Intermediate Python

John Strickler

Version 1.0, November 2021

Table of Contents

About this course

Welcome!

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

Instructor name:

Instructor e-mail:

Have Fun!

Classroom etiquette for in-person learning

- 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.

Classroom etiquette for remote learning

- Please turn your mic off when you’re not speaking. If multiple mics are on, it makes it difficult for all to hear
- The instructor doesn’t know you need help unless you tell them. It’s ok to ask for help often.
- Ask questions. Ask questions. Ask questions.
- INTERACT with the instructor and other students.
- Log off the remote S/W at the end of the day

Course Outline

Half-Day 1

Chapter 1 Pythonic Programming
Chapter 2 Functions, Modules, and Packages

Half-Day 2

Chapter 3 Intermediate Classes
Chapter 4 Metaprogramming

Half-Day 3

Numpy
Matplotlib

Half-Day 4

Pandas

Half-Day 5

Chapter 5 Developer tools
Chapter 6 Unit Testing with PyTest
Chapter 7 Database access

Half-Day 6

Chapter 8 Multiprogramming
Chapter 9 Network Programming

Half-Day 7

Chapter 10 Effective Scripting
Chapter 11 Serializing Data

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

Student files

You will need to load some 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 **py3cirrusint**.

What's in the files?

py3cirrusint contains data and other files needed for the exercises
py3cirrusint/EXAMPLES contains the examples from the course manuals.
py3cirrusint/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 **py3cirrusint.zip**. Extract all files to your desktop. This will create the folder **py3cirrusint**.

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

Copy or download **py3cirrusint.tar.gz** to your home directory. In your home directory, type

```
tar xzvf py3cirrusint.tar.gz
```

This will create the **py3cirrusint** 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 py3cirrusint/ANSWERS

Appendices

- Appendix A: Python Bibliography

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
- Frequently used with list of tuples
- Make code more readable

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
```

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 be a list of all values not assigned to other variables.

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']
```

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.extend([i, i + .5])

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

fruits = ["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(fruits)  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"""
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",
]


def strip_articles(title):  ①
    title = title.lower()
    for article in 'a ', 'an ', 'the ':
        if title.startswith(article):
            title = title[len(article):]  ②
            break
    return title


for book in sorted(books, key=strip_articles):  ③
    print(book)
```

① create function which takes element to compare and returns comparison key

② remove article by using a slice that starts after article + space`

③ 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']

# {KEY: VALUE for VAR ... in ITERABLE if CONDITION}
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 past the sequence to the **set()** constructor.

Example

set_comprehension.py

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

import re

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

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

set_comprehension.py

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

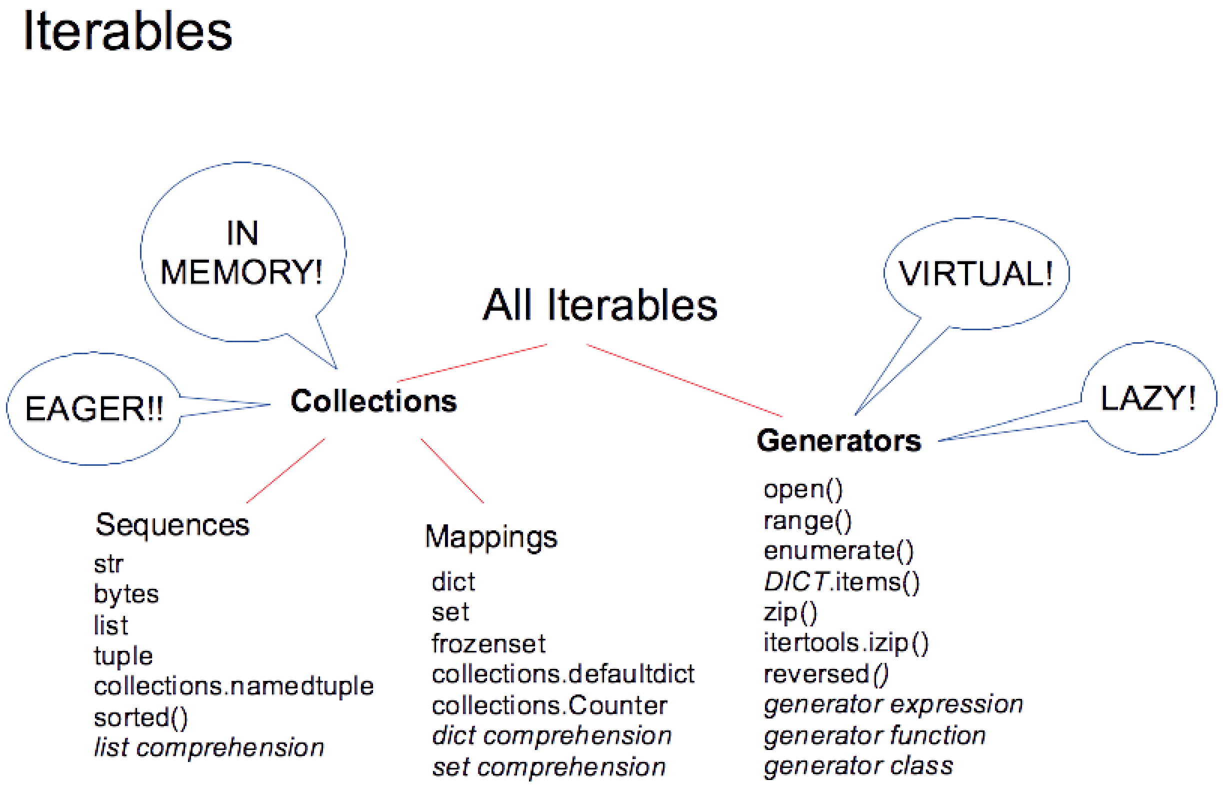
Iterables

- 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.



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()

with open("../DATA/mary.txt") as page:
    m = max(len(line) for line in page)  ③
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:
            yield line.rstrip('\n\r')  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 0011000000111001
A   38
B  127
C    9
```

f-strings

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

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

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

In normal 3.x formatting:

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

f-strings let you do this:

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

Example

f_strings.py

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

import sys

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

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

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

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

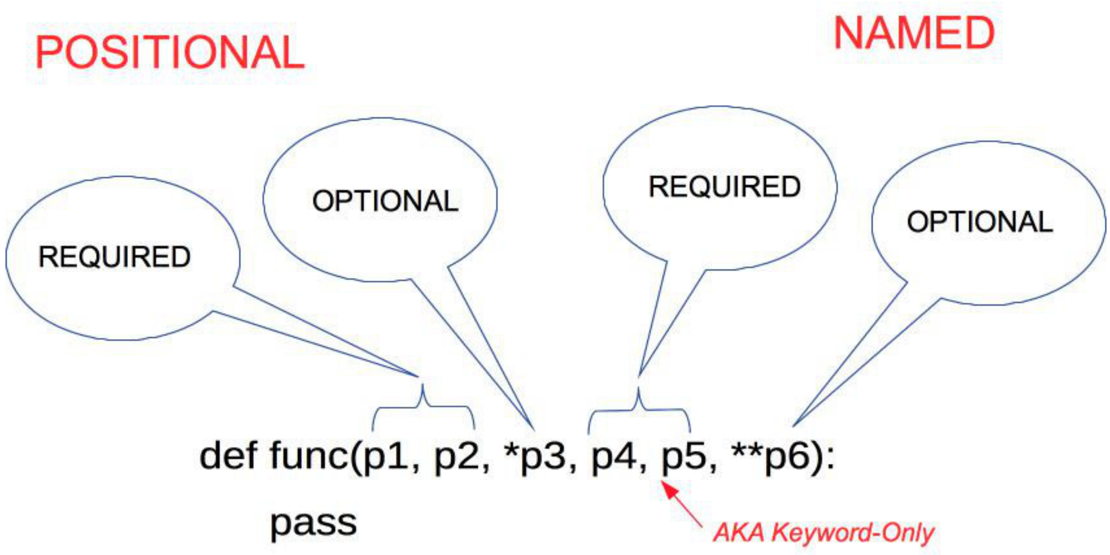
Function parameters

- 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 {}\n".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 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 _***builtins***_ module)

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

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

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

Example

scope_examples.py

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

x = 42  ①


def function_a():
    y = 5  ②

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

    return function_b


f = function_a()  ⑧
f()  ⑨
```

① global variable

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

③ local variable

④ local scope

⑤ nested (nonlocal) scope

⑥ global scope

⑦ builtin scope

⑧ calling function_a, which returns function_b

⑨ calling function_b

scope_examples.py

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

The global statement

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

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

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

Modules

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

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

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

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

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

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

Modules and packages should be documented with *docstrings*.

Using import

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

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

Variation 1

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

Variation 2

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

Variation 3

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

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

Example

samplelib.py

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

# 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.

Examples

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:\Users\bob\Documents and settings\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 both run directly and 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
from sound.effects.surround import dolby
import sound.filters.equalizer as eq
```

Example: Core Django packages

```
django
django.apps
django.conf.urls
django.conf.urls.i18n
django.contrib.admin
django.contrib.admindocs
django.contrib.auth
django.contrib.auth.backends
django.contrib.auth.forms
django.contrib.auth.hashers
django.contrib.auth.middleware
django.contrib.auth.password_validation
django.contrib.auth.signals
django.contrib.auth.views
django.contrib.contenttypes
django.contrib.contenttypes.admin
django.contrib.contenttypes.fields
django.contrib.contenttypes.forms
django.contrib.flatpages
django.contrib.gis
django.contrib.gis.admin
django.contrib.gis.db.backends
django.contrib.gis.db.models
django.contrib.gis.db.models.functions
django.contrib.gis.feeds
django.contrib.gis.forms
django.contrib.gis.forms.widgets
django.contrib.gis.gdal
django.contrib.gis.geoip2
django.contrib.gis.geos
django.contrib.gis.measure
django.contrib.gis.serializers.geojson
django.contrib.gis.utils
django.contrib.gis.utils.layermapping
django.contrib.gis.utils.ogrinspect
django.contrib.humanize
django.contrib.messages
django.contrib.messages.middleware
django.contrib.postgres
django.contrib.postgres.aggregates
django.contrib.postgres.constraints
```

```
django.contrib.postgres.indexes
django.contrib.postgres.validators
django.contrib.redirects
django.contrib.sessions
django.contrib.sessions.middleware
django.contrib.sitemaps
django.contrib.sites
django.contrib.sites.middleware
django.contrib.staticfiles
django.contrib.syndication
django.core.checks
django.core.exceptions
django.core.files
django.core.files.storage
django.core.files.uploadedfile
django.core.files.uploadhandler
django.core.mail
django.core.management
django.core.paginator
django.core.signals
django.core.signing
django.core.validators
django.db
django.db.backends
django.db.backends.base.schema
django.db.migrations
django.db.migrations.operations
django.db.models
django.db.models.constraints
django.db.models.fields
django.db.models.fields.related
django.db.models.functions
django.db.models.indexes
django.db.models.lookups
django.db.models.options
django.db.models.signals
django.db.transaction
django.dispatch
django.forms
django.forms.fields
django.forms.formsets
django.forms.models
django.forms.renderers
django.forms.widgets
django.http
django.middleware
django.middleware.cache
django.middleware.clickjacking
django.middleware.common
django.middleware.csrf
django.middleware.gzip
django.middleware.http
django.middleware.locale
django.middleware.security
django.shortcuts
django.template
django.template.backends
django.template.backends.django
django.template.backends.jinja2
django.template.loader
django.template.response
django.test
django.test.signals
django.test.utils
django.urls
django.urls.conf
django.utils
django.utils.cache
django.utils.dateparse
django.utils.decorators
django.utils.encoding
django.utils.feedgenerator
django.utils.functional
django.utils.html
django.utils.http
django.utils.log
django.utils.module_loading
django.utils.safestring
django.utils.text
django.utils.timezone
django.utils.translation
django.views
django.views.decorators.cache
django.views.decorators.csrf
django.views.decorators.gzip
django.views.decorators.http
django.views.decorators.vary
django.views.generic.dates
django.views.i18n
```

Configuring import with __init__.py

- Provide package documentation
- Load modules into package's namespace for convenience
    - Specify modules to load when * is used
- Execute startup code

The docstring in **__init__.py** is used to document the package itself. This is used by IDEs as well as **pydoc**.

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.

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

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

```
from PKG import *
```

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

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

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

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

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

Documenting modules and packages

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

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

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

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

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

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

Python style

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

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

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

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

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

Chapter 2 Exercises

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

Create a module named **potus** (potus.py) 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

> Random historic note: if I hadn’t chosen the __dunder__ naming scheme for Python language internals long ago, dunders would have been an obscure feature of the C preprocessor.
>
> — Guido van Rossum, Twitter, April 2020

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.

NOTE For an entertaining explanation of how classes work in Python, see Raymond Hettinger’s talk ***Python’s Class Development Toolkit***: https://www.youtube.com/watch?v=HTLu2DFOdTg

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!
```

Note that you can just create an attribute named *toast* without defining it anywhere. However, 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__()
- Implicitly called when object is created
- **self** is object it*self*

If a class defines a method named **__init__()**, 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__()** . 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* method retrieves data (e.g., from a private variable) from **self**. A *setter* method 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 "*@propertyname*.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

```
class Animal():
    count = 0  ①

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

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

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

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

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

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

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


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

    for animal in leo, garfield, felix:
        print(animal.name, "is a", animal.species, "--", end=' ')
        animal.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():  ②
    def fly(self):
        print("I'm flying")


class Dog(CanBark, AnimalBase):  ③
    pass


class Sparrow(CanFly, AnimalBase):  ③
    pass


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

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

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

① create primary base class

② create additional (mixin) base class

③ inherit from primary base class plus mixin

④ all animals have id()

⑤ dogs can bark() (from mixin)

⑥ sparrows can fly() (from mixin)

multiple_inheritance.py

```
Dennis
woof-woof

Steve
I'm flying

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

Abstract base classes

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

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

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

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

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

Example

abstract_base_classes.py

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

class Animal(metaclass=ABCMeta):  ①

    @abstractmethod   ②
    def speak(self):
        pass

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

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

class Duck(Animal): ③
    pass  ⑤

d = Dog()
d.speak()

c = Cat()
c.speak()

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

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

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

③ Inherit from abstract base class Animal

④ speak() **must** be implemented

⑤ Duck does not implement speak()

⑥ Duck throws a TypeError if instantiated

abstract_base_classes.py

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

Special Methods

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

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

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

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

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

Table 1. Special Methods and Variables

Method or Variables	Description
__new__(cls,…)	Returns new object instance; Called before __init__()
__init__(self,…)	Object initializer (constructor)
__del__(self)	Called when object is about to be destroyed
__repr__(self)	Called by repr() builtin
__str__(self)	Called by str() builtin

__eq__(self, other)
__ne__(self, other)
__gt__(self, other)
__lt__(self, other)
__ge__(self, other)
__le__(self, other)
	Implement comparison operators ==, !=, >, <, >=, and ⇐. self is object on the left.

__cmp__(self, other)	Called by comparison operators if __eq__, etc., are not defined
__hash__(self)	Called by hash() builtin, also used by dict, set, and frozenset operations
__bool__(self)	Called by bool() builtin. Implements truth value (boolean) testing. If not present, bool() uses len()
__unicode__(self)	Called by unicode() builtin

__getattr__(self, name)
__setattr__(self, name, value)
__delattr__(self, name)
	Override normal fetch, store, and deleter

__getattribute__(self, name)	Implement attribute access for new-style classes
__get__(self, instance)	__set__(self, instance, value)
__del__(self, instance)	Implement descriptors
__slots__ = variable-list	Allocate space for a fixed number of attributes.
__metaclass__ = callable	Called instead of type() when class is created.
__instancecheck__(self, instance)	Return true if instance is an instance of class
__subclasscheck__(self, instance)	Return true if instance is a subclass of class
__call__(self, …)	Called when instance is called as a function.
__len__(self)	Called by len() builtin
__getitem__(self, key)	Implements self[key]
__setitem__(self, key, value)	Implements self[key] = value

Method or Variables	**Description**
__selitem__(self, key)	Implements del self[key]
__iter__(self)	Called when iterator is applied to container
__reversed__(self)	Called by reversed() builtin
__contains__(self, object)	Implements in operator

__add__(self, other)
__sub__(self, other)
__mul__(self, other)
__floordiv__(self, other)
__mod__(self, other)
__divmod__(self, other)
__pow__(self, other[, modulo])
__lshift__(self, other)
__rshift__(self, other)
__and__(self, other)
__xor__(self, other)
__or__(self, other)
	Implement binary arithmetic operators +, -, *, //, %, **, <<, >>, &, ^, and |. Self is object on left side of expression.

__div__(self,other)
__truediv__(self,other)
	Implement binary division operator /. __truediv__ is called if __future__.division is in effect.

__radd__(self, other)
__rsub__(self, other)
__rmul__(self, other)
__rdiv__(self, other)
__rtruediv__(self, other)
__rfloordiv__(self, other)
__rmod__(self, other)
__rdivmod__(self, other)
__rpow__(self, other)
__rlshift__(self, other)
__rrshift__(self, other)
__rand__(self, other)
__rxor__(self, other)
__ror__(self, other)
	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)

Method or Variables	Description

__iadd__(self, other)
__isub__(self, other)
__imul__(self, other)
__idiv__(self, other)
__itruediv__(self, other)
__ifloordiv__(self, other)
__imod__(self, other)
__ipow__(self, other[, modulo])
__ilshift__(self, other)
__irshift__(self, other)
__iand__(self, other)
__ixor__(self, other)
__ior__(self, other)
	Implement augmented (+=, -=, etc.) arithmetic operators

__neg__(self)
__pos__(self)
__abs__(self)
__invert__(self)
	Implement unary arithmetic operators -, +, abs(), and ~

__oct__(self)
__hex__(self)
	Implement oct() and hex() builtins

__index__(self)	Implement operator.index()
__coerce__(self, other)	Implement "mixed-mode" numeric arithmetic.

specialmethods.py

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

class Special():

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

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

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

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

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


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

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

① all Special instances are strings

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

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

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

⑤ define equality between two Special valuess

⑥ parameter to Special() is converted to a string

⑦ add two Special instances

⑧ multiply a Special instance by an integer

specialmethods.py

