.path.local</code> instance unique to each test
tmpdir_factory	Return <code>TempdirFactory</code> instance for the test session.

Configuring fixtures

- Create **conftest.py**
- Automatically included
- Provides
 - Fixtures
 - Hooks
 - Plugins
- Directory scope

The **conftest.py** file can be used to contain user-defined fixtures, as well has hooks and plugins. Subfolders can have their own conftest.py, which will only apply to tests in that folder.

In a test folder, define one or more fixtures in conftest.py, and they will be available to all tests in that folder, as well as any subfolders.

Hooks

Hooks are predefined functions that will automatically be called at various points in testing. All hooks start with *pytest_*. A `pytest.Function` object, which contains the actual test function, is passed into the hook.

For instance, `pytest_runtest_setup()` will be called before each test.

NOTE

A complete list of hooks can be found here: <https://docs.pytest.org/en/latest/reference.html#hooks>

Plugins

There are many pytest plugins to provide helpers for testing code that uses common libraries, such as **Django** or **redis**.

You can register plugins in conftest.py like so:

```
pytest_plugins = "plugin1", "plugin2",
```

This will load the plugins.

Example

pytests/stuff/conftest.py

```
#!/usr/bin/env python
from pytest import fixture

@fixture
def common_fixture(): ①
    return "DATA"

def pytest_runtst_setup(item): ②
    print("Hello from setup,", item)
```

① user-defined fixture

② predefined hook (all hooks start with *pytest_*)

Example

pytests/stuff/test_stuff.py

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

def test_one(): ①
    print("WHOOPEE")
    assert(1)

def test_two(common_fixture): ②
    assert(common_fixture == "DATA")

if __name__ == '__main__':
    pytest.main([__file__, "-s"]) ③
```

① unit test that writes to STDOUT

② unit test that uses fixture from conftest.py

③ run tests (without stdout/stderr capture) when this script is run

pytests/stuff/test_stuff.py

```
===== test session starts =====
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, infile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 2 items

pytests/stuff/test_stuff.py Hello from setup, <Function 'test_one'>
WHOOPEE
.Hello from setup, <Function 'test_two'>
.

=====
2 passed in 0.01 seconds =====
```

Parametrizing tests

- Run same test on multiple values
- Add parameters to fixture decorator
- Test run once for each parameter
- Use `pytest.mark.parametrize()`

Many tests require testing a method or function against many values. Rather than writing a loop in the test, you can automatically repeat the test for a set of inputs via **parametrizing**.

Apply the `@pytest.mark.parametrize` decorator to the test. The first argument is a string with the comma-separated names of the parameters; the second argument is the list of parameters. The test will be called once for each item in the parameter list. If a parameter list item is a tuple or other multi-value object, the items will be passed to the test based on the names in the first argument.

NOTE

For more advanced needs, when you need some extra work to be done before the test, you can do indirect parametrizing, which uses a parametrized fixture. See `test_parametrize_indirect.py` for an example.

NOTE

The authors of pytest deliberately spelled it "parametrizing", not "parameterizing".

Example

pytests/test_parametrization.py

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

def triple(x): ①
    return x * 3

test_data = [(5, 15), ('a', 'aaa'), ([True], [True, True, True])] ②

@pytest.mark.parametrize("input,result", test_data) ③
def test_triple(input, result): ④
    print("input {} result {}".format(input, result)) ④
    assert triple(input) == result ⑤

if __name__ == "__main__":
    pytest.main([__file__, '-s'])
```

① Function to test

② List of values for testing containing input and expected result

③ Parametrize the test with the test data; the first argument is a string defining parameters to the test and mapping them to the test data

④ The test expects two parameters (which come from each element of test data)

⑤ Test the function with the parameters

pytests/test_parametrization.py

```
===== test session starts =====
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 3 items

pytests/test_parametrization.py input 5 result 15:
.input a result aaa:
.input [True] result [True, True, True]:
.

===== 3 passed in 0.01 seconds =====
```

Marking tests

- Create groups of tests ("test cases")
- Can create multiple groups
- Use `@pytest.mark.somemark()`

You can mark tests with labels so that they can be run as a group. Use `@pytest.mark.mark()`, where *mark* is the name of the mark, which can be any alphanumeric string.

Then you can run select tests which contain or match the mark, as described in the next topic.

In addition, you can register markers in the **[pytest]** section of `pytest.ini`:

```
[pytest]
markers =
    internet: test requires internet connection
    slow: tests that take more time (omit with '-m "not slow")
```

```
pytest -m "mark"
pytest -m "not mark"
```

Example

pytests/test_mark.py

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

@pytest.mark.alpha ①
def test_one():
    assert 1

@pytest.mark.alpha ①
def test_two():
    assert 1

@pytest.mark.beta ②
def test_three():
    assert 1

if __name__ == '__main__':
    pytest.main([__file__, '-m alpha']) ③
```

① Mark with label **alpha**

② Mark with label **beta**

③ Only tests marked with **alpha** will run (equivalent to 'pytest -m alpha' on command line)

pytests/test_mark.py

```
===== test session starts =====
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 3 items / 1 deselected

pytests/test_mark.py .. [100%]

===== 2 passed, 1 deselected in 0.01 seconds =====
```

Running tests (advanced)

- Run all tests
- Run by
 - function
 - class
 - module
 - name match
 - group

`pytest` provides many ways to select which tests to run.

Running all tests

To run all tests in the current and any descendent directories, use

Use `-s` to disable capturing, so anything written to STDOUT is displayed. Use `-s` for verbose output.

```
pytest
pytest -v
pytest -s
pytest -vs
```

Running by component

Use the node ID to select by component, such as module, class, method, or function name:

```
file::class
file::class::test
file::::test
```

```
pytest test_president.py::test_dates
pytest test_president.py::test_dates::test_birth_date
```

Running by name match

Use **-k** to run all tests whose name includes a specified string

`pytest -k date` *run all tests whose name includes 'date'*

Skipping and failing

- Conditionally skip tests
- Completely ignore tests
- Decorate with
 - `@pytest.mark.xfail`
 - `@pytest.mark.skip`

To skip tests conditionally (or unconditionally), use `@pytest.mark.skip()`. This is useful if some tests rely on components that haven't been developed yet, or for tests that are platform-specific.

To fail on purpose, use `@pytest.mark.xfail`). This reports the test as "XPASS" or "xfail", but does not provide traceback. Tests marked with xfail will not fail the test suite. This is useful for testing not-yet-implemented features, or for testing objects with known bugs that will be resolved later.

Example

pytests/test_skip.py

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

def test_one(): ①
    assert 1

@pytest.mark.skip(reason="can not currently test") ②
def test_two():
    assert 1

@pytest.mark.skipif(sys.platform != 'win32', reason="only implemented on Windows")
③
def test_three():
    assert 1

@pytest.mark.xfail ④
def test_four():
    assert 1

@pytest.mark.xfail ④
def test_five():
    assert 0

if __name__ == '__main__':
    pytest.main([__file__, '-v'])
```

① Normal test

② Unconditionally skip this test

③ Skip this test if current platform is not Windows

pytests/test_skip.py

```
===== test session starts =====
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0 --
/Users/jstrick/.pyenv/versions/anaconda3-2018.12/bin/python
cachedir: .pytest_cache
rootdir: /Users/jstrick/curr/courses/python/examples3, configparser:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collecting ... collected 5 items

pytests/test_skip.py::test_one PASSED [ 20%]
pytests/test_skip.py::test_two SKIPPED [ 40%]
pytests/test_skip.py::test_three SKIPPED [ 60%]
pytests/test_skip.py::test_four XPASS [ 80%]
pytests/test_skip.py::test_five xfail [100%]

===== 1 passed, 2 skipped, 1 xfailed, 1 xpassed in 0.07 seconds =====
```

Mocking data

- Simulate behavior of actual objects
- Replace expensive dependencies (time/resources)
- Use `unittest.mock` or `pytest-mock`

Some objects have dependencies which can make unit testing difficult. These dependencies may be expensive in terms of time or resources.

The solution is to use a **mock** object, which pretends to be the real object. A mock object behaves like the original object, but is restricted and controlled in its behavior.

For instance, a class may have a dependency on a database query. A mock object may accept the query, but always returns a hard-coded set of results.

A mock object can record the calls made to it, and assert that the calls were made with correct parameters.

A mock object can be preloaded with a return value, or a function that provides dynamic (or random) return values.

A *stub* is an object that returns minimal information, and is also useful in testing. However, a mock object is more elaborate, with record/playback capability, assertions, and other features.

pymock objects

- Use `pytest-mock` plugin
 - Can also use `unittest.mock.Mock`
- Emulate resources

`pytest` can use `unittest.mock`, from the standard library, or the `pytest-mock` plugin, which provides a wrapper around `unittest.mock`.

In either case, there are two primary ways of using mock. One is to provide a replacement class, function, or data object that mimics the real thing. The second is to monkey-path a library, which temporarily (just during the test) replaces a component with a mock version.

Once the `pytest-mock` module is installed, it provides a fixture named **mocker**, from which you can create mock objects.

Example

pytests/test_mock_unittest.py

```
#!/usr/bin/env python
#
import pytest
from unittest.mock import Mock

ham = Mock() ①

# system under test
class Spam(): ②
    def __init__(self, param):
        self._value = ham(param) ③

    @property
    def value(self): ④
        return self._value

# dependency to be mocked -- not used in test
# def ham(n):
#     pass

def test_spam_calls_ham(): ⑤
    _ = Spam(42) ⑥
    ham.assert_called_once_with(42) ⑦

if __name__ == '__main__':
    pytest.main([__file__])
```

① Create mock version of ham() function

② System (class) under test

③ Calls ham() (doesn't know if it's fake)

④ Property to return result of ham()

⑤ Actual unit test

⑥ Create instance of Spam, which calls ham()

⑦ Check that spam.value correctly returns return value of ham()

pytests/test_mock_unittest.py

```
===== test session starts =====
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, infile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 1 item

pytests/test_mock_unittest.py . [100%]

===== 1 passed in 0.01 seconds =====
```

Example

pytests/test_mock_pymock.py

```
#!/usr/bin/env python
import pytest ①
import re ②

class SpamSearch(): ③
    def __init__(self, search_string, target_string):
        self.search_string = search_string
        self.target_string = target_string

    def findit(self): ④
        return re.search(self.search_string, self.target_string)

def test_spam_search_calls_re_search(mocker): ⑤
    mocker.patch('re.search') ⑥
    s = SpamSearch('bug', 'lightning bug') ⑦
    _ = s.findit() ⑧
    re.search.assert_called_once_with('bug', 'lightning bug') ⑨

if __name__ == '__main__':
    pytest.main([__file__, '-s']) ⑩
```

① Needed for test runner

② Needed for test (but will be mocked)

③ System under test

④ Specific method to test (uses re.search)

⑤ Unit test

⑥ Patch re.search (i.e., replace re.search with a Mock object that records calls to it)

⑦ Create instance of SpamSearch

⑧ Call the method under test

⑨ Check that method was called just once with the expected parameters

⑩ Start the test runner

pytests/test_mock_pymock.py

```
===== test session starts =====
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: /Users/jstrick/curr/courses/python/examples3, infile:
plugins: remotedata-0.3.1, openfiles-0.3.1, mock-2.0.0, doctestplus-0.2.0,
arraydiff-0.3
collected 1 item

pytests/test_mock_pymock.py .

===== 1 passed in 0.08 seconds =====
```

Example

pytests/test_mock_play.py

```
#!/usr/bin/env python
import pytest
from unittest.mock import Mock

@pytest.fixture
def small_list(): ①
    return [1, 2, 3]

def test_m1_returns_correct_list(small_list):
    m1 = Mock(return_value=small_list) ②
    mock_result = m1('a', 'b') ③
    assert mock_result == small_list ④

m2 = Mock() ⑤

m2.spam('a', 'b') ⑥
m2.ham('wombat') ⑥
m2.eggs(1, 2, 3) ⑥

print("mock calls:", m2.mock_calls) ⑦

m2.spam.assert_called_with('a', 'b') ⑧
```

- ① Create fixture that provides a small list
- ② Create mock object that "returns" a small list
- ③ Call mock object with arbitrary parameters
- ④ Check the mocked result
- ⑤ Create generic mock object
- ⑥ Call fake methods on mock object
- ⑦ Mock object remembers all calls
- ⑧ Assert that spam() was called with parameters 'a' and 'b'

pytests/test_mock_play.py

```
mock calls: [call.spam('a', 'b'), call.ham('wombat'), call_eggs(1, 2, 3)]
```

Pytest and Unittest

- Run Unittest-based tests
- Use Pytest test runner

The Pytest builtin test runner will detect Unittest-based tests as well. This can be handy for transitioning legacy code to Pytest.

Chapter 13 Exercises

Exercise 13-1 (test_president.py)

Using **pytest**, Create a unit test for the President class you created earlier.¹

Suggestions for tests:

- What happens when an out-of-range term number is given?
- President 1's first name is "George"
- All 45 presidential terms match the correct last name (use list of last names and **parametrize**)
- Confirm date fields return an object of type **datetime.date**

¹ If there was not an exercise where you created a President class, there is one in the top-level folder of the student guide.

Chapter 14: Serializing Data

Objectives

- Have a good understanding of the XML format
- Know which modules are available to process XML
- Use lxml ElementTree to create a new XML file
- Parse an existing XML file with ElementTree
- Using XPath for searching XML nodes
- Load JSON data from strings or files
- Write JSON data to strings or files
- Read and write CSV data
- Read and write YAML data

About XML

- Variant of SGML
- All data contained within tags

An XML document consists of a single element, which contains sub-elements, which can have further sub-elements inside them. Elements are indicated by tags in the text. Tags are always inside angle brackets < >. Elements can either contain content, or they can be empty.

Tags can contain attributes, indicated by attribute="value". Tags can either appear in begin/end pairs, in which the end tag starts with a slash, or as a single tag, in which case the tag ends with a slash. Attributes must be surrounded by double quotes. All tag names and attribute names should be lower case.

Normal Approaches to XML

- SAX
 - One scan through file
 - Good for large files
 - Uses callbacks on XML parsing events
- DOM
 - Builds a document tree
 - Python supports both through many libraries

There are two approaches normally used in working with XML.

The first is called SAX, which stands for Simple API for XML . It processes an XML file as a single stream, and so is appropriate for large XML files.

SAX processing consists of attaching callback functions to SAX events. These events are created when the XML reader encounters all the various components of an XML file – begin tags, end tags, data, and so forth.

The second approach is called the DOM, (Document Object Model), which parses an XML document into a tree that's fully resident in memory.

A top-level Document instance is the root of the tree, and has a single child which is the top-level Element instance; this Element has child nodes representing the content and any sub-elements, which may in turn have further children and so forth. Classes such as Text, Comment, CDATASEction, EntityReference, provide access to XML structure and data. Nodes have methods for accessing the parent and child nodes, accessing element and attribute values, insert and delete nodes, and converting the tree back into XML.

The DOM is often useful for modifying XML documents, because you can create a DOM, modify it by adding new nodes and moving sub-trees around, and then produce a new XML document as output.

While the DOM specification doesn't require that the entire tree be resident in memory at one time, many of the Python DOM implementations keep the whole tree in RAM, which can limit the size of the file being processed.

Which module to use?

