Esri Advanced Python Workshop

John Strickler

Version 2.0, February 2022

Table of Contents

About this course

Welcome!

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

Instructor name:

Instructor e-mail:

Have Fun!

Classroom etiquette

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

Please turn off cell phone ringers and other noisemakers.

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

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

IMPORTANT	Please do not bring killer rabbits to class. They might maim, dismember, or otherwise disturb your fellow students.

Course Outline

Day 1

Chapter 1 Pythonic Programming
Chapter 2 Using OpenPyXL with Excel Spreadsheets
Chapter 3 Database Access
Chapter 4 Network Programming

Day 2

Chapter 5 Serializing Data
Chapter 6 Multiprogramming
Chapter 7 Unit Testing with PyTestData
Chapter 8 Effective Scripts

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 **py3esriadv**.

What's in the files?

py3esriadv contains data and other files needed for the exercises
py3esriadv/EXAMPLES contains the examples from the course manuals.
py3esriadv/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 **py3esriadv.zip**. Extract all files to your desktop. This will create the folder **py3esriadv**.

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

Copy or download **py3esriadv.tgz** to your home directory. In your home directory, type

```
tar xzvf py3esriadv.tgz
```

This will create the **py3esriadv** 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 py3esriadv/ANSWERS

Appendices

- Appendix A: Where do I go from here?
- Appendix B: Python Bibliography
- Appendix B: String Formatting

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', 'Brown', 'Madison', 'WI'),
    ('Jose', 'Ramirez', 'Ames', 'IA'),
]

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

for person in people:  ③
    display_person(*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 Brown 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

LINE_FORMAT = '{:6.1f} {:8.2f} {:8.2f}'
HEADER_FORMAT = '{:>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((HEADER_FORMAT.format(*HEADINGS)))  ①
        for size in SIZE_RANGE:
            lengths = get_length(size, flag)
            print(LINE_FORMAT.format(size, *lengths))

        print()

def get_length(size, mens=True):
    start_size = MENS_START_SIZE if mens else 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_example.py

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

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

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

print(sorted_fruits)
```

① The lambda function takes one fruit name and returns a tuple containing the length of the name and the name in lower case. This sorts first by length, then by name.

lambda_example.py

```
['KIWI', 'Apple', 'guava', 'LEMON', 'Mango', 'apricot', '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

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

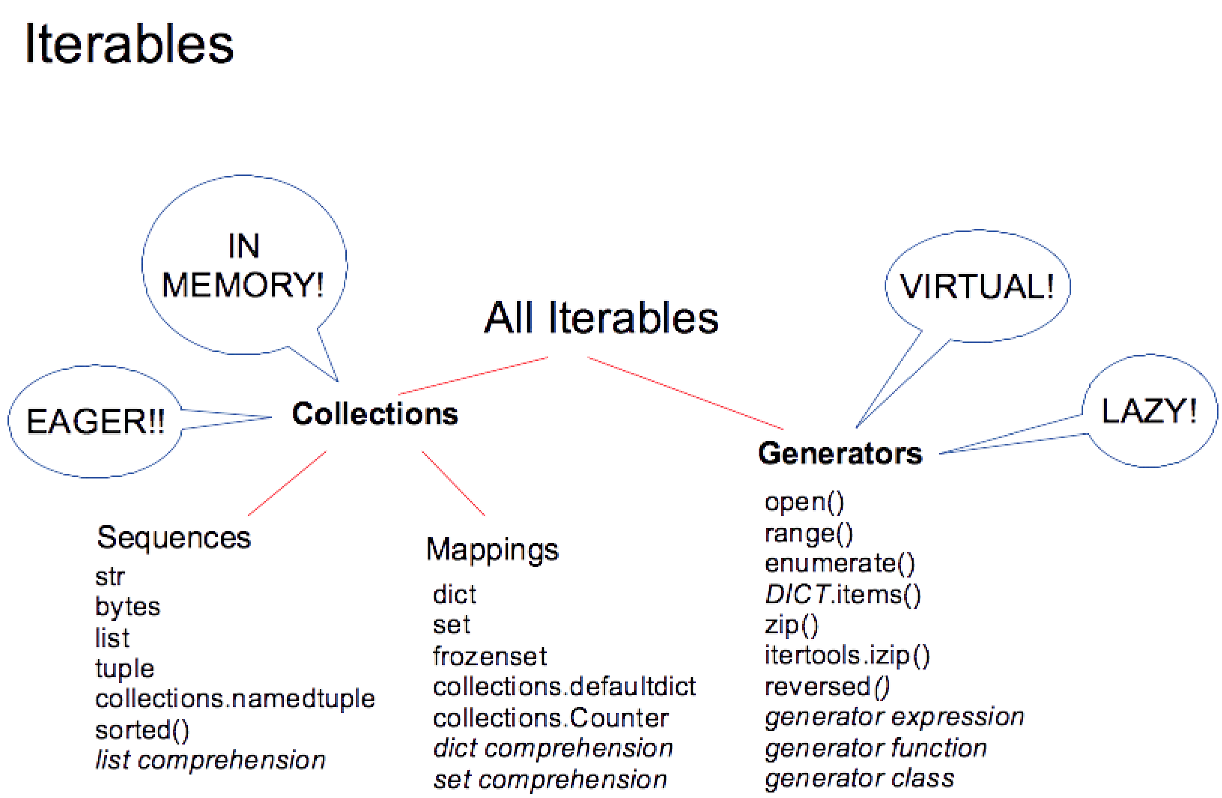

Iterables

- 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: Using openpyxl with Excel spreadsheets

Objectives

- Learn the basics of **openpyxl**
- Open **Excel** spreadsheets and extract data
- Update spreadsheets
- Create new spreadsheets
- Add styles and conditional formatting

The openpyxl module

- Provides full read/write access to Excel spreadsheets
- Creates new workbooks/worksheets
- Does not require Excel

The **openpyxl** module allows you to read, write, and create **Excel** spreadsheets.

openpyxl does not require Excel to be installed.

You can open existing workbooks or create new ones. You can do most anything that you could do manually in a spreadsheet – update or insert data, create formulas, add or change styles, even hide columns.

When you open an existing spreadsheet or create a new one, openpyxl creates an instance of `Workbook`. A Workbook contains one or more `Worksheet` objects.

From a worksheet you can access cells directly, or create a range of cells.

The data in each cell can be manipulated through its `.value` property. Other properties, such as `.font` and `.number_format`, control the display of the data.

View the full documentation at http://openpyxl.readthedocs.org/en/latest/index.html.

TIP To save typing, import **openpyxl** as **px**.

Reading an existing spreadsheet

- Use `load_workbook()` to open file
- Get active worksheet with `WB.active`
- List all worksheets with `get_sheet_by_name()`
- Get named worksheet with `WB['worksheet-name']`

To open and read an existing spreadsheet, use the `load_workbook()` function. This returns a WorkBook object.

There are several ways to get a worksheet from a workbook. To list all the sheets by name, use `WB.get_sheet_names()`.

To get a particular worksheet, index the workbook by sheet name, e.g. `WB['sheetname']`.

A workbook is also an iterable of all the worksheets it contains, so to work on all the worksheets one at a time, you can loop over the workbook.

```
for ws in WB:
    print(ws.title)
```

The `.active` property of a workbook is the currently active worksheet. `WB.active` may be used as soon as a workbook is open.

The `.title` property of a worksheet lets you get or set the title (name) of the worksheet.