```
s + s spamspam
s + t spameggs
t + t eggseggs
s * 10 spamspamspamspamspamspamspamspamspamspam
t * 3 eggseggseggs
str(s)=SPAM  str(t)=EGGS
id(s)=140335238330768 id(t)=140335238330832 id(u)=140335238331344
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: 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 0x7f9bf00ddcd0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'eggs': <function eggs at 0x7f9bf01a87a0>,
 'ham': 'Smithfield',
 'pprint': <function pprint at 0x7f9bf01a8290>,
 '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=26, function='<module>', code_context=['print("Current frame:",
inspect.getframeinfo(inspect.currentframe()))  # <7>\n'], index=0)
```

Table 2. 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 MethodType().

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'
```

Callable classes

- Convenient for one-method classes
- Really "callable instances"
- Implement __call__
- Convenient for one-method classes
- Useful for decorators

Any class instance may be made callable by implementing the special method *call*. This means that rather than saying:

```
sc = SomeClass()
sc.some_method()
```

you can say

```
sc = SomeClass()
sc()
```

What’s the advantage? Really, not too much. It just saves having to call a method from the instance, letting you call the instance itself. The use case is for classes that only have one method.

You can think of a callable class as a function that can also keep some state. As with many object-oriented features, its main purpose is to simplify the user interface.

One good use of callable classes is for implementing decorators as classes, rather than functions.

Example

callable_class.py

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

class TagWrapper():
    def __init__(self, tag):
        self._tag = tag

    def wrap(self, text):
        return '<{0}>{1}</{0}>'.format(self._tag, text)

class HTMLWrapper():

    def __init__(self, tag):
        self._tag = tag

    def __call__(self, text):  ①
        return '<{0}>{1}</{0}>'.format(self._tag, text)

if __name__ == '__main__':
    # non-callable class
    t = TagWrapper('h1')
    print(t.wrap('foo'))
    print(t.wrap('bar'))
    print()

    # callable class
    h1 = HTMLWrapper('h1')  ②
    print(h1('spam'))  ③
    div = HTMLWrapper('div')
    print(div('ham'))
    print(div('toast'))
    print(div('jam'))
```

① Define function to be called when instance is called

② Create instance of "callable class"

③ Instance is callable — essentially h1.__call__('spam') *callable_class.py*

```
<h1>foo</h1>
<h1>bar</h1>

<h1>spam</h1>
<div>ham</div>
<div>toast</div>
<div>jam</div>
```

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, as discussed earlier).

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.

Table 3. Decorators in the standard library

Decorator	Description
@abc.abstractmethod	Indicate abstract method (must be implemented).
@abc.abstractproperty	Indicate abstract property (must be implemented). **DEPRECATED**
@asyncio.coroutine	Mark generator-based coroutine.
@atexit.register	Register function to be executed when interpreter (script) exits.
@classmethod	Indicate class method (receives class object, not instance object)
@contextlib.contextmanager	Define factory function for **with** statement context managers (no need to create __enter__() and __exit__() methods)
@functools.lru_cache	Wrap a function with a memoizing callable
@functools.singledispatch	Transform function into a single-dispatch generic function.
@functools.total_ordering	Supply all other comparison methods if class defines at least one.
@functools.wraps	Invoke update_wrapper() so decorator’s replacement function keeps original function’s name and other properties.
@property	Indicate a class property.
@staticmethod	Indicate static method (passed neither instance nor class object).
@types.coroutine	Transform generator function into a coroutine function.
@unittest.mock.patch	Patch target with a new object. When the function/with statement exits patch is undone.
@unittest.mock.patch.dict	Patch dictionary (or dictionary-like object), then restore to original state after test.
@unittest.mock.patch.multiple	Perform multiple patches in one call.
@unittest.mock.patch.object	Patch object attribute with mock object.
@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

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 or classes and having them return the appropriate thing.

Table 4. Implementing Decorators

Implemented as	**Decorates**	**Takes parameters**	**How to do it**
function	function	N	decorator function returns replacement function
function	function	Y	decorator function accept params and returns function that returns replacement function
function	class	N	decorator function returns replacement class
function	class	Y	decorator function accepts params and returns function that returns replacement class
class	function	N	*instance*.__call__() *is* replacement function
class	function	Y	*instance*.__call__() accepts params and *returns* replacement function
class	class	N	*instance*.__call__() accepts original class, returns replacement class (which is usually same as orginal class)
class	class	Y	*instance*.__call__() accepts params and returns function that returns replacement class

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(thing_being_decorated):
    return 42  ①

name = "Guido"
x = void(name)

@void  ②
def hello():
    print("Hello, world")

@void
def howdy():
    print("Howdy, world")

print(hello, type(hello)) ③
print(howdy, type(howdy)) 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'>
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 simple 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. More complex decorators expect parameters to the decorator itself. In this case the decorator returns a function that expects the original function, and returns the replacement function.

The new function should be defined with generic arguments (*args, **kwargs) so it can handle any combination of arguments for the original function.

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__() 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("{0}! ".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, {}. Here are your delicious fried eggs.".format(self._name,
))


s = Spam('Mrs. Higgenbotham')  ③
s.eggs()  ④


def scrambled(self):  ⑤
    print("Hello, {}. Enjoy your scrambled eggs".format(self._name, ))


setattr(Spam, "eggs", scrambled)  ⑥

s.eggs()  ⑦
```

① create normal class

② add normal method

③ create instance of class

④ call method

⑤ define new method outside of class

⑥ monkey patch the class with the new method

⑦ call the monkey-patched method from the instance *meta_monkey.py*

```
Good morning, Mrs. Higgenbotham. Here are your delicious fried eggs.
Hello, Mrs. Higgenbotham. Enjoy your scrambled eggs
```

Do you need a Metaclass?

- Deep magic
- Used in frameworks such as Django
- YAGNI (You Ain’t Gonna Need It)

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**.

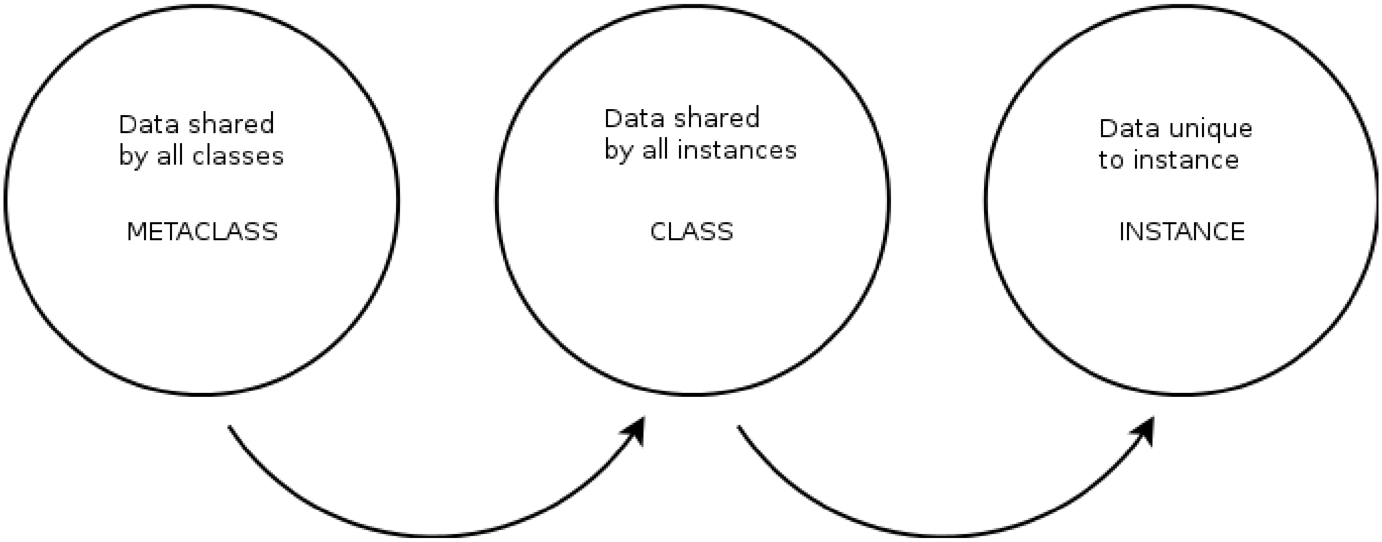
As we saw earlier ,you can create a class from a metaclass by passing in the new class's name, a tuple of base classes (which can be empty), and a dictionary of class attributes (which also can be empty).

```
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.

__prepare__() __new__() __init__() __call__()

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 0x7f972802c710>}
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 0x7f972802c710>})
============================================================
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 0x7f972802ca70>}
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 0x7f972802ca70>})
------------------------------------------------------------
in metaclass (class=A)__call__()
------------------------------------------------------------
in metaclass (class=B)__call__()
------------------------------------------------------------
in metaclass (class=A)__call__()
------------------------------------------------------------
in metaclass (class=B)__call__()
------------------------------------------------------------
animal: wombat id: 100
```

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 metaclas

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 140241554520464 1
ThingA 140241554520464 1
ThingA 140241554520464 1
ThingB 140241554520528 4
ThingB 140241554520528 4
ThingB 140241554520528 4
```

Chapter 4 Exercises

Exercise 4-1 (pres_attr.py)

Instantiate the President class. Get the first name, last name, and party attributes using getattr().

Exercise 4-2 (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 4-3 (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 4-4 (doubledeco.py)

Write a decorator to double the return value of any function. If a function returns 5, after decoration it should return 10. If it returns "spam", after decoration it should return "spamspam", etc.

Exercise 4-5 (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 5: 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.

pylint usage:

```
pylint filename(s) or directory
pylint -ry filename(s) or directory
```

The **-ry** option says to generate a full detailed report.

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.

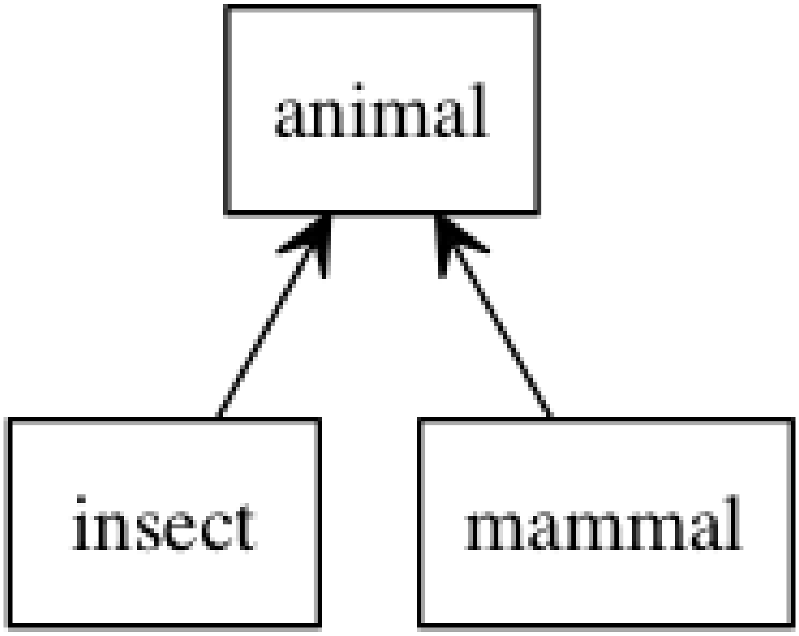
Use ``-A' to search all ancestors, `-p`` to specify the project name, `-o` to specify output type (e.g., **pdf**, **png**, **jpg**).

NOTE **pyreverse** requires **Graphviz**, a graphics tool that must be installed separately from Python

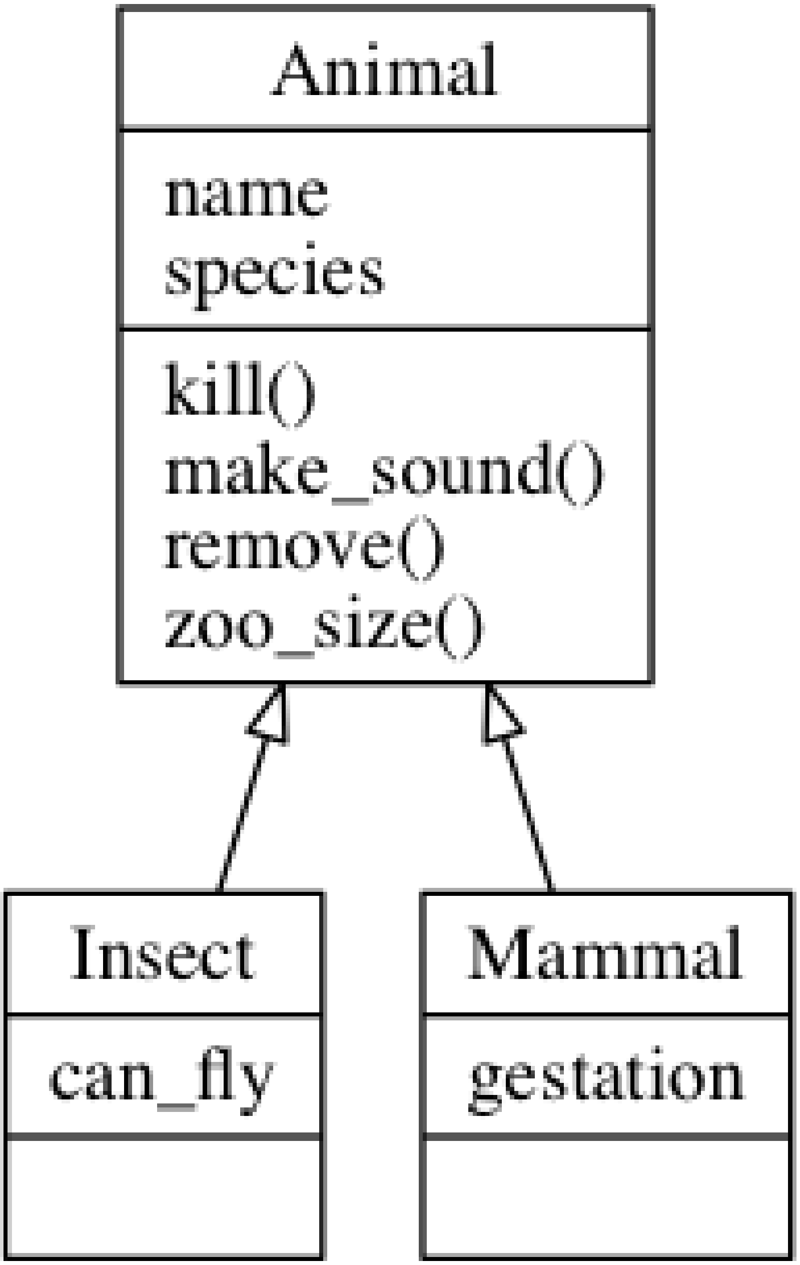
Example

```
pyreverse -o png -p MyProject -A animal.py mammal.py insect.py
```

packages_MyProject.png



classes_MyProject.png



NOTE **pyreverse** requires the (non-Python) Graphviz utility to be installed.

The Python debugger

- Implemented via pdb module
- Supports breakpoints and single stepping
- Based on gdb

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The pdb module provides debugging facilities for Python.

The usual way to use pdb is from the command line:

```
python -mpdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

Starting debug mode

- Syntax

```
python -m pdb script
```

or

```
import pdb
pdb.run('function')
```

pdb is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import some_module
>>> pdb.run('some_module.function_to_text()')
> <string>(0)?()
(Pdb) c    # (c)ontinue
> <string>(1)?()
(Pdb) c    # (c)ontinue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

To get help, type **h** at the debugger prompt.