- Bewildering array of XML modules
- Some are SAX, some are DOM
- Use `xml.etree.ElementTree`

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, `xml.etree.ElementTree` is fast, provides both SAX and DOM style parsing, and unlike many of the other modules, has a Pythonic interface.

If available, use `lxml.etree`, which is a superset of `ElementTree` with some nice extra features, such as pretty-printing.

Table 12. XML Modules in the Python 3 Standard Library

Module	Description
<code>xml.parsers.expat</code>	Fast XML parsing using Expat
<code>xml.dom</code>	The Document Object Model API
<code>xml.dom.minidom</code>	Lightweight DOM implementation
<code>xml.dom.pulldom</code>	Support for building partial DOM trees
<code>xml.sax</code>	Support for SAX2 parsers
<code>xml.sax.handler</code>	Base classes for SAX handlers
<code>xml.sax.saxutils</code>	SAX Utilities
<code>xml.sax.xmlreader</code>	Interface for XML parsers
<code>xml.etree.ElementTree</code>	Pythonic interface to XML

Getting Started With ElementTree

- Import `xml.etree.ElementTree` (or `lxml.etree`) as `ET` for convenience
- Parse XML or create empty `ElementTree`

`ElementTree` is part of the Python standard library; `lxml` is included with the Anaconda distribution.

Since putting "`xml.etree.ElementTree`" in front of its methods requires a lot of extra typing , it is typical to alias `xml.etree.ElementTree` to just `ET` when importing it: `import xml.etree.ElementTree as ET`

You can check the version of `ElementTree` via the `VERSION` attribute:

```
import xml.etree.ElementTree as ET
print(ET.VERSION)
```

How ElementTree Works

- ElementTree contains root Element
- Document is tree of Elements

In ElementTree, an XML document consists of a nested tree of Element objects. Each Element corresponds to an XML tag.

An ElementTree object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use ElementTree.parse(); this creates the ElementTree wrapper and the tree of Elements. You can then navigate to, or search for, Elements within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

```
element = root.find('sometag')
for subelement in element:
    print(subelement.tag)
print(element.get('someattribute'))
```

Elements

- Element has
 - Tag name
 - Attributes (implemented as a dictionary)
 - Text
 - Tail
 - Child elements (implemented as a list) (if any)
- SubElement creates child of Element

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** class is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the get() method).

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is any text following the element, before the next element.

The **SubElement** class is a convenient way to add children to an existing Element.

TIP Only the tag property of an Element is required; other properties are optional.

Table 13. Element properties and methods

Property	Description
append(element)	Add a subelement element to end of subelements
attrib	Dictionary of element's attributes
clear()	Remove all subelements
find(path)	Find first subelement matching path
.findall(path)	Find all subelements matching path
findtext(path)	Shortcut for find(path).text
get(attr)	Get an attribute; Shortcut for attrib.get()
getiterator()	Returns an iterator over all descendants
getiterator(path)	Returns an iterator over all descendants matching path
insert(pos,element)	Insert subelement element at position pos
items()	Get all attribute values; Shortcut for attrib.items()
keys()	Get all attribute names; Shortcut for attrib.keys()
remove(element)	Remove subelement element
set(attrib,value)	Set an attribute value; shortcut for attr[attrib] = value
tag	The element's tag
tail	Text following the element
text	Text contained within the element

Table 14. ElementTree properties and methods

Property	Description
find(path)	Finds the first toplevel element with given tag; shortcut for getroot().find(path).
findall(path)	Finds all toplevel elements with the given tag; shortcut for getroot().findall(path).
findtext(path)	Finds element text for first toplevel element with given tag; shortcut for getroot().findtext(path).
getiterator(path)	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
getroot()	Return the root node of the document
parse(filename) parse(fileobj)	Parse an XML source (filename or file-like object)
write(filename,encoding)	Writes XML document to filename, using encoding (Default us-ascii).

Creating a New XML Document

- Create root element
- Add descendants via SubElement
- Use keyword arguments for attributes
- Add text after element created
- Create ElementTree for import/export

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once the root element is created, use SubElement to add elements to the root element, and then nested Elements as needed. SubElement returns the new element, so you can assign the contents of the tag to the **text** attribute.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text. From the ElementTree object, call `write`.

To output an XML string from your elements, call `ET.tostring()`, passing the root of the element tree as a parameter. It will return a bytes object (pure ASCII), so use `.decode()` to convert it to a normal Python string.

For an example of creating an XML document from a data file, see `xml_create_knights.py` in the EXAMPLES folder

Example

xml_create_movies.py

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

# from xml.etree import ElementTree as ET
import lxml.etree as ET

movie_data = [
    ('Jaws', 'Spielberg, Stephen'),
    ('Vertigo', 'Alfred Hitchcock'),
    ('Blazing Saddles', 'Brooks, Mel'),
    ('Princess Bride', 'Reiner, Rob'),
    ('Avatar', 'Cameron, James'),
]

movies = ET.Element('movies')

for name, director in movie_data:
    movie = ET.SubElement(movies, 'movie', name=name)
    ET.SubElement(movie, 'director').text = director

print(ET.tostring(movies, pretty_print=True).decode())

doc = ET.ElementTree(movies)

doc.write('movies.xml')
```

xml_create_movies.py

```
<movies>
  <movie name="Jaws">
    <director>Spielberg, Stephen</director>
  </movie>
  <movie name="Vertigo">
    <director>Alfred Hitchcock</director>
  </movie>
  <movie name="Blazing Saddles">
    <director>Brooks, Mel</director>
  </movie>
  <movie name="Princess Bride">
    <director>Reiner, Rob</director>
  </movie>
  <movie name="Avatar">
    <director>Cameron, James</director>
  </movie>
</movies>
```

Parsing An XML Document

- Use `ElementTree.parse()`
- returns an `ElementTree` object
- Use `get*` or `find*` methods to select an element

Use the `parse()` method to parse an existing XML document. It returns an `ElementTree` object, from which you can find the root, or any other element within the document.

To get the root element, use the `getroot()` method.

Example

Navigating the XML Document

- Use `find()` or `findall()`
- Element is iterable of its children
- `findtext()` retrieves text from element

To find the first child element with a given tag, use `find('tag')`. This will return the first matching element. The `findtext('tag')` method is the same, but returns the text within the tag.

To get all child elements with a given tag, use the `findall('tag')` method, which returns a list of elements.

to see whether a node was found, say

```
if node is None:
```

but to check for existence of child elements, say

```
if len(node) > 0:
```

A node with no children tests as false because it is an empty list, but it is not `None`.

TIP

The `ElementTree` object also supports the `find()` and `findall()` methods of the `Element` object, searching from the root object.

Example

xml_planets_nav.py

```
#!/usr/bin/env python
'''Use etree navigation to extract planets from solar.xml'''
import lxml.etree as ET

def main():
    '''Program entry point'''
    doc = ET.parse('../DATA/solar.xml') ①

    solar_system = doc.getroot() ②

    print(solar_system)
    print()

    inner = solar_system.find('innerplanets') ③
    print('Inner:')

    for planet in inner: ④
        if planet.tag == 'planet':
            print('\t', planet.get("planetname", "NO NAME"))

    outer = solar_system.find('outerplanets')
    print('Outer:')

    for planet in outer:
        print('\t', planet.get("planetname"))

    plutoids = solar_system.find('dwarfplanets')
    print('Dwarf:')

    for planet in plutoids:
        print('\t', planet.get("planetname"))

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

xml_planets_nav.py

```
<Element solarsystem at 0x10a28ad48>
```

Inner:

- Mercury
- Venus
- Earth
- Mars

Outer:

- Jupiter
- Saturn
- Uranus
- Neptune

Dwarf:

- Pluto

Example

`xml_read_movies.py`

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

# import xml.etree.ElementTree as ET
import lxml.etree as ET

movies_doc = ET.parse('movies.xml') ①

movies = movies_doc.getroot() ②

for movie in movies: ③
    print('{} by {}'.format(
        movie.get('name'), ④
        movie.findtext('director'), ⑤
    ))
    )
```

- ① read and parse the XML file
- ② get the root element (<movies>)
- ③ loop through children of root element
- ④ get 'name' attribute of movie element
- ⑤ get 'director' attribute of movie element

`xml_read_movies.py`

```
Jaws by Spielberg, Stephen
Vertigo by Alfred Hitchcock
Blazing Saddles by Brooks, Mel
Princess Bride by Reiner, Rob
Avatar by Cameron, James
```

Using XPath

- Use simple XPath patterns Works with find* methods

When a simple tag is specified, the find* methods only search for subelements of the current element. For more flexible searching, the find* methods work with simplified **XPath** patterns. To find all tags named 'spam', for instance, use `./spam`.

```
./movie  
presidents/president/name/last
```

Example

`xml_planets_xpath1.py`

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

# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml') ①

inner_nodes = doc.findall('innerplanets/planet') ②

outer_nodes = doc.findall('outerplanets/planet') ③

print('Inner:')
for planet in inner_nodes: ④
    print('\t', planet.get("planetname")) ⑤

print('Outer:')
for planet in outer_nodes: ④
    print('\t', planet.get("planetname")) ⑤
```

- ① parse XML file
- ② find all elements (relative to root element) with tag "planet" under "innerplanets" element
- ③ find all elements with tag "planet" under "outerplanets" element
- ④ loop through search results
- ⑤ print "name" attribute of planet element

xml_planets_xpath1.py

```
Inner:  
    Mercury  
    Venus  
    Earth  
    Mars  
Outer:  
    Jupiter  
    Saturn  
    Uranus  
    Neptune
```

Example

xml_planets_xpath2.py

```
#!/usr/bin/env python  
  
# import xml.etree.ElementTree as ET  
import lxml.etree as ET  
  
doc = ET.parse('../DATA/solar.xml')  
  
jupiter = doc.find('.//planet[@planetname="Jupiter"]')  
  
if jupiter is not None:  
    for moon in jupiter:  
        print(moon.text) # grab attribute
```

xml_planets_xpath2.py

```
Metis
Adrastea
Amalthea
Thebe
Io
Europa
Ganymede
Callista
Themisto
Himalia
Lysithea
Elara
```

Table 15. ElementTree XPath Summary

Syntax	Meaning
tag	Selects all child elements with the given tag. For example, “spam” selects all child elements named “spam”, “spam/egg” selects all grandchildren named “egg” in all child elements named “spam”. You can use universal names (“{url}local”) as tags.
*	Selects all child elements. For example, “*/egg” selects all grandchildren named “egg”.
.	Select the current node. This is mostly useful at the beginning of a path, to indicate that it’s a relative path.
//	Selects all subelements, on all levels beneath the current element (search the entire subtree). For example, “./egg” selects all “egg” elements in the entire tree.
..	Selects the parent element.
[@attrib]	Selects all elements that have the given attribute. For example, “./a[@href]” selects all “a” elements in the tree that has a “href” attribute.
[@attrib='value']	Selects all elements for which the given attribute has the given value. For example, “./div[@class='sidebar']” selects all “div” elements in the tree that has the class “sidebar”. In the current release, the value cannot contain quotes.
[tag]	Selects all elements that has a child element named tag. In the current version, only a single tag can be used (i.e. only immediate children are supported).

About JSON

- Lightweight, human-friendly format for data
- Contains dictionaries and lists
- Stands for JavaScript Object Notation
- Looks like Python
- Basic types: Number, String, Boolean, Array, Object
- White space is ignored
- Stricter rules than Python

JSON is a lightweight and human-friendly format for sharing or storing data. It was developed and popularized by Douglas Crockford starting in 2001.

A JSON file contains objects and arrays, which correspond exactly to Python dictionaries and lists.

White space is ignored, so JSON may be formatted for readability.

Data types are Number, String, and Boolean. Strings are enclosed in double quotes (only); numbers look like integers or floats; Booleans are represented by true or false; null (None in Python) is represented by null.

Reading JSON

- json module in standard library
- json.load() parse from file-like object
- json.loads() parse from string
- Both methods return Python dict or list

To read a JSON file, import the json module. Use json.loads() to parse a string containing valid JSON. Use json.load() to read JSON from a file-like object.

Both methods return a Python dictionary containing all the data from the JSON file.

Example

json_read.py

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

import json

with open('../DATA/solar.json') as solar_in: ①
    solar = json.load(solar_in) ②

# json.loads(STRING)
# json.load(FILE_OBJECT)

# print(solar)

print(solar['innerplanets']) ③
print('*' * 60)
print(solar['innerplanets'][0]['name'])
print('*' * 60)
for planet in solar['innerplanets'] + solar['outerplanets']:
    print(planet['name'])

print("*" * 60)
for group in solar:
    if group.endswith('planets'):
        for planet in solar[group]:
            print(planet['name'])
```

① open JSON file for reading

② load from file object and convert to Python data structure

③ solar is just a Python dictionary

json_read.py

```
[{'name': 'Mercury', 'moons': None}, {'name': 'Venus', 'moons': None}, {'name': 'Earth', 'moons': ['moon']}, {'name': 'Mars', 'moons': ['Deimos', 'Phobos']}]
*****
Mercury
*****
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
*****
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto
```

Writing JSON

- Use `json.dumps()` or `json.dump()`

To output JSON to a string, use `json.dumps()`. To output JSON to a file, pass a file-like object to `json.dump()`. In both cases, pass a Python data structure as the data to be output.

Example

`json_write.py`

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

import json

george = [
    {
        'num': 1,
        'lname': 'Washington',
        'fname': 'George',
        'dstart': [1789, 4, 30],
        'dend': [1797, 3, 4],
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'dbirth': [1732, 2, 22],
        'ddeath': [1799, 12, 14],
        'assassinated': False,
        'party': None,
    },
    {
        'spam': 'ham',
        'eggs': [1.2, 2.3, 3.4],
        'toast': {'a': 5, 'm': 9, 'c': 4},
    }
] ①

js = json.dumps(george, indent=4) ②
print(js)

with open('george.json', 'w') as george_out: ③
    json.dump(george, george_out, indent=4) ④
```

- ① Python data structure
- ② dump structure to JSON string
- ③ open file for writing
- ④ dump structure to JSON file using open file object

json_write.py

```
[  
  {  
    "num": 1,  
    "lname": "Washington",  
    "fname": "George",  
    "dstart": [  
      1789,  
      4,  
      30  
    ],  
    "dend": [  
      1797,  
      3,  
      4  
    ],  
    "birthplace": "Westmoreland County",  
    "birthstate": "Virginia",  
    "dbirth": [  
      1732,  
      2,  
      22  
    ],  
    "ddeath": [  
      1799,  
      12,  
      14  
    ],  
    "assassinated": false,  
    "party": null  
  },  
  {  
    "spam": "ham",  
    "eggs": [  
      1.2,  
      2.3,  
      3.4  
    ]  
  }]
```

```
3.4
],
"toast": {
    "a": 5,
    "m": 9,
    "c": 4
}
]
}
```

Customizing JSON

- JSON data types limited
- simple cases — dump dict
- create custom encoders

The JSON spec only supports a limited number of datatypes. If you try to dump a data structure contains dates, user-defined classes, or many other types, the json encoder will not be able to handle it.

You can a custom encoder for various data types. To do this, write a function that expects one Python object, and returns some object that JSON can parse, such as a string or dictionary. The function can be called anything. Specify the function with the **default** parameter to `json.dump()`.

The function should check the type of the object. If it is a type that needs specially handling, return a JSON-friendly version, otherwise just return the original object.