Example

px_load_worksheet.py

```python
#!/usr/bin/env python
import openpyxl as px

def main():
    wb = px.load_workbook('../DATA/presidents.xlsx')

# three ways to get to a worksheet:

    # 1
    print(wb.sheetnames, '\n')
    ws = wb['US Presidents']
    print(ws, '\n')

    # 2
    for ws in wb:
        print(ws.title, ws.dimensions)
    print()

    # 3
    ws = wb.active
    print(ws, '\n')

    print(ws['B2'].value)


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

px_load_worksheet.py

```
['US Presidents', 'President Names']

<Worksheet "US Presidents">

US Presidents A1:J47
President Names B2:C47

<Worksheet "President Names">

Washington
```


Worksheet info

- Worksheet attributes
  - dimensions
  - min_row
  - max_row
  - min_column
  - max_column
  - *many others…*

Once a worksheet is opened, you can get information about the worksheet directly from the worksheet object.

The dimensions are based on the extent of the cells that actually contain data.

NOTE Worksheets can have a maximum of 1,048,576 rows and 16,384 columns.

Example

px_worksheet_info.py

```
#!/usr/bin/env python
import openpyxl as px


def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    print("Title:", ws.title)
    print("Dimensions:", ws.dimensions)
    print("Minimum column:", ws.min_column)
    print("Minimum row:", ws.min_row)
    print("Maximum column:", ws.max_column)
    print("Maximum row:", ws.max_row)
    print("Parent:", ws.parent)
    print("Active cell:", ws.active_cell)

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

px_worksheet_info.py

```
Title: US Presidents
Dimensions: A1:J47
Minimum column: 1
Minimum row: 1
Maximum column: 10
Maximum row: 47
Parent: <openpyxl.workbook.workbook.Workbook object at 0x7fc5a07fdf90>
Active cell: J48
```


Table 1. Useful Worksheet Attributes

Attribute	Data type	Description
`active_cell`	str	coordinates ("A1"-style) of active cell
`columns`	generator	iterable of all columns, as tuples of Cell objects
`dimensions`	str	coordinate range ("A1:B2") of all cells containing data
`encoding`	str	text encoding of worksheet
`max_column`	int	maximum column index (1-based)
`max_row`	int	maximum row index (1-based)
`mime_type`	str	MIME type of document
`min_column`	int	minimum column index (1-based)
`min_row`	int	minimum row index (1-based)
`parent`	Workbook	Workbook object that this worksheet belongs to
`rows`	generator	iterable of all rows, as tuples of Cell objects
`selected_cell`	str	coordinates of currently selected cell
`tables`	dict	dictionary of tables
`title`	str	title of this worksheet
`values`	generator	iterable of all values in the worksheet (actual values, not Cell objects)

NOTE min and max row/column refer to extent of cells containing data

Accessing cells

- Each cell is instance of Cell
- Attributes
  - value
  - number_format
  - font
  - *and others*
- Get cell with
  - `ws["COORDINATES"]`
  - `ws.cell(row, column)`

A worksheet consists of *cells*. There are two ways to access an individual cell:

- lookup the cell using the cell coordinates, e.g. `ws["B3"]`.
- specify the row and column as integers (1-based) using the `.cell` method of the worksheet, e.g. `ws.cell(4, 5)`. You can used named arguments for the row and column: `ws.cell(row=4, column=5)`.

In both cases, to get the actual value, use the `.value` attribute of the cell.

NOTE Cell coordinates are case-insensitive.

Example

px_access_cells.py

```
#!/usr/bin/env python
import openpyxl as px

def main():
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    # access cell by cell name
    print(ws['A1'].value)
    print(ws['C2'].value, ws['B2'].value)
    print()

    # same, but lower-case
    print(ws['a1'].value)
    print(ws['c2'].value, ws['b2'].value)
    print()

    # access cell by row/column (1-based)
    print(
        ws.cell(row=27, column=3).value,  # "C27"
        ws.cell(row=27, column=2).value,  # "B27"
    )
    print()

    # same, without argument names
    print(
        ws.cell(27, 3).value,  # "C27"
        ws.cell(27, 2).value,  # "B27"
    )
    print()

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

px_access_cells.py

```
Term
George Washington

Term
George Washington

Theodore Roosevelt

Theodore Roosevelt
```


Table 2. Cell attributes

Attribute	Type	Description
`alignment`	openpyxl.styles.alignment.Alignment	display alignment info
`border`	openpyxl.styles.borders.Border	border info
`col_idx`	int	column index as integer
`column`	int	column index as integer
`column_letter`	str	column index as string
`comment`	any	cell comment
`coordinate`	str	coordinate of cell (e.g. ("B2")
`data_type`	str	data type code (s=str, n=numeric, etc)
`encoding`	str	text encoding (e.g. 'utf-8')
`fill`	openpyxl.styles.fills.PatternFill	fill (background) style
`font`	openpyxl.styles.fonts.Font	font color, family, style
`has_style`	bool	True if cell has style assigned
`hyperlink`	openpyxl.worksheet.hyperlink.Hyperlink	hyperlink for cell
`internal_value`	any	value of cell
`is_date`	bool	True if value is a date
`number_format`	str	Code for number format, e.g. "0.0"
`parent`	openpyxl.worksheet.worksheet.Worksheet	worksheet in which cell is located
`protection`	openpyxl.styles.protection.Protection	protection settings (e.g., hidden, locked)
`quotePrefix`	bool	character used for quoting
`row`	int	row index
`style`	str	name of style
`value`	any	actual value of cell

Getting raw values

- Use `worksheet.values`
- Returns row generator
- Each row is a tuple of column values

To iterate over all the values in the spreadsheet, use `worksheet.values`. It is a generator of the rows in the worksheet. Each element is a tuple of column values.

Only populated cells will be part of the returned data.

Example

px_raw_values.py

```
#!/usr/bin/env python
import openpyxl as px

def main():
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']  ①
    headers = next(ws.values)   ②
    for row in ws.values:  ③
        print(row[:5])   ④


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

① get active sheet

② read first row from generator

③ loop over rows in generator

④ print first 5 elements of row tuple

px_raw_values.py

```
('Term', 'Last Name', 'First Name', 'Birth Date', 'Death Date')
(1, 'Washington', 'George', '1732-02-22', '1799-12-14')
(2, 'Adams', 'John', '1735-10-30', '1826-07-04')
(3, 'Jefferson', 'Thomas', '1743-04-13', '1826-07-04')
(4, 'Madison', 'James', '1751-03-16', '1836-06-28')
(5, 'Monroe', 'James', '1758-04-28', '1831-07-04')
(6, 'Adams', 'John Quincy', '1767-07-11', '1848-02-23')
(7, 'Jackson', 'Andrew', '1767-03-15', '1845-06-08')
(8, 'Van Buren', 'Martin', '1782-12-05', '1862-07-24')
(9, 'Harrison', 'William Henry', '1773-02-09', '1841-04-04')
```

```
...
```

```
(37, 'Nixon', 'Richard Milhous', '1913-01-09', '1994-04-22')
(38, 'Ford', 'Gerald Rudolph', '1913-07-14', '2006-12-26')
(39, 'Carter', "James Earl 'Jimmy'", '1924-10-01', 'NONE')
(40, 'Reagan', 'Ronald Wilson', '1911-02-06', '2004-06-05')
(41, 'Bush', 'George Herbert Walker', '1924-06-12', datetime.datetime(2018, 11, 30, 0,
0))
(42, 'Clinton', "William Jefferson 'Bill'", '1946-08-19', 'NONE')
(43, 'Bush', 'George Walker', '1946-07-06', 'NONE')
(44, 'Obama', 'Barack Hussein', '1961-08-04', 'NONE')
(45, 'Trump', 'Donald J', '1946-06-14', 'NONE')
(46, 'Biden', 'Joseph Robinette', datetime.datetime(1942, 11, 10, 0, 0), 'NONE')
```

Working with ranges

- Range represents a rectangle of cells
- Use slice notation
- Iterate through rows, then columns

To get a range of cells, use slice notation on the worksheet object and standard cell notation, e.g. `WS['A1':'M9']` or `WS['A1:M9']`. Note that the range can consist of one string containing the range, or two strings separated by `:`.

The range is a virtual list of rows, and so can be iterated over. Each element of a row is a Cell object. Use the .value attribute to get or set the cell value.