Stepping through a program

- s single-step, stepping into functions
- n single-step, stepping over functions
- r return from function
- c run to next breakpoint or end

The debugger provides several commands for stepping through a program. Use **s** to step through one line at a time, stepping into functions.

Use **n** to step over functions; use **r** to return from a function; use **c** to continue to next breakpoint or end of program.

Pressing Enter repeats most commands; if the previous command was list, the debugger lists the next set of lines.

Setting breakpoints

- Syntax

```
b list all breakpoints
b linenumber (, condition)
b file:linenumber (, condition)
b function name (, condition)
```

Breakpoints can be set with the b command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The **tbreak** command creates a one-time breakpoint that is deleted after it is hit the first time.

Profiling

- Use the **profile** module from the command line
- Shows where program spends the most time
- Output can be tweaked via options

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

To use the profiler, execute the following at the command line:

```
python -m profile scriptname.py
```

This will output a simple report to STDOUT. You can also specify an output file with the -o option, and the sort order with the -s option. See the docs for more information.

TIP The **pycallgraph2** module (third-party module) 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
from timeit import Timer

setup_code = """
values = []
"""  ①

test_code_one = '''
for i in range(10000):
    values.append(i)
values.clear()
'''  ②

test_code_two = '''
i = 0
while i < 10000:
    values.append(i)
    i += 1
values.clear()
'''  ②

t1 = Timer(test_code_one, setup_code)  ③
t2 = 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:
0.581884131

test two:
0.8679698449999999
```

Chapter 5 Exercises

Exercise 5-1

Pick several of your scripts (from class, or from real life) and run pylint on them.

Exercise 5-2

Use the builtin debugger or one included with your IDE to step through any of the scripts you have written so far.

Chapter 6: Unit Testing 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.

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 xUnit[1]

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 (and does not display) 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**. While newer versions support that name, 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 0.30000000000000004)

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.

It is convenient to put fixtures into a separate 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_capsysbinary(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 5. Pytest Builtin Fixtures

Fixture	Brief Description
cache	Return cache object to persist state between testing sessions.
capsys	Enable capturing of writes (text mode) to **sys.stdout** and **sys.stderr**
capsysbinary	Enable capturing of writes (binary mode) to **sys.stdout** and **sys.stderr**
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 **dict** that will be injected into namespace of doctests
pytestconfig	Session-scoped fixture that returns **_pytest.config.Config** 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 **monkeypatch** fixture providing monkeypatching tools
recwarn	Return **WarningsRecorder** instance that records all warnings emitted by test functions.
tmp_path	Return **pathlib.Path** instance with unique temp directory
tmp_path_factory	Return a **_pytest.tmpdir.TempPathFactory** instance for the test session.
tmpdir	Return **py.path.local** instance unique to each test
tmpdir_factory	Return **TempdirFactory** 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 as 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_runtest_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 3.7.6, pytest-6.2.3, py-1.9.0, pluggy-0.13.1
PyQt5 5.9.2 -- Qt runtime 5.9.7 -- Qt compiled 5.9.6
rootdir: /Users/jstrick/curr/courses/python/examples3
plugins: common-subject-1.0.4, fixture-order-0.1.3, lambda-1.2.0, hypothesis-5.5.4,
arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0, cov-2.11.1, mock-3.3.1, django-4.1.0,
doctestplus-0.5.0, qt-3.3.0, astropy-header-0.1.2, assert-utils-0.2.1
collected 2 items

pytests/stuff/test_stuff.py Hello from setup, <Function test_one>
WHOOPEE
.Hello from setup, <Function test_two>
.

============================== 2 passed in 0.11s ===============================
```

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 3.7.6, pytest-6.2.3, py-1.9.0, pluggy-0.13.1
PyQt5 5.9.2 -- Qt runtime 5.9.7 -- Qt compiled 5.9.6
rootdir: /Users/jstrick/curr/courses/python/examples3
plugins: common-subject-1.0.4, fixture-order-0.1.3, lambda-1.2.0, hypothesis-5.5.4,
arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0, cov-2.11.1, mock-3.3.1, django-4.1.0,
doctestplus-0.5.0, qt-3.3.0, astropy-header-0.1.2, assert-utils-0.2.1
collected 3 items

pytests/test_parametrization.py input 5 result 15:
.input a result aaa:
.input [True] result [True, True, True]:
.

============================== 3 passed in 0.09s ===============================
```

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.marker()`, where *marker* is the marker (label), which can be any alphanumeric string.

Then you can run select tests which contain or match the marker, as described in the next topic.

In addition, you can register markers in the **[pytest]** section of **pytest.ini**, so they will be listed with `pytest --markers`:

```
[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 3.7.6, pytest-6.2.3, py-1.9.0, pluggy-0.13.1
PyQt5 5.9.2 -- Qt runtime 5.9.7 -- Qt compiled 5.9.6
rootdir: /Users/jstrick/curr/courses/python/examples3
plugins: common-subject-1.0.4, fixture-order-0.1.3, lambda-1.2.0, hypothesis-5.5.4,
arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0, cov-2.11.1, mock-3.3.1, django-4.1.0,
doctestplus-0.5.0, qt-3.3.0, astropy-header-0.1.2, assert-utils-0.2.1
collected 3 items / 1 deselected / 2 selected

pytests/test_mark.py ..                                                  [100%]

=============================== warnings summary ===============================
pytests/test_mark.py:4
  /Users/jstrick/curr/courses/python/examples3/pytests/test_mark.py:4:
PytestUnknownMarkWarning: Unknown pytest.mark.alpha - is this a typo?  You can register
custom marks to avoid this warning - for details, see
https://docs.pytest.org/en/stable/mark.html
    @pytest.mark.alpha  ①

pytests/test_mark.py:8
  /Users/jstrick/curr/courses/python/examples3/pytests/test_mark.py:8:
PytestUnknownMarkWarning: Unknown pytest.mark.alpha - is this a typo?  You can register
custom marks to avoid this warning - for details, see
https://docs.pytest.org/en/stable/mark.html
    @pytest.mark.alpha  ①

pytests/test_mark.py:12
  /Users/jstrick/curr/courses/python/examples3/pytests/test_mark.py:12:
PytestUnknownMarkWarning: Unknown pytest.mark.beta - is this a typo?  You can register
custom marks to avoid this warning - for details, see
https://docs.pytest.org/en/stable/mark.html
    @pytest.mark.beta  ②

-- Docs: https://docs.pytest.org/en/stable/warnings.html
================= 2 passed, 1 deselected, 3 warnings in 0.03s ==================
```

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 aas 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 3.7.6, pytest-6.2.3, py-1.9.0, pluggy-0.13.1 --
/Users/jstrick/opt/anaconda3/bin/python
cachedir: .pytest_cache
hypothesis profile 'default' ->
database=DirectoryBasedExampleDatabase('/Users/jstrick/curr/courses/python/examples3/.hyp
othesis/examples')
PyQt5 5.9.2 -- Qt runtime 5.9.7 -- Qt compiled 5.9.6
rootdir: /Users/jstrick/curr/courses/python/examples3
plugins: common-subject-1.0.4, fixture-order-0.1.3, lambda-1.2.0, hypothesis-5.5.4,
arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0, cov-2.11.1, mock-3.3.1, django-4.1.0,
doctestplus-0.5.0, qt-3.3.0, astropy-header-0.1.2, assert-utils-0.2.1
collecting ... collected 5 items

pytests/test_skip.py::test_one PASSED                                    [ 20%]
pytests/test_skip.py::test_two SKIPPED (can not currently test)          [ 40%]
pytests/test_skip.py::test_three SKIPPED (only implemented on Windows)   [ 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 0.03s ==============
```

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

Once the pytest-mock module is installed, it provides a fixture named **mocker**, from which you can create mock objects.

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-patch a library, which temporarily (just during the test) replaces a component with a mock version. The **mocker.patch()** function replaces a component with a mock object. Any calls to the component are now recorded.

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 3.7.6, pytest-6.2.3, py-1.9.0, pluggy-0.13.1
PyQt5 5.9.2 -- Qt runtime 5.9.7 -- Qt compiled 5.9.6
rootdir: /Users/jstrick/curr/courses/python/examples3
plugins: common-subject-1.0.4, fixture-order-0.1.3, lambda-1.2.0, hypothesis-5.5.4,
arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0, cov-2.11.1, mock-3.3.1, django-4.1.0,
doctestplus-0.5.0, qt-3.3.0, astropy-header-0.1.2, assert-utils-0.2.1
collected 1 item

pytests/test_mock_unittest.py .                                          [100%]

============================== 1 passed in 0.02s ===============================
```

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 3.7.6, pytest-6.2.3, py-1.9.0, pluggy-0.13.1
PyQt5 5.9.2 -- Qt runtime 5.9.7 -- Qt compiled 5.9.6
rootdir: /Users/jstrick/curr/courses/python/examples3
plugins: common-subject-1.0.4, fixture-order-0.1.3, lambda-1.2.0, hypothesis-5.5.4,
arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0, cov-2.11.1, mock-3.3.1, django-4.1.0,
doctestplus-0.5.0, qt-3.3.0, astropy-header-0.1.2, assert-utils-0.2.1
collected 1 item

pytests/test_mock_pymock.py .                                            [100%]

============================== 1 passed in 0.02s ===============================
```

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 plugins

- Common plugins
    - **pytest-qt**
    - **pytest-django**

There are some plugins for **pytest** that that integrate various frameworks which would otherwise be difficult to test directly.

The **pytest-qt** plugin provides a `qtbot` fixture that can attach widgets and invoke events. This makes it simpler to test your custom widgets.

The **pytest-django** plugin allows you to run Django with **pytest**-style tests rather than the default **unittest** style.

See https://docs.pytest.org/en/latest/reference/plugin_list.html for a complete list of plugins. There are currently 880 plugins!

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 6 Exercises

Exercise 6-1 (test_president_pytest.py)

Using **pytest**, Create some unit tests 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, you can use **president.py** in the top-level folder of the student guide.

Chapter 7: Database Access

Objectives

- Understand the Python DB API architecture
- Connect to a database
- Execute simple and parameterized queries
- Fetch single and multiple row results
- Get metadata about a query
- Execute non-query statements
- Start transactions and commit or rollback as needed

The DB API

- Most popular DB interface
- Specification, not abstract class
- Many modules for different DBMSs
- Hides actual DBMS implementation

To make database programming simpler, Python has the DB API. This is an API to standardize working with databases. When a package is written to access a database, it is written to conform to the API, and thus programmers do not have to learn a new set of methods and functions.

DB API objects and methods

```
conn = package.connect(server, db)
cursor = conn.cursor()
num_lines = cursor.execute(query)
num_lines = cursor.execute(query-with-placeholders, param-iterable)
num_lines = cursor.executemany(query-with-placeholders, nested-param-iterable)
all_rows = cursor.fetchall()
some_rows = cursor.fetchmany(n)
one_row = cursor.fetchone()
conn.commit()
conn.rollback()
```

Table 6. Available Interfaces (using Python DB API-2.0)

Database	Python package
Firebird (and Interbase)	KInterbasDB
IBM DB2	ibm-db
Informix	informixdb
Ingres	ingmod
Microsoft SQL Server	pymssql
MySQL	pymysql
ODBC	pyodbc
Oracle	cx_oracle
PostgreSQL	psycopg2
SAP DB (also known as "MaxDB")	sapdbapi
SQLite	sqlite3
Sybase	Sybase

NOTE This list is not comprehensive, and there may be additional interfaces to some of the listed DBMSs.

Connecting to a Server

- Import appropriate library
- Use connect() to get a database object
- Specify host, database, username, password

To connect to a database server, import the package for the specific database. Use the package's `connect()` method to get a database object, specifying the host, initial database, username, and password. If the username and password are not needed, use None.

Argument names for the `connect()` method may not be consistent across packages. Most `connect()` methods use individual arguments, such as ***host***, ***database***, etc., but some use a single string argument.

When finished with the connection, call the `close()` method on the connection object.

Many database modules support the context manager (`with` statement), and will automatically close the database when the `with` block is exited. Check the documentation to see how this is implemented for a specific database.

Example

```
import pymysql

conn = pymysql.connect (host = "dbserver",
                           user = "adeveloper",
                           passwd = "s3cr3t",
                           db = "samples")
# Interact with database here ...
conn.close()
```

```
import sqlite3

with sqlite3.connect('sample.db') as conn:
    # Interact with database here ...
```

Table 7. connect() examples

Package	Database	Connection
IBM DB2	ibm-db	`import ibm_db_dbi as db2 + conn = db2.connect("DATABASE=testdb;HOSTNAME=localhost;PORT=50000;PROTOCOL=TCPIP;UID=db2inst1;PWD=scripts;", "", "")`
cx-Oracle	Oracle	`ip = 'localhost' + port = 1521 + SID = 'YOURSIDHERE' + dsn_tns = cx_Oracle.makedsn(ip, port, SID) + + db = cx_Oracle.connect('adeveloper', '$3cr3t', dsn_tns)`
PostgreSQL	psychopg	`psycopg2.connect (''' + host='localhost' + user='adeveloper' + password='$3cr3t' + dbname='testdb' + ''')` *note: connect() has one (string) parameter, not multiple parameters*

MS-SQL	pymssql	`pymssql.connect ( + host="localhost", + user="adeveloper", + passwd="$3cr3t", + db="testdb", + )`
		
		`pymssql.connect ( + dsn="DSN", + )`

MySQL	pymysql	`pymysql.connect ( + host="localhost", + user="adeveloper", + passwd="$3cr3t", + db="testdb", + )`

ODBC-compliant DB	pyodbc	`pyodbc.connect(''' + DRIVER={SQL Server}; + SERVER=localhost; + DATABASE=testdb; + UID=adeveloper; + PWD=$3cr3t + ''')`
		
		`pyodbc.connect('DSN=testdsn;PWD=$3cr3t')`
		note: connect() has one (string) parameter, not multiple parameters

SqlLite3	sqlite3	`sqlite3.connect('testdb')` *on-disk database (single file)* `sqlite3.connect(':memory:')` *in-memory database*

Creating a Cursor

- Cursor can execute SQL statements
- Create with `cursor()` method
- Multiple cursors available
    - Standard cursor
        - Returns tuples
    - Other cursors
        - Returns dictionaries
        - Leaves data on server

Once you have a connection object, you can call `cursor()` to create a cursor object. A cursor is an object that can execute SQL code and fetch results. One connection may have one or more active cursors.

The default cursor for most packages returns each row as a tuple of values. There are different types of cursors that can return data in different formats, or that control whether data is stored on the client or the server.

NOTE

See **db_*.py** for examples using DB2, Postgres, MySQL, and MS-SQL. Most of the **sqlite3** examples in this chapter are also implemented for MySQL, Postgres, and DB2, plus a few extras.