*Table 16. Python types that
JSON can encode*

Python	JSON
dict	object
list	array
str	string
int	number (int)
float	number (real)
True	true
False	false
None	null

NOTE

see the file `json_custom_singledispatch.py` in EXAMPLES for how to use the `singledispatch` module (which is not in the standard library) to handle multiple data types.

Example

json_custom_encoding.py

```
#!/usr/bin/env python
#
import json
from datetime import date

class Parrot(): ①
    def __init__(self, name, color):
        self._name = name
        self._color = color

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

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

parrots = [ ③
    Parrot('Polly', 'green'), #
    Parrot('Peggy', 'blue'),
    Parrot('Roger', 'red'),
]

def encode(obj): ④
    if isinstance(obj, date): ⑤
        return obj.ctime() ⑥
    elif isinstance(obj, Parrot): ⑦
        return {'name': obj.name, 'color': obj.color} ⑧
    return obj ⑨

data = { ⑩
    'spam': [1, 2, 3],
    'ham': ('a', 'b', 'c'),
    'toast': date(2014, 8, 1),
    'parrots': parrots,
```

```
}
```

```
print(json.dumps(data, default=encode, indent=4)) ⑪
```

- ① sample user-defined class (not JSON-serializable)
- ② JSON does not understand arbitrary properties
- ③ list of Parrot objects
- ④ custom JSON encoder function
- ⑤ check for date object
- ⑥ convert date to string
- ⑦ check for Parrot object
- ⑧ convert Parrot to dictionary
- ⑨ if not processed, return object for JSON to parse with default parser
- ⑩ dictionary of arbitrary data
- ⑪ convert Python data to JSON data; 'default' parameter specifies function for custom encoding;
'indent' parameter says to indent and add newlines for readability

json_custom_encoding.py

```
{  
    "spam": [  
        1,  
        2,  
        3  
    ],  
    "ham": [  
        "a",  
        "b",  
        "c"  
    ],  
    "toast": "Fri Aug 1 00:00:00 2014",  
    "parrots": [  
        {  
            "name": "Polly",  
            "color": "green"  
        },  
        {  
            "name": "Peggy",  
            "color": "blue"  
        },  
        {  
            "name": "Roger",  
            "color": "red"  
        }  
    ]  
}
```

Reading and writing YAML

- yaml module from PYPI
- Same syntax as JSON
- yaml.load(), dump() parse from/to file-like object
- yaml.loads(), dumps() parse from/to string

Reading and writing YAML uses the same syntax as JSON, other than using the **yaml** module, which is NOT in the standard library. To install the **yaml** module:

```
pip install pyyaml
```

To read a YAML file (or string) into a Python data structure, use `yaml.load(file_object)` or `yaml.loads(string)`.

To write a data structure to a YAML file or string, use `yaml.dump(data, file_object)` or `yaml.dumps(data)`.

You can also write custom YAML processors.

Example

yaml_read_solar.py

```
#!/usr/bin/env python
# (c) 2015 John Strickler
#
import yaml

PLANET_SECTIONS = "inner outer plutoid".split()

with open('../DATA/solar.yaml') as solar_in:
    solar_data = yaml.load(solar_in)

star = solar_data['star']
print("Our star is {}".format(star))

for section in PLANET_SECTIONS:
    for planet in solar_data[section]:
        print(planet['name'])
        for moon in planet['moons']:
            print("\t{}".format(moon))
```

yaml_read_solar.py

```
Our star is Sun
```

```
Mercury
```

```
    None
```

```
Venus
```

```
    None
```

```
Earth
```

```
    Moon
```

```
Mars
```

```
    Deimos
```

```
    Phobos
```

```
    Metis
```

```
Jupiter
```

```
    Adrastea
```

```
    Amalthea
```

```
    Thebe
```

```
    Io
```

```
    Europa
```

```
    Ganimede
```

```
    Callista
```

```
    Themisto
```

```
    Himalia
```

```
    Lysithea
```

```
    Elara
```

```
Saturn
```

```
    Rhea
```

```
    Hyperion
```

```
    Titan
```

```
    Iapetus
```

```
    Mimas
```

...

Example

yaml_create_file.py

```
#!/usr/bin/env python
# (c) 2015 John Strickler
#
import sys
from datetime import date
import yaml

potus = {
    1: {
        'lname': 'Washington',
        'fname': 'George',
        'dob': date(1732, 2, 22),
        'dod': date(1799, 12, 14),
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'term': [ date(1789, 4, 30), date(1797, 3, 4) ],
        'assassinated': False,
        'party': None,
    },
    2: {
        'lname': 'Adams',
        'fname': 'John',
        'dob': date(1735, 10, 30),
        'dod': date(1826, 7, 4),
        'birthplace': 'Braintree, Norfolk',
        'birthstate': 'Massachusetts',
        'term': [date(1797, 3, 4), date(1801, 3, 4)],
        'assassinated': False,
        'party': 'Federalist',
    },
}

with open('potus.yaml', 'w') as potus_out:
    yaml.dump(potus, potus_out)

yaml.dump(potus, sys.stdout)
```

yaml_create_file.py

```
1:  
assassinated: false  
birthplace: Westmoreland County  
birthstate: Virginia  
dob: 1732-02-22  
dod: 1799-12-14  
fname: George  
lname: Washington  
party: null  
term: [1789-04-30, 1797-03-04]  
2:  
assassinated: false  
birthplace: Braintree, Norfolk  
birthstate: Massachusetts  
dob: 1735-10-30  
dod: 1826-07-04  
fname: John  
lname: Adams  
party: Federalist  
term: [1797-03-04, 1801-03-04]
```

Reading CSV data

- Use csv module
- Create a reader with any iterable (e.g. file object)
- Understands Excel CSV and tab-delimited files
- Can specify alternate configuration
- Iterate through reader to get rows as lists of columns

To read CSV data, use the reader() method in the csv module.

To create a reader with the default settings, use the reader() constructor. Pass in an iterable – typically, but not necessarily, a file object.

You can also add parameters to control the type of quoting, or the output delimiters.

Example

csv_read.py

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

with open('../DATA/knights.csv') as knights_in:
    rdr = csv.reader(knights_in) ①
    for name, title, color, quest, comment, number, ladies in rdr: ②
        print('{:4s} {:9s} {}'.format(
            title, name, quest
        ))
```

① create CSV reader

② Read and unpack records one at a time; each record is a list

csv_read.py

```
King Arthur    The Grail
Sir Lancelot   The Grail
Sir Robin      Not Sure
Sir Bedevere   The Grail
Sir Gawain    The Grail
```

...

Nonstandard CSV

- Variations in how CSV data is written
- Most common alternate is for Excel
- Add parameters to reader/writer

You can customize how the CSV parser and generator work by passing extra parameters to `csv.reader()` or `csv.writer()`. You can change the field and row delimiters, the escape character, and for output, what level of quoting.

You can also create a "dialect", which is a custom set of CSV parameters. The `csv` module includes one extra dialect, **excel**, which handles CSV files generated by Microsoft Excel. To use it, specify the *dialect* parameter:

```
rdr = csv.reader(csvfile, dialect='excel')
```

Table 17. CSV reader()/writer() Parameters

Parameter	Meaning
quotechar	One-character string to use as quoting character (default: "")
delimiter	One-character string to use as field separator (default: ',')
skipinitialspace	If True, skip white space after field separator (default: False)
lineterminator	The character sequence which terminates rows (default: depends on OS)
quoting	When should quotes be generated when writing CSV csv.QUOTE_MINIMAL – only when needed (default) csv.QUOTE_ALL – quote all fields csv.QUOTE_NONNUMERIC – quote all fields that are not numbers csv.QUOTE_NONE – never put quotes around fields
escapechar	One-character string to escape delimiter when quoting is set to csv.QUOTE_NONE
doublequote	Control quote handling inside fields. When True, two consecutive quotes are read as one, and one quote is written as two. (default: True)

Example

`csv_nonstandard.py`

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

with open('../DATA/computer_people.txt') as computer_people_in:
    rdr = csv.reader(computer_people_in, delimiter=';') ①

    for first_name, last_name, known_for, birth_date in rdr: ②
        print('{}: {}'.format(last_name, known_for))
```

① specify alternate field delimiter

② iterate over rows of data — csv reader is a generator

`csv_nonstandard.py`

```
Gates: Gates Foundation
Jobs: Apple
Wall: Perl
Allen: Microsoft
Ellison: Oracle
Gates: Microsoft
Zuckerberg: Facebook
Brin: Google
Page: Google
Torvalds: Linux
```

Using csv.DictReader

- Returns each row as dictionary
- Keys are field names
- Use header or specify

Instead of the normal reader, you can create a dictionary-based reader by using the DictReader class.

If the CSV file has a header, it will parse the header line and use it as the field names. Otherwise, you can specify a list of field names with the **fieldnames** parameter. For each row, you can look up a field by name, rather than position.

Example

csv_dictreader.py

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

field_names = ['term', 'firstname', 'lastname', 'birthplace', 'state', 'party'] ①

with open('../DATA/presidents.csv') as presidents_in:
    rdr = csv.DictReader(presidents_in, fieldnames=field_names) ②
    for row in rdr: ③
        print('{:25s} {:12s} {}'.format(row['firstname'], row['lastname'],
row['party'])) ④
        # string .format can use keywords from an unpacked dict as well:
        # print('{firstname:25s} {lastname:12s} {party}'.format(**row))
```

① field names, which will become dictionary keys on each row

② create reader, passing in field names (if not specified, uses first row as field names)

③ iterate over rows in file

④ print results with formatting

csv_dictreader.py

George	Washington	no party
John	Adams	Federalist
Thomas	Jefferson	Democratic - Republican
James	Madison	Democratic - Republican
James	Monroe	Democratic - Republican
John Quincy	Adams	Democratic - Republican
Andrew	Jackson	Democratic
Martin	Van Buren	Democratic
William Henry	Harrison	Whig
John	Tyler	Whig
James Knox	Polk	Democratic
Zachary	Taylor	Whig
Millard	Fillmore	Whig
Franklin	Pierce	Democratic
James	Buchanan	Democratic
Abraham	Lincoln	Republican
Andrew	Johnson	Republican
Ulysses Simpson	Grant	Republican
Rutherford Birchard	Hayes	Republican
James Abram	Garfield	Republican

...

Writing CSV Data

- Use `csv.writer()`
- Parameter is file-like object (must implement `write()` method)
- Can specify parameters to writer constructor
- Use `writerow()` or `writerows()` to output CSV data

To output data in CSV format, first create a writer using `csv.writer()`. Pass in a file-like object.

For each row to write, call the `writerow()` method of the writer, passing in an iterable with the values for that row.

To modify how data is written out, pass parameters to the writer.

TIP

On Windows, to prevent double-spaced output, add `lineterminator='\n'` when creating a CSV writer.

Example

csv_write.py

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

data = (
    ('February', 28, 'The shortest month, with 28 or 29 days'),
    ('March', 31, 'Goes out like a "lamb"'),
    ('April', 30, 'Its showers bring May flowers'),
)

with open('../TEMP/stuff.csv', 'w') as stuff_in:
    if sys.platform == 'win32':
        wtr = csv.writer(stuff_in, lineterminator='\n') ①
    else:
        wtr = csv.writer(stuff_in) ①
    for row in data:
        wtr.writerow(row) ②
```

① create CSV writer from file object that is opened for writing; on windows, need to set output line terminator to '\n'

② write one row (of iterables) to output file

Pickle

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Python uses the pickle module for data serialization.

To create pickled data, use either pickle.dump() or pickle.dumps(). Both functions take a data structure as the first argument. dumps() returns the pickled data as a string. dump () writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use pickle.load(), which takes a file-like object that has been open for writing, or pickle.loads() which reads from a string. Both functions return the original data structure that had been pickled.

NOTE

The syntax of the **json** module is based on the **pickle** module.

Example

pickling.py

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

@author: jstrick
Created on Sat Mar 16 00:47:05 2013

"""

import pickle
from pprint import pprint

①
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

data = [ ②
    colors,
    airports,
]

with open('../TEMP/pickled_data.pic', 'wb') as pic_out: ③
    pickle.dump(data, pic_out) ④

with open('../TEMP/pickled_data.pic', 'rb') as pic_in: ⑤
    pickled_data = pickle.load(pic_in) ⑥

pprint(pickled_data) ⑦
```

- ① some data structures
- ② list of data structures
- ③ open pickle file for writing in binary mode
- ④ serialize data structures to pickle file
- ⑤ open pickle file for reading in binary mode
- ⑥ de-serialize pickle file back into data structures
- ⑦ view data structures

pickling.py

```
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
 {'EWR': 'Newark',
  'IAD': 'Dulles',
  'LAX': 'Los Angeles',
  'MGW': 'Morgantown',
  'ORD': 'Chicago',
  'RDU': 'Raleigh-Durham'}]
```

Chapter 14 Exercises

Exercise 14-1 (xwords.py)

Using ElementTree, create a new XML file containing all the words that start with 'x' from words.txt. The root tag should be named 'words', and each word should be contained in a 'word' tag. The finished file should look like this:

```
<words>
    <word>xanthan</word>
    <word>xanthans</word>
    and so forth
</words>
```

Exercise 14-2 (xpresidents.py)

Use ElementTree to parse presidents.xml. Loop through and print out each president's first and last names and their state of birth.

Exercise 14-3 (jpresidents.py)

Rewrite xpresidents.py to parse presidents.json using the json module.

Exercise 14-4 (cpresidents.py)

Rewrite xpresidents.py to parse presidents.csv using the csv module.

Exercise 14-5 (pickle_potus.py)

Write a script which reads the data from presidents.csv into a dictionary where the key is the term number, and the value is another dictionary of data for one president.

Using the pickle module, Write the entire dictionary out to a file named presidents.pic.

Exercise 14-6 (unpickle_potus.py)

Write a script to open presidents.pic, and restore the data back into a dictionary.

Then loop through the array and print out each president's first name, last name, and party.

Chapter 15: 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: 48839
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Via: 1.1 vegur
Via: 1.1 varnish
Accept-Ranges: bytes
Date: Thu, 20 Feb 2020 11:37:46 GMT
Via: 1.1 varnish
Age: 1722
X-Served-By: cache-bwi5138-BWI, cache-lga21946-LGA
X-Cache: HIT, HIT
X-Cache-Hits: 3, 4
X-Timer: S1582198666.418588,VS0,VE0
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains
```

```
<!doctype html>
<!--[if lt IE 7]>    <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">    <![endif]-->
<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]-->
<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">          <![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr">  <!--<![endif]-->

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <link rel="prefetch" href="//ajax.googleapis.com/ajax/libs/jqu
```

Example

read_pdf_urllib.py

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

import sys
import os
from urllib.request import urlopen
from urllib.error import HTTPError