Example

px_get_ranges.py

```python
#!/usr/bin/env python
import openpyxl as px


def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    print_first_and_last_names(ws)


def print_first_and_last_names(ws):
    """Print first and last names of all presidents"""
    pres_range = ws['B2':'C47']  # cell range
    for row in pres_range:  # row object
        print(row[1].value, row[0].value)


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

px_get_ranges.py

```
George Washington
John Adams
Thomas Jefferson
James Madison
James Monroe
John Quincy Adams
Andrew Jackson
Martin Van Buren
William Henry Harrison
John Tyler
```

...

Modifying a worksheet

- Assign to cells
  - WS.cell(row=ROW, column=COLUMN).value = *value*
  - WS.cell(ROW, COLUMN).value = *value*
  - WS[*coordinate*] = *value*

To modify a worksheet, you can either iterate through rows and columns as described above, or assign to the `.value` attribute of individual cells using either `WS.cell()` or `WS["coordinates"]`.

Use `ws.append(iterable)` to append a new row to the spreadsheet.

Use `workbook.save('name.xlsx')` to save the changes. To save to the original workbook, use its name.

TIP Assigning to *cell* is a shortcut for assigning to *cell*.value(). That is, you can say `ws['B4'] = 10`.

NOTE See the later section on inserting and moving rows and columns.

Example

px_modify_sheet.py

```
#!/usr/bin/env python
from datetime import date
import openpyxl as px


def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    add_age_at_inauguration(ws)

    wb.save('presidents1.xlsx')  # save as ...


def make_date(date_str):
    """Convert date string returned by CELL.value into Python date object"""
    year, month, day = date_str.split('-')
    return date(int(year), int(month), int(day))


def add_age_at_inauguration(ws):
    """Add a new column with age of inauguration"""
    new_col = ws.max_column + 1
    print(new_col)
    ws.cell(row=1, column=new_col).value = 'Age at Inauguration'
    for row in range(2, 47):
        birth_date = make_date(ws.cell(row=row, column=4).value)  # treat date as string
        inaugural_date = make_date(ws.cell(row=row, column=8).value)
        raw_age_took_office = inaugural_date - birth_date
        age_took_office = raw_age_took_office.days / 365.25
        ws.cell(row=row, column=new_col).value = age_took_office


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


Working with formulas

- Assign to cell value as a string
- Be sure to start with '='

To add or update a formula, assign the formula as a string to the cell value.

NOTE Remember that **openpyxl** can not *recalculate* a worksheet.

Example

px_formulas.py

```
#!/usr/bin/env python
import openpyxl as px

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    add_age_at_inauguration(ws)

    wb.save('presidents_formula.xlsx')

def add_age_at_inauguration(ws):
    """Add a new column with age of inauguration"""
    new_col = ws.max_column + 1
    print(new_col)
    ws.cell(row=1, column=new_col).value = 'Age at Inauguration'
    for row in range(2, 47):
        new_cell = ws.cell(row=row, column=new_col)
        new_cell.value = '=(H{0}-D{0})/365.25'.format(row)
        new_cell.number_format = '0.0'


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

Creating a new spreadsheet

- Use the Workbook() function
- One worksheet created by default
- Add worksheets with WB.create_sheet(n)
- Copy worksheets with WB.copy_sheet(n)
- Add data rows with WS.append(iterable)

To create a new spreadsheet file, use the Workbook() function. It creates a new workbook, with a default worksheet named "Sheet1".

Add worksheets with `WB.create_sheet(n)`. The parameter indicates where to insert the new worksheet; if not specified, it is appended.

To get or set the name of the worksheet, use its `.title` property.

To easily add rows to the worksheet, use `WS.append(iterable)`, where *iterable* is an iterable of column values.

Example

px_create_worksheet.py

```python
#!/usr/bin/env python
import openpyxl as px

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

wb = px.Workbook()

ws = wb.active

ws.title = 'fruits'

ws.append(['Fruit', 'Length'])

for fruit in fruits:
    ws.append([fruit, len(fruit)])

# hard way
# for i, fruit in enumerate(fruits, 1):
#     ws.cell(row=i, column=1).value = fruit
#     ws.cell(row=i, column=2).value = len(fruit)

wb.save('fruits.xlsx')
```

Inserting, Deleting, and moving cells

- Insert
  - `ws.insert_rows(row_index, num_rows=1)`
  - `ws.insert_cols(col_index, num_cols=1)`
- Delete
  - `ws.delete_rows(row_index, num_rows)`
  - `ws.delete_cols(col_index, num_cols)`
- Move
  - `ws.move_range(range, rows=row_delta, cols=col_delta)`
- Append
  - `ws.append(iterable)`

To insert one or more blank rows or columns, use `ws.insert_rows()` or `ws.insert_cols()`. The first argument is the positional index of the row or column (1-based), and the second argument is how many columns to insert.

To delete rows or columns, use `ws.delete_rows()` or `ws.delete_cols()`. The first argument is the index of the first row or column to delete; the second is the number of rows or columns.

To move a range of rows and columns, use `ws.move_range()`. The first argument is a range string such as `A1:F10`. Add named arguments `rows` and `cols` to specify how many cells to move. Positive values move down or right, and negative values move up or left. Existing data will be overwritten at the new location of the moved cells.

To append a row of data to a worksheet, pass an iterable to `ws.append()`.

Example

px_insert_delete_move.py

```python
#!/usr/bin/env python
import openpyxl as px

RAW_DATA = [47, "Mouse", "Mickey", None, None, "Anaheim", "California", "2025-01-20",
None, "Imagineer"]

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    insert_cells(ws)
    delete_cells(ws)
    move_cells(ws)
    append_cells(ws)

    print(ws.dimensions)

    wb.save('presidents_insert_delete_move.xlsx')

def append_cells(ws):
    ws.append(RAW_DATA)

def insert_cells(ws):
    ws.insert_rows(1, 3)  # insert three rows at top
    ws.insert_cols(5)  # insert one col at position 5

def delete_cells(ws):
    ws.delete_rows(15,5)
    ws.delete_cols(6)

def move_cells(ws):
    ws.move_range('A43:K45', rows=6, cols=3)


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

Hiding and freezing columns and sheets

- Hide
  - `ws.column_dimensions[column]`
  - `ws.column_dimensions.group(column, …)`
- Freeze
  - `ws.freeze_panes = 'coordinate'`
- Hide entire sheet
  - `ws.sheet_state = 'hidden'`

You can hide a column by using the `.column_dimensions` property of a worksheet. Specify a column letter inside square brackets, and assign `True` to the `.hidden` property. To hide multiple columns, use `.column_dimensions.group()`. Specify start and end columns, and set the `hidden` argument to `True`

```
ws.column_dimensions['M'].hidden = True
ws.column_dimensions.group(start="C", end="F", hidden=True)
```

To freeze rows and columns for scrolling, assign the coordinates of the first row and column that you want to scroll to `ws.freeze_panes`. For example, to freeze the first 3 columns and start scrolling with column 'D', use

```
ws.freeze_panes = 'A4'
```

You can also hide a worksheet by setting `ws.sheet_state` to `"hidden"`.

Example

px_hide_freeze.py

```
#!/usr/bin/env python
import openpyxl as px


def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    hide_columns(ws)

    wb.save('presidents_hidden.xlsx')

    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    freeze_columns(ws)

    create_hidden_sheet(wb)

    wb.save('presidents_frozen.xlsx')

    wb = px.load_workbook('../DATA/presidents.xlsx')

    create_hidden_sheet(wb)

    wb.save('presidents_hidden_sheet.xlsx')


def hide_columns(ws):
    """Hide single columnn and multiple columns"""

    # hide birthplace column
    ws.column_dimensions['F'].hidden = True

    # hide inauguration columns
    ws.column_dimensions.group(start='H', end='I', hidden=True)

def freeze_columns(ws):
    """Freeze the first 2 columns"""
    ws.freeze_panes = "C1"

def create_hidden_sheet(wb):
    """Add a hidden worksheet to the workbook"""
```

```
    ws = wb.create_sheet(title="secret plans")
    ws.sheet_state = "hidden"


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


Setting Styles

- Must be set on each cell individually
- Cannot change, once assigned (but can be replaced)
- Copy style to make changes

Each cell has a group of attributes that control its styles and formatting. Most of these have a corresponding class; to change styles, create an instance of the appropriate class and assign it to the attribute.

You can also make a copy of an existing style object, and just change the attributes you need.

Example

px_styles.py

```
#!/usr/bin/env python
import openpyxl as px

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    update_last_names(ws)

    wb.save('presidents_styles.xlsx')

def update_last_names(ws):
    """Make the last name column blue and bold"""
    for row in ws['B2:B47']:
        cell = row[0]
        cell.value = cell.value.upper()
        cell.font = px.styles.Font(color='FF0000FF')