Example

```
import sqlite3
conn = sqlite3.connect("sample.db")
cursor = conn.cursor()
```

Executing a query statement

- Gets all data from query
- Use `_cursor_.fetch{splat}` to retrieve.
- Returns # rows in result set

Once you have a cursor, you can use it to execute queries via the `execute()` method. The first argument to `execute()` is a string containing one SQL statement.

For queries, `__cursor__.execute()` returns the number of rows in the result set.

> **NOTE** In Sqlite3, `__cursor__.execute()` returns the cursor object, so you can say `__cursor__.execute(__query__).fetchall()`.

Example

```
cursor.execute("select hostname,ostype,user from hostinfo")
cursor.execute('insert into hostinfo values
("foo",5,"2.6","arch","net",2055,3072,"bob",0)')
```

Fetching Data

- Use one of the fetch methods from the cursor object
- Syntax
  - `rec = cursor.fetchone()`
  - `recs = cursor.fetchall()`
  - `recs = cursor.fetchmany()`

Cursors provide three methods for returning query results.

`fetchone()` returns the next available row from the query results.

`fetchall()` returns a tuple of all rows.

`fetchmany(n)` returns up to n rows. This is useful when the query returns a large number of rows.

In all cases, each row is returned as a tuple of values.

Example

db_sqlite_basics.py

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

import sqlite3

with sqlite3.connect("../DATA/presidents.db") as conn:  ①

    cursor = conn.cursor()  ②

    # select first name, last name from all presidents
    cursor.execute('''
        select *
        from presidents
    ''')  ③

    print("Sqlite3 does not provide a row count\n")  ④

    for row in cursor.fetchall():  ⑤
        print(row)
#        print(' '.join(row))  ⑥
```

① connect to the database

② get a cursor object

③ execute a SQL statement

④ (included for consistency with other DBMS modules)

⑤ fetchall() returns all rows

⑥ each row is a tuple

db_sqlite_basics.py

```
(37, 'Nixon', 'Richard Milhous', '1969-01-20', '1974-08-09', 'Yorba Linda', 'California', '1913-01-09', '1994-04-22', 'Republican')
(38, 'Ford', 'Gerald Rudolph', '1974-08-09', '1977-01-20', 'Omaha', 'Nebraska', '1913-07-14', '2006-12-26', 'Republican')
(39, 'Carter', "James Earl 'Jimmy'", '1977-01-20', '1981-01-20', 'Plains', 'Georgia', '1924-10-01', None, 'Democratic')
(40, 'Reagan', 'Ronald Wilson', '1981-01-20', '1989-01-20', 'Tampico', 'Illinois', '1911-02-06', '2004-06-05', 'Republican')
(41, 'Bush', 'George Herbert Walker', '1989-01-20', '1993-01-20', 'Milton', 'Massachusetts', '1924-06-12', None, 'Republican')
(42, 'Clinton', "William Jefferson 'Bill'", '1993-01-20', '2001-01-20', 'Hope', 'Arkansas', '1946-08-19', None, 'Democratic')
(43, 'Bush', 'George Walker', '2001-01-20', '2009-01-20', 'New Haven', 'Connecticut', '1946-07-06', None, 'Republican')
(44, 'Obama', 'Barack Hussein', '2009-01-20', '2017-01-20', 'Honolulu', 'Hawaii', '1961-08-04', None, 'Democratic')
(45, 'Trump', 'Donald J', '2017-01-20', '2021-01-20', 'Queens, NYC', 'New York', '1946-06-14', None, 'Republican')
(46, 'Biden', 'Joseph Robinette', '2021-01-20', None, 'Scranton', 'Pennsylvania', '1942-11-10', None, 'Democratic')
```

Non-query statements

- Updates database
- Returns # rows in result set
- Must commit changes

The `execute()`method is also used to execute non-query statements.

As with queries, the first argument is a string containing one SQL statement. The optional second argument is an iterable of values to fill in placeholders in a parameterized statement.

For most DB packages, `execute()` returns the number of rows affected.

Example

db_sqlite_add_row.py

```
#!/usr/bin/env python
from datetime import date
import sqlite3


with sqlite3.connect("../DATA/presidents.db") as s3conn:  ①

    sql_insert = """
    insert into presidents
    (termnum, lastname, firstname, birthdate, deathdate, birthplace, birthstate,
termstart, termend, party)
    values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
    """



    new_row_data = [47, 'Ramirez', 'Mary', date(1968, 9, 22), None,
                    'Topeka', 'Kansas', date(2024, 1, 20), None, 'Independent']

    cursor = s3conn.cursor()

    try:
        cursor.execute(sql_insert, new_row_data)
    except (sqlite3.OperationalError, sqlite3.DatabaseError, sqlite3.DataError) as err:
        print(err)
        s3conn.rollback()
    else:
        s3conn.commit()

    cursor.close()
```

Example

db_sqlite_delete_row.py

```
#!/usr/bin/env python
from datetime import date
import sqlite3


with sqlite3.connect("../DATA/presidents.db") as conn:  ①

    sql_delete = """
    delete from presidents
    where TERMNUM = 47
    """

    cursor = conn.cursor()

    try:
        cursor.execute(sql_delete)
    except (sqlite3.DatabaseError, sqlite3.OperationalError, sqlite3.DataError) as err:
        print(err)
        conn.rollback()
    else:
        conn.commit()

    cursor.close()
```

SQL Injection

- "Hijacks" SQL code
- Result of string formatting
- Always use parameterized statements

One kind of vulnerability in SQL code is called *SQL injection*. This occurs when an attacker embeds SQL commands in input data. This can happen when naively using string formatting to build SQL statements.

Since the programmer is generating the SQL code as a string, there is no way to check for malicious SQL code. It is best practice to use parameterized statements.

Example

db_sql_injection.py

```
#!/usr/bin/env python
#
good_input = 'Google'
malicious_input = "'; drop table customers; -- "  ①

naive_format = "select * from customers where company_name = '{}' and company_id != 0"

good_query = naive_format.format(good_input)  ②
malicious_query = naive_format.format(malicious_input)  ②

print("Good query:")
print(good_query)  ③
print()

print("Bad query:")
print(malicious_query)  ④
```

① input would come from a web form, for instance

② string formatting naively adds the user input to a field, expecting only a customer name

③ non-malicious input works fine

④ query now drops a table (-- is SQL comment) *db_sql_injection.py*

```
Good query:
select * from customers where company_name = 'Google' and company_id != 0

Bad query:
select * from customers where company_name = ''; drop table customers; -- ' and
company_id != 0
```

NOTE see http://www.xkcd.com/327 for a well-known web comic on this subject.

Parameterized Statements

- Prevent SQL injection
- More efficient updates
- Use placeholders in query
  - Placeholders vary by DB
- Pass iterable of parameters
- Use cursor.execute() or cursor.executemany()

For efficiency, you can iterate over of sequence of input datasets when performing a non-query SQL statement. The `execute()` method takes a query, plus an iterable of values to fill in the placeholders. The database manager will only parse the query once, then reuse it for subsequent calls to execute().

All SQL statements may be parameterized, including queries.

Parameterized statements also protect against SQL injection attacks.

Different database modules use different placeholders. To see what kind of placeholder a module uses, check MODULE.paramstyle. Types include *pyformat*, meaning *%s*, and *qmark*, meaning *?*.

The `executemany()` method takes a query, plus an iterable of iterables. It will call `execute()` once for each nested iterable.

Table 8. Placeholders for SQL Parameters

Python package	Placeholder for parameters
pymysql	%s
cx_oracle	:param_name
pyodbc	?
pymssql	%d for int, %s for str, etc.
Psychopg	%s or %(param_name)s
sqlite3	? or :param_name

TIP with the exception of **pymssql** the same placeholder is used for all column types.

Example

db_sqlite_parameterized.py

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

import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()

    party_query = '''
    select firstname, lastname
    from presidents
        where party = ?
    '''  ①

    for party in 'Federalist', 'Whig':
        print(party)
        s3cursor.execute(party_query, (party,))  ②
        print(s3cursor.fetchall())
        print()
```

① ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use different placeholders

② second argument to execute() is iterable of values to fill in placeholders from left to right

db_sqlite_parameterized.py

```
Federalist
[('John', 'Adams')]

Whig
[('William Henry', 'Harrison'), ('John', 'Tyler'), ('Zachary', 'Taylor'), ('Millard',
'Fillmore')]
```

Example

db_sqlite_bulk_insert.py

```
#!/usr/bin/env python
import sqlite3
import os
import csv

DATA_FILE = '../DATA/fruit_data.csv'

DB_NAME = 'fruits.db'
DB_TABLE = 'fruits'

SQL_CREATE_TABLE = f"""
create table {DB_TABLE} (
    id integer primary key,
    name varchar(30),
    unit varchar(30),
    unitprice decimal(6, 2)
)
"""  ②

SQL_INSERT_ROW = f'''
insert into {DB_TABLE} (name, unit, unitprice) values (?, ?, ?)
'''  ③

SQL_SELECT_ALL = f"""
select name, unit, unitprice from {DB_TABLE}
"""


def main():
    """
    Program entry point.

    :return: None
    """
    conn, cursor = get_connection()
    create_database(cursor)
    populate_database(conn, cursor)
    read_database(cursor)

    cursor.close()
    conn.close()


def get_connection():
    """
    Get a connection to the PRODUCE database

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)  ④

    conn = sqlite3.connect(DB_NAME)  ⑤
    cursor = conn.cursor()
    return conn, cursor


def create_database(cursor):
    """
    Create the fruit table

    :param conn: The database connection
    :return: None
    """
    cursor.execute(SQL_CREATE_TABLE)  ⑥


def populate_database(conn, cursor):
    """
    Add rows to the fruit table

    :param conn: The database connection
    :return: None
    """
    with open(DATA_FILE) as file_in:
        fruit_data = csv.reader(file_in, quoting=csv.QUOTE_NONNUMERIC)

        try:
            cursor.executemany(SQL_INSERT_ROW, fruit_data)  ⑦
        except sqlite3.DatabaseError as err:
            print(err)
            conn.rollback()
        else:
            conn.commit()  ⑧

def read_database(cursor):
    cursor.execute(SQL_SELECT_ALL)
    for name, unit, unitprice in cursor.fetchall():
        print('{:12s} {:5.2f}/{}'.format(name, unitprice, unit))


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

① set name of database

② SQL statement to create table

③ parameterized SQL statement to insert one record

④ remove existing database if it exists

⑤ connect to (new) database

⑥ run SQL to create table

⑦ iterate over list of pairs and add each pair to the database

⑧ commit the inserts; without this, no data would be saved

⑨ build list of tuples containing fruit, price pairs

db_sqlite_bulk_insert.py

```
pomegranate   0.99/each
cherry        2.25/pound
apricot       3.49/pound
date          1.20/pound
apple         0.55/pound
lemon         0.69/each
kiwi          0.88/each
orange        0.49/each
lime          0.49/each
watermelon    4.50/each
guava         2.88/pound
papaya        1.79/pound
fig           2.29/pound
pear          1.10/pound
banana        0.65/pound
```

Dictionary Cursors

- Indexed by column name
- Not standardized in the DB API

The standard cursor provided by the DB API returns a tuple for each row. Most DB packages provide other kinds of cursors, including user-defined versions.

A very common cursor is a dictionary cursor, which returns a dictionary for each row, where the keys are the column names. Each package that provides a dictionary cursor has its own way of providing the dictionary cursor, although they all work the same way.

Table 9. Dictionary Cursors

Python package	**How to get a dictionary cursor**

pymysql	`import pymysql.cursors + conn = pymysql.connect(..., + cursorclass = pymysql.cursors.DictCursor + ) + dcur = conn.cursor()`
	all cursors will be dict cursors
	`dcur = conn.cursor( pymysql.cursors.DictCursor)`
	only this cursor will be a dict cursor

cx_oracle	*Not available*
pyodbc	*Not available*
pgdb	*Not available*
pymssql	`conn = pymssql.connect (..., as_dict=True) + dcur = conn.cursor()`
psychopg	`import psycopg2.extras + dcur = conn.cursor(cursor_factory=psycopg.extras.DictCursor)`

sqlite3	`conn = sqlite3.connect (...,`
	`row_factory=sqlite3.Row) + dcur = conn.cursor()`
	`conn.row_factory = sqlite3.Row + dcur = conn.cursor()`

Example

db_sqlite_dict_cursor.py

```
#!/usr/bin/env python
import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")
# uncomment to make _all_ cursors dictionary cursors
# conn.row_factory = sqlite3.Row

NAME_QUERY = '''
    select firstname, lastname
    from presidents
    where termnum < 5
'''

cur = s3conn.cursor()

# select first name, last name from all presidents
cur.execute(NAME_QUERY)

for row in cur.fetchall():
    print(row)
print('-' * 50)

dict_cursor = s3conn.cursor()  ①

# make _this_ cursor a dictionary cursor
dict_cursor.row_factory = sqlite3.Row  ②

# select first name, last name from all presidents
dict_cursor.execute(NAME_QUERY)

for row in dict_cursor.fetchall():
    print(row['firstname'], row['lastname'])  ③

print('-' * 50)
```

db_sqlite_dict_cursor.py

```
('George', 'Washington')
('John', 'Adams')
('Thomas', 'Jefferson')
('James', 'Madison')
--------------------------------------------------
George Washington
John Adams
Thomas Jefferson
James Madison
--------------------------------------------------
```

Metadata

- cursor.description returns tuple of tuples
- Fields
    - name
    - type_code
    - display_size
    - internal_size
    - precision
    - scale
    - null_ok

Once a query has been executed, the cursor's description attribute is a tuple with metadata about the columns in the query. It contains one tuple for each column in the query, containing 7 values describing the column.

For instance, to get the names of the columns, you could say `names = [d[0] for d in cursor.description]`

For non-query statements, cursor.description returns None.

The names are based on the query (with possible aliases), and not necessarily on the names in the table.

NOTE Sqlite3 only provides column names.

Generic alternate cursors

- Create generator function
  - Get column names from *cursor*.description()
  - For each row
    - Make object from column names and values
      - Dictionary
      - Named tuple
      - Dataclass

Many database modules have a dictionary cursor built in. For those that don't the `iterrows_asdict()` function can be used with a cursor from any DB API-compliant package.

The example uses the metadata from the cursor to get the column names, and forms a dictionary by zipping the column names with the column values. `db_iterrows` also provides `iterrows_asnamedtuple()`, which returns each row as a named tuple.

The functions in `db_iterrows` return generator objects. When you loop over the generator object, each element is a dictionary or a named tuple, depending on which function you called.

Example

db_iterrows.py

```
#!/usr/bin/env python
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
from collections import namedtuple
from dataclasses import make_dataclass

def get_column_names(cursor):
    return [desc[0] for desc in cursor.description]

def iterrows_asdict(cursor):
    '''Generate rows as dictionaries'''
    column_names = get_column_names(cursor)
    for cursor_row in cursor.fetchall():
        row_dict = dict(zip(column_names, cursor_row))
        yield row_dict

def iterrows_asnamedtuple(cursor):
    '''Generate rows as named tuples'''
    column_names = get_column_names(cursor)
    Row = namedtuple('Row', column_names)
    for row in cursor.fetchall():
        yield Row(*row)

def iterrows_asdataclass(cursor):
    '''Generate rows as dataclass instances'''
    column_names = get_column_names(cursor)
    Row = make_dataclass('row_tuple', column_names)

    for cursor_row in cursor.fetchall():
        row_instance = Row(*cursor_row)
        yield row_instance
```

db_iterrows.py

```
```

Transactions

- Transactions allow safer control of updates
- commit() to save transactions
- rollback() to discard

Sometimes a database task involves more than one change to your database (i.e., more than one SQL statement). You don't want the first SQL statement to succeed and the second to fail; this would leave your database in a corrupt state.

To be certain of data integrity, use **transactions**. This lets you make multiple changes to your database and only commit the changes if all the SQL statements were successful.

For all packages using the Python DB API, a transaction is started when you connect. At any point, you can call `__CONNECTION__.commit()` to save the changes, or `__CONNECTION__.rollback()` to discard the changes. If you don't call `commit()` after modify a table, the data will not be saved.

You can also turn on *autocommit*, which calls `commit()` after every statement. See the table below for how autocommit is implemented in various DB packages.

Table 10. How to turn on autocommit

Package	Method/Attribute
cx_oracle	`__conn__.autocommit = True`
ibm_db_api	`__conn__.set_autocommit(True)`
pymysql	`` pymysql.connect(..., autocommit=True) + __or__ + `__conn__.autocommit(True) ``
psycopg2	`__conn__.autocommit = True`
sqlite3	`sqlite3.connect(__dbname__, isolation_level=None)`

NOTE **pymysql** only supports transaction processing when using the **InnoDB** engine

Example