# url to download a PDF file of a NASA ISS brochure

url =
'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure%202012%20Screenres
%203-8-13.pdf' ①

saved_pdf_file = 'nasa_iss.pdf' ②

try:
    URL = urlopen(url) ③
except HTTPError as e: ④
    print("Unable to open URL:", e)
    sys.exit(1)

pdf_contents = URL.read() ⑤
URL.close()

with open(saved_pdf_file, 'wb') as pdf_in:
    pdf_in.write(pdf_contents) ⑥

if sys.platform == 'win32': ⑦
    cmd = saved_pdf_file
elif sys.platform == 'darwin':
    cmd = 'open ' + saved_pdf_file
else:
    cmd = 'acroread ' + saved_pdf_file

os.system(cmd) ⑧
```

① target URL

② name of PDF file for saving

- ③ open the URL
- ④ catch any HTTP errors
- ⑤ read all data from URL in binary mode
- ⑥ write data to a local file in binary mode
- ⑦ select platform and choose the app to open the PDF file
- ⑧ launch the app

Consuming Web services

- Use `urllib.parse` to URL encode the query.
- Use `urllib.request.Request`
- Specify data type in header
- Open URL with `urlopen` Read data and parse as needed

To consume Web services, use the `urllib.request` module from the standard library. Create a `urllib.request.Request` object, and specify the desired data type for the service to return.

If needed, add a `headers` parameter to the request. Its value should be a dictionary of HTTP header names and values.

For URL encoding the query, use `urllib.parse.urlencode()`. It takes either a dictionary or an iterable of key/value pairs, and returns a single string in the format "K1=V1&K2=V2&..." suitable for appending to a URL.

Pass the Request object to `urlopen()`, and it will return a file-like object which you can read by calling its `read()` method.

The data will be a bytes object, so to use it as a string, call `decode()` on the data. It can then be parsed as appropriate, depending on the content type.

NOTE

the example program on the next page queries the 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

    with open(saved_pdf_file, 'wb') as pdf_in: ⑤
        pdf_in.write(response.content) ⑥

    if sys.platform == 'win32': ⑦
        cmd = saved_pdf_file
    elif sys.platform == 'darwin':
        cmd = 'open ' + saved_pdf_file
    else:
        cmd = 'acroread ' + saved_pdf_file

    os.system(cmd) ⑧
```

- ① target URL
- ② name of PDF file for saving
- ③ open the URL
- ④ check status code
- ⑤ open local file
- ⑥ write data to a local file in binary mode; response.content is data from URL
- ⑦ select platform and choose the app to open the PDF file
- ⑧ launch the app

Example

`web_content_consumer_requests.py`

```
#!/usr/bin/env python
from pprint import pprint
import sys
import requests

BASE_URL = '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:
        # pprint(response.content.decode())
        # print('-' * 60)
        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('f1'))
    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 "maker's mark"

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 18. 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.

Once 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".

Paramiko is used by Ansible and other sys admin tools.

NOTE

[Find out more about paramiko at http://www.lag.net/paramiko/](http://www.lag.net/paramiko/)

[Find out more about Ansible at http://www.ansible.com/](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 8
drwx-----+ 3 python  staff  96 Apr 16 2014 Desktop
drwx-----+ 3 python  staff  96 Apr 16 2014 Documents
drwx-----+ 4 python  staff  128 Apr 16 2014 Downloads
drwx-----+ 28 python  staff  896 Dec 17 2017 Library
drwx-----+ 3 python  staff  96 Apr 16 2014 Movies
drwx-----+ 3 python  staff  96 Apr 16 2014 Music
drwx-----+ 3 python  staff  96 Apr 16 2014 Pictures
drwxr-xr-x+ 5 python  staff  160 Apr 16 2014 Public
-rw-r--r--  1 python  staff  519 Jul 27 2016 remote_processes.py
drwxr-xr-x  2 python  staff   64 Jan  1 17:55 text_files

-----
STDOUT:
-rw-r--r-- 1 root  wheel  6946 Oct 17 18:39 /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('../DATA/alice.txt', remote_file) ⑦
    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)

stdin, stdout, stderr = ssh.exec_command('rm -f
{}-betsy.txt'.format(REMOTE_DIR))
stdin, stdout, stderr = ssh.exec_command('rmdir {}'.format(REMOTE_DIR))
stdin, stdout, stderr = ssh.exec_command('ls -l')
print(stdout.read().decode())
print('-' * 60)
```

- ① create paramiko Transport instance
- ② connect to remote host
- ③ create SFTP client using Transport instance
- ④ get list of items on default (login) folder (listdir_iter() returns a generator)
- ⑤ create 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----- 1 504 20 96 16 Apr 2014 Music
-rw----- 1 504 20 3 16 Apr 2014 .CFUserTextEncoding
drwx----- 1 504 20 96 16 Apr 2014 Pictures
drwxr-xr-x 1 504 20 64 01 Jan 17:55 text_files
drwx----- 1 504 20 96 16 Apr 2014 Desktop
drwx----- 1 504 20 896 17 Dec 2017 Library
drwxr-xr-x 1 504 20 160 16 Apr 2014 Public
drwx----- 1 504 20 96 06 Feb 2015 .ssh
drwx----- 1 504 20 96 16 Apr 2014 Movies
drwx----- 1 504 20 96 16 Apr 2014 Documents
-rw-r--r-- 1 504 20 519 27 Jul 2016 remote_processes.py
drwx----- 1 504 20 128 16 Apr 2014 Downloads
-rw----- 1 504 20 2614 23 May 2019 .bash_history
```

Chapter 15 Exercises

Exercise 15-1 (`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 15-2 (`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 BeautifulSoup 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 15-3 (`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 16: Accessing git

Objectives

- Access git repos
- Execute various git commands
- Create and update git repos

What is source code control?

- Tracks changes to files
- Allows collaboration
- Provides for tagging a repo at given point

A source code control system tracks changes to files. When a file is first created, and then at any point afterwards, the file is *committed* to the code repository. Any previously committed version can be retrieved.

At a particular point in development, when all of the files are ready for production (or beta testing), you can *tag* the current state of the repository as "version X.y", so you can retrieve all files that comprise a particular version of the software.

Such systems are also designed for multiple developers to collaborate on a set of source files.

Other source code control systems include Subversion, Mercurial, and Microsoft Team Foundation Server.

What is git?

- Distributed source code control
- Keeps track of changes to your code
- Created by Linux Torvalds

git, created by Linux Torvalds, is the most popular modern source code control system.

git is distinguished from other systems by having no official "main" repository. It is designed for easy collaboration, and to make it easy to merge changes from files being updated by multiple developers.

With git, every developer has their own repo, and commits locally. Developers can also update to or from other repos.

git workflow

- Getting started
 - git init
- Saving changes
 - git add
 - git commit
- Other commands

The normal git workflow can be divided into several areas.

Getting started

To get started, create a repo with `git init`. Then add any existing files with `git add ...` to stage the files (tell git to track them) and `git commit ...` (tell git to add to or update the repo).

Then as you create new files or make changes to files, commit the changes by repeating `git add ...` and `git commit ...`. You can update another repo with your changes with `git push`, or update your repo with someone else's with `git pull`.

Branching and merging

At some point, you'll want to create a *branch*. A branch is a separate copy of the repository. After working on the branch (e.g. for a bug fix or a new feature), you can *merge* your changes back into the primary repo.

Tagging

At any time, you can *tag* the repo with a label (e.g. "1.0") across all files, so that you can retrieve your entire project with a known state.

Using the command line

- git *command*
- Dozens of commands

A common way to use git is from the command line. git has many commands, each of which has options, arguments or subcommands.

For instance, to create a new git repository and add existing files:

```
git init  
git add .  
git commit -a -m "initial commit"
```

Later, you would typically repeat the following

```
git add foobar.py  
git commit -m "added foobar.py" foobar.py
```

or

```
git add .  
git commit -a -m "fixed issue #1234"
```

Common git commands: **add, commit, push, status, merge, branch, pull, fetch, clone**

TIP

Many IDEs, such as PyCharm, Visual Studio Code, and Eclipse, have git integration built in.

NOTE

see <https://git-scm.com/doc> for the full git documentation

Using the *gitpython* module

- Install **gitpython**
- Create Repo object
 - from path to repo
 - init new repo
- Create Git object from repo
- Methods map to git cli commands

The **gitpython** module is a convenient wrapper for the git command line utility. To install with **pip**:

```
pip install gitpython
```

To get started, import the **Git** and **Repo** objects from **git**.

For an existing repo, initialize Repo with the path the repo. For a new repo, call Repo.init() with the desired path.

```
repo = Repo('/path/to/repo')
```

You can get some information, such as untracked files, from the Repo object, but for most tasks you'll need a Git object, which is initialized from the Repo object.

The Git object has methods that map to the commands of the **git** utility.

```
git = Git(repo)
print(git.status())
```

Example

git_status.py

```
#!/usr/bin/env python
import os
from git import Git, Repo

repo_dir = 'myproject' ①

repo = Repo(repo_dir) ②

os.chdir(repo_dir) ③

g = Git(repo) ④
print(g.status()) ⑤
```

① Folder of repository

② Create Repo object

③ Go to repo folder (not always required)

④ Create Git() object; this is used for most git commands

⑤ Get status

git_status.py

```
On branch myfeature
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

list_comprehensions.py

```
nothing added to commit but untracked files present (use "git add" to track)
```

Getting repo info

- Use Repo object
- Many methods and properties
- Use `git.log()` for file changes

To get general information about a repo, you can access and instance of `git.Repo`.

Some of the information available is * Untracked files * List of branches * Whether the repo is "dirty" (has modified files) * List of remote repos

Use `git.log()` to print the log of recent changes to the codebase.

Example

git_info.py

```
#!/usr/bin/env python
from git import Repo

repo_dir = 'myproject'

repo = Repo(repo_dir)

print("Untracked files:", repo.untracked_files, '\n') ①

print("Branches:")
for branch in repo.branches: ②
    print(branch.name, branch.path)
print()

print(repo.is_dirty(), '\n') ③

print(repo.remotes) ④
```

① Get list of untracked files

② Iterate over branch objects

③ Are there any changed files?

④ Get list of remotes

git_info.py

```
Untracked files: ['list_comprehensions.py']

Branches:
master refs/heads/master
myfeature refs/heads/myfeature

False

[<git.Remote "origin">]
```

Example

git_log.py

```
#!/usr/bin/env python
import os
from git import Git, Repo

repo_dir = 'myproject'

repo = Repo(repo_dir)

os.chdir(repo_dir)

g = Git(repo)
print(g.log()) ①
print('-' * 60)
print(g.log('unicode.py')) ②
print('-' * 60)
print(g.log('fizzbuzz1.py')) ②
```

① get log for entire repo

② get log for specific file

git_log.py

```
commit 8739fe57a40029c0d02eb1b1a7a969d957647bc8
Author: jstrickler <jstrickler@gmail.com>
Date:   Thu Feb 20 05:35:22 2020 -0500
```

new file

```
commit 6adcd0fb72dd9aa8595e8a80cfa45c9cb13981ba
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:52:33 2020 -0500
```

added hello.py

```
commit f631c7e5ca139d8be637caaae9d7aa7685386bd8
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:50:47 2020 -0500
```

added hello.py

```
commit c46bef1c9506a157c18813654065346aa6e256ea
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:33:05 2020 -0500
```

added exploding head emoji

```
commit 1b1c2686a8df4817de84f7a3027f6eb3f1412f4f
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:31:38 2020 -0500
```

initial commit

```
commit 6adcd0fb72dd9aa8595e8a80cfa45c9cb13981ba
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:52:33 2020 -0500
```

added hello.py

```
commit c46bef1c9506a157c18813654065346aa6e256ea
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:33:05 2020 -0500
```

added exploding head emoji

```
commit 1b1c2686a8df4817de84f7a3027f6eb3f1412f4f
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:31:38 2020 -0500
```

initial commit

```
commit 1b1c2686a8df4817de84f7a3027f6eb3f1412f4f
Author: jstrickler <jstrickler@gmail.com>
Date:   Wed Feb 19 06:31:38 2020 -0500
```

initial commit

Creating a repo

- Use `repo.init()`
- Stage files with `git.add()`
- Commit files with `git.commit()`

Use `repo.init()` to create a new repo.

Use `git.add(filename)` to stage a particular file for commit, or `git.add('.)` to stage all untracked files in the repo.

Use `git.commit(filename, "commit message")` to commit one file, or `git.commit('.', "commit message")` to commit changes to all files that have been staged.

TIP

Create a `.gitignore` file in the repo containing files and folders that git should not track.

TIP

Use `os.chdir()` to change directory to the repo to avoid accidentally using a repo in a parent folder.

Example

git_complete.py

```
#!/usr/bin/env python
import os
import shutil
from git import Git, Repo

repo_dir = 'newproject'

r = Repo.init(repo_dir) ①
os.chdir(repo_dir)

file_to_add = 'creating_dicts.py'
g = Git(r) ②
shutil.copy('../' + file_to_add, ".") ③

g.add(file_to_add) ④
g.commit(file_to_add, message="initial commit") ⑤
print(g.log()) ⑥
```

- ① Create new empty repo
- ② Create Git() object from repo
- ③ Copy file to add (typically created with IDE, so no need to copy)
- ④ Stage file for commit
- ⑤ Commit file
- ⑥ Show repo log

Pushing to remote repos

- Use `_repo_.add_remote()`

An important feature of `git` is the ability to update another repo from your local repo. Even though `git` is decentralized (i.e., does not depend on a central server), most organizations designate a "primary" repo that has the most up-to-date versions of a codebase.

The non-local repo is called, appropriately enough, the *remote*.

The remote repo must exist.

To add a remote, use `git.remote('add', remote_name, remote)`, where *name* is the name and *remote* is the path (if the remote is on the same computer) or URL (if on a different machine) of the remote repo.

```
g.remote('add', 'origin', 'https://github.com/jstrickler/myproject.git') ⑤
```

To automatically track the remote (i.e., `push` will default to the remote

```
g.push('-u', 'origin', 'master')
```

Now, `git.push()` with no arguments will update the remote.

You can have multiple remotes. In this case pass the name of the remote to `git.push()`.

```
git.push() # push local commits to configured remote  
git.push("myremote") # push to specific remot
```

Example

git_remote.py

```
#!/usr/bin/env python
import os
from git import Git, Repo

repo_dir = 'myproject'

repo = Repo(repo_dir)

os.chdir(repo_dir)

g = Git(repo)

g.remote('add', 'origin', 'https://github.com/jstrickler/myproject.git') ①

g.push('-u', 'origin', 'master') ②
```

① Add remote repo named 'origin'

② Push local files to remote and track 'origin' as the "master" repo so push() will automatically use it

Cloning repos

- Use Repo.clone_from()

To clone (make a local copy of) a repo, use Repo.clone_from() with the URL of the repo you want to clone and the local folder to clone into.

```
Repo.clone_from("remote-URL", "local-path")
```

Example

git_clone.py

```
#!/usr/bin/env python
import os
from git import Git, Repo

repo_dir = 'myproject'

repo = Repo(repo_dir)

os.chdir(repo_dir)

Repo.clone_from(
    'https://github.com/jstrickler/myproject.git',
    '../TEMP/myprojectcopy'
) ①
```

① Clone existing repo to new repo

Working with branches

- Create branch for a particular task
- Merge branch back into main stream
- Use `git.checkout()`

Branches are used to make a "private" copy of a repo. git tracks changes to each branch separately, so updates to a branch don't affect other branches. This is so developers can collaborate on a codebase without changing each other's files.

To create a new branch and begin working on it:

```
git.checkout('-b', 'branch-name')
```

The name can be anything you want.

To work on an existing branch:

```
git.checkout('branch-name')
```

NOTE

See <https://nvie.com/posts/a-successful-git-branching-model/> for a good description of `git` workflow with branches.