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

Table 3. openpyxl Cell Style Attributes

Cell attribute	Class	Parameters	Default value
font	Font		
		name	'Calibri'
		size	11
		bold	False
		italic	False
		vertAlign	None
		underline	'none'
		strike	False
		color	'FF000000'
fill	PatternFill		
		fill_type	None
		start_color	'FFFFFFFF'
		end_color	'FF000000'
border	Border		
		left	Side(border_style=None, color='FF000000')
		right	Side(border_style=None, color='FF000000')
		top	Side(border_style=None, color='FF000000')
		bottom	Side(border_style=None, color='FF000000')
		diagonal	Side(border_style=None, color='FF000000')
		diagonal_direction	
		outline	Side(border_style=None, color='FF000000')
		vertical	Side(border_style=None, color='FF000000')
		horizontal	Side(border_style=None, color='FF000000')
alignment	Alignment		

Cell attribute	Class	Parameters	Default value
		horizontal	'general'
		vertical	'bottom'
		text_rotation	0
		wrap_text	False
		shrink_to_fit	False
		indent	0
number_format	None	N/A	'General'
protection	Protection		
		locked	True
		hidden	False

Conditional Formatting

- Apply styles per values
- Types
    - Builtin
    - Standard
    - Custom
- Components
    - Differential Style
    - Rule
        - Formula

Conditional formatting means applying styles to cells based on their values. In openpyxl, conditional formatting can be a little complicated.

There are three kinds of rules for conditional formatting:

- Builtin — predefined rules with predefined styles
- Standard — predefined rules with custom styles
- Custom — custom rules with custom styles

Because this is complicated, there are some convenience functions for generating some formats.

Components

Formatting requres *styles*, which are either builtin or configured via a `DifferentialStyle` object, and *rules*, which are embedded in the formats. For custom rules, you provide a *formula* that defines when the rule should be used.

Builtin formats

There are three conditional formats: `ColorScale`, `IconSet`, and `DataBar`. These formats contain various settings, which compare the value to ann integer using one of these types: `num`, `percent`, `max`, `min`, `formula`, or `percentile`.

ColorScale

`ColorScale` provides a rule for a gradient from one color to another for the values within a range. You can add a second `ColorScale` for two gradients.

The convenience function for `ColorScale` is `openpyxl.formatting.rule.ColorScaleRule()`.

IconSet

`IconSet` provides a rule for applying different icons to different values.

The convenience function for `IconSet` is `openpyxl.formatting.rule.IconSetRule()`.

DataBar

`DataBar` provides a rule for adding "data bars", similar to the bars used by mobile phones to indicate signal strength.

The convenience function for `DataBar` is `openpyxl.formatting.rule.DataBarRule()`.

Example

px_conditional_styles.py

```
#!/usr/bin/env python
import openpyxl as px
from openpyxl.formatting.rule import (
    Rule, ColorScale, FormatObject, IconSet, DataBar
)

from openpyxl.styles import Font, PatternFill, Color
from openpyxl.styles.differential import DifferentialStyle

CONDITIONAL_CONFIG = {
    'Republican': {
        'font_color': "FF0000",
        'fill': "FFC0CB",
    },
    'Democratic': {
        'font_color': "0000FF",
        'fill': "ADD8E6",
    },
    'Whig': {
        'font_color': "008000",
        'fill': "98FB98",
    }
}

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    colorscale_values(ws)

    color_potus_parties(ws)

    icon_values(ws)

    wb.save('presidents_conditional.xlsx')

    wb = px.load_workbook('../DATA/columns_of_numbers.xlsx')
    icon_values(wb.active)
    databar_values(wb.active)
    wb.save('columns_with_icons.xlsx')

def colorscale_values(ws):
    """
```

```
    Add conditional style to the "TERM" column using a builtin
    type.

    :param ws: the worksheet
    :return: None
    """
    first = FormatObject(type="min")
    last = FormatObject(type="max")
    colors = [Color('AA0000'), Color('00AA00')]
    cs2 = ColorScale(cfvo=[first, last], color=colors)
    rule = Rule(type='colorScale', colorScale=cs2)

    last_row = ws.max_row
    ws.conditional_formatting.add(f'A2:A{last_row}', rule)



def color_potus_parties(ws):
    """
        Make Republicans red and Democrats blue, etc.

        This is a custom rule with a custom formula.

        :param ws: Worksheet to format
        :returns: None
    """
    for text, config in CONDITIONAL_CONFIG.items():
        font = Font(color=config['font_color'])
        fill = PatternFill(bgColor=config['fill'])
        dxf = DifferentialStyle(font=font, fill=fill)

        # make a rule for this condition
        rule = Rule(type="expression", dxf=dxf)

        # add an Excel formula to the rule. Cell must be first cell of
        # range; otherwise formatting is offset by difference from first
        # cell to specified cell
        #
        # can use any Excel text operations here
        rule.formula=[f'EXACT("{text}",$J2)']

        # add the rule to desired range
        ws.conditional_formatting.add('J2:J47', rule)

def icon_values(ws):
    """
    Add icons for numeric values in column.
```

```
    :param ws: worksheet to format
    :return: None
    """
    thresholds = [0, 33, 67]
    icons = [FormatObject(type='percent', val=t) for t in thresholds]
    iconset = IconSet(iconSet='3TrafficLights1', cfvo=icons)
    rule = Rule(type='iconSet', iconSet=iconset)
    format_range = f"A2:A{ws.max_row}"
    ws.conditional_formatting.add(format_range, rule)

def databar_values(ws):
    """
    Add conditional databars to worksheet.

    :param ws: worksheet to format
    :return: None
    """
    first = FormatObject(type='min')
    second = FormatObject(type='max')
    data_bar = DataBar(cfvo=[first, second], color="638EC6")
    rule = Rule(type='dataBar', dataBar = data_bar)
    format_range = f"F2:F{ws.max_row}"
    ws.column_dimensions['F'].width = 25  # make column wider for data bar
    ws.conditional_formatting.add(format_range, rule)


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

Chapter 2 Exercises

Exercise 2-1 (age_of_geeks.py,age_of_geeks_formula.py)

Write a script to compute the average age of the people on the worksheet 'people' in **computer_people.xlsx**. First, you'll have to calculate the age from the birth date. (Some of the people in the worksheet have died. Just include them for purposes of this exercise).

TIP	TO calculate the age, get today's date (`datetime.datetime.now()`), subtract the DOB cell value from today's date. This gets a **timedelta** object. Use the **days** attribute of the timedelta divided by 365 to get the age.

NOTE	Another way to do this lab (if Excel is available) is to use this Excel formula: `=DATEDIF(DOB,TODAY(),"y")` where DOB is the cell containing the birthdate, such as "D2". Add an additional column. Then add a formula to average the values in this new column. Since OpenPyXL can't recalculate a sheet, open the sheet in Excel to the the results. (You could also use Libre Office or Open Office to open the workbook).

Print out the average.

Exercise 2-2 (knights_to_spreadsheet.py, knights_to_spreadsheet_extra.py

Write a script to create a new spreadsheet with data from the knights.txt file. The first row of the spreadsheet should have the column headings:

Name, Title, Favorite Color, Quest, Comment

The data should start after that.

Save the workbook as knights.xlsx.

NOTE	For extra fun, make the headers bold, the name fields red, and the comments in italics.

Chapter 3: 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 4. 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 5. connect() examples

<table>
<thead>
<tr><th>Package</th><th>Database</th><th>Connection</th></tr>
</thead>
<tbody>
<tr><td>IBM DB2</td><td>ibm-db</td><td>

```
import ibm_db_dbi as db2
conn = db2.connect(

"DATABASE=testdb;HOSTNAME=localhost;PORT=50000;PROTOCOL=TCPIP;
UID=db2inst1;PWD=scripts;",
    "",
    ""
)
```

</td></tr>
<tr><td>cx-Oracle</td><td>Oracle</td><td>

```
ip = 'localhost'
port = 1521
SID = 'YOURSIDHERE'
dsn_tns = cx_Oracle.makedsn(ip, port, SID)
db = cx_Oracle.connect('adeveloper', '$3cr3t', dsn_tns)
```

</td></tr>
<tr><td>PostgreSQL</td><td>psychopg</td><td>