```
try:
    for info in list_of_tuples:
         cursor.execute(query,info)
except SQLError:
    dbconn.rollback()
else:
    dbconn.commit()
```

Object-relational Mappers

- No SQL required
- Maps a class to a table
- All DB work is done by manipulating objects
- Most popular Python ORMs
  - SQLAlchemy
  - Django (which is a complete web framework)

An Object-relational mapper is a module or framework that creates a level of abstraction above the actual database tables and SQL queries. As the name implies, a Python class (object) is mapped to the actual table.

The two most popular Python ORMs are SQLAlchemy which is a standalone ORM, and Django ORM. Django is a comprehensive Web development framework, which provides an ORM as a subpackage. SQLAlchemy is the most fully developed package, and is the ORM used by Flask and some other Web development frameworks.

Instead of querying the database, you call a search method on an object representing a table. To add a row to the table, you create a new instance of the table class, populate it, and call a method like save(). You can create a large, complex database system, complete with foreign keys, composite indices, and all the other attributes near and dear to a DBA, without writing the first line of SQL.

You can use Python ORMs in two ways.

One way is to design the database with the ORM. To do this, you create a class for each table in the database, specifying the columns with predefined classes from the ORM. Then you run an ORM command which executes the queries needed to build the database. If you need to make changes, you update the class definitions, and run an ORM command to synchronize the actual DBMS to your classes.

The second way is to map tables to an existing database. You create the classes to match the schemas that have already been defined in the database. Both SQLAlchemy and the Django ORM have tools to automate this process.

NoSQL

- Non-relational database
- Document-oriented
- Can be hierarchical (nested)
- Examples
    - MongoDB
    - Cassandra
    - Redis

A current trend in data storage are called "NoSQL" or non-relational databases. These databases consist of *documents*, which are indexed, and may contain nested data.

NoSQL databases don’t contain tables, and do not have relations.

While relational databases are great for tabular data, they are not as good a fit for nested data. Geospatial, engineering diagrams, and molecular modeling can have very complex structures. It is possible to shoehorn such data into a relational database, but a NoSQL database might work much better. Another advantage of NoSQL is that it can adapt to changing data structures, without having to rebuild tables if columns are added, deleted, or modified.

Some of the most common NoSQL database systems are MongoDB, Cassandra and Redis.

Example

mongodb_example.py

```
#!/usr/bin/env python
import re
from pymongo import MongoClient, errors

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

mc = MongoClient()  ②

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

db = mc["presidents"]  ④

coll = db.presidents  ⑤

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

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

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

for field in FIELD_NAMES:
    print("{0:15s} {1}".format(field.upper(), abe[field]))  ⑩

print('-' * 50)

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

rx_lastname = re.compile('^roo', re.IGNORECASE)
for president in coll.find({'lastname': rx_lastname}):  ⑫
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

for president in coll.find({"birthstate": 'Virginia'}):  ⑬
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))

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


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

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


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

animals = db.animals

print(animals, '\n')

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

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

① define some field name

② get a Mongo client

③ delete *presidents* database if it exists

④ create a new database named *presidents*

⑤ get the collection from presidents db

⑥ open a data file

⑦ insert a record into collection

⑧ get list of collections

⑨ search collection for doc where termnumber == 16

⑩ print all fields for one record

⑪ loop through all records in collection

⑫ find record using regular expression

⑬ find record searching multiple fields

⑭ delete record

⑮ get count of records

mongodb_example.py

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

wombat
ocelot
honey badger
```

Chapter 7 Exercises

Exercise 7-1 (president_sqlite.py)

For this exercise, you can use the SQLite3 database provided, or use your own DBMS. The mkpres.sql script is generic and should work with any DBMS to create and populate the presidents table. The SQLite3 database is named **presidents.db** and is located in the DATA folder of the student files.

The data has the following layout

Table 11. Layout of President Table

Field Name	**Data Type**	**Null**	**Default**
termnum	int(11)	YES	NULL
lastname	varchar(32)	YES	NULL
firstname	varchar(64)	YES	NULL
termstart	date	YES	NULL
termend	date	YES	NULL
birthplace	varchar(128)	YES	NULL
birthstate	varchar(32)	YES	NULL
birthdate	date	YES	NULL
deathdate	date	YES	NULL
party	varchar(32)	YES	NULL

Refactor the **president.py** module to get its data from this table, rather than from a file. Re-run your previous scripts that used president.py; now they should get their data from the database, rather than from the flat file.

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

Exercise 7-2 (add_pres_sqlite.py)

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

SQL syntax for adding a record is

```
INSERT INTO table ("COL1-NAME",..) VALUES ("VALUE1",...)
```

To do a parameterized insert (the right way!):

```
INSERT INTO table ("COL1-NAME",..) VALUES (%s,%s,...)  # MySQL
INSERT INTO table ("COL1-NAME",..) VALUES (?,?,...)    # SQLite
```

or whatever your database uses as placeholders

NOTE There are also MySQL versions of the answers.

Chapter 8: Multiprogramming

Objectives

- Understand multiprogramming
- Differentiate between threads and processes
- Know when threads benefit your program
- Learn the limitations of the GIL
- Create a threaded application
- Implement a queue object
- Use the multiprocessing module
- Develop a multiprocessing application

Multiprogramming

- Parallel processing
- Three main ways to achieve it
    - threading
    - multiple processes
    - asynchronous communication
- All three supported in standard library

Computer programs spend a lot of their time doing nothing. This occurs when the CPU is waiting for the relatively slow disk subsystem, network stack, or other hardware to fetch data.

Some applications can achieve more throughput by taking advantage of this slack time by seemingly doing more than one thing at a time. With a single-core computer, this doesn’t really happen; with a multicore computer, an application really can be executing different instructions at the same time. This is called multiprogramming.

The three main ways to implement multiprogramming are threading, multiprocessing, and asynchronous communication:

Threading subdivides a single process into multiple subprocesses, or threads, each of which can be performing a different task. Threading in Python is good for IO-bound applications, but does not increase the efficiency of compute-bound applications.

Multiprocessing forks (spawns) new processes to do multiple tasks. Multiprocessing is good for both CPU-bound and IO-bound applications.

Asynchronous communication uses an event loop to poll multiple I/O channels rather than waiting for one to finish. Asynch communication is good for IO-bound applications.

The standard library supports all three.

What Are Threads?

- Like processes (but lighter weight)
- Process itself is one thread
- Process can create one more more additional threads
- Similar to creating new processes with fork()

Modern operating systems (OSs) use time-sharing to manage multiple programs which appear to the user to be running simultaneously. Assuming a standard machine with only one CPU, that simultaneity is only an illusion, since only one program can run at a time, but it is a very useful illusion. Each program that is running counts as a process. The OS maintains a process table, listing all current processes. Each process will be shown as currently being in either Run state or Sleep state.

A thread is like a process. A thread might even be a process, depending on the implementation. In fact, threads are sometimes called “lightweight” processes, because threads occupy much less memory, and take less time to create, than do processes.

A process can create any number of threads. This is similar to a process calling the fork() function. The process itself is a thread, and could be considered the "main" thread.

Just as processes can be interrupted at any time, so can threads.

The Python Thread Manager

- Python uses underlying OS's threads
- Alas, the GIL – Global Interpreter Lock
- Only one thread runs at a time
- Python interpreter controls end of thread's turn
- Cannot take advantage of multiple processors

Python "piggybacks" on top of the OS's underlying threads system. A Python thread is a real OS thread. If a Python program has three threads, for instance, there will be three entries in the OS's thread list.

However, Python imposes further structure on top of the OS threads. Most importantly, there is a global interpreter lock, the famous (or infamous) GIL. It is set up to ensure that (a) only one thread runs at a time, and (b) that the ending of a thread's turn is controlled by the Python interpreter rather than the external event of the hardware timer interrupt.

The fact that the GIL allows only one thread to execute Python bytecode at a time simplifies the Python implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines. The takeaway is that Python does not currently take advantage of multi-processor hardware.

NOTE *GIL* is pronounced "jill", according to Guido__

For a thorough discussion of the GIL and its implications, see http://www.dabeaz.com/python/UnderstandingGIL.pdf.

The threading Module

- Provides basic threading services
- Also provides locks
- Three ways to use threads
  - Instantiate **Thread** with a function
  - Subclass **Thread**
  - Use pool method from **multiprocessing** module

The threading module provides basic threading services for Python programs. The usual approach is to subclass threading.Thread and provide a run() method that does the thread’s work.

Threads for the impatient

- No class needed (created "behind the scenes")
- For simple applications

For many threading tasks, all you need is a run() method and maybe some arguments to pass to it.

For simple tasks, you can just create an instance of Thread, passing in positional or keyword arguments.

Example

thr_noclass.py

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

import threading
import random
import time


def doit(num):  ①
    time.sleep(random.randint(1, 3))
    print("Hello from thread {}".format(num))


for i in range(10):
    t = threading.Thread(target=doit, args=(i,))  ②
    t.start()  ③

print("Done.")  ④
```

① function to launch in each thread

② create thread

③ launch thread

④ "Done" is printed immediately — the threads are "in the background" *thr_noclass.py*

```
Done.
Hello from thread 0
Hello from thread 6
Hello from thread 2
Hello from thread 3
Hello from thread 7
Hello from thread 8
Hello from thread 9
Hello from thread 1
Hello from thread 4
Hello from thread 5
```

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 1
Hello from thread 3
Hello from thread 4
Hello from thread 9
Hello from thread 0
Hello from thread 2
Hello from thread 7
Hello from thread 5
Hello from thread 6
Hello from thread 8
```

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 9 (fig)
Hello from thread 10 (elderberry)
Hello from thread 5 (papaya)
Hello from thread 4 (peach)
Hello from thread 6 (cherry)
Hello from thread 1 (apple)
Hello from thread 3 (mango)
Hello from thread 7 (lemon)
Hello from thread 2 (banana)
Hello from thread 8 (watermelon)
['FIG', 'ELDERBERRY', 'PAPAYA', 'PEACH', 'CHERRY', 'APPLE', 'MANGO', 'LEMON', 'BANANA',
'WATERMELON']
```

Using queues

- Queue contains a list of objects
- Sequence is FIFO
- Worker threads can pull items from the queue
- Queue structure has builtin locks

Threaded applications often have some sort of work queue data structure. When a thread becomes free, it will pick up work to do from the queue. When a thread creates a task, it will add that task to the queue.

The queue must be guarded with locks. Python provides the Queue module to take care of all the lock creation, locking and unlocking, and so on, so that you don't have to bother with it.

Example

thr_queue.py

```
#!/usr/bin/env python
import random
import queue
from threading import Thread, Lock as tlock
import time

NUM_ITEMS = 25000
POOL_SIZE = 100

q = queue.Queue(0)  ①

shared_list = []
shlist_lock = tlock()  ②
stdout_lock = tlock()  ②


class RandomWord():  ③
    def __init__(self):
        with open('../DATA/words.txt') as words_in:
            self._words = [word.rstrip('\n\r') for word in words_in.readlines()]
        self._num_words = len(self._words)

    def __call__(self):
        return self._words[random.randrange(0, self._num_words)]
```

```
class Worker(Thread):  ④

    def __init__(self, name):  ⑤
        Thread.__init__(self)
        self.name = name

    def run(self):  ⑥
        while True:
            try:
                s1 = q.get(block=False)  ⑦
                s2 = s1.upper() + '-' + s1.upper()
                with shlist_lock:  ⑧
                    shared_list.append(s2)

            except queue.Empty:  ⑨
                break


⑩
random_word = RandomWord()
for i in range(NUM_ITEMS):
    w = random_word()
    q.put(w)

start_time = time.ctime()

⑪
pool = []
for i in range(POOL_SIZE):
    worker_name = "Worker {:c}".format(i + 65)
    w = Worker(worker_name)  ⑫
    w.start()  ⑬
    pool.append(w)

for t in pool:
    t.join()  ⑭

end_time = time.ctime()

print(shared_list[:20])

print(start_time)
print(end_time)
```

① initialize empty queue

② create locks

③ define callable class to generate words

④ worker thread

⑤ thread constructor

⑥ function invoked by thread

⑦ get next item from thread

⑧ acquire lock, then release when done

⑨ when queue is empty, it raises Empty exception

⑩ fill the queue

⑪ populate the threadpool

⑫ add thread to pool

⑬ launch the thread

⑭ wait for thread to finish

thr_queue.py

```
['UNCELEBRATED-UNCELEBRATED', 'REHYDRATE-REHYDRATE', 'RESTORATION-RESTORATION',
'BOMBAZINES-BOMBAZINES', 'RESILES-RESILES', 'MESHING-MESHING', 'SENECIOS-SENECIOS',
'CLAMPS-CLAMPS', 'THIOL-THIOL', 'TROPHICALLY-TROPHICALLY', 'NECESSITATIONS-
NECESSITATIONS', 'COUNTERIRRITANT-COUNTERIRRITANT', 'NAVIGATOR-NAVIGATOR', 'ALKALIZE-
ALKALIZE', 'OVIPAROUS-OVIPAROUS', 'STRIKEOUTS-STRIKEOUTS', 'BROIDERIES-BROIDERIES',
'PROJECTIONIST-PROJECTIONIST', 'PLUMPEST-PLUMPEST', 'QUAKIEST-QUAKIEST']
Fri Nov 12 13:45:03 2021
Fri Nov 12 13:45:03 2021
```

Debugging threaded Programs

- Harder than non-threaded programs
- Context changes abruptly
- Use pdb.trace
- Set breakpoint programmatically

Debugging is always tough with parallel programs, including threads programs. It's especially difficult with pre-emptive threads; those accustomed to debugging non-threads programs find it rather jarring to see sudden changes of context while single-stepping through code. Tracking down the cause of deadlocks can be very hard. (Often just getting a threads program to end properly is a challenge.)

Another problem which sometimes occurs is that if you issue a "next" command in your debugging tool, you may end up inside the internal threads code. In such cases, use a "continue" command or something like that to extricate yourself.

Unfortunately, threads debugging is even more difficult in Python, at least with the basic PDB debugger.

One cannot, for instance, simply do something like this:

```
pdb.py buggyprog.py
```

This is because the child threads will not inherit the PDB process from the main thread. You can still run PDB in the latter, but will not be able to set breakpoints in threads.

What you can do, though, is invoke PDB from within the function which is run by the thread, by calling pdb.set trace() at one or more points within the code:

```
import pdb
pdb.set_trace()
```

In essence, those become breakpoints.

For example, we could add a PDB call at the beginning of a loop:

```
import pdb
while True:
    pdb.set_trace() # app will stop here and enter debugger
    k = c.recv(1)
    if k == ▯▯:
        break
```

You then run the program as usual, NOT through PDB, but then the program suddenly moves into debugging mode on its own. At that point, you can then step through the code using the n or s commands, query the values of variables, etc.

PDB’s c (“continue”) command still works. Can you still use the b command to set additional breakpoints? Yes, but it might be only on a one-time basis, depending on the context.

The multiprocessing module

- Drop-in replacement for the threading module
- Doesn’t suffer from GIL issues
- Provides interprocess communication
- Provides process (and thread) pooling

The multiprocessing module can be used as a replacement for threading. It uses processes rather than threads to spread out the work to be done. While the entire module doesn’t use the same API as threading, the multiprocessing.Process object is a drop-in replacement for a threading.Thread object. Both use run() as the overridable method that does the work, and both use start() to launch. The syntax is the same to create a process without using a class:

```
def myfunc(filename):
      pass

p = Process(target=myfunc, args=('/tmp/info.dat', ))
```

This solves the GIL issue, but the trade-off is that it’s slightly more complicated for tasks (processes) to communicate. However, the module does the heavy lifting of creating pipes to share data.

The **Manager** class provided by multiprocessing allows you to create shared variables, as well as locks for them, which work across processes.

NOTE On windows, processes must be started in the "if __name__ == __main__" block, or they will not work.