Example

git_branch.py

```
#!/usr/bin/env python
import os
from git import Git, Repo

repo_dir = 'myproject'

repo = Repo(repo_dir)

os.chdir(repo_dir)

g = Git(repo)

g.checkout('-b', 'myfeature') ①

g.add('tyger.py') ②
g.commit('tyger.py', message='new file')
```

① Create new branch named "myfeature"

② tyger.py is only added and committed to branch "myfeature", not main branch

Tagging

- Creates a snapshot of the repo
- Mark all files in repo
- Creates specific version of codebase
- Use `repo.create_tag()`

At some point you'll want to "tag" the current state of a repository. This typically is tied to a "release" of a particular version of your project. A typical tag might be "v1.2.7a", representing major, minor, and micro versions of the project. However, the tag can be anything you want, such as "Beta9" or "weaseldust".

Tags are created from the `Repo` object, with `create_tag()`.

```
repo.create_tag('v1.2.7a')
```

Once you have created a tag, you can then fetch, pull, or clone the repo and specify the tag. This will use the versions of the files at the time the tag was added.

Example

git_create_tag.py

```
#!/usr/bin/env python
import os
from git import Repo

repo_dir = 'myproject'

repo = Repo(repo_dir)

repo.create_tag("v1.0.0", message="First Public Release") ①
```

① Create tag with descriptive message

Example

git_list_tags.py

```
#!/usr/bin/env python
from git import Repo

repo = Repo('myproject')

for tag in repo.tags: ①
    print(tag.name)
```

① Iterate over all tags for repo

git_list_tags.py

```
v1.0.0
```

Exercise 16-1 (funwithgit.py)

Clone <https://github.com/jstrickler/myproject.git> into a folder named myclone. Using PyCharm (or other IDE), create a new file in that folder named "your_name.py" (this is so everyone doesn't use the same name). Add the new file, commit the file, and finally push the file to the remote.

Chapter 17: Using Docker from Python

Objectives

- Understand what Docker is (and isn't)
- Use the Python Docker API
- List and run containers
- Create and provision containers via python scripts

What is Docker?

- Container for running programs
- Lighter weight than full VM
- Extremely flexible

Docker is an open-source package that makes it easy to deploy applications inside a specified software environment, known as a *container*.

Unlike using a virtual machine (VM), Docker shares the *same* OS kernel across all containers. This saves not only gigabytes of memory, but many machine resources as well. This lower resource requirement can save money when deploying containers in the cloud.

A docker container boots very quickly compared to a VM, less than a second for most images, compared to many seconds or even minutes for VMs.

The advantage of Docker is providing a consistent environment for software to run, independent of any upgrades, installations, or other changes to the "real" environment.

NOTE

There are other containerization platforms, but Docker is by far the most commonly used.

Docker terminology

Docker engine

Client/server app that makes Docker possible

Docker daemon

Docker server that manages containers, images, and other objects

Docker client

API for interacting with Docker. One implementation is the CLI

Image

Container template (read-only). Containers are built from images; new images may also start with an existing image

Container

Runnable instance of an image

Creating and running a client

- Use `docker.from_env()`
- Add parameters as needed

To create a new client, use `docker.from_env()`. This will create a new local client from which you can run, list, and provision clients. The client is configured from standard Docker environment variables:

DOCKER_HOST

The URL of the Docker host

DOCKER_TLS_VERIFY

Verify the host against a CA certificate

DOCKER_CERT_PATH

Path to folder with TLS certs

To add a timeout to containers that are run, add the `timeout` parameter.

Other parameters give more control over the client.

Table 19. Parameters for `from_env()`

Parameter	Type	Description
version	str	What version of Docker to use; defaults to auto
timeout	int	Default timeout (in seconds) for API calls
ssl_version	int	SSL version
assert_hostname	bool	Verify server hostname
environment	dict	Where to get environment variables; defaults to os.environ
credstore_env	dict	Values to override variables when calling credential store

Example

docker_simple.py

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

client = docker.from_env() ①

result = client.containers.run("ubuntu", 'echo "Hello, world"') ②
print(result, '\n') ③
print(result.decode(), '\n') ④
print('-' * 60)

result = client.containers.run("ubuntu", 'ls -l') ②
print(result, '\n') ③
print(result.decode(), '\n') ④
```

① Create local client

② Run a container

③ Result is a bytes objects

④ Use .decode() to convert bytes object to Python string

docker_simple.py

```
b'Hello, world\n'
```

```
Hello, world
```

```
-----  
b'total 64\ndrwxr-xr-x  2 root root 4096 Jan 12 21:10 bin\ndrwxr-xr-x  2 root root  
4096 Apr 24 2018 boot\ndrwxr-xr-x  5 root root 340 Feb 20 11:37 dev\ndrwxr-xr-x  
1 root root 4096 Feb 20 11:37 etc\ndrwxr-xr-x  2 root root 4096 Apr 24 2018  
home\ndrwxr-xr-x  8 root root 4096 May 23 2017 lib\ndrwxr-xr-x  2 root root 4096  
Jan 12 21:10 lib64\ndrwxr-xr-x  2 root root 4096 Jan 12 21:09 media\ndrwxr-xr-x  2  
root root 4096 Jan 12 21:09 mnt\ndrwxr-xr-x  2 root root 4096 Jan 12 21:09 opt\ndr-  
xr-xr-x 160 root root 0 Feb 20 11:37 proc\ndrwx----- 2 root root 4096 Jan 12  
21:10 root\ndrwxr-xr-x  1 root root 4096 Jan 16 01:20 run\ndrwxr-xr-x  1 root root  
4096 Jan 16 01:20 sbin\ndrwxr-xr-x  2 root root 4096 Jan 12 21:09 srv\ndr-xr-xr-x  
13 root root 0 Feb 16 20:24 sys\ndrwxrwxrwt  2 root root 4096 Jan 12 21:10  
tmp\ndrwxr-xr-x  1 root root 4096 Jan 12 21:09 usr\ndrwxr-xr-x  1 root root 4096  
Jan 12 21:10 var\n'
```

```
total 64
```

```
drwxr-xr-x  2 root root 4096 Jan 12 21:10 bin  
drwxr-xr-x  2 root root 4096 Apr 24 2018 boot  
drwxr-xr-x  5 root root 340 Feb 20 11:37 dev  
drwxr-xr-x  1 root root 4096 Feb 20 11:37 etc  
drwxr-xr-x  2 root root 4096 Apr 24 2018 home  
drwxr-xr-x  8 root root 4096 May 23 2017 lib  
drwxr-xr-x  2 root root 4096 Jan 12 21:10 lib64  
drwxr-xr-x  2 root root 4096 Jan 12 21:09 media  
drwxr-xr-x  2 root root 4096 Jan 12 21:09 mnt  
drwxr-xr-x  2 root root 4096 Jan 12 21:09 opt  
dr-xr-xr-x 160 root root 0 Feb 20 11:37 proc  
drwx----- 2 root root 4096 Jan 12 21:10 root  
drwxr-xr-x  1 root root 4096 Jan 16 01:20 run  
drwxr-xr-x  1 root root 4096 Jan 16 01:20 sbin  
drwxr-xr-x  2 root root 4096 Jan 12 21:09 srv  
dr-xr-xr-x 13 root root 0 Feb 16 20:24 sys  
drwxrwxrwt  2 root root 4096 Jan 12 21:10 tmp  
drwxr-xr-x  1 root root 4096 Jan 12 21:09 usr  
drwxr-xr-x  1 root root 4096 Jan 12 21:10 var
```

List containers

- Use `client.containers.list()`
- Returns list of container objects

`client.containers.list()` will return a list of currently running container objects. Each object has many attributes, such as name and id.

Example

`docker_listing.py`

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

client = docker.from_env() ①

client.containers.run('ubuntu', 'sleep 30', detach=True)
client.containers.run('ubuntu', 'sleep 30', detach=True)
client.containers.run('ubuntu', 'sleep 30', detach=True)

for container in client.containers.list():
    print(container)
```

① Create local client

`docker_listing.py`

```
<Container: 398ab8dd43>
<Container: 09d8d3ab9f>
<Container: a5e746a0de>
```

Table 20. Container attributes

id	Container ID
image	container image
labels	Container labels, as a dict
name	Container name
short_id	Container ID truncated to 10 chars
status	Container status (e.g. running or exited)
attach()	Attach to this container.
attach_socket()	Like attach(), but returns underlying socket-like object
commit()	Commit container to image (cf. docker commit)
diff()	Report changes to container's filesystem
export()	Export contents of container's filesystem
get_archive()	Retrieve file or folder from the container as tar file
kill()	Kill (or send signal to) container.
logs()	Get container's logs
pause()	Pause all container processes
put_archive()	Insert file or folder in container
reload()	Reload this object from server
remove()	Remove container
rename()	Rename container
resize()	Resize tty session
restart()	Restart container
start()	Start container
stats()	Stream stats for container
stop()	Stop container
top()	Display running processes

id	Container ID
unpause()	Unpause all processes
update()	Update resource config
wait()	Block until container stops

Working with images

- Use `docker.images.*`
 - `.list()` to list
 - `.pull()` to download
 - `.remove()` to delete

The `images` object has methods for working with images. For instance, `.pull()` will download and install an image from DockerHub. `.remove` will remove any image.

Example

`docker_images.py`

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

client = docker.from_env() ①

for image in client.images.list(): ②
    print(image.tags, image.short_id) ③

result = client.images.pull("python") ④
print(result) ⑤
```

- ① Create local client
- ② Iterate over currently running clients
- ③ Use client data as needed
- ④ Pull an image from the repository
- ⑤ Show the contents of the image

docker_images.py

```
[ 'python-fizz:latest' ] sha256:f5bbb12a93
[ 'python-fizzbuzz:latest' ] sha256:748c61b546
[] sha256:acd2b798b9
[ 'python:2-alpine3.10' ] sha256:868f6308b8
[ 'python:2-alpine', 'python:2-alpine3.11' ] sha256:7ec8514e7b
[ 'python:2-slim-stretch' ] sha256:f97540f370
[ 'python:2-stretch' ] sha256:7e525a2edc
[ 'python:2-slim', 'python:2-slim-buster' ] sha256:814454f69a
[ 'python:2-buster' ] sha256:09a181e16b
[ 'ubuntu:latest' ] sha256:ccc6e87d48
[ 'python:2-alpine3.9' ] sha256:3f0e580ded
[ 'continuumio/anaconda3:latest' ] sha256:5fbf7bac70
[ 'python:2-slim-jessie' ] sha256:14c92db9df
[ 'python:2-jessie' ] sha256:96f10bf3d8
[ 'python:2-alpine3.8' ] sha256:fcbce699af
[ 'python:2-wheezy' ] sha256:008e192e2e
[ 'python:2-alpine3.6' ] sha256:dd4c05fcb8
[ 'python:2-alpine3.7' ] sha256:ec0218200a
[ 'requestbin_app:latest' ] sha256:03152fd310
[ 'python:2.7-alpine' ] sha256:7c306adf1b
[ 'python:2-onbuild' ] sha256:3f246dd60a
[ 'busybox:latest' ] sha256:8c811b4aec
[ 'python:2-alpine3.4' ] sha256:5fdd069daf
[ 'hello-world:latest' ] sha256:e38bc07ac1
[<Image: 'python:2-alpine3.10'>, <Image: 'python:2-alpine', 'python:2-alpine3.11'>,
<Image: 'python:2-slim-stretch'>, <Image: 'python:2-stretch'>, <Image: 'python:2-slim',
 'python:2-slim-buster'>, <Image: 'python:2-buster'>, <Image: 'python:2-alpine3.9'>,
 <Image: 'python:2-slim-jessie'>, <Image: 'python:2-jessie'>, <Image: 'python:2-alpine3.8'>,
 <Image: 'python:2-wheezy'>, <Image: 'python:2-alpine3.6'>,
 <Image: 'python:2-alpine3.7'>, <Image: 'python:2.7-alpine'>, <Image: 'python:2-onbuild'>,
 <Image: 'python:2-alpine3.4'>]
```

Table 21. docker.images attributes

Attribute	Description
build()	Build image and return it
get()	Retrieve an image
get_registry_data()	Retrieve image registry data
list()	List images on server
load()	Load previously saved image
prune()	Delete unused images
pull()	Pull image from repository
push()	Push image or repository to registry
remove()	Remove image
search()	Find image on Docker Hub

Building an image

- Use `client.images.build()`
- Need
 - Folder for image
 - Dockerfile
 - Script to run

You can build a new container using `client.images.build()`. The minimum it needs is the path to the docker image folder, but you probably want to add at least a tag, using the **tag** parameter.

You can use Python tools to copy needed items into the folder, including the **Dockerfile** itself. **shutil** is a great module for copying files, and is more portable than using `subprocess.run()` with external copy commands.

Example

docker_build.py

```
#!/usr/bin/env python
import os
import shutil
import docker

PYTHON_SCRIPT = "fizzbuzz1.py"    ①
DOCKER_DIR = "docker_build"      ②
DOCKERFILE = "../DATA/Dockerfile" ③

client = docker.from_env()        ④

os.makedirs(DOCKER_DIR, exist_ok=True) ⑤
shutil.copy(DOCKERFILE, DOCKER_DIR)    ⑥
shutil.copy(PYTHON_SCRIPT, DOCKER_DIR) ⑦

image, log_gen = client.images.build( ⑧
    path=DOCKER_DIR,
    tag="python-fizz",
)
output = client.containers.run('python-fizz') ⑨
print(output.decode())
print()

output = client.containers.run(image) ⑩
print(output.decode())
```

① Python script to deploy

② Build folder for image

③ Dockerfile to copy into build folder

④ Create Docker client

⑤ Create build folder if it doesn't exist

⑥ Copy Dockerfile to build folder

⑦ Copy python script to build folder

⑧ Build the image and add to local repository; an image object is returned

⑨ Run the image by tag

⑩ Run the image object

docker_build.py

```
1, 2, FIZZ, 4, BUZZ, FIZZ, 7, 8, FIZZ, BUZZ, 11, FIZZ, 13, 14, FIZZBUZZ, 16, 17,  
FIZZ, 19, BUZZ, FIZZ, 22, 23, FIZZ, BUZZ, 26, FIZZ, 28, 29, FIZZBUZZ, 31, 32, FIZZ,  
34, BUZZ, FIZZ, 37, 38, FIZZ, BUZZ, 41, FIZZ, 43, 44, FIZZBUZZ, 46, 47, FIZZ, 49,  
BUZZ,
```

```
1, 2, FIZZ, 4, BUZZ, FIZZ, 7, 8, FIZZ, BUZZ, 11, FIZZ, 13, 14, FIZZBUZZ, 16, 17,  
FIZZ, 19, BUZZ, FIZZ, 22, 23, FIZZ, BUZZ, 26, FIZZ, 28, 29, FIZZBUZZ, 31, 32, FIZZ,  
34, BUZZ, FIZZ, 37, 38, FIZZ, BUZZ, 41, FIZZ, 43, 44, FIZZBUZZ, 46, 47, FIZZ, 49,  
BUZZ,
```

Chapter 17 Exercises

Exercise 17-1

Using the Python Docker API, create a new Docker image that will run one of the scripts you created earlier in class.

List all images to confirm your image got built.

Run the new image in a container.

Working with Flask

The following chapters describe using Flask for RESTful services

Chapter 18: Diving Right In

Objectives

- Using variables
- Understanding the "big picture" of Flask
- Creating a minimal Flask app
- Connecting a route to a view function
- Starting the development server

Flask "Hello, world!"

- Import Flask
- Create app object
- Map a view function to a route
- Start the app

The simplest Flask application is just a single page view. Create an instance of Flask (AKA a Flask app). Define a view function that returns some HTML. Use the route decorator to map the route '/' to your function.

Start the app.

Wasn't that easy?

TIP

A good place for Flask support is <https://www.reddit.com/r/flask/>

Example

flask_hello.py