```
psycopg2.connect ('''
    host='localhost'
    user='adeveloper'
    password='$3cr3t'
    dbname='testdb'
''')
```

NOTE `connect()` has one (string) parameter, not multiple parameters

</td></tr>
<tr><td>MS-SQL</td><td>pymssql</td><td>

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

</td></tr>
</tbody>
</table>

<table>
<tr><th>Package</th><th>Database</th><th>Connection</th></tr>
<tr><td>MySQL</td><td>pymysql</td><td>

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

</td></tr>
<tr><td>ODBC-compliant DB</td><td>pyodbc</td><td>

```
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

</td></tr>
<tr><td>SqlLite3</td><td>sqlite3</td><td>

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

</td></tr>
</table>

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 6. 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 7. 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
```


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 8. 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 3 Exercises

Exercise 3-1 (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 9. 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 3-2 (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 4: 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 10. 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 11. **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

Authentication with requests

- Options
  - Basic-Auth
  - Digest
  - Custom
- Use **auth** argument

requests makes it eaasy to provide basic authentication to a web site.

In the simplest case, create a `requests.auth.HTTPBasicAuth` object with the username and password, then pass that to requests with the `auth` argument. Since this is a common use case, you can also just pass a `(user, password)` tuple to the `auth` parameter.

For digest authentication, use `requests.auth.HTTPDigestAuth` with the username and password.

For custom authentication, you can create your own auth class by inheriting from `requests.auth.AuthBase`.

For OAuth 1, OAuth 2, and OpenID, install `requests-oauthlib`. This additional module provides auth objects that can be passed in with the `auth` parameter, as above.

See https://docs.python-requests.org/en/latest/user/authentication/ for more details.

Example

basic_auth_requests.py

```
import requests
from requests.auth import HTTPBasicAuth, HTTPDigestAuth

# base URL for httpbin
BASE_URL = 'https://httpbin.org'

# formats for httpbin
BASIC_AUTH_FMT = "/basic-auth/{}/{}"
DIGEST_AUTH_FMT = "/digest-auth/{}/{}/{}"

USERNAME = "spam"
PASSWORD = "ham"
BAD_PASSWORD = "toast"

REPORT_FMT = "{:35s} {}"

def main():
    basic_auth()
    digest()

def basic_auth():
    auth = HTTPBasicAuth(USERNAME, PASSWORD)
    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth good password", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=(USERNAME, PASSWORD),
    )
    print(REPORT_FMT.format("Basic auth good password (shortcut)", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, BAD_PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth bad password", response))

def digest():
    auth = HTTPDigestAuth(USERNAME, PASSWORD)
    response = requests.get(
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
```

```
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth good password", response))

    auth = HTTPDigestAuth(USERNAME, BAD_PASSWORD)
    response = requests.get(
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth bad password", response))


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

basic_auth_requests.py

```
Basic auth good password            <Response [200]>
Basic auth good password (shortcut) <Response [200]>
Basic auth bad password             <Response [401]>
Digest auth good password           <Response [200]>
Digest auth bad password            <Response [401]>
```

Grabbing a web page the hard way

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

While **requests** simplifies creating an HTTP client, the standard library module **urllib.request** includes **urlopen()**. It 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, the preferred approach is to install and use **requests**.

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: 50002
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: Sun, 20 Feb 2022 17:21:07 GMT
Age: 136
X-Served-By: cache-iad-kcgs7200114-IAD, cache-pdk17835-PDK
X-Cache: HIT, HIT
X-Cache-Hits: 4, 1
X-Timer: S1645377668.772546,VS0,VE1
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains



<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]-->
<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">                 <![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr">  <!--<![endif]-->

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <link rel="prefetch" href="//ajax.googleapis.com/ajax/libs/jqu
```

Example

read_pdf_urllib.py

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

import sys
import os
from urllib.request import urlopen
from urllib.error import HTTPError

# url to download a PDF file of a NASA ISS brochure

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

saved_pdf_file = 'nasa_iss.pdf'  ②

try:
    URL = urlopen(url)  ③
except HTTPError as e:  ④
    print("Unable to open URL:", e)
    sys.exit(1)

pdf_contents = URL.read()  ⑤
URL.close()

with open(saved_pdf_file, 'wb') as pdf_in:
    pdf_in.write(pdf_contents)  ⑥

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

os.system(cmd)  ⑧
```

① target URL

② name of PDF file for saving

③ open the URL

④ catch any HTTP errors

⑤ read all data from URL in binary mode

⑥ write data to a local file in binary mode

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

⑧ launch the app

Consuming Web services 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 Feb 18 14:47 alice.txt
drwxr-xr-x   2 python  staff      64 May 27  2021 foo
drwxr-xr-x   2 python  staff      64 May 27  2021 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 18 Feb 14:47 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 2021  testing
drwxr-xr-x   1 503      20            128 11 Feb 2021  Public
drwxr-xr-x   1 503      20             64 27 May 2021  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 4 Exercises

Exercise 4-1 (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 4-2 (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".

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.

NOTE	For detailed screen-scraping, you can use the Beautiful Soup module.

Exercise 4-3 (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 5: 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 12. 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 13. 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 0x7fc1f016c780>

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 14. 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 15. 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 16. 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 5 Exercises

Exercise 5-1 (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 5-2 (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 5-3 (jpresidents.py)

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

Exercise 5-4 (cpresidents.py)

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

Exercise 5-5 (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 5-6 (unpickle_potus.py)

Write a script to open presidents.pic, and restore the data back into a dictionary.

Then loop through the array and print out each president's first name, last name, and party.

Chapter 6: Multiprogramming

Objectives

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

Multiprogramming

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

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

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

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

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

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

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

The standard library supports all three.

What Are Threads?

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

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

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

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

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

The Python Thread Manager

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

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

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

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

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

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

The threading Module

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

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

Threads for the impatient

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

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

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

Example

thr_noclass.py

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

import threading
import random
import time


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


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

print("Done.")  ④
```

① function to launch in each thread

② create thread

③ launch thread

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

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


Creating a thread class

- Subclass Thread
- *Must* call base class's __init__()
- *Must* implement run()
- Can implement helper methods

A thread class is a class that starts a thread, and performs some task. Such a class can be repeatedly instantiated, with different parameters, and then started as needed.

The class can be as elaborate as the business logic requires. There are only two rules: the class must call the base class's __init__(), and it must implement a run() method. Other than that, the run() method can do pretty much anything it wants to.

The best way to invoke the base class __init__() is to use super().

The run() method is invoked when you call the start() method on the thread object. The start() method does not take any parameters, and thus run() has no parameters as well.

Any per-thread arguments can be passed into the constructor when the thread object is created.

Example

thr_simple.py

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

from threading import Thread
import random
import time


class SimpleThread(Thread):
    def __init__(self, num):
        super().__init__()  ①
        self._threadnum = num

    def run(self):  ②
        time.sleep(random.randint(1, 3))
        print("Hello from thread {}".format(self._threadnum))


for i in range(10):
    t = SimpleThread(i)  ③
    t.start()  ④

print("Done.")
```

① call base class constructor — REQUIRED

② the function that does the work in the thread

③ create the thread

④ launch the thread

thr_simple.py

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

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

Using queues

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

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

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

Example

thr_queue.py

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

```
['SCRUPULOUSLY-SCRUPULOUSLY', 'DEVITRIFICATIONS-DEVITRIFICATIONS', 'SLEAZO-SLEAZO',
'INQUIET-INQUIET', 'RESEEKING-RESEEKING', 'BENDERS-BENDERS', 'FAINTING-FAINTING',
'VERISMOS-VERISMOS', 'REBRED-REBRED', 'REABSORB-REABSORB', 'INTERLOBULAR-INTERLOBULAR',
'VOLTAISMS-VOLTAISMS', 'FLASHBACK-FLASHBACK', 'HOMINOID-HOMINOID', 'WEATHERLY-WEATHERLY',
'OVERBEATEN-OVERBEATEN', 'REPACIFIED-REPACIFIED', 'IMMUNOCOMPETENCE-IMMUNOCOMPETENCE',
'PERTUSSIS-PERTUSSIS', 'ALEURON-ALEURON']
Sun Feb 20 12:21:20 2022
Sun Feb 20 12:21:20 2022
```