Example

multi_processing.py

```
#!/usr/bin/env python
import sys
import random
from multiprocessing import Manager, Lock, Process, Queue, freeze_support
from queue import Empty
import time

NUM_ITEMS = 25000  ①
POOL_SIZE = 100
```

```
class RandomWord():  ②
    def __init__(self):
        with open('../DATA/words.txt') as words_in:
            self._words = [word.rstrip('\n\r') for word in words_in]
        self._num_words = len(self._words)

    def __call__(self):  ③
        return self._words[random.randrange(0, self._num_words)]


class Worker(Process):  ④

    def __init__(self, name, queue, lock, result):  ⑤
        Process.__init__(self)
        self.queue = queue
        self.result = result
        self.lock = lock
        self.name = name

    def run(self):  ⑥
        while True:
            try:
                word = self.queue.get(block=False)  ⑦
                word = word.upper()  ⑧
                with self.lock:
                    self.result.append(word)  ⑨

            except Empty:  ⑩
                break


if __name__ == '__main__':
    if sys.platform == 'win32':
        freeze_support()

    word_queue = Queue()  ⑪

    manager = Manager()  ⑫
    shared_result = manager.list()  ⑬
    result_lock = Lock()  ⑭

    random_word = RandomWord()  ⑮
    for i in range(NUM_ITEMS):
        w = random_word()
        word_queue.put(w)  ⑯

    start_time = time.ctime()
    pool = []  ⑰
    for i in range(POOL_SIZE):  ⑱
        worker_name = "Worker {:03d}".format(i)
        w = Worker(worker_name, word_queue, result_lock, shared_result)  ⑲
        #
        w.start()  ⑳
        pool.append(w)

    for t in pool:
        t.join()

    end_time = time.ctime()

    print((shared_result[-50:]))
    print(len(shared_result))
    print(start_time)
    print(end_time)
```

① set some constants

② callable class to provide random words

③ will be called when you call an instance of the class

④ worker class — inherits from Process

⑤ initialize worker process

⑥ do some work — will be called when process starts

⑦ get data from the queue

⑧ modify data

⑨ add to shared result

⑩ quit when there is no more data in the queue

⑪ create empty Queue object

⑫ create manager for shared data

⑬ create list-like object to be shared across all processes

⑭ create locks

⑮ create callable RandomWord instance

⑯ fill the queue

⑰ create empty list to hold processes

⑱ populate the process pool

⑲ create worker process

⑳ actually start the process — note: in Windows, should only call X.start() from main(), and may not work inside an IDE

add process to pool

wait for each queue to finish

print last 50 entries in shared result

multi_processing.py

```
['DRAG', 'GASELIER', 'STENOTYPY', 'FRAME', 'SNOWBALLS', 'ASSIZE', 'DUSTCOVERS',
'MICROMERES', 'BUNGEES', 'OVERNIGHTS', 'CRUMBERS', 'BARRELSFUL', 'NATURALNESS',
'GANTLETING', 'TEXTILES', 'DEGRINGOLADE', 'CZARDOMS', 'SYNCHROMESH', 'SUNDOWNERS',
'BOUSTROPHEDONIC', 'BLACKLEGS', 'NONCONTRADICTORY', 'JAGS', 'SUBMUNITIONS', 'EFFULGES',
'AMBIDEXTERITIES', 'CAYMAN', 'LITHOLOGIES', 'SPURGALLING', 'SUBBASE', 'OWSEN',
'SEXAGENARIAN', 'PURCHASER', 'GINGIVAE', 'EMISSION', 'CLIMBERS', 'UNSWEPT',
'INVALIDATORS', 'FROCKED', 'HEARTLAND', 'COMMISSARS', 'LACUNOSE', 'DELICACY',
'NONFLUENCIES', 'PLATTERFULS', 'WORKMATE', 'REPLANS', 'LOCALITES', 'SERVOMOTOR',
'DOURER']
25000
Fri Nov 12 13:45:04 2021
Fri Nov 12 13:45:05 2021
```

Using pools

- Provided by **multiprocessing**
- Both thread and process pools
- Simplifies multiprogramming tasks

For many multiprocessing tasks, you want to process a list (or other iterable) of data and do something with the results. This is easily accomplished with the Pool object provided by the **multiprocessing** module.

This object creates a pool of *n* processes. Call the **.map()** method with a function that will do the work, and an iterable of data. map() will return a list the same size as the list that was passed in, containing the results returned by the function for each item in the original list.

For a thread pool, import **Pool** from **multiprocessing.dummy**. It works exactly the same, but creates threads.

Example

proc_pool.py

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

import random
from multiprocessing import Pool

POOL_SIZE = 30  ①

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in]  ②

random.shuffle(WORDS)  ③


def my_task(word):  ④
    return word.upper()


if __name__ == '__main__':
    ppool = Pool(POOL_SIZE)  ⑤

    WORD_LIST = ppool.map(my_task, WORDS)  ⑥

    print(WORD_LIST[:20])  ⑦

    print("Processed {} words.".format(len(WORD_LIST)))
```

① number of processes

② read word file into a list, stripping off \n

③ randomize word list

④ actual task

⑤ create pool of POOL_SIZE processes

⑥ pass wordlist to pool and get results; map assigns values from input list to processes as needed

⑦ print last 20 words

proc_pool.py

```
['GUIDING', 'SONORANTS', 'CORPORATIONS', 'THWACK', 'INVEIGLERS', 'ADDRESSES',
'AGREEABILITY', 'METTLES', 'SUPERROAD', 'COMBUSTIBLES', 'JABS', 'BROMIDIC', 'GELLING',
'MURDEREES', 'DESTRUCTS', 'TOPI', 'MERDES', 'TACK', 'SPARKLE', 'MANDATORS']
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

```
['OUTSWEARING', 'PEEKED', 'BORANES', 'SOLUBILIZES', 'MAN', 'THWARTER', 'OUTMUSCLING',
'NONSEGREGATION', 'CHEAPLY', 'HYPERIRRITABLE', 'TYRANNOUS', 'PENICILLIA', 'STAB',
'CLACKS', 'OTOSCOPY', 'CHAIN', 'TICKTACKED', 'TONOPLAST', 'TENSIOMETERS', 'RESENSITIZE']
Processed 173466 words.
```

Example

thr_pool_mw.py

```
#!/usr/bin/env python
from multiprocessing.dummy import Pool  ①
from pprint import pprint
import requests

POOL_SIZE = 4

BASE_URL = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/'  ②

API_KEY = 'b619b55d-faa3-442b-a119-dd906adc79c8' ③

search_terms = [  ④
    'wombat',
    'frog', 'muntin', 'automobile', 'green', 'connect',
    'vial', 'battery', 'computer', 'sing', 'park',
    'ladle', 'ram', 'dog', 'scalpel'
]


def fetch_data(term):  ⑤
    try:
        response = requests.get(
            BASE_URL + term,
            params={'key': API_KEY},
        )  ⑥
    except requests.HTTPError as err:
        print(err)
        return []
    else:
        data = response.json()  ⑦
        parts_of_speech = []
        for entry in data: ⑧
            if isinstance(entry, dict):
                meta = entry.get("meta")
                if meta:
                    part_of_speech = entry.get("fl")
                    if part_of_speech:
                        parts_of_speech.append(part_of_speech)
        return sorted(set(parts_of_speech))  ⑨


p = Pool(POOL_SIZE)  ⑩

results = p.map(fetch_data, search_terms)  ⑪
```

```
for search_term, result in zip(search_terms, results):  ⑫
    print("{}:".format(search_term.upper()))
    if result:
        print(result)
    else:
        print("** no results **")
```

① .dummy has Pool for threads

② base url of site to access

③ credentials to access site

④ terms to search for; each thread will search some of these terms

⑤ function invoked by each thread for each item in list passed to map()

⑥ make the request to the site

⑦ convert JSON to Python structure

⑧ loop over entries matching search terms

⑨ return list of parsed entries matching search term

⑩ create pool of POOL_SIZE threads

⑪ launch threads, collect results

⑫ iterate over results, mapping them to search terms

…

Alternatives to multiprogramming

- asyncio
- Twisted

Threading and forking are not the only ways to have your program do more than one thing at a time. Another approach is asynchronous programming. This technique putting events (typically I/O events) in a list, or queue, and starting an event loop that processes the events one at a time. If the granularity of the event loop is small, this can be as efficient as multiprogramming.

Asynchronous programming is only useful for improving I/O throughput, such as networking clients and servers, or scouring a file system. Like threading (in Python), it will not help with raw computation speed.

The **asyncio** module in the standard library provides the means to write asynchronous clients and servers.

The **Twisted** framework is a large and well-supported third-party module that provides support for many kinds of asynchronous communication. It has prebuilt objects for servers, clients, and protocols, as well as tools for authentication, translation, and many others. Find Twisted at twistedmaxtrix.com/trac.

Chapter 8 Exercises

For each exercise, ask the questions: Should this be multi-threaded or multi-processed? Distributed or local?

Exercise 8-1 (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 8-2 (folder_scanner.py)

Write a program that takes in a directory name on the command line, then traverses all the files in that directory tree and prints out a count of:

- how many total files
- how many total lines (count \n)
- how many bytes (len() of file contents)

HINT: Use either a thread or a process pool in combination with **os.walk()**.

FOR ADVANCED STUDENTS

Exercise 8-3 (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 8-4 (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 9: Network Programming

Objectives

- Download web pages or file from the Internet
- Consume web services
- Send e-mail using a mail server
- Learn why requests is the best HTTP client

Making HTTP requests

- Use the **requests** module
- Pythonic front end to urllib, urllib2, httplib, etc
- Makes HTTP transactions simple

The standard library provides the **urllib** package. It and its friends are powerful libraries, but their interfaces are complex for non-trivial tasks. There is a lot of code to write 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 the **Anaconda** distribution, or is readily available from **PyPI**.

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.

To use **requests**, import the module and then call **requests.*VERB***, where *VERB* is "get", "post", "put", "patch", "delete", or "head". The first argument to any of these methods is the URL, followed by any of the named parameters for fine-tuning the request.

These methods return an HTTPResponse object, which contains the headers and data returned from the HTTP server. If the URL refers to a web page, then the **text** attribute contains the text of the page as a Python string.

In all cases, the **content** attribute contains the raw content from the server as a **bytes** string. If the returned data is a JSON string, the **json()** method converts the JSON data into a Python nested list or dictionary.

The **status_code** attribute contains the HTTP status code, normally 200 for a successful request.

For GET requests, URL parameters can be specified as a dictionary, using the **params** parameter.

For POST, PUT, or PATCH requests, the data to be uploaed can be specified as a dictionary using the **data** parameter.

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

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("{:20.20s} {}".format(header, value))
print()

print(response.text[:200])   ③
print('...')
print(response.text[-200:])   ④
```

① requests.get() returns HTTP response object

② response.headers is a dictionary of the headers

③ The text is returned as a bytes object, so it needs to be decoded to a string; print the first 200 bytes

④ print the last 200 bytes

Example

read_pdf_requests.py

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

import sys
import os

import requests

url =
'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure%202012%20Screenres%203-
8-13.pdf'  ①
saved_pdf_file = 'nasa_iss.pdf'  ②

response = requests.get(url)  ③
if response.status_code == requests.codes.OK:  ④
    if response.headers.get('content-type') == 'application/pdf':
        with open(saved_pdf_file, 'wb') as pdf_in:  ⑤
            pdf_in.write(response.content)  ⑥

        if sys.platform == 'win32':  ⑦
            cmd = saved_pdf_file
        elif sys.platform == 'darwin':
            cmd = 'open ' + saved_pdf_file
        else:
            cmd = 'acroread ' + saved_pdf_file

        os.system(cmd)  ⑧
```

① target URL

② name of PDF file for saving

③ open the URL

④ check status code

⑤ open local file

⑥ write data to a local file in binary mode; response.content is data from URL

⑦ select platform and choose the app to open the PDF file

⑧ launch the app

Example

web_content_consumer_requests.py

```
import sys
import requests

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

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


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

    response = requests.get(
        BASE_URL + args[0],
        params={'key': API_KEY},
        # ssl, proxy, cookies, headers, etc.
    )  ③

    if response.status_code == requests.codes.OK:  # 200?
        data = response.json()  ④
        for entry in data: ⑤
            if isinstance(entry, dict):
                meta = entry.get("meta")
                if meta:
                    part_of_speech = '({})'.format(entry.get('fl'))
                    word_id = meta.get("id")
                    print("{} {}".format(word_id.upper(), part_of_speech))
                if "shortdef" in entry:
                    print('\n'.join(entry['shortdef']))
                print()
            else:
                print(entry)

    else:
        print("Sorry, HTTP response", response.status_code)

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

① base URL of resource site

② credentials

③ send HTTP request and get HTTP response

④ convert JSON content to Python data structure

⑤ check for results

web_content_consumer_requests.py wombat

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

Table 12. 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

NOTE These can be used with any of the HTTP request types, as appropriate.

Table 13. **requests.Response** *attributes*

Attribute	Definition
apparent_encoding	Returns the apparent encoding
close()	Closes the connection to the server
content	Content of the response, in bytes
cookies	A CookieJar object with the cookies sent back from the server
elapsed	A timedelta object with the time elapsed from sending the request to the arrival of the response
encoding	The encoding used to decode r.text
headers	A dictionary of response headers
history	A list of response objects holding the history of request (url)
is_permanent_redirect	True if the response is the permanent redirected url, otherwise False
is_redirect	True if the response was redirected, otherwise False
iter_content()	Iterates over the response
iter_lines()	Iterates over the lines of the response
json()	A JSON object of the result (if the result was written in JSON format, if not it raises an error)
links	The header links
next	A PreparedRequest object for the next request in a redirection
ok	True if status_code is less than 400, otherwise False
raise_for_status()	If an error occur, this method a HTTPError object
reason	A text corresponding to the status code
request	The request object that requested this response
status_code	A number that indicates the status (200 is OK, 404 is Not Found)
text	The content of the response, in unicode
url	The URL of the response

Grabbing a web page the hard way

- import urlopen() from urllib.request
- urlopen() similar to open()
- Read response
- Use info() 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.

In general, if you can install **requests** and use it, that is the preferred approach.

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: 50697
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Via: 1.1 vegur, 1.1 varnish, 1.1 varnish
Accept-Ranges: bytes
Date: Fri, 12 Nov 2021 18:45:07 GMT
Age: 174
X-Served-By: cache-bwi5144-BWI, cache-pdk17882-PDK
X-Cache: HIT, HIT
X-Cache-Hits: 3, 1
X-Timer: S1636742707.472437,VS0,VE7
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 the hard way

- 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
```

sending e-mail

- import smtplib module
- Create an SMTP object specifying server
- Call sendmail() method from SMTP object

You can send e-mail messages from Python using the smtplib module. All you really need is one smtplib object, and one method – sendmail().

Create the smtplib object, then call the sendmail() method with the sender, recipient(s), and the message body (including any headers).

The recipients list should be a list or tuple, or could be a plain string containing a single recipient.

Example

email_simple.py

```
#!/usr/bin/env python
from getpass import getpass  ①
import smtplib  ②
from email.message import EmailMessage  ③
from datetime import datetime

TIMESTAMP = datetime.now().ctime()  ④

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

MESSAGE_BODY = """
Hello at {}.

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

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

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

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

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

① module for hiding password

② module for sending email

③ module for creating message

④ get a time string for the current date/time

⑤ get password (not echoed to screen)

⑥ connect to SMTP server

⑦ log into SMTP server

⑧ create empty email message

⑨ add the message body

⑩ add the message subject

⑪ add the sender address

⑫ add a list of recipients

⑬ send the message

⑭ disconnect from SMTP server

Email attachments

- Create MIME multipart message
- Create MIME objects
- Attach MIME objects
- Serialize message and send

To send attachments, you need to create a MIME multipart message, then create MIME objects for each of the attachments, and attach them to the main message. This is done with various classes provided by the **email.mime** module.

These modules include **multipart** for the main message, **text** for text attachments, **image** for image attachments, **audio** for audio files, and **application** for miscellaneous binary data.

One the attachments are created and attached, the message must be serialized with the **as_string()** method. The actual transport uses **smptlib**, just like simple email messages described earlier.

Example

email_attach.py

```
#!/usr/bin/env python
import smtplib
from datetime import datetime
from imghdr import what  ①
from email.message import EmailMessage ②
from getpass import getpass ③


SMTP_SERVER = "smtp2go.com"  ④
SMTP_PORT = 2525

SMTP_USER = 'pythonclass'

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


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


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


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

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


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

    return smtpserver


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


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

① module to determine image type

② module for creating email message

③ module for reading password privately

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

⑤ create instance of EmailMessage to hold message

⑥ set content (message text) and various headers

⑦ read data for text attachment

⑧ add text attachment to message

⑨ read data for binary attachment

⑩ get type of binary data

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

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

⑬ create SMTP server connection

⑭ log into SMTP connection

⑮ send message

Remote Access

- Use paramiko (not part of standard library)
- Create ssh client
- Create transport object to use sftp and other tools

For remote access to other computers, you generally use the SSH protocol. Python has several ways to use SSH.