```
#!/usr/bin/env python
from flask import Flask

app = Flask(__name__) ①

@app.route('/') ②
def index(): ③
    return '<h1>Hello, Flask world!</h1>' ④

# app.register_route(index, '/')

if __name__ == '__main__':
    app.run(debug=True) ⑤
```

Creating a Flask app

- Import flask
- Instantiate a Flask instance
- First argument is the app name To get started with Flask, import the flask module, and create a Flask object.

The first argument to Flask() is the name of the application. If you are creating a simple app contained in just one script, you should use the *name* builtin variable

```
from flask import Flask
app = Flask(__name__)
```

If your script is part of a package, you should hard-code the package name. That is, if your application (and package) is named wombat, then you might put your main code in app.py, in the wombat package, so the file layout is:

```
wombat/
    __init__.py      # package init
    app.py          # main module
```

In this case, you should hard-code the app name into the package:

```
from flask import Flask
app = Flask('wombat')
```

This will make life easier as your application grows more complex.

The app object will contain the configuration for your app, as well as the registered view functions, the URL rules, the templates, extensions, and more.

The Flask() constructor takes additional parameters to control the location of templates, static files, and other items.

NOTE

The name "app" is not required; it could be anything, but "app" is short and convenient.

Defining view functions

- Returns content to browser
- Typically returns HTML
- Can return nearly anything

A view function is just a function that returns content to the web browser or other HTTP client.

A view function typically returns an HTTP header, some HTML content, and an HTTP status code. Flask takes care of some of this for you; in simple cases you can just return HTML.

Usually, though, you will use a template to create the HTML, so you can embed data in the page and have a consistent look-and-feel across your application.

Example

```
def index():
    return '<h1>Hello, Flask world!</h1>'
```

Configuring Routes

- Associate a URL with a function
- Uses a decorator

Just defining view functions is not enough. An app will certainly have more than one view function. Each view function must be associated with a specific URL in your application.

Flask makes this easy by providing a `route()` decorator which is part of your `app` object. The decorator takes the URL (aka route) as its argument, and then, when the web server asks for a particular URL, it is passed via the WSGI server to the Flask app, which then parses the route and selects the appropriate function.

Example

```
@app.route('/')
def index():
    return '<h1>Hello, Flask world!</h1>'

@app.route('/president/<int:termnum>/')
def president_by_term(termnum):
    term = int(termnum)
    p = President(term)
    return render_template('president_results.html', presidents=[p])
```

Deploying the app

- Flask provides development server
- Do NOT use in production
- Just call run() on the app object

Flask provides a development WSGI server for convenience. This can be used while you are getting your application ready for production.

Do NOT use the development server in any public setting. It does not have appropriate security features. We will cover real-world deployment later in the course.

To use the development server, just call the run() method on your application object:

```
if __name__ == '__main__':
    app.run(debug=True)
```

This statement is normally wrapped in "if *name* == 'main'". This ensures that the app server will only start when the script is run directly, as opposed to being imported by some other script.

The debug flag should be set to True for debugging – it will fire up the builtin debugger when your code has an unhandled exception, and it will also reload the server when your source code changes. This is a great time-saver, eliminating the need to kill and restart the server whenever you make a change in the source.

We will be looking at more flexible ways of starting your app in a few chapters.

There are many other methods that can be called from the application object.

Chapter 18 Exercises

Exercise 18-1 (powers_of_two.py)

Create a Flask app that displays a single page containing the powers of two from 2^0 through 2^{31} .

Chapter 19: Creating Models

Objectives

- Understand the concept of models
- Connect to a SQLite or PostgreSQL database
- Provide forms for retrieving and updating the database
- Learn about other options for models

Creating models for a Flask Application

- Models not specified by Flask
- Use any DB API module
- Use ORM (SQLAlchemy)

Flask is not integrated with any database managers. The developer is free to use any database package, or implement a custom package.

Given that, most developers do use existing database tools.

For direct access to databases, the many modules that implement the DB API are available. Typically, you might abstract these databases into classes that hide the DB details from Flask.

You could also use the Flask-SQLAlchemy extension, and use ORM to create classes that represent database tables. In this case, SQLAlchemy will generate the actual SQL commands based on DB classes that you define. If you already have a database you want to use, SQLAlchemy can automatically create classes that map to the existing tables.

For NoSQL databases, the extensions Flask-CouchDB and Flask-PyMongo provide access,

Connecting to a database

- Typically connect in the main script or *init.py*
- Connection details from environment or config.py

A Flask app typically connects to the database at the main script level, maybe in *init.py*, depending on the project layout.

This connection can then be shared by different modules in the project.

You should not store sensitive information like passwords in source code, so that information can come from a separate file, user input, or a config module that is part of your project.

Creating a get_db() function

- Convenient to make a universal function
- Use the g (global) variable from the app context

For some apps, it is convenient to create a global function that will retrieve the database connection.

You might also create a function that executes a query and returns the result set as tuples, dictionaries, or whatever your app needs. It would combine the execute() and fetch() methods.

Most apps, however, create the database object in the models file, and import it where needed.

Using Flask-SQLAlchemy

- Create models (classes that map to DB tables)
- Generates SQL as needed
- Can create DB from scratch
- Interact with DB via objects, not SQL

SQL-Alchemy is a popular choice for supporting Flask apps. It is relatively easy to use, and works with SQLite3, MySQL, PostgreSQL, Oracle, Microsoft SQL Server, and others.

Creating models

- Create class that inherits from db.Model
- Add columns as class variables
- Add str() or repr() methods as desired
- Add constructor (*init*) to populate the model

To create a model in SQL Alchemy, define a class that inherits from the Model base class. To add columns, add class variables of type Column. The arguments to Column define the data type and other properties of the column

Be sure to add an ID field in each Model.

Add a *repr* method for displaying the raw model.

See the SQLAlchemy docs for details on defining models

Initializing the database

- Call `create_all()` from the DB object

Once the models are defined, call `create_all()` to generate and execute the SQL to build the initial database.

TIP

You could add something like `initialize_db` as a command in your manage script

Example

presidents_one/models.py

```
#!/usr/bin/env python
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

print(db.Model.metadata.tables.items())

class President(db.Model):
    __table__ = db.Model.metadata.tables['president']

    def __repr__(self):
        return '<President {} {}>'.format(self.firstname, self.lastname)

# class President(db.Model):
#     id = db.Column(db.Integer, primary_key=True)
#     termnum = db.Column(db.Integer, primary_key=True)
#     fname = db.Column(db.String(80))
#     lname = db.Column(db.String(80))
#     dbirth = db.Column(db.Date())
#     ddeath = db.Column(db.Date())
#     birthplace = db.Column(db.String(80))
#     birthstate = db.Column(db.String(80))
#     dstart = db.Column(db.Date())
#     dend = db.Column(db.Date())
#     party = db.Column(db.String(32))
#
#     def __repr__(self):
#         return '<President {} {}>'.format(self.fname, self.lastname)
```

Adding and accessing data

- To use the database, create instances of models
- Add model instances to db session with db.session.add()
- Call session.commit() to save all added instances

To add data to the database, create an instance of a model, and populate it with appropriate data. Add the instance to the database session. When one or more instances have been added, use db.session.commit() to save all instances added since the last commit.

Example

presidents_one/main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from models import db, President

app = Flask(__name__)

# in Real Life, get from config or file or environment
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///DATA/presidents.db'

db.init_app(app)

@app.route('/')
def index():
    return("<h1>Try /president/#</h1>")

# @app.route('/initdb')
# def initdb():
#     db.create_all()
#     return("<h1>Database initialized</h1>")

@app.route('/president/<int:termnum>')
def show_pres(termnum):
    # select from president ....
```

```
p = President.query.filter(President.termnum == termnum).first()
if p:
    html = '<html><head><title>President
#{}</title></head><body>'.format(termnum)
    html += 'President #{}<br/>\n'.format(termnum)
    html += 'Name: {} {}<br/>\n'.format(p.fname, p.lname)
    html += 'Lived: {} to {}<br/>\n'.format(p.dbirth, p.ddeath)
    html += 'Born in: {}, {}<br/>\n'.format(p.birthplace, p.birthstate)
    html += 'Served: {} to {}<br/>\n'.format(p.dstart, p.dend)
    html += 'Party: {}<br/>\n'.format(p.party)
    html += '</body></html>'

    return html, 200
else:
    return "No such president", 200

if __name__ == '__main__':
    app.run(debug=True)
```

Chapter 19 Exercises

Exercise 19-1 (`knights_one/*`)

Create a Flask project that creates a SQLite3 (or Postgres or MySQL) database to hold information on, you guessed it, knights. To keep things simple, you can create a single table named 'knight'. If you want to play with table relationships, you can create a separate table name 'title', and make it a foreign key relationship with the knight table.

You can create a `manage.py` shell, as described earlier, to manually add a couple of knights, or use the database's CLI.

Chapter 20: Implementing REST with Flask

Objectives

- Learn what REST means
- Implement a REST API
- Write view functions to respond to REST requests

What is REST?

- REpresentational State Transfer
- Architectural style, not protocol
- Programmatic access to data via web server
- Replaces older protocols (e.g. SOAP)

REST is an architectural style for providing web-based services. It is not a protocol. It replaces older styles, such as RPC and SOAP.

REST specifies six constraints which must be met in order to be called RESTful. Since it is a guideline and not a protocol, all requirements do not necessarily have to be met.

The commonly recognizable part of REST is that typical REST servers are accessed by sending URL path information in the request, rather than URL queries.

In other words, a traditional request might look like <http://yourserver.com/presidents?term=26>. The REST equivalent might look like <http://yourserver.com/presidents/term/26> or just <http://yourserver.com/presidents/26>.

Your application's RESTful API is defined in terms of such URLs.

Base REST constraints

The base constraints of a web-based REST system are:

- Client-server architecture via HTTP
- Stateless communication (each request is independent of the last)
- Cacheable
- Layered system
- Code on demand (optional)
- Uniform interface
 - Hypertext (HTML) links to reference state (i.e., retrieve data)
 - Hypertext links to get related resource
 - Standard media types (typically JSON, but sometimes XML or others)

Designing a REST API

- It's all about the URL
- Include version in paths
- Human-readable

When designing a RESTful API, it's all about the URL. All communication from the client to the server is done in terms of URL requests. This includes queries (GET) as well as updates (POST, PUT) and deletes (DELETE).

Because a service has no control over the clients that are using it, a service cannot abruptly change the API. For this reason, you should include the API version in URLs. Once you have announced an API change, the client can update its code.

For instance, the first version of a site might use the URL

```
http://myserver.com/api/v1.0/president/term/26
```

But the next version might change it to just

```
http://myserver.com/api/v2.0/president/26
```

This allows the user to present the new version while keeping the old version available.

URLs in your API should be as short as possible while remaining descriptive and human-readable.

Example

```
http://myserver.com/api/v2.0/presidents  
http://myserver.com/api/v2.0/president/26  
http://myserver.com/api/v2.0/presidents/roosevelt
```

If the service allows data to be modified, rather than reflect this in the URL, it is controlled by HTTP verbs. Instead of providing deletion with

```
http://myserver.com/api/v2.0/president/delete/26
```

provide it with the same URL as the query, but with the client providing a DELETE request, rather than GET:

```
http://myserver.com/api/v2.0/president/26
```

Here is a good presentation on the design of APIs:

```
https://speakerdeck.com/dzuelke/designing-http-interfaces-and-restful-web-services-  
sfliveparis2012-2012-06-08
```

Setting up RESTful routes

*Use view functions *Parameter types specified with converters *Path information passed into view functions

To set up a RESTful URL, create a view function as usual. In the route string, enclose one or more parameters in angle brackets, like

```
/president/<int:term>
```

The word before the colon is a converter, and is used to parse a data type from the path information.

The view function's parameter names must match the names in the route:

```
def president_by_term(term):  
    pass
```

There can be any number of path parameters in a URL.

Choosing the return type

- Return type is typically JSON
- Other types requested via HTTP header
- Put logic in view function, or use decorator

RESTful services typically return data as JSON (JavaScript Object Notation). However, the client may request another data type via the HTTP ACCEPT header. You can decide to provide alternate data types, or just return an error status if that data type is not available.

Example

flask_rest_response.py

```
#!/usr/bin/env python
#
from datetime import date
import xml.etree.ElementTree as ET
from flask import Flask, request, render_template, jsonify
from flask_bootstrap import Bootstrap

from president import President

app = Flask(__name__)
Bootstrap(app)

def pres_to_dict(pres):
    """Convert one president object to a dictionary"""
    pres_dict = {}
    for prop_name in dir(pres):
        if not prop_name.startswith('_'):
            prop_value = getattr(pres, prop_name)
            if isinstance(prop_value, date):
                prop_value = '{0.year:4d}-{0.month:02d}-
{0.day:02d}'.format(prop_value)
            pres_dict[prop_name] = prop_value
    return pres_dict
```

```
def pres_list_to_xml(pres_list):
    """Convert list of presidents to XML"""
    root_element = ET.Element('presidents')
    for pres in pres_list:
        pres_element = ET.Element('president')
        fname = ET.Element('firstname')
        fname.text = pres.first_name
        pres_element.append(fname)
        lname = ET.Element('lastname')
        lname.text = pres.last_name
        pres_element.append(lname)
        root_element.append(pres_element)
    return ET.tostring(root_element)

@app.route('/api/1.0/potus')
def index():
    """Main (and only) page; returns list of all presidents"""
    presidents = []
    for i in range(1, 46):
        presidents.append(President(i))
    accept_type = request.headers.get('ACCEPT')
    response = ''
    print("Accept type:", accept_type)
    if accept_type.startswith('text/html'):
        response = render_template('president_list_bs.html', presidents=presidents)
    elif accept_type.startswith('application/xml'):
        response = pres_list_to_xml(presidents)
    elif accept_type == 'application/json':
        presidents_as_dicts = [pres_to_dict(p) for p in presidents]
        response = jsonify(presidents=presidents_as_dicts)
    # handle error here if non-expected type

    return response

if __name__ == '__main__':
    app.run(debug=True)
```

Using jsonify

- jsonify() creates JSON suitable for the response
- Pass keyword arguments which becomes keys in JSON dictionary flask.jsonify() returns a JSON response created from named parameters.

Example

```
from flask import jsonify  
  
...  
  
return jsonify(movie='Jaws', directory='Spielberg')
```

Handling invalid requests

- Use app.errorhandler()

Flask's builtin exceptions typically return human-friendly HTML. This is great for humans, but not so great for API clients.

The solution is to define a custom exception type and install an error handler for it.

Example

```
class InvalidAPIRequest(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, data=None):
        Exception.__init__(self)
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self._data = data

    def to_dict(self):
        return_value = dict(self.data or ())
        return_value['message'] = self.message
        return return_value

@app.errorhandler(InvalidAPIRequest)
def handle_invalid_usage(error):
    response = jsonify(error.to_dict())
    response.status_code = error.status_code
    return response

@app.route('/president/<int:term>')
def get_president(term):
    if term < 1 or term > 44:
        raise InvalidUsage('Term out of range', status_code=410)
```

Receiving data from POST

- Use the `request.form` object
- `form` is a dictionary of fields

To receive data from a form, use a route that specifies a POST request, and use the `request.form` object to retrieve specific fields.