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

```
['GADABOUT', 'SOLUMS', 'FATBIRD', 'EROTICISTS', 'GLIMMERINGS', 'STHENIC', 'DURABILITIES',
'GEYSERITES', 'ENDPOINTS', 'INCONGRUOUSNESS', 'PRIVATELY', 'APHIDES', 'MOONY',
'PEARLITES', 'DIGITIZES', 'ANTIFEMINISM', 'TURNSTONES', 'EMBATTLEMENT', 'PROKARYOTIC',
'ESTERIFIED', 'GEOPOLITICS', 'ENDOSKELETON', 'MULTICHARACTER', 'SPRITES', 'WITNESSING',
'BOVIDS', 'BLOCKHOUSES', 'EXEMPTION', 'CHANCELLERY', 'CAPSULATED', 'BURAN', 'SHOWED',
'PREGNENOLONE', 'TITI', 'ORRISROOTS', 'BIOCHEMICAL', 'TOTALITARIANIZE', 'SPLENDOUR',
'DEFUZE', 'CHRYSALID', 'SPONGINS', 'SQUAWKER', 'OVERCOLD', 'COURSEWARE', 'NONOBSERVANT',
'CERTIFIERS', 'DELICIOUSNESS', 'MISERLINESS', 'PURIFIES', 'EXORCIZE']
25000
Sun Feb 20 12:21:21 2022
Sun Feb 20 12:21:22 2022
```

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

```
['WAINSCOT', 'EMPRISE', 'ALRIGHT', 'INTERVENERS', 'CATEGORIZING', 'BIRDED', 'FRIBBLER',
'REVISALS', 'VICTIMIZER', 'UPROOT', 'PYROLYSES', 'FUMELIKE', 'DIGITALINS', 'GAMESOME',
'CANOLAS', 'PSEUDOALLELE', 'COOKHOUSE', 'HONKIE', 'LITERATELY', 'ALPENHORNS']
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

```
['NUCLEATORS', 'GRITTILY', 'HILLER', 'CATS', 'ADVERTISINGS', 'UNIVALENT', 'TETRACID',
'LUSTFUL', 'HECTOMETERS', 'TRACKLAYINGS', 'SWATH', 'LECHER', 'PELAGES', 'PUBLICIZE',
'HUMBLER', 'COTTONMOUTHS', 'PROOFREAD', 'EROTICIZE', 'PEDAL', 'MUKLUKS']
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 6 Exercises

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

Exercise 6-1 (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 6-2 (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 6-3 (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 6-4 (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 7: 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 17. 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 0.01s ===============================
```

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 0.02s =============================
```


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 0.02s ==================
```

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 0.02s ==============
```

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 0.01s ===============================
```

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 0.01s ===============================
```

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 7 Exercises

Exercise 7-1 (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 8: 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/ncvoter32.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 18. 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 Jan 22 16:16 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 Jan 22 16:16 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 19. 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 20. 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 21. 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 2022-02-20 12:21:31,831 INFO this is information
root 2022-02-20 12:21:31,831 WARNING this is a warning
root 2022-02-20 12:21:31,831 INFO this is information
root 2022-02-20 12:21:31,831 CRITICAL this is critical
```


Table 22. 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 8 Exercises

Exercise 8-1 (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.

Appendix A: Where do I go from here?

Resources for learning Python

These are from Jessica Garson, who, among other things, teaches Python classes at NYU. (Used with permission).

Run the script **where_do_i_go.py** to display a web page with live links.

Resources for Learning Python [https://dev.to/jessicagarson/resources-for-learning-python-hd6]

Just getting started

Here are some resources that can help you get started learning how to code.

- Code Newbie Podcast [https://www.codenewbie.org/podcast]
- Dive into Python3 [http://www.diveintopython3.net]
- Learn Python the Hard Way [https://learnpythonthehardway.org/python3]
- Learn Python the Hard Way [https://learnpythonthehardway.org/python3]
- Automate the Boring Stuff with Python [https://automatetheboringstuff.com]
- Automate the Boring Stuff with Python [https://automatetheboringstuff.com]

So you want to be a data scientist?

- Data Wrangling with Python [https://www.amazon.com/Data-Wrangling-Python-Tools-Easier/dp/1491948817]
- Data Analysis in Python [http://www.data-analysis-in-python.org/index.html]
- Titanic: Machine Learning from Disaster [https://www.kaggle.com/c/titanic/discussion/5105]
- Deep Learning with Python [https://www.manning.com/books/deep-learning-with-python]
- How to do X with Python [https://chrisalbon.com/]
- Machine Learning: A Probabilistic Prospective [https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020]

So you want to write code for the web?

- Learn flask, some great resources are listed here [https://www.fullstackpython.com/flask.html]
- Django Polls Tutorial [https://docs.djangoproject.com/en/2.0/intro/tutorial01/]
- Hello Web App [https://www.amazon.com/Hello-Web-App-Learn-Build-ebook/dp/B00U5MMZ2E/ref=sr_1_1?ie=UTF8&qid=1510599119&sr=8-1&keywords=hello+web+app]
- Hello Web App Intermediate [https://www.amazon.com/Hello-Web-App-Intermediate-Concepts/dp/0986365920]

- Test-Driven-Development for Web Programming [https://www.obeythetestinggoat.com/pages/book.html#toc]
- 2 Scoops of Django [https://www.amazon.com/Two-Scoops-Django-1-11-Practices-ebook/dp/B076D5FKFX/ref=sr_1_1?s=books&ie=UTF8&qid=1510598897&sr=1-1&keywords=2+scoops+of+django]
- HTML and CSS: Design and Build Websites [https://www.amazon.com/HTML-CSS-Design-Build-Websites/dp/1118008189/ref=sr_1_1?ie=UTF8&qid=1510599157&sr=8-1&keywords=css+and+html]
- JavaScript and JQuery [https://www.amazon.com/JavaScript-JQuery-Interactive-Front-End-Development/dp/1118531647]

Not sure yet, that's okay!

Here are some resources for self guided learning. I recommend trying to be very good at Python and the rest should figure itself out in time.

- Python 3 Crash Course [https://www.amazon.com/Python-Crash-Course-Hands-Project-Based/dp/1593276036]
- Base CS Podcast [https://www.codenewbie.org/basecs]
- Writing Idiomatic Python [https://www.amazon.com/Writing-Idiomatic-Python-Jeff-Knupp-ebook/dp/B00B5VXMRG]
- Fluent Python [https://www.amazon.com/dp/1491946008?aaxitk=o7.Y1C9z7oJp87fs3ev30Q&pd_rd_i=1491946008&hsa_cr_id=1406361870001]
- Pro Python [https://www.amazon.com/Pro-Python-Marty-Alchin/dp/1484203356/ref=sr_1_1?s=books&ie=UTF8&qid=1510598874&sr=1-1&keywords=pro+python]
- Refactoring [https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672/ref=sr_1_1?ie=UTF8&qid=1510598784&sr=8-1&keywords=refactoring+martin+fowler]
- Clean Code [https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=sr_1_1?s=books&ie=UTF8&qid=1510598926&sr=1-1&keywords=clean+code]
- Write music with Python, since that's my favorite way to learn a new language [https://github.com/reckoner165/soundmodular]

Appendix B: Python Bibliography

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

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

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

Appendix C: String Formatting

Overview

- Strings have a format() method
- Allows values to be inserted in strings
- Values can be formatted
- Add a field as placeholders for variable
- Field syntax: `{SELECTOR:FORMATTING}`
- Selector can be empty, index, or keyword
- Formatting controls alignment, width, padding, etc.

Python provides a powerful and flexible way to format data. The string method format() takes one or more parameters, which are inserted into the string via placeholders.

The placeholders, called fields, consist of a pair of braces enclosing parameter selectors and formatting directives.

The selector can be followed by a set of formatting directives, which always start with a colon. The simplest directives specify the type of variable to be formatted. For instance, {1:d} says to format the second parameter as an integer; {0:.2f} says to format the first parameter as a float, rounded to two decimal points.

The formatting part can consist of the following components, which will be explained in detail in the following pages:

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

Parameter Selectors

- Null for auto-numbering
- Can be numbers or keywords
- Start at 0 for numbers

Selectors refer to which parameter will be used in a placeholder.

Null (empty) selectors — the most common — will be treated as though they were filled in with numbers from left to right, beginning with 0. Null selectors cannot be mixed with numbered or named selectors — either all of the selectors or none of the selectors must be null.

Non-null selectors can be either numeric indices or keywords (strings). Thus, {0} will be replaced with the first parameter, {4} will be replaced with the fifth parameter, and so on. If using keywords, then {name} will be replaced by the value of keyword *name*, and {age} will be replaced by keyword *age*.

Parameters do not have to be in the same order in which they occur in the string, although they typically are. The same parameter can be used in multiple fields.

If positional and keyword parameters are both used, the keyword parameters must come after all positional parameters.

Example

fmt_params.py

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

person = 'Bob'
age = 22

print("{0} is {1} years old.".format(person, age))  ①
print("{0}, {0}, {0} your boat".format('row'))  ②
print("The {1}-year-old is {0}".format(person, age))  ③
print("{name} is {age} years old.".format(name=person, age=age))  ④
print()
print("{} is {} years old.".format(person, age))  ⑤
print("{name} is {} and his favorite color is {}".format(22, 'blue', name='Bob'))  ⑥
```