The current 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 the Anaconda distribution.

NOTE

Paramiko is used by Ansible and other sys admin tools.

Find out more about paramiko at http://www.lag.net/paramiko/
Find out more about Ansible at http://www.ansible.com/
Find out more about **ssh2-python**, an alternative to Paramiko, at https://parallel-ssh.org/post/ssh2-python/

Auto-adding hosts

- Interactive SSH prompts to add new host
- Programmatic interface can't do that
- Use **set_missing_host_key_policy()**
- Adds to list of known hosts.

The first time you connect to a new host with SSH, you get the following message:

```
The authenticity of host HOSTNAME can't be established.
ECDSA key fingerprint is HOSTNAME
Are you sure you want to continue connecting...
```

To avoid the message when using Paramiko, call **set_missing_host_key_policy()** from the Paramiko SSH client object:

```
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

Remote commands

- Use SSHClient
- Access standard I/O channels

To run commands on a remote computer, use SSHClient. Once you connect to the remote host, you can execute commands and access the standard I/O of the remote program.

The **exec_command()** method executes a command on the remote host, and returns a 3-tuple with the remote command's stdin, stdout, and stderr as file-like objects.

You can read from stdout and stderr, and write to stdin.

NOTE With some versions of **paramiko**, the *stdin* object returned by **exec_command()** must be explicitly set to **None**, or deleted with **DEL** after use. Otherwise, an error will be raised.

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)

    del stdin  # workaround for paramiko bug!
```

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 384
drwx------+  3 python  staff      96 Feb 11  2021 Desktop
drwx------+  3 python  staff      96 Feb 11  2021 Documents
drwx------+  3 python  staff      96 Feb 11  2021 Downloads
drwx------@ 50 python  staff    1600 Sep 14 07:12 Library
drwx------+  3 python  staff      96 Feb 11  2021 Movies
drwx------+  3 python  staff      96 Feb 11  2021 Music
drwx------+  3 python  staff      96 Feb 11  2021 Pictures
drwxr-xr-x+  4 python  staff     128 Feb 11  2021 Public
-rw-r--r--   1 python  staff  148544 May 27 16:16 alice.txt
drwxr-xr-x   2 python  staff      64 May 27 16:00 foo
drwxr-xr-x   2 python  staff      64 May 27 16:16 testing
drwxr-xr-x   3 python  staff      96 Feb 18  2021 text_files

------------------------------------------------------------
STDOUT:
-rw-r--r--  1 root  wheel  6946 Jun  5  2020 /etc/passwd

STDERR:
ls: /etc/horcrux: No such file or directory

------------------------------------------------------------
```

Copying files with SFTP

- 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("testing")

    # sftp.put(local-file)
    # sftp.put(local-file, remote-file)
    sftp.put('../DATA/alice.txt', 'text_files/betsy.txt')  ⑥
    sftp.put('../DATA/alice.txt', 'alice.txt')
    sftp.put('../DATA/alice.txt', 'text_files')
    sftp.get(remote_file, 'eileen.txt')  ⑦
```

① create paramiko Transport instance

② connect to remote host

③ create SFTP client using Transport instance

④ get list of items on default (login) folder (listdir_iter() returns a generator)

⑤ create path for remote file

⑥ create a folder on the remote host

⑦ copy a file to the remote host

⑧ copy a file from the remote host

⑨ use SSHClient to confirm operations (not needed, just for illustration) *paramiko_copy_files.py*

```
drwx------   1 503      20             96 11 Feb 2021  Music
-r--------   1 503      20              7 14 Sep 07:09 .CFUserTextEncoding
drwx------   1 503      20             96 11 Feb 2021  Pictures
drwxr-xr-x   1 503      20             96 18 Feb 2021  text_files
-rw-r--r--   1 503      20         148544 27 May 16:16 alice.txt
-rw-------   1 503      20            135 14 Sep 07:13 .zsh_history
drwx------   1 503      20             96 11 Feb 2021  Desktop
drwx------   1 503      20           1600 14 Sep 07:12 Library
drwxr-xr-x   1 503      20             64 27 May 16:16 testing
drwxr-xr-x   1 503      20            128 11 Feb 2021  Public
drwxr-xr-x   1 503      20             64 27 May 16:00 foo
drwx------   1 503      20             96 11 Feb 2021  Movies
drwx------   1 503      20             96 11 Feb 2021  Documents
drwx------   1 503      20             96 11 Feb 2021  Downloads
------------------------------------------------------------
```

Interactive remote access

- Write to stdin
- Read response from stdout

To interact with a remote program, write to the stdin object returned by ***ssh_object*.exec_command()**.

```
stdin.write("command input....\n")
```

Be sure to add a newline (|*n*) for each line of input you send.

To get the response, read the next line(s) of code with *stdout*.readline()

Example

paramiko_interactive.py

```
#!/usr/bin/env python
import paramiko
# bc is an interactive calculator that comes with Unix-like systems (Linux, Mac, etc.)

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('bc')  ④

    stdin.write("17 + 25\n")  ⑤
    result = stdout.readline()  ⑥
    print("Result is:", result)

    stdin.write("scale = 3\n")  ⑦
    stdin.write("738.3/191.9\n")
    result = stdout.readline()
    print("Result is:", result)

    stdin.write("quit\n")  ⑧
    stdin = None   ⑨
```

① create paramiko SSH client

② auto-add remote host

③ log into to remote host

④ execute command; returns file-like objects representing stdio

⑤ write to command’s stdin

⑥ read output of command

⑦ set scale (# decimal points) to 3 (bc-specific command)

paramiko_interactive.py

```
Result is: 42

Result is: 3.847
```

Chapter 9 Exercises

Exercise 9-1 (fetch_xkcd_requests.py, fetch_xkcd_urllib.py)

Write a script to fetch the following image from the Internet and display it. http://imgs.xkcd.com/comics/python.png

Exercise 9-2 (wiki_links_requests.py, wiki_links_urllib.py)

Write a script to count how many links are on the home page of Wikipedia. To do this, read the page into memory, then look for occurrences of the string "href". (For *real* screen-scraping, you can use the Beautiful Soup module.)

You can use the string method **find()**, which can be called like S.find(*text*, start, stop), which finds on a slice of the string, moving forward each time the string is found.

Exercise 9-3 (send_chimp.py)

If the class conditions allow it (i.e., if you have access to the Internet, and an SMTP account), send an email to yourself with the image **chimp.bmp** (from the DATA folder) attached.

Chapter 10: Effective Scripts

Objectives

- Launch external programs
- Check permissions on files
- Get system configuration information
- Store data offline
- Create Unix-style filters
- Parse command line options
- Configure application logging

Using glob

- Expands wildcards
- Windows and non-windows
- Useful with **subprocess** module

When executing external programs, sometimes you want to specify a list of files using a wildcard. The **glob** function in the **glob** module will do this. Pass one string containing a wildcard (such as `*.txt`) to glob(), and it returns a sorted list of the matching files. If no files match, it returns an empty list.

Example

glob_example.py

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

from glob import glob

files = glob('../DATA/*.txt') ①
print(files, '\n')

no_files = glob('../JUNK/*.avi')
print(no_files, '\n')
```

① expand file name wildcard into sorted list of matching names *glob_example.py*

```
['../DATA/presidents_plus_biden.txt', '../DATA/columns_of_numbers.txt',
'../DATA/poe_sonnet.txt', '../DATA/computer_people.txt', '../DATA/owl.txt',
'../DATA/eggs.txt', '../DATA/world_airport_codes.txt', '../DATA/stateinfo.txt',
'../DATA/fruit2.txt', '../DATA/us_airport_codes.txt', '../DATA/parrot.txt',
'../DATA/http_status_codes.txt', '../DATA/fruit1.txt', '../DATA/alice.txt',
'../DATA/littlewomen.txt', '../DATA/spam.txt', '../DATA/world_median_ages.txt',
'../DATA/phone_numbers.txt', '../DATA/sales_by_month.txt', '../DATA/engineers.txt',
'../DATA/underrated.txt', '../DATA/tolkien.txt', '../DATA/tyger.txt',
'../DATA/example_data.txt', '../DATA/states.txt', '../DATA/kjv.txt', '../DATA/fruit.txt',
'../DATA/areacodes.txt', '../DATA/float_values.txt', '../DATA/unabom.txt',
'../DATA/chaos.txt', '../DATA/noisewords.txt', '../DATA/presidents.txt',
'../DATA/bible.txt', '../DATA/breakfast.txt', '../DATA/Pride_and_Prejudice.txt',
'../DATA/nsfw_words.txt', '../DATA/mary.txt',
'../DATA/2017FullMembersMontanaLegislators.txt', '../DATA/badger.txt',
'../DATA/README.txt', '../DATA/words.txt', '../DATA/primeministers.txt',
'../DATA/nc_counties_avg_wage.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 14. CalledProcessError attributes

Attribute	Description
args	The arguments used to launch the process. This may be a list or a string.
returncode	Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully. A negative value -N indicates that the child was terminated by signal N (POSIX only).
stdout	Captured stdout from the child process. A bytes sequence, or a string if run() was called with an encoding or errors. None if stdout was not captured. If you ran the process with stderr=subprocess.STDOUT, stdout and stderr will be combined in this attribute, and stderr will be None. stderr

subprocess convenience functions

- run(), check_call() , check_output()
- Simpler to use than Popen

subprocess defines convenience functions, **call()**, **check_call()**, and **check_output()**.

```
proc subprocess.run(cmd, ...)
```

Run command with arguments. Wait for command to complete, then return a **CompletedProcess** instance.

```
subprocess.check_call(cmd, ...)
```

Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute.

```
check_output(cmd, ...)
```

Run command with arguments and return its output as a byte string. If the exit code was non-zero it raises a CalledProcessError. The CalledProcessError object will have the return code in the returncode attribute and output in the output attribute.

NOTE run() is only implemented in Python 3.5 and later.

Example

subprocess_conv.py

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

import sys
from subprocess import check_call, check_output, CalledProcessError
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

try:
    check_call(full_cmd)
except CalledProcessError as err:
    print("Command failed with return code", err.returncode)

print('-' * 60)

try:
    output = check_output(full_cmd)
    print("Output:", output.decode(), sep='\n')
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

subprocess_conv.py

```
-rw-r--r--  1 jstrick  staff  3178541 Nov  2  2020 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff      297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff     2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff    73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff      834 Feb 14  2016 ../DATA/tyger.txt
------------------------------------------------------------
Output:
-rw-r--r--  1 jstrick  staff  3178541 Nov  2  2020 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff      297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff     2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff    73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff      834 Feb 14  2016 ../DATA/tyger.txt

--------------------------------------------------
```

NOTE showing Unix/Linux/Mac output – Windows will be similar

TIP (Windows only) The following commands are *internal* to CMD.EXE, and must be preceded by `cmd /c` or they will not work: ASSOC, BREAK, CALL ,CD/CHDIR, CLS, COLOR, COPY, DATE, DEL, DIR, DPATH, ECHO, ENDLOCAL, ERASE, EXIT, FOR, FTYPE, GOTO, IF, KEYS, MD/MKDIR, MKLINK (vista and above), MOVE, PATH, PAUSE, POPD, PROMPT, PUSHD, REM, REN/RENAME, RD/RMDIR, SET, SETLOCAL, SHIFT, START, TIME, TITLE, TYPE, VER, VERIFY, VOL

Capturing stdout and stderr

- Add stdout, stderr args
- Assign subprocess.PIPE

To capture stdout and stderr with the subprocess module, import **PIPE** from subprocess and assign it to the stdout and stderr parameters to run(), check_call(), or check_output(), as needed.

For check_output(), the return value is the standard output; for run(), you can access the **stdout** and **stderr** attributes of the CompletedProcess instance returned by run().

> **NOTE** output is returned as a bytes object; call decode() to turn it into a normal Python string.

Example

subprocess_capture.py

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

import sys
from subprocess import check_output, Popen, CalledProcessError, STDOUT, PIPE ①
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

②
try:
    output = check_output(full_cmd) ③
    print("Output:", output.decode(), sep='\n') ④
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

```
⑤
try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=STDOUT) ⑥
    stdout, stderr = proc.communicate() ⑦
    print("Output:", stdout.decode()) ⑧
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)

try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=PIPE) ⑨
    stdout, stderr = proc.communicate() ⑩
    print("Output:", stdout.decode()) ⑪
    print("Error:", stderr.decode()) ⑪
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

① need to import PIPE and STDOUT

② capture only stdout

③ check_output() returns stdout

④ stdout is returned as bytes (decode to str)

⑤ capture stdout and stderr together

⑥ assign PIPE to stdout, so it is captured; assign STDOUT to stderr, so both are captured together

⑦ call communicate to get the input streams of the process; it returns two bytes objects representing stdout and stderr

⑧ decode the stdout object to a string

⑨ assign PIPE to stdout and PIPE to stderr, so both are captured individually

⑩ now stdout and stderr each have data

⑪ decode from bytes and output

subprocess_capture.py

```
Output:
-rw-r--r--  1 jstrick  staff  3178541 Nov  2  2020 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff      297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff     2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff   106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff   284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff    73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff      834 Feb 14  2016 ../DATA/tyger.txt

--------------------------------------------------
Output: -rw-r--r--  1 jstrick  staff     3178541 Nov  2  2020 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff         297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff        2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff      106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff      284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff       73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff         834 Feb 14  2016 ../DATA/tyger.txt
-rw-r--r--  1 jstrick  students       22 Nov 11 11:26 spam.txt

--------------------------------------------------
Output: -rw-r--r--  1 jstrick  staff     3178541 Nov  2  2020 ../DATA/tate_data.zip
-rwxr-xr-x  1 jstrick  staff         297 Nov 17  2016 ../DATA/testscores.dat
-rwxr-xr-x  1 jstrick  staff        2198 Feb 14  2016 ../DATA/textfiles.zip
-rw-r--r--  1 jstrick  staff      106960 Jul 26  2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick  staff      284160 Jul 26  2017 ../DATA/titanic3.xls
-rwxr-xr-x  1 jstrick  staff       73808 Feb 14  2016 ../DATA/tolkien.txt
-rwxr-xr-x  1 jstrick  staff         834 Feb 14  2016 ../DATA/tyger.txt
-rw-r--r--  1 jstrick  students       22 Nov 11 11:26 spam.txt

Error:
--------------------------------------------------
```

Permissions

- Simplest is os.access()
- Get mode from os.lstat()
- Use binary AND with permission constants

Each entry in a Unix filesystem has a inode. The inode contains low-level information for the file, directory, or other filesystem entity. Permissions are stored in the *mode*, which is a 16-bit unsigned integer. The first 4 bits indicate what kind of entry it is, and the last 12 bits are the permissions.

To see if a file or directory is readable, writable, or executable use os.access(). To test for specific permissions, use the os.lstat() method to return a tuple of inode data, and use the S_IMODE () method to get the mode information as a number. Then use predefined constants such as stat.S_IRUSR, stat.S_IWGRP, etc. to test for permissions.

Example

file_access.py

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

import sys
import os

if len(sys.argv) < 2:
    start_dir = "."
else:
    start_dir = sys.argv[1]

for base_name in os.listdir(start_dir):  ①
    file_name = os.path.join(start_dir, base_name)
    if os.access(file_name, os.W_OK):  ②
        print(file_name, "is writable")
```

① os.listdir() lists the contents of a directory

② os.access() returns True if file has specified permissions (can be os.W_OK, os.R_OK, or os.X_OK, combined with | (OR))

file_access.py ../DATA

```
../DATA/hyper.xlsx is writable
../DATA/presidents.csv is writable
../DATA/Bicycle_Counts.csv is writable
../DATA/wetprf is writable
../DATA/uri-schemes-1.csv is writable
../DATA/presidents.html is writable
../DATA/presidents.xlsx is writable
../DATA/pokemon_data.csv is writable
../DATA/presidents_plus_biden.txt is writable
../DATA/baby_names 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 15. 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
  - 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 16. 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 17. Logging Levels

Level	Value
CRITICAL FATAL	50
ERROR	40
WARN WARNING	30
INFO	20
DEBUG	10
UNSET	0

Example

logging_simple.py

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

import logging

logging.basicConfig(
    filename='../TEMP/simple.log',
    level=logging.WARNING,
) ①

logging.warning('This is a warning') ②
logging.debug('This message is for debugging') ③
logging.error('This is an ERROR') ④
logging.critical('This is ***CRITICAL***') ⑤
logging.info('The capital of North Dakota is Bismark') ⑥
```

① setup logging; minimal level is WARN

② message will be output

③ message will NOT be output

④ message will be output

⑤ message will be output

⑥ message will not be output

simple.log

```
WARNING:root:This is a warning
ERROR:root:This is an ERROR
CRITICAL:root:This is ***CRITICAL***
```

Formatting log entries

- Add format=format to basicConfig() parameters
- Format is a string containing directives and (optionally) other text
- Use directives in the form of %(item)type
- Other text is left as-is

To format log entries, provide a format parameter to the basicConfig() method. This format will be a string contain special directives (i.e. Placeholders) and, optionally, other text. The directives are replaced with logging information; other data is left as-is.

