Advanced Python for JPMC

TTPS4804

John Strickler

Version 1.1, April 2023

Table of Contents

About Advanced Python for JPMC: TTPS4804 . . . 1
- Course Outline . . . 2
- Student files . . . 3
- Examples . . . 4
- Appendices . . . 5
- Classroom etiquette . . . 6

Chapter 1: Idiomatic Data Handling . . . 7
- Deep vs shallow copying . . . 8
- Using `ordereddict` . . . 11
- Default dictionary values . . . 12
- Counting with Counter . . . 15
- Named Tuples . . . 17
- Printing data structures . . . 20
- Zipped archives . . . 23
- Tar Archives . . . 25
- Archives the easy way . . . 27
- Serializing Data . . . 28

Chapter 2: Container Classes . . . 32
- Container classes . . . 33
- Builtin containers . . . 34
- Containers in the standard library . . . 35
- The array module . . . 39
- Emulating builtin types . . . 42
- Creating list-like containers . . . 48
- Creating dict-like containers . . . 51
- Free-form containers . . . 54

Chapter 3: Functional Tools . . . 56
- Higher-order functions . . . 57
- Lambda functions . . . 58
- The operator module . . . 61
- The functools module . . . 63
- map() . . . 64
- reduce() . . . 65
- Partial functions . . . 68
- Single dispatch . . . 71
- The itertools module . . . 74

Infinite iterators . . . 75
Extended iteration . . . 77
Grouping . . . 80
Combinatoric generators . . . 82
Chapter 4: Metaprogramming . . . 85
Metaprogramming . . . 86
globals() and locals() . . . 87
The inspect module . . . 90
Working with attributes . . . 94
Adding instance methods . . . 96
Callable classes . . . 99
Decorators . . . 101
Applying decorators . . . 103
Trivial Decorator . . . 105
Decorator functions . . . 106
Decorator Classes . . . 109
Decorator parameters . . . 113
Creating classes at runtime . . . 115
Monkey Patching . . . 118
Do you need a Metaclass? . . . 121
About metaclasses . . . 122
Mechanics of a metaclass . . . 123
Singleton with a metaclass . . . 127
Chapter 5: Generators and Other Iterables . . . 132
Iterables . . . 133
Unpacking function arguments . . . 134
Iterable unpacking . . . 137
Extended iterable unpacking . . . 140
What exactly is an iterable? . . . 143
List comprehensions . . . 144
Generators . . . 147
Generator Expressions . . . 148
Generator functions . . . 151
Coroutines . . . 153
Generator classes . . . 156
Chapter 6: PyParsing . . . 160
Text Parsing Tools . . . 161
About pyparsing . . . 162

Defining a Grammar . . . 163
A Simple Parser . . . 164
Parsing functions . . . 169
Accessing tokens . . . 170
Case Study: A URL decomposed . . . 173
The URL parser grammar . . . 174
Building a parser for the grammar . . . 175
Taking Action . . . 180
Chapter 7: Multiprogramming . . . 183
Multiprogramming . . . 184
What Are Threads? . . . 185
The Python Thread Manager . . . 186
The threading Module . . . 187
Threads for the impatient . . . 188
Creating a thread class . . . 190
Variable sharing . . . 192
Using queues . . . 195
Debugging threaded Programs . . . 198
The multiprocessing module . . . 200
Using pools . . . 203
Alternatives to `POOL.map()` . . . 208
Alternatives to threading and multiprocessing . . . 209
Chapter 8: Design Patterns . . . 211
Coupling and cohesion . . . 212
Interfaces . . . 213
What are design patterns? . . . 215
Why use design patterns? . . . 216
Types of patterns . . . 217
Singleton . . . 222
Module as Singleton . . . 223
Classic Singleton . . . 225
The Borg . . . 228
Strategy . . . 232
Decorators . . . 237
Using decorators . . . 239
Creating decorator functions . . . 241
Decorators with arguments . . . 243
Adapters . . . 246

Abstract Factory . . . 248
Builder . . . 249
Facade . . . 253
Flyweight . . . 258
Command . . . 259
Mediator . . . 262
Appendix A: Python Bibliography . . . 264
Index . . . 267

About Advanced Python for JPMC: TTPS4804

Course Outline

Half-day 1

Chapter 1 Idiomatic Data Handling
Chapter 2 Container Classes

Half-day 2

Chapter 3 Functional Tools
Chapter 4 Metaprogramming

Half-day 3

Chapter 5 Generators and Other Iterables
Chapter 6 PyParsing

Half-day 4

Chapter 7 Multiprogramming

Time permitting

Chapter 8 Design Patterns

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

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **py3jpmcadv**. See the setup guides for details.