① Placeholders can be numbered

② Placeholders can be reused

③ They do not have to be in order (but usually are)

④ Selectors can be named

⑤ Empty selectors are autonumbered (but all selectors must either be empty or explicitly numbered)

⑥ Named and numbered selectors can be mixed

fmt_params.py

```
Bob is 22 years old.
row, row, row your boat
The 22-year-old is Bob
Bob is 22 years old.

Bob is 22 years old.
Bob is 22 and his favorite color is blue
```

f-strings

- **f** in front of literal strings
- More readable
- Same rules as `__string__.format()`

Starting with version 3.6, Python also supports *f-strings*.

The big difference from the `format()` method is that the parameters are inside the `{}` placeholders. Place formatting details after a `:` as usual.

Since the parameters are part of the placeholders, parameter numbers are not used.

All of the following formatting tools work with both `__string__.format()` and f-strings.

Example

fmt_fstrings.py

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

person = 'Bob'
age = 22

print(f"{person} is {age} years old.")
print(f"The {age}-year-old is {person}.")
print()
```

fmt_fstrings.py

```
Bob is 22 years old.
The 22-year-old is Bob.
```


Data types

- Fields can specify data type
- Controls formatting
- Raises error for invalid types

The type part of the format directive tells the formatter how to convert the value. Builtin types have default formats – *s* for strings, *d* for integers, *f* for float.

Some data types can be specified as either upper or lower case. This controls the output of letters. E.g, {:x} would format the number 48879 as *beef*, but {:X} would format it as *BEEF*.

The type must generally match the type of the parameter. An integer cannot be formatted with type *s*. Integers can be formatted as floats, but not the other way around. Only integers may be formatted as binary, octal, or hexadecimal.

Example

fmt_types.py

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

person = 'Bob'
value = 488
bigvalue = 3735928559
result = 234.5617282027

print('{:s}'.format(person))    ①
print('{name:s}'.format(name=person))    ②
print('{:d}'.format(value))    ③
print('{:b}'.format(value))    ④
print('{:o}'.format(value))    ⑤
print('{:x}'.format(value))    ⑥
print('{:X}'.format(bigvalue))    ⑦
print('{:f}'.format(result))    ⑧
print('{:.2f}'.format(result))    ⑨
```

① String

② String

③ Integer (displayed as decimal)

④ Integer (displayed as binary)

⑤ Integer (displayed as octal)

⑥ Integer (displayed as hex)

⑦ Integer (displayed as hex with uppercase digits)

⑧ Float (defaults to 6 places after the decimal point)

⑨ Float rounded to 2 decimal places

fmt_types.py

```
Bob
Bob
488
111101000
750
1e8
DEADBEEF
234.561728
234.56
```

Table 23. Formatting Types

b	Binary – converts number to base 2
c	Character – converts to corresponding character, like chr()
d	Decimal – outputs number in base 10
e, E	Exponent notation. *e* prints the number in scientific notation using the letter *e* to indicate the exponent. *E* is the same, except it uses the letter *E*
f,F	Floating point. *F* and *f* are the same.
g	General format. For a given precision p >= 1, rounds the number to p significant digits and then formats the result in fixed-point or scientific notation, depending on magnitude. This is the default for numbers
G	Same as g, but upper-cases *e*, *nan*, and 'inf"
n	Same as d, but uses locale setting for number separators
o	Octal – converts number to base 8
s	String format. This is the default type for strings
x, X	Hexadecimal – convert number to base 16; A-F match case of *x* or *X*
%	Percentage. Multiplies the number by 100 and displays in fixed (*f*) format, followed by a percent sign.

Field Widths

- Specified as {0:width.precision}
- Width is really minimum width
- Precision is either maximum width or # decimal points

Fields can specify a minimum width by putting a number before the type. If the parameter is shorted than the field, it will be padded with spaces, on the left for numbers, and on the right for strings.

The precision is specified by a period followed by an integer. For strings, precision means the maximum width. Strings longer than the maximum will be truncated. For floating point numbers, precision means the number of decimal places displayed, which will be padded with zeros as needed.

Width and precision are both optional. The default width for all fields is 0; the default precision for floating point numbers is 6.

It is invalid to specify precision for an integer.

Example

fmt_width.py

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

name = 'Ann Elk'
value = 10000
airspeed = 22.347
# note: [] are used to show blank space, and are not part of the formatting
print('[{:s}]'.format(name))     ①
print('[{:10s}]'.format(name))   ②
print('[{:3s}]'.format(name))    ③
print('[{:3.3s}]'.format(name))  ④
print()
print('[{:8d}]'.format(value))       ⑤
print('[{:8f}]'.format(value))       ⑥
print('[{:8f}]'.format(airspeed))    ⑦
print('[{:.2f}]'.format(airspeed))   ⑧
print('[{:8.3f}]'.format(airspeed))  ⑨
```

① Default format — no padding

② Left justify, 10 characters wide

③ Left justify, 3 characters wide, displays entire string

④ Left justify, 3 characters wide, truncates string to max width

⑤ Right justify, decimal, 8 characters wide (all numbers are right-justified by default)

⑥ Right justify int as float, 8 characters wide

⑦ Right justify float as float, 8 characters wide

⑧ Right justify, float, 3 decimal places, no maximum width

⑨ Right justify, float, 3 decimal places, maximum width 8

fmt_width.py

```
[Ann Elk]
[Ann Elk   ]
[Ann Elk]
[Ann]

[   10000]
[10000.000000]
[22.347000]
[22.35]
[  22.347]
```


Alignment

- Alignment within field can be left, right, or centered
    - < left align
    - > right align
    - ^ center
    - = right align but put padding after sign

You can align the data to be formatted. It can be left-aligned (the default), right-aligned, or centered. If formatting signed numbers, the minus sign can be placed on the left side.

Example

fmt_align.py

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

name = 'Ann'
value = 12345
nvalue = -12345

①
print('[{0:10s}]'.format(name))    ②
print('[{0:<10s}]'.format(name))   ③
print('[{0:>10s}]'.format(name))   ④
print('[{0:^10s}]'.format(name))   ⑤
print()
print('[{0:10d}] [{1:10d}]'.format(value, nvalue))    ⑥
print('[{0:>10d}] [{1:>10d}]'.format(value, nvalue))  ⑦
print('[{0:<10d}] [{1:<10d}]'.format(value, nvalue))  ⑧
print('[{0:^10d}] [{1:^10d}]'.format(value, nvalue))  ⑨
print('[{0:=10d}] [{1:=10d}]'.format(value, nvalue))  ⑩
```

① note: all of the following print in a field 10 characters widedd

② Default (left) alignment

③ Explicit left alignment

④ Right alignment

⑤ Centered

⑥ Default (right) alignment

⑦ Explicit right alignment

⑧ Left alignment

⑨ Centered

⑩ Right alignment, but pad *after* sign

fmt_align.py

```
[Ann       ]
[Ann       ]
[       Ann]
[   Ann    ]

[     12345] [    -12345]
[     12345] [    -12345]
[12345     ] [-12345    ]
[  12345   ] [  -12345  ]
[     12345] [-    12345]
```

Fill characters

- Padding character must precede alignment character
- Default is one space
- Can be any character except }

By default, if a field width is specified and the data does not fill the field, it is padded with spaces. A character preceding the alignment character will be used as the fill character.