Example

```
@app.route('/president', methods=['POST'])
def add_president():
    first_name = request.form['firstname']
    last_name = request.form['lastname']
```

Handling a DELETE request

- No special URL
- Use route with DELETE method

To handle a delete request, use a route that specifies the DELETE method.

```
@app.route('/president/<int:term>', methods=['DELETE'])
def delete_president(term):
    pass # delete president from database here...
```

Chapter 20 Exercises

Exercise 20-1 (`restful_knights.py`)

Write a script to provide access to knight data via a RESTful interface. Design a simple API.

You can use the Knight class from `knight.py` for data.

`/knight/name` (to get one knight)

or

`/knights` (to get a list of all knights)

Return the data as JSON. Use postman or a similar tool to test (or just use a browser).

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

Title	Author	Publisher
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
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

Title	Author	Publisher
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		
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

Title	Author	Publisher
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: Object Oriented Design

Object Oriented Design Principles

Before introducing the details of building classes in Python, it is important to realize that there are many design principles that have been defined over the years. These design principles are intended to assist in the development of clean and modular code.

Several object oriented design principles have become very prevalent to developers over the years. These principles are designed to simplify developing software that is both easy to maintain and extend.

One such principle is the **DRY** principle. The **DRY** acronym stands for **D**on't **R**epeat **Y**ourself. The principle was defined by Andy Hunt and Dave Thomas in the book *The Pragmatic Programmer*, Addison-Wesley Professional - 1999.

The `setUp()` method of the `unittest.TestCase` class from the TDD examples is a good example of using the DRY principle to factor out the duplicated set-up code into a single method that is automatically called by the framework prior to each test method being called. That is where the instantiation of an object was done once, instead of inside each of the test methods themselves.

Simply put; if code is being copied and pasted from one part of the code to another, then the DRY principle is not being adhered to. The example that follows demonstrates another violation of the DRY principle. Once the problem is stated, the second example will demonstrate modifying the code to adhere to the DRY principle.

Example

oodesign/dry/wetdesign.py

```
#!/usr/bin/env python3
def do_something(an_object):
    if an_object is None:
        raise TypeError("NoneType is not allowed")
    # business logic would go here

def do_something_else(an_object):
    if an_object is None:
        raise TypeError("NoneType is not allowed")
    # business logic would go here
```

The example above breaks the DRY principle since there are two places where an object is being tested to ensure it is not `None`.

The following example factors out the duplication into a function call that can be reused by both of the original functions.

Example

oodesign/dry/drydesign.py

```
#!/usr/bin/env python3
def do_something(an_object):
    do_not_allow_none(an_object)
    # business logic would go here

def do_something_else(an_object):
    do_not_allow_none(an_object)
    # business logic would go here

def do_not_allow_none(an_object):
    if an_object is None:
        raise TypeError("NoneType is not allowed")
```

The example above raises an exception if `an_object` is `None`. A different version might rely on a Boolean return value by defining a function similar to the one shown below instead of `do_not_allow_none()`.

```
def is_not_none(an_object):
    return True if an_object is not None else False
```

SOLID Design Principles

SOLID is a mnemonic acronym for five of the object oriented design principles developed by Robert C. Martin who is also commonly known as Uncle Bob. While the principles were developed by Uncle Bob, the mnemonic itself was coined by Michael Feathers.

The five **SOLID** principles of class design are as follows:

S Single Responsibility Principle (SRP)

This principle states that a class should have one, and only one reason to change.

O Open Closed Principle (OCP)

This principle states that you should be able to extend a class's behavior, without modifying it.

L Liskov Substitution Principle (LSP)

Derived classes must be substitutable for their base classes.

I Interface Segregation Principle (ISP)

Make fine grained interfaces that are client specific.

D Dependency Inversion Principle (DIP)

Depend on abstractions not concretions.

The Single Responsibility Principle (SRP)

The "S" in SOLID stands for the Single Responsibility Principle. As mentioned earlier, this principle states that a class should have one, and only one reason to change. This can also be thought of as a class should have only one responsibility.

As an example of the SRP, we will look at designing a `Book` class that has a title, author, pages and a variable to track the current page within the book.

Example

oodesign/srp/too_much_responsibility.py

```
#!/usr/bin/env python3
class Book:

    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
        self.current_page = 0

    def __str__(self):
        fmt = "Title: {0}\nAuthor: {1}\nPages: {2}\nCurrent Page: {3}"
        return fmt.format(self.title, self.author, self.pages,
                         self.current_page)
```

The main responsibility of the `Book` class is to represent a `title`, an `author`, and a number of `pages`. Adding the concept of storing the current page would add additional responsibility which would break the SRP. That data might be better factored out into a class called `Bookmark` that maintains information about a `Book` and a page within that book.

One might then be tempted to add a reference to a `Bookmark` in the composition of a `Book`, but this once again would break the SRP. A better approach may be to define a `Bookmark` as being composed of the `Book` that it belongs to and the page within that book that is referenced.

Example

oodesign/srp/bookmark.py

```
#!/usr/bin/env python3
class Bookmark:

    def __init__(self, book, page):
        self.book = book
        self.page = page

    def __str__(self):
        fmt = "Bookmark:\nPage: {1}\nBook : {2}"
        return fmt.format(self.page, self.book.title)
```

The requirement of having the ability to track the current page within a Book has been accomplished with the above class. It did not require any change to the `Book` class in order to accomplish the requirement. As such the `Book` class maintained its original responsibility without having to change.

The next example defines a function that reads the contents of a file into a list and then prints them out. The fact that it reads, populates and prints is a good indication that it has too much responsibility.

Example

oodesign/srp/srp_01.py

```
#!/usr/bin/env python3
def words(filename):
    words = []
    with open(filename) as the_file:
        text = the_file.read()
    for each_line in [line for line in text.split("\n")]:
        words.extend(each_line.split())
    print(words)

def main():
    words("declaration_of_independance.txt")

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

The previous application is rewritten below to better adhere to the SRP.

Example

oodesign/srp/srp_02.py

```
#!/usr/bin/env python3
def words(text):
    words = []
    for each_line in [line for line in text.split("\n")]:
        words.extend(each_line.split())
    return words

def read_file(filename):
    with open(filename) as the_file:
        text = the_file.read()
    return text

def print_iterable(an_iterable, per_line=1):
    for counter, value in enumerate(an_iterable):
        print(value, end=" ")
        if counter % per_line == per_line - 1:
            print()

def main():
    result = read_file("declaration_of_independance.txt")
    data = words(result)
    print_iterable(data, 10)

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

Composite vs. Aggregate Types

When dealing with the SRP and deciding upon composition within a class definition, the types can often be thought of as one of two types.

- Composite types
 - Composite types are thought of as being "owned" by the objects that are created from them.
 - When the owner object is destroyed, the composite types within it are usually also destroyed, as they serve no purpose outside of the object.
- Aggregate types.
 - Aggregate types are often thought of as being "used" by the objects that are created with them.
 - When the owner object is destroyed, the aggregate objects live on, as they are probably being used by more than one object at a time within the application.

To better understand the difference, the `Book` and a potential `Shelf` classes will be compared as to what each is composed of.

- The `Book` class is composed of string and number objects that outside of the `Book` object where they were used probably serve no purpose. The title, author and pages attributes of the `Book` class are thus often referred to as composite data types.
- A `Shelf` class may be composed of a collection of `Book` objects that may exist long after the shelf they were stored on is destroyed. While it is debatable whether the list itself is a composite or aggregate type within the `Shelf` class, the `Book` objects within the list are definitely an example of aggregate types in that the `Book` objects may be used in other places.

In Python aggregation is referred to as composition as the distinction between composition and aggregation is not as important.

The Open Closed Principle (OCP)

The **O** in SOLID stands for the Open Closed Principle.

The principle originates from a book entitled "*Object Oriented Software Construction*" written by Bertrand Meyer in 1988.

The principle stated the following:

- *A good module structure should be both closed and open.*
 - *Closed, because clients need the module's services to proceed with their own development, and once they have settled on a version of the module should not be affected by the introduction of new services they do not need.*
 - *Open, because there is no guarantee that we will include right from the start every service potentially useful to some client.*

And according to Meyer, this can be achieved through inheritance:

But inheritance solves it. A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as a parent, adding new features and redeclaring inherited features; in this process there is no need to change the original or to disturb its clients.

When the SOLID principles were developed by Uncle Bob, he restated the above as follows:

Modules, classes, and even methods and functions are "Open For Extension" and "Closed for Modification"

In his Clean Code Blog, Uncle Bob has an entry from March 8th 2013 entitled "An Open and Closed Case".

- In the blog he discusses the revisions to his original statement about OCP and ultimately simplifies its meaning to:

This principle has a high-falutin' definition, but a simple meaning: You should be able to change the environment surrounding a module without changing the module itself.

While the `words` function from the SRP examples was made to have one responsibility, as written it violates the Open Closed Principle in that while it may be closed for modification, which is good, it is not open for extension. The code only knows how to work with a list that it creates internally. This does not permit the ability to extend the functions use to someone who wants a set instead.

The example that follows is a rewrite of the `srp_02.py` example that is an attempt to have the `words` function also adhere to OCP. It does this by allowing the caller of the function to pass either a list or a set as an argument to the function. This has the added benefit of loose coupling between the function and the collection being used.

Example

`oodesign/ocp/ocp_01.py`

```
#!/usr/bin/env python3
def words(text, set_or_list):
    if type(set_or_list) == list:
        add_all = list.extend
    elif type(set_or_list) == set:
        add_all = set.update
    else:
        raise TypeError()

    for each_line in [line for line in text.split("\n")]:
        add_all(set_or_list, each_line.split())

def read_file(filename):
    with open(filename) as the_file:
        text = the_file.read()
    return text

def print_iterable(an_iterable, per_line=1):
    for counter, value in enumerate(an_iterable):
        print(value, end=" ")
        if counter % per_line == per_line - 1:
            print()

def main():
    result = read_file("declaration_of_independance.txt")
    data = []
    words(result, data)
    print_iterable(data, 10)

    print("*" * 50)

    data = set()
    words(result, data)
    print_iterable(data, 10)

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

The example as written, although coming closer to adhering to the OCP, includes type checking which often goes against the Python's idea of it is easier to ask for forgiveness than permission. It also limits the use of the function to lists and sets - so it is not as extensible as it could be.

For this particular function, a generator may be a much better and cleaner choice as shown in the revised version below.

Example

oodesign/ocp/ocp_02.py

```
#!/usr/bin/env python3
def words(text):
    yield from text.split()

def read_file(filename):
    with open(filename) as the_file:
        text = the_file.read()
    return text

def print_iterable(an_iterable, per_line=1):
    for counter, value in enumerate(an_iterable):
        print(value, end=" ")
        if counter % per_line == per_line - 1:
            print()

def main():
    result = read_file("declaration_of_independance.txt")
    data = words(result)
    print_iterable(data, 10)

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

OCP applies to classes as well as functions. Often the easiest way of working with OCP and classes is the use of the `AbstractBaseClass` in the `abc` module.

The example that follows defines a `Shape` class that is closed for modification. Everything a `Shape`

represents is encapsulated in the abstract base class of `Shape`. It is open for extension as shown in the various subclasses that extend from `Shape` and add, but are not limited to, the minimum requirements of a shape.

Example

oodesign/ocp/shapes/shape.py

```
#!/usr/bin/env python3
import abc

class Shape(abc.ABC):
    id = 100

    def __init__(self, name):
        self.name = name
        self.number = Shape.id
        Shape.id += 1

    @abc.abstractmethod
    def area(self):
        pass # Intended to be implemented by subclasses

    @abc.abstractmethod
    def perimeter(self):
        pass # Intended to be implemented by subclasses

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

    @name.setter
    def name(self, name): self._name = name

    def __str__(self):
        return "Name:{} id:{}".format(self.name, self.number)
```

oodesign/ocp/shapes/circle.py

```
#!/usr/bin/env python3
import math

from shape import Shape

class Circle(Shape):
    def __init__(self, name, radius):
        super().__init__(name)
        self.radius = radius

    def __str__(self):
        fmt = "{} Radius:{}"
        return fmt.format(super().__str__(), self.radius)

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius
```

oodesign/ocp/shapes/rectangle.py

```
#!/usr/bin/env python3
from shape import Shape

class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def __str__(self):
        fmt = "{} Length:{} Width:{}"
        return fmt.format(super().__str__(), self.length,
                          self.width)

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width
```

oodesign/ocp/shapes/rectangle.py

```
#!/usr/bin/env python3
from shape import Shape

class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def __str__(self):
        fmt = "{} Length:{} Width:{}"
        return fmt.format(super().__str__(), self.length,
                          self.width)

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width
```

The application below creates a list of **Shape** objects and polymorphism ensures that the correct methods are called on the correct subclasses automatically

Example

oodesign/ocp/shapes/create_shapes.py

```
#!/usr/bin/env python3
from circle import Circle
from square import Square
from rectangle import Rectangle

def main():
    shapes = [Circle("Circle 1", 10),
              Square("Square 1", 5),
              Rectangle("Rectang1e 1", 5, 10)]

    for shape in shapes:
        print(shape)
        print("Area:", shape.area(), "Perimeter:", shape.perimeter())
        print("*" * 50)

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

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle was initially introduced by Barbara Liskov in 1987. Barbara Liskov and Jeannette Wing described the principle succinctly in 1994. The principle states that derived classes (subclasses) must be substitutable for their base(parent) classes.

An overridden method of a subclass needs to accept, at minimum the same parameters as the method of the superclass. If the subclass takes additional parameters - these would be required to be optional so that not passing them is acceptable in the application that calls the parent method.

The subclass methods can implement less restrictive rules, but would not be allowed to be stricter. Otherwise, code that calls the method on an object of the parent class might cause an exception that otherwise would not have happened in the parent class' implementation.

At first glance, the square class seems to benefit from extending from the rectangle class in the previous example. Upon closer examination though, A rectangle can have its length or width changed independently of the other. The Square class as written violates the LSP because it cannot do what its parent class does, in that the length and width are always the same.

The same type of problem would exist from extending a `Circle` class from an `Ellipse` class that extends `Shape`. In fact, this problem is usually called the "Circle-Ellipse Problem".

One way to modify the design of the hierarchy so that it does not violate the Liskov Substitution Principle is to enforce that all Shape objects are immutable. This would ensure that if a square needs to be different, a new square would have to be instantiated as oppose to the existing instance being changed.

The example that follows is a rewrite of the previous shapes example. The code relies on the `@dataclass` decorator from the `dataclasses` module to freeze the state of each instance of an object so that any attempt to change it results in a `dataclasses.FrozenInstanceError` being raised.

Example

`odesign/lsp/shapes/shape.py`

```
import abc
from dataclasses import dataclass, field
from typing import ClassVar

@dataclass(frozen=True)
class Shape():
    id: ClassVar[int] = 100
    name: str

    def __post_init__(self):
        self.__dict__["number"] = Shape.id
        Shape.id += 1

    @abc.abstractmethod
    def area(self):
        pass # Intended to be implemented by subclasses

    @abc.abstractmethod
    def perimeter(self):
        pass # Intended to be implemented by subclasses

    def __str__(self):
        return "Name:{} id:{}".format(self.name, self.number)
```

oodesign/lsp/shapes/circle.py

```
#!/usr/bin/env python3
import math
from dataclasses import dataclass

from shape import Shape

@dataclass(frozen=True)
class Circle(Shape):
    radius: int

    def __str__(self):
        fmt = "{} Radius:{}"
        return fmt.format(super().__str__(), self.radius)

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius
```

oodesign/lsp/shapes/rectangle.py

```
#!/usr/bin/env python3
from dataclasses import dataclass

from shape import Shape

@dataclass(frozen=True)
class Rectangle(Shape):
    length: int
    width: int

    def __str__(self):
        fmt = "{} Length:{} Width:{}"
        return fmt.format(super().__str__(), self.length,
                          self.width)

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width
```

oodesign/lsp/shapes/square.py

```
#!/usr/bin/env python3
from dataclasses import dataclass

from rectangle import Rectangle

@dataclass(frozen=True)
class Square(Rectangle):
    # implemented with __init__ otherwise using type hinting would require
    # the width be passed as an argument to the constructor also
    def __init__(self, name, length):
        super().__init__(name, length, length)
```

The application below creates a list of **Shape** objects and polymorphism ensures that the correct methods are called on the correct subclasses automatically

Example

oodesign/lsp/shapes/create_shapes.py

```
#!/usr/bin/env python3
import dataclasses
import sys

from circle import Circle
from square import Square
from rectangle import Rectangle


def main():
    shapes = [Circle("Circle 1", 10),
              Square("Square 1", 5),
              Rectangle("Rectangle 1", 5, 10)]

    for shape in shapes:
        print(shape)
        print("Area:", shape.area(), "Perimeter:", shape.perimeter())
        print("*" * 50)

    try:
        shapes[1].width = 88
    except dataclasses.FrozenInstanceError as fie:
        print("Frozen Instance Error", fie, file=sys.stderr)

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

The code inside of the `try` block above will generate an exception since the object is immutable.