Directives are in the form %(item)type, where item is the data field, and type is the data type.

Example

logging_formatted.py

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

import logging

logging.basicConfig(
    format='%(name)s %(asctime)s %(levelname)s %(message)s', ①
    filename='../TEMP/formatted.log',
    level=logging.INFO,
)

logging.info("this is information")
logging.warning("this is a warning")
logging.info("this is information")
logging.critical("this is critical")
```

① set the format for log entries

formatted.log

```
root 2021-11-12 13:45:11,372 INFO this is information
root 2021-11-12 13:45:11,372 WARNING this is a warning
root 2021-11-12 13:45:11,372 INFO this is information
root 2021-11-12 13:45:11,373 CRITICAL this is critical
```

Table 18. Log entry formatting directives

Directive	**Description**
%(name)s	Name of the logger (logging channel)
%(levelno)s	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL)
%(levelname)s	Text logging level for the message ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
%(pathname)s	Full pathname of the source file where the logging call was issued (if available)
%(filename)s	Filename portion of pathname
%(module)s	Module (name portion of filename)
%(lineno)d	Source line number where the logging call was issued (if available)
%(funcName)s	Function name
%(created)f	Time when the LogRecord was created (time.time() return value)
%(asctime)s	Textual time when the LogRecord was created
%(msecs)d	Millisecond portion of the creation time
%(relativeCreated)d	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded (typically at application startup time)
%(thread)d	Thread ID (if available)
%(threadName)s	Thread name (if available)
%(process)d	Process ID (if available)
%(message)s	The result of record.getMessage(), computed just as the record is emitted

Logging exception information

- Use logging.exception()
- Adds exception info to message
- Only in **except** blocks

The logging.exception() function will add exception information to the log message. It should only be called in an **except** block.

Example

logging_exception.py

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

import logging

logging.basicConfig( ①
    filename='../TEMP/exception.log',
    level=logging.WARNING,  ②
)

for i in range(3):
    try:
        result = i/0
    except ZeroDivisionError:
        logging.exception('Logging with exception info') ③
```

① configure logging

② minimum level

③ add exception info to the log

exception.log

```
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "logging_exception.py", line 12, in <module>
    result = i/0
ZeroDivisionError: division by zero
```

Logging to other destinations

- Use specialized handlers to write to other destinations
- Multiple handlers can be added to one logger
    - NTEventLogHandler for Windows event log
    - SysLogHandler for syslog
    - SMTPHandler for logging via email

The logging module provides some preconfigured log handlers to send log messages to destinations other than a file.

Each handler has custom configuration appropriate to the destination. Multiple handlers can be added to the same logger, so a log message will go to a file and to email, for instance, and each handler can have its own minimum level. Thus, all messages could go to the message file, but only CRITICAL messages would go to email.

Be sure to read the documentation for the particular log handler you want to use

NOTE On Windows, you must run the example script (logging.altdest.py) as administrator. You can find `Command Prompt (admin)` on the main Windows 8/10 menu. You can also right-click on `Command Prompt` from the Windows 7 menu and choose "Run as administrator".

Example

logging_altdest.py

```
#!/usr/bin/env python
import sys
import logging
import logging.handlers

logger = logging.getLogger('ThisApplication')  ①
logger.setLevel(logging.DEBUG)  ②

if sys.platform == 'win32':
    eventlog_handler = logging.handlers.NTEventLogHandler("Python Log Test")  ③
    logger.addHandler(eventlog_handler)  ④
else:
    syslog_handler = logging.handlers.SysLogHandler()  ⑤
    logger.addHandler(syslog_handler)  ⑥

# note -- use your own SMTP server...
email_handler = logging.handlers.SMTPHandler(
    ('smtpcorp.com', 8025),
    'LOGGER@pythonclass.com',
    ['jstrick@mindspring.com'],
    'ThisApplication Log Entry',
    ('jstrickpython', 'python(monty)'),
)  ⑦

logger.addHandler(email_handler)  ⑧

logger.debug('this is debug')  ⑨
logger.critical('this is critical')  ⑨
logger.warning('this is a warning')  ⑨
```

① get logger for application

② minimum log level

③ create NT event log handler

④ install NT event handler

⑤ create syslog handler

⑥ install syslog handler

⑦ create email handler

⑧ install email handler

⑨ goes to all handlers

Chapter 10 Exercises

Exercise 10-1 (copy_files.py)

Write a script to find all text files (only the files that end in ".txt") in the DATA folder of the student files and copy them to C:\TEMP (Windows) or /tmp (non-windows). On Windows, create the C:\TEMP folder if it does not already exist.

Add logging to the script, and log each filename at level INFO.

TIP use shutil.copy() to copy the files.

Chapter 11: 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

Which XML 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, use **lxml.etree**, which is based on **ElementTree** with some nice extra features, such as pretty-printing. While not part of the core Python library, it is provided by the Anaconda bundle.

If `lxml.etree` is not available, you can use **xml.etree.ElementTree** from the core library.

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 19. 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 20. 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

```
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

root = doc.getroot()
```

Navigating the XML Document

- Use find() or findall()
- Element is iterable of it 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 0x7fb2b00caf00>

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
Gannymede
Callista
Themisto
Himalia
Lysithea
Elara
```

Table 21. 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.
parent_tag[*child_tag*]	Selects all parent elements that has a child element named *child_tag*. In the current version, only a single tag can be used (i.e. only immediate children are supported). Parent tag can be *.

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 object0.

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
        ],
        "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 special handling, return a JSON-friendly version, otherwise just return the original object.

Table 22. 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** decorator (in the **functools** module 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
- syntax like **json** module
- yaml.load(), dump() parse from/to file-like object
- yaml.loads(), dumps() parse from/to string

YAML is a structured data format which is a superset of JSON. However, YAML allows for a more compact and readable format.

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.

NOTE YAML parsers will parse JSON data

Example

yaml_read_solar.py

```
import yaml

PLANET_SECTIONS = "inner outer plutoid".split()

with open('../DATA/solar.yaml') as solar_in:
    solar_data = yaml.load(solar_in, Loader=yaml.FullLoader)

star = solar_data['star']
print("Our star is {}\n".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
    Gannymede
    Callista
    Themisto
    Himalia
    Lysithea
    Elara
Saturn
    Rhea
    Hyperion
    Titan
    Iapetus
    Mimas
```

...

Example

yaml_create_file.py

```
import sys
from datetime import date
import yaml

potus = {
    'presidents': [
         {
            'lastname': 'Washington',
            'firstname': '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,
        },
        {
            'lastname': 'Adams',
            'firstname': '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

```
presidents:
- assassinated: false
  birthplace: Westmoreland County
  birthstate: Virginia
  dob: 1732-02-22
  dod: 1799-12-14
  firstname: George
  lastname: Washington
  party: null
  term:
  - 1789-04-30
  - 1797-03-04
- assassinated: false
  birthplace: Braintree, Norfolk
  birthstate: Massachusetts
  dob: 1735-10-30
  dod: 1826-07-04
  firstname: John
  lastname: 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 23. 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

chicago_data = [
    ['Name', 'Position Title', 'Department', 'Employee Annual Salary'],
    ['BONADUCE,  MICHAEL J', 'POLICE OFFICER', 'POLICE', '$80724.00'],
    ['MELLON,  MATTHEW J "Matt"', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FIERI,  JOHN J', 'FIREFIGHTER-EMT', 'FIRE', '$75342.00'],
    ['GALAHAD,  MERLE S', 'CLERK III', 'BUSINESS AFFAIRS', '$45828.00'],
    ['ORCATTI,  JENNIFER L', 'FIRE COMMUNICATIONS OPERATOR I', 'OEMC', '$63121.68'],
    ['ASHE,  JOHN W', 'FOREMAN OF MACHINISTS', 'AVIATION', '$96553.60'],
    ['SADINSKY BLAKE,  MICHAEL G', 'POLICE OFFICER', 'POLICE', '$78012.00'],
    ['GRANT,  CRAIG A', 'SANITATION LABORER', 'STREETS & SAN', '$69576.00'],
    ['MILLER,  JONATHAN D', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FRANK,  ARTHUR R',
    'POLICE OFFICER/EXPLSV DETECT, K9 HNDLR',
    'POLICE',
    '$87918.00'],
    ['POVOTTI,  JAMES S "Jimmy P"', 'TRAFFIC CONTROL AIDE-HOURLY', 'OEMC', '$19167.20'],
    ['TRAWLER,  DANIEL J', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SCUBA,  ANDREW G', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SWINE,  MATTHEW W', 'SERGEANT', 'POLICE', '$99756.00'],
    ['''RYDER,  MYRTA T "Lil'Myrt"''', 'POLICE OFFICER', 'POLICE', '$83706.00'],
    ['KORSHAK,  ROMAN', 'PARAMEDIC', 'FIRE', '$75372.00']
]

with open('../TEMP/chi_data.csv', 'w') as chi_out:
    # if sys.platform == 'win32':
    wtr = csv.writer(chi_out, lineterminator='\n') ①
    # else:
    #     wtr = csv.writer(stuff_in) ①
    for data_row in chicago_data:
        data_row[-1] = data_row[-1].lstrip('$')  # strip leading $ from last field
        wtr.writerow(data_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
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 11 Exercises

Exercise 11-1 (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</words>
   and so forth
</words>
```

Exercise 11-2 (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 11-3 (jpresidents.py)

Rewrite xpresidents.py to parse presidents.json using the json module.

Exercise 11-4 (cpresidents.py)

Rewrite xpresidents.py to parse presidents.csv using the csv module.

Exercise 11-5 (pickle_potus.py)

Write a script which reads the data from presidents.csv into an 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 11-6 (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.

Appendix A: Python Bibliography

Title	Author	Publisher
Data Science		
Building machine learning systems with Python	William Richert, Luis Pedro Coelho	Packt Publishing
High Performance Python	Mischa Gorlelick and Ian Ozsvald	O'Reilly Media
Introduction to Machine Learning with Python	Sarah Guido	O'Reilly & Assoc.
iPython Interactive Computing and Visualization Cookbook	Cyril Rossant	Packt Publishing
Learning iPython for Interactive Computing and Visualization	Cyril Rossant	Packt Publishing
Learning Pandas	Michael Heydt	Packt Publishing
Learning scikit-learn: Machine Learning in Python	Raúl Garreta, Guillermo Moncecchi	Packt Publishing
Mastering Machine Learning with Scikit-learn	Gavin Hackeling	Packt Publishing
Matplotlib for Python Developers	Sandro Tosi	Packt Publishing
Numpy Beginner's Guide	Ivan Idris	Packt Publishing
Numpy Cookbook	Ivan Idris	Packt Publishing
Practical Data Science Cookbook	Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta	Packt Publishing
Python Text Processing with NLTK 2.0 Cookbook	Jacob Perkins	Packt Publishing
Scikit-learn cookbook	Trent Hauck	Packt Publishing
Python Data Visualization Cookbook	Igor Milovanovic	Packt Publishing
Python for Data Analysis	Wes McKinney	O'Reilly & Assoc.
Design Patterns		
Design Patterns: Elements of Reusable Object-Oriented Software	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides	Addison-Wesley Professional

Title	Author	Publisher
Head First Design Patterns	Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra	O’Reilly Media
Learning Python Design Patterns	Gennadiy Zlobin	Packt Publishing
Mastering Python Design Patterns	Sakis Kasampalis	Packt Publishing
General Python development		
Expert Python Programming	Tarek Ziadé	Packt Publishing
Fluent Python	Luciano Ramalho	O’Reilly & Assoc.
Learning Python, 2nd Ed.	Mark Lutz, David Asher	O’Reilly & Assoc.
Mastering Object-oriented Python	Stephen F. Lott	Packt Publishing
Programming Python, 2nd Ed.	Mark Lutz	O’Reilly & Assoc.
Python 3 Object Oriented Programming	Dusty Phillips	Packt Publishing
Python Cookbook, 3nd. Ed.	David Beazley, Brian K. Jones	O’Reilly & Assoc.
Python Essential Reference, 4th. Ed.	David M. Beazley	Addison-Wesley Professional
Python in a Nutshell	Alex Martelli	O’Reilly & Assoc.
Python Programming on Win32	Mark Hammond, Andy Robinson	O’Reilly & Assoc.
The Python Standard Library By Example	Doug Hellmann	Addison-Wesley Professional
Misc		
Python Geospatial Development	Erik Westra	Packt Publishing
Python High Performance Programming	Gabriele Lanaro	Packt Publishing
Networking		
Python Network Programming Cookbook	Dr. M. O. Faruque Sarker	Packt Publishing
Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers	T J O’Connor	Syngress
Web Scraping with Python	Ryan Mitchell	O’Reilly & Assoc.
Testing		

Title	Author	Publisher
Python Testing Cookbook	Greg L. Turnquist	Packt Publishing
Learning Python Testing	Daniel Arbuckle	Packt Publishing
Learning Selenium Testing Tools, 3rd Ed.	Raghavendra Prasad MG	Packt Publishing
Web Development		
Building Web Applications with Flask	Italo Maia	Packt Publishing
Django 1.0 Website Development	Ayman Hourieh	Packt Publishing
Django 1.1 Testing and Development	Karen M. Tracey	Packt Publishing
Django By Example	Antonio Melé	Packt Publishing
Django Design Patterns and Best Practices	Arun Ravindran	Packt Publishing
Django Essentials	Samuel Dauzon	Packt Publishing
Django Project Blueprints	Asad Jibran Ahmed	Packt Publishing
Flask Blueprints	Joel Perras	Packt Publishing
Flask by Example	Gareth Dwyer	Packt Publishing
Flask Framework Cookbook	Shalabh Aggarwal	Packt Publishing
Flask Web Development	Miguel Grinberg	O’Reilly & Assoc.
Full Stack Python (e-book only)	Matt Makai	Gumroad (or free download)
Full Stack Python Guide to Deployments (e-book only)	Matt Makai	Gumroad (or free download)
High Performance Django	Peter Baumgartner, Yann Malet	Lincoln Loop
Instant Flask Web Development	Ron DuPlain	Packt Publishing
Learning Flask Framework	Matt Copperwaite, Charles O Leifer	Packt Publishing
Mastering Flask	Jack Stouffer	Packt Publishing
Two Scoops of Django: Best Practices for Django 1.11	Daniel Roy Greenfeld, Audrey Roy Greenfeld	Two Scoops Press
Web Development with Django Cookbook	Aidas Bendoraitis	Packt Publishing

Index