Example

fmt_fill.py

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

name = 'Ann'
value = 123

print('[{:>10s}]'.format(name))    ①
print('[{:.>10s}]'.format(name))   ②
print('[{:->10s}]'.format(name))    ③
print('[{:.10s}]'.format(name))    ④
print()
print('[{:10d}]'.format(value))    ⑤
print('[{:010d}]'.format(value))   ⑥
print('[{:_>10d}]'.format(value))  ⑦
print('[{:+>10d}]'.format(value))  ⑧
```

① Right justify string, pad with space (default)

② Right justify string, pad with .

③ Right justify string, pad with -

④ Left justify string, pad with .

⑤ Right justify number, pad with space (default

⑥ Right justify number, pad with zeroes

⑦ Right justfy, pad with _ (> required)

⑧ Right justfy, pad with + (> required)

fmt_fill.py

```
[       Ann]
[.......Ann]
[-------Ann]
[Ann]

[       123]
[0000000123]
[_______123]
[+++++++123]
```

Signed numbers

- Can pad with any character except *{}*
- Sign can be +, -, or space
- Only appropriate for numeric types

The sign character follows the alignment character, and can be plus, minus, or space.

A plus sign means always display + or – preceding non-zero numbers.

A minus sign means only display a sign for negative numbers.

A space means display a – for negative numbers and a space for positive numbers.

Example

fmt_signed.py

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

values = 123, -321, 14, -2, 0

for value in values:
    print("default: |{:d}|".format(value))  ①
print()

for value in values:
    print("   plus: |{:+d}|".format(value))  ②
print()

for value in values:
    print("  minus: |{:-d}|".format(value)) ③
print()

for value in values:
    print("  space: |{: d}|".format(value)) ④
print()
```

① default (pipe symbols just to show white space)

② plus sign puts + on positive numbers (and zero) and - on negative

③ minus sign only puts - on negative numbers

④ space puts - on negative numbers and space on others *fmt_signed.py*

```
default: |123|
default: |-321|
default: |14|
default: |-2|
default: |0|

   plus: |+123|
   plus: |-321|
   plus: |+14|
   plus: |-2|
   plus: |+0|

  minus: |123|
  minus: |-321|
  minus: |14|
  minus: |-2|
  minus: |0|

  space: | 123|
  space: |-321|
  space: | 14|
  space: |-2|
  space: | 0|
```

Parameter Attributes

- Specify elements or properties in template
- No need to repeat parameters
- Works with sequences, mappings, and objects

When specifying container variables as parameters, you can select elements in the format rather than in the parameter list. For sequences or dictionaries, index on the selector with []. For object attributes, access the attribute from the selector with . (period).

Example

fmt_attrib.py

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

from datetime import date

fruits = 'apple', 'banana', 'mango'
values = [5, 18, 27, 6]
dday = date(1944, 6, 6)
pythons = {'Idle': 'Eric', 'Cleese': 'John', 'Gilliam': 'Terry',
           'Chapman': 'Graham', 'Palin': 'Michael', 'Jones': 'Terry'}

print('{0[0]} {0[2]}'.format(fruits))  ①
print('{f[0]} {f[2]}'.format(f=fruits))  ②
print()
print('{0[0]} {0[2]}'.format(values))  ③
print()
print('{0[Palin]} {0[Cleese]}'.format(pythons))  ④
print('{names[Palin]} {names[Cleese]}'.format(names=pythons))  ⑤
print()
print('{0.month}-{0.day}-{0.year}'.format(dday))  ⑥
```

① select from tuple

② named parameter + select from tuple

③ Select from list

④ select from dict

⑤ named parameter + select from dict

⑥ select attributes from date

fmt_attrib.py

```
apple mango
apple mango

5 27

Michael John
Michael John

6-6-1944
```


Formatting Dates

- Special formats for dates
- Pull appropriate values from date/time objects

To format dates, use special date formats. These are placed, like all formatting codes, after a colon. For instance, {0:%B %d, %Y} will format a parameter (which must be a datetime.datetime or datetime.date) as "Month DD, YYYY".

Example

fmt_dates.py

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

from datetime import datetime

event = datetime(2016, 1, 2, 3, 4, 5)

print(event)  ①
print()

print("Date is {0:%m}/{0:%d}/{0:%y}".format(event))  ②
print("Date is {:%m/%d/%y}".format(event))  ③
print("Date is {:%A, %B %d, %Y}".format(event))  ④
```

① Default string version of date

② Use three placeholders for month, day, year

③ Format month, day, year with a single placeholder

④ Another single placeholder format

fmt_dates.py

```
2016-01-02 03:04:05

Date is 01/02/16
Date is 01/02/16
Date is Saturday, January 02, 2016
```

Table 24. Date Formats

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

1. When used with the strptime() method, the %f directive accepts from one to six digits and zero pads on the right. %f is an extension to the set of format characters in the C standard (but

implemented separately in datetime objects, and therefore always available).

2. When used with the strptime() method, the %p directive only affects the output hour field if the %I directive is used to parse the hour.
3. The range really is 0 to 61; according to the Posix standard this accounts for leap seconds and the (very rare) double leap seconds. The time module may produce and does accept leap seconds since it is based on the Posix standard, but the datetime module does not accept leap seconds instrptime() input nor will it produce them in strftime() output.
4. When used with the strptime() method, %U and %W are only used in calculations when the day of the week and the year are specified.
5. For example, if utcoffset() returns timedelta(hours=-3, minutes=-30), %z is replaced with the string *-0330*.

Run-time formatting

- Use parameters to specify alignment, precision, width, and type
- Use {} placeholders for runtime values for the above

To specify formatting values at runtime, use a {} placeholder for the value, and insert the desired value in the parameter list. These placeholders are numbered along with the normal placeholders.

Example

fmt_runtime.py

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

FIRST_NAME = 'Fred'
LAST_NAME = 'Flintstone'
AGE = 35

print("{0} {1}".format(FIRST_NAME, LAST_NAME))

WIDTH = 12
print("{0:{width}s} {1:{width}s}".format(  ①
    FIRST_NAME,
    LAST_NAME,
    width=WIDTH,
))

PAD = '-'
WIDTH = 20
ALIGNMENTS = ('<', '>', '^')

for alignment in ALIGNMENTS:
    print("{0:{pad}{align}{width}s} {1:{pad}{align}{width}s}".format(  ②
        FIRST_NAME,
        LAST_NAME,
        width=WIDTH,
        pad=PAD,
        align=alignment,
    ))
```

① value of WIDTH used in format spec

② values of PAD, WIDTH, ALIGNMENTS used in format spec

fmt_runtime.py

```
Fred Flintstone
Fred         Flintstone
Fred---------------- Flintstone----------
----------------Fred ----------Flintstone
--------Fred-------- -----Flintstone-----
```

Miscellaneous tips and tricks

- Adding commas to large numbers {n:,}
- Auto-converting parameters to strings (!s)
- Non-decimal prefixes

- Adding commas to large numbers {n:,}

You can add a comma to the format to add commas to numbers greater than 999.

Using a format type of !s will call str() on the parameter and force it to be a string.

Using a # (pound sign) will cause binary, octal, or hex output to be preceded by *0b*, *0o*, or *0x*. This is only valid with type codes b, o, and x.

Example

fmt_misc.py

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

'''Demonstrate misc formatting'''

big_number = 2303902390239

print("Big number: {:,d}".format(big_number))  ①
print()

value = 27

print("Binary: {:#010b}".format(value))  ②
print("Octal:  {:#010o}".format(value))  ③
print("Hex:    {:#010x}".format(value))  ④
print()
```


① Add commas for readability

② Binary format with leading 0b

③ Octal format with leading 0o

④ Hexadecimal format with leading 0x

fmt_misc.py

```
Big number: 2,303,902,390,239

Binary: 0b00011011
Octal:  0o00000033
Hex:    0x0000001b
```

Index