The Interface Segregation Principle (ISP)

The Interface Segregation Principle states that clients should not be forced to implement interfaces they don't use. Many small interfaces are preferred over one large interface. An abstract class that has 10 methods defined might be better broken into several abstract classes. This allows implementation of only what is needed instead of implementing everything - including things that do not apply to the subclass.

The example that follows defines an interface (abstract class) that requires 4 things of every subclass.

Example

oodesign/isp/bad/printer.py

```
import os
from abc import ABC, abstractmethod

class Printer(ABC):
    @abstractmethod
    def print_it(self):
        pass

    @abstractmethod
    def scan_it(self, file_name):
        pass

    @abstractmethod
    def fax_it(self, fax_number):
        pass

    @abstractmethod
    def email_it(self, email_address):
        pass
```

The following two examples extend the abstract `Printer` class. * The `AllInOnePrinter` is capable of providing all 4 of the required methods. * The `PrintAndScan` printer is only capable of providing two of the methods. * But it is still responsible for defining all 4 of the methods as shown.

Example

oodesign/isp/bad/allinone.py

```
from printer import Printer

class AllInOnePrinter(Printer):
    def __init__(self, data):
        self.data = data

    def print_it(self):
        print("Printing")
        print(self.data)

    def scan_it(self, file_name):
        print("Scanning to:", file_name)
        with open(file_name, "w") as scan_to:
            scan_to.write(self.data)

    def fax_it(self, fax_number):
        print("Faxing to:", fax_number)
        # remainder of faxing logic here

    def email_it(self, email_address):
        print("Emailing to", email_address)
        # remainder of emailing logic here
```

oodesign/isp/bad/nocomms.py

```
from printer import Printer

class PrintAndScan(Printer):
    def __init__(self, data):
        self.data = data

    def print_it(self):
        print("Printing")
        print(self.data)

    def scan_it(self, file_name):
        print("Scanning to:", file_name)
        with open(file_name, "w") as scan_to:
            scan_to.write(self.data)

    def fax_it(self, fax_number):
        raise NotImplementedError("Faxing Not Supported")

    def email_it(self, email_address):
        raise NotImplementedError("Emailing Not Supported")
```

This break the ISP by requiring all things of all child classes. The example below re-writes the **Printer** class by breaking it into separate abstract classes as shown below.

Example

[odesign/isp/good/printer_abstractions.py](#)

```
import os
from abc import ABC, abstractmethod

class Printable(ABC):
    @abstractmethod
    def print_it(self):
        pass

class Scannable(ABC):
    @abstractmethod
    def scan_it(self, file_name):
        pass

class Faxable(ABC):
    @abstractmethod
    def fax_it(self, fax_number):
        pass

class Emailable(ABC):
    @abstractmethod
    def email_it(self, email_address):
        pass
```

Now each subclass only has to extend the particular classes it is capable of providing as behavior. This can be seen in the re-write of the two subclasses.

Example

oodesign/isp/good/allinone.py

```
from printer_abstractions import Printable, Scannable, Faxable, Emailable

class AllInOnePrinter(Printable, Scannable, Faxable, Emailable):
    def __init__(self, data):
        self.data = data

    def print_it(self):
        print("Printing")
        print(self.data)

    def scan_it(self, file_name):
        print("Scanning to:", file_name)
        with open(file_name, "w") as scan_to:
            scan_to.write(self.data)

    def fax_it(self, fax_number):
        print("Faxing to:", fax_number)
        # remainder of faxing logic here

    def email_it(self, email_address):
        print("Emailing to", email_address)
        # remainder of emailing logic here
```

oodesign/isp/good/nocomms.py

```
from printer import Printable, Scannable

class PrintAndScan(Printable, Scannable):
    def __init__(self, data):
        self.data = data

    def print_it(self):
        print("Printing")
        print(self.data)

    def scan_it(self, file_name):
        print("Scanning to:", file_name)
        with open(file_name, "w") as scan_to:
            scan_to.write(self.data)
```

The Dependency Inversion Principle (DIP)

The Dependency Inversion Principle states the following two things: 1. High-level modules should not depend on low-level modules. (Both should depend on abstractions.) 2. Abstractions should not depend on details. (Details should depend on abstractions.)

The high-level depends on an abstraction, and the low-level depends on the same abstraction.

The following example shows a class that depends heavily on the `os` module to get environment settings.

Example

oodesign/dip/dependent.py

```
import os

class DependsOnEnv:
    def get_home(self):
        return os.getenv("HOME")

    def get_user(self):
        return os.getenv("USER")

doe = DependsOnEnv()

print(doe.get_home())
print(doe.get_user())
```

The next example removes the dependency by defining an `AbstractEnvironment` class that defines the necessary behavior.

Subclasses are then defined that rely on an abstract class. This permits the `DependsOnEnv` class to rely on the abstract class but not be dependent upon the specific subclasses it uses. It accomplishes this through a term referred to as Dependency Injection, by taking it in as a parameter to the constructor instead of instantiating it itself inside of the constructor.

Example

oodesign/dip/notdependent.py

```
import os
from abc import ABC, abstractmethod

class AbstractEnvironment(ABC):
    @abstractmethod
    def get_home(self):
        pass

    @abstractmethod
    def get_user(self):
        pass

class OSEnvironment(AbstractEnvironment):
    def get_home(self):
        return os.getenv("HOME")

    def get_user(self):
        return os.getenv("USER")

class HardCodedEnvironment(AbstractEnvironment):
    def get_home(self):
        return "/home/default"

    def get_user(self):
        return "default"

class DependsOnEnv:
    def __init__(self, abstract_environment):
        self.abstract_environment = abstract_environment

    def get_home(self):
        return self.abstract_environment.get_home()

    def get_user(self):
        return self.abstract_environment.get_user()
```

```
doe = DependsOnEnv(HardCodedEnvironment())

print(doe.get_home())
print(doe.get_user())


doe = DependsOnEnv(OSEnvironment())

print(doe.get_home())
print(doe.get_user())
```

Index

- @
 - .gitignore, 486
 - @pytest.mark.mark, 377
 - _call_(){empty}, 249
 - _enter_, 116, 118
 - _exit_, 116, 118
 - _init_(){empty}, 249
 - _init_.py, 72
 - _iter_, 154
 - _new_(){empty}, 249
 - _next_, 154
 - _prepare_(){empty}, 249
 - _pycache_, 61
- A
 - abstract base classes, 104
 - Anaconda, 399
 - argparse, 341
 - assert, 360
 - assertions, 359
 - asynchronous communication, 274
 - asyncio, 304
 - attributes, 215
 - awaitable, 310
- B
 - benchmarking, 268
 - binary mode, 446
 - builtin types
 - emulating, 170
- C
 - callback, 316
 - class
 - defining at runtime, 236
 - class data, 92
 - class method, 93
 - classes, 80
- constructors, 86
- defining, 81
- inheritance, 95
- client.containers.list(), 502
- collection vs generator, 31
- collections, 130
- command line scripts, 337
- comments, 257
- conftest.py, 371
- constructors, 86
- container, 498
- Container classes, 160
- context manager, 116-117
- context protocol, 117
- contextlib, 123
- contextlib.closing, 124
- contextlib.contextmanager, 126
- contextlib.suppress, 127
- coroutine, 151, 306
- Coroutines, 305
- Counter, 163
- creating Unix-style filters, 338
- CSV, 432
 - nonstandard, 434
- csv
 - DictReader, 436
 - csv.reader(), 432
 - csv.writer(), 438
- D
 - debugger
 - setting breakpoints, 265
 - starting, 263
 - stepping through a program, 264
 - decorator class, 229
 - decorator function, 226
 - decorator parameters, 233
 - decorators, 221

decorators in the standard library, 224
defaultdict, 163
delattr(), 215
DELETE request, 546
deque, 163
dict, 161
dictionary
 custom, 179
dictionary comprehension, 28
Django, 371
Django framework, 246
Docker, 498
 advantages of, 498
 container attributes, 504
 create client, 499
 environment variables, 499
 listing containers, 502
 managing images, 505
 terminology, 498
Docker API, 499
docker.containers.list(), 502
docker.from_env(), 499
docker.images
 attributes, 507
DOM, 397-398
Douglas Crockford, 416

E

Eclipse, 479
Element, 181, 400-401
ElementTree, 399
 find(), 408
 findall(), 408
email
 attachments, 463
 sending, 460
email.mime, 463
event loop, 305
exception, 315, 362

F

fixtures, 358, 364
for, 130
for loop, 140
function parameters, 49
 named, 49
 optional, 49
 positional, 49
 required, 49
functions, 46
functools.wraps, 226
Future, 310
future, 308

G

generator, 130, 148
generator class, 144, 154
generator expression, 144-145, 33
generator function, 144
generator.send(), 151
generators, 144
 how to create, 144
GET, 453
getattr(), 215
getroot(), 407
getters, 87
GIL, 276
git, 477
 command line, 479
 definition, 477
 documentation, 479
 workflow, 478
git workflow, 478
git.Git, 480
git.Repo, 480
gitpython, 480
glob, 322
global, 59
globals(), 209
grabbing a web page, 446

Graphviz, 261

H

hasattr(), 215

hooks, 371

HTTP verbs, 453

I

import *, 66

import {star}, 65

in operator, 31

inheritance, 95

 multiple, 101

inspect module, 212

instance attributes, 83

instance methods, 85

iterable, 130

J

Java, 359

JSON, 416

 custom encoding, 423

 types, 416

json module, 417

json.dumps(), 420

json.loads(), 417

L

lambda function, 24

list, 161, 176

list comprehension, 141, 26

lists

 custom, 178

locals(), 209

logging

 alternate destinations, 353

 exceptions, 351

 formatted, 348

 simple, 346

lxml

 Element, 401

SubElement, 401

lxml.etree, 398

M

markers, 359

Mercurial, 476

metaclass, 244, 246

metaclasses, 243

metaprogramming, 208

Microsoft Team Foundation Server, 476

mock object, 384

modules, 60

 documenting, 75

 executing as scripts, 68

 importing, 61

 search path, 67

monkey patches, 240

multiprocessing, 274, 295

 Manager, 291

 multiprocessing module, 291

 multiprocessing.dummy, 295

 multiprocessing.dummy.Pool, 295

 multiprocessing.Pool, 295

 multiprogramming, 274

 alternatives to, 301

N

namedtuple, 163

node ID, 379

nonlocal, 59

O

object instance, 82

OrderedDict, 163

os.chdir(), 486

P

packages, 70

 configuring, 72

parametrizing, 374

paramiko, 467

parsing the command line, 342
PEP 20, 10
PEP 8, 76
permissions, 332
 checking, 332
pipelines, 151
plugins, 371
Popen, 325
POST, 453
POST data, 545
preconfigured log handlers, 353
profiler, 266
properties, 88
PUT, 453
py.test, 361
pycallgraph, 267
PyCharm, 361, 479
pychecker, 258
pyflakes, 258
pylint, 258
 customizing, 259
pymock, 384
pyreverse, 260
pytest, 359-360
 builtin fixtures, 367
 configuring fixtures, 371
 output capture, 361
 special assertions, 362
 user-defined fixtures, 365
 verbose, 361
pytest-mock, 385
pytest.approx(), 362
pytest.fixture, 365
pytest.raises(), 362
python debugger, 262
python style, 76
PYTHONPATH, 67

R

redis, 371
remote access, 467

requests, 453
 methods
 keyword parameters, 459
REST, 536
 converter, 540
REST API
 designing, 538
running tests, 380
 by component, 380
 by mark, 380
 by name, 380

S

SAX, 397-398
scope
 builtin, 56
 global, 56
 nonlocal, 56
sendmail(), 460
set, 161
set comprehension, 30
setattr(), 215
setters, 87
SFTP, 470
shlex.split(), 324
shutil, 334
singledispatch, 423
smtplib, 460
SOAP, 536
sorted(), 19
sorting
 custom key, 21
source code control system, 476-477
special methods, 107, 170
ssh protocol, 467
static method, 113
StopIteration, 154
struct, 167
SubElement, 401
subprocess, 325-326
 capturing stdout/stderr, 329

check_call(), 326
check_output(), 326
run(), 326
Subversion, 476
super(), 96
super(), 100
sys.path, 67

T

Task, 313
test case, 358
test cases, 358
test runner, 359, 361
test runners, 358
tests
 messages, 360
thread, 275
thread class
 creating, 280
threading, 274
threading module, 277
threading.Thread, 277
threads
 debugging, 290
 locks, 283
 queue, 286
 simple, 278
 variable sharing, 283
Tim Peters, 10
timeit, 268
timsort, 10
tolerance
 pytest.approx, 362
tuple, 11, 161
type, 244
type(), 236

U

unit test, 358
unit test components, 358
unit tests

failing, 381
mock objects, 385
running, 361
skipping, 381
unittest.mock, 384-385
unpacking function parameters, 131, 14
urllib.parse.urlencode(), 450
urllib.request, 446, 450
urllib.request.Request, 450
urlopen(), 446

V

variable scope, 56
virtual machine, 498
Visual Studio Code, 479

W

web services
 consuming, 450
web-based services, 536
with statement, 116

X

xfail, 381
XML, 397-398
 root element, 404
xml.etree.ElementTree, 398-399
XPASS, 381
XPath, 412
xUnit, 359

Y

yield, 126, 148, 33

Z

Zen of Python, 10