What's in the files?

py3jpmcadv contains all files necessary for the class
py3jpmcadv/EXAMPLES/ contains the examples from the course manuals.
py3jpmcadv/ANSWERS/ contains sample answers to the labs.
py3jpmcadv/DATA/ contains data used in examples and answers
py3jpmcadv/SETUP/ contains any needed setup scripts (may be empty)

The following folders may be present in some classes:

py3jpmcadv/BIG_DATA/ contains large data files used in examples and answers
py3jpmcadv/NOTEBOOKS/ Jupyter notebooks for use in class
py3jpmcadv/TEMP/ initially empty; used by some examples for output files
py3jpmcadv/LOGS/ initially empty; used by some examples to write log files

NOTE	The student files do not contain Python itself. It will need to be installed separately. This may already have been done.

Examples

Most of the examples from the course manual are provided in EXAMPLES subdirectory.

It will look like this:

Example

cmd_line_args.py

```
import sys   # Import the sys module

print(sys.argv) # Print all parameters, including script itself

name = sys.argv[1]  # Get the first actual parameter
print("name is", name)
```

cmd_line_args.py apple mango 123

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

Appendices

Appendix A Python Bibliography

Classroom etiquette

Remote learning

- Mic off when you're not speaking. If multiple mics are on, it makes it difficult to hear
- The instructor doesn't know you need help unless you let them know via voice or chat.
- It's ok to ask for help a lot.
  - Ask questions. Ask questions. Ask questions.
  - ***INTERACT*** with the instructor and other students.
- Log off the remote S/W at the end of the day

In-person learning

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

Please turn off cell phone ringers and other noisemakers.

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

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

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

Chapter 1: Idiomatic Data Handling

Objectives

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

Deep vs shallow copying

- Normal assignments create aliases
- New objects are shallow copies
- Use `copy.deepcopy()`

Consider the following code:

```
colors = ['red', 'blue', 'green']
c1 = colors
```

The assignment to variable `c1` does not create a new object; it is another name that is *bound* to the same list object as the variable `colors`. Another way to say this is that `c1` is an alias for `colors`.

To create a *new* list, you can either use the `list()` constructor, or use a slice which contains all elements:

```
c2 = list(colors)
c3 = colors[::]
```

In both cases, `c2` and `c3` are each distinct objects.

However, the elements of `c2` and `c3` are not copied, but are still bound to the same objects as the elements of `colors`.

For example:

```
data1 = [ [1, 2, 3], [4, 5, 6] ]
d1 = list(data)
d1[0].append(99)
print(data1)
```

This will show that the first element of `data1` contains the value 99, because `data1[0]` and `d1[0]` are both bound to the same object. To do a "deep" (recursive) copy, use the `deepcopy()` from the `copy` module:

Example

deep_copy.py

```
import copy

data = [
    [1, 2, 3],
    [4, 5, 6],
]

d1 = data # Bind d1 to same object as data
d2 = list(data) # Make shallow copy of data and store in d2
d3 = copy.deepcopy(data) # Make deep copy of data and store in d3

d1.append("d1")  # Append to d1 (same as appending to data)
d1[0].append(50) # Append to first element of d1 (same first element as data)

d2.append("d2")   # Append to d2 (does not affect data)
d2[0].append(99)  # Append to first element of d2 (same first element as data and d1)

d3.append("d3")   # Append to d3 (does not affect data)
d3[0].append(500) # Append to first element of d3 (does not affect data, d1, or d2)

print("data:", data, id(data))
print("d1:", d1, id(d1))
print("d2:", d2, id(d2))
print("d3:", d3, id(d3))
print()

print("id(d1[0]):", id(d1[0]))
print("id(d2[0]):", id(d2[0]))
print("id(d3[0]):", id(d3[0]))
```

deep_copy.py

```
data: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 140367316495168
d1: [[1, 2, 3, 50, 99], [4, 5, 6], 'd1'] 140367316495168
d2: [[1, 2, 3, 50, 99], [4, 5, 6], 'd2'] 140367316495232
d3: [[1, 2, 3, 500], [4, 5, 6], 'd3'] 140367316495616

id(d1[0]): 140367316475520
id(d2[0]): 140367316475520
id(d3[0]): 140367316495808
```

Using `ordereddict`

- Remembers order items were added
- Good for exporting data
- Obsolete as of 3.6

Before version 3.6, the order of elements in a dictionary was undefined. This could create issues when converting dictionaries to **JSON**, **XML**, **YAML**, and other formats, where you want to create fields in a specific order.

To solve this problem, the `collections` module provides the `ordereddict` type. It's the same as a normal dictionary, but it uses the order in which elements were added when you iterate over `some_dict.items()`, `some_dict.keys()`, or `some_dict.values()`.

As of Python 3.6 (unofficially) and 3.7 (officially), normal dictionaries preserve the insertion order, so `ordereddict` is now obsolete.

Default dictionary values

- Use `collections.defaultdict`
- Specify function that provides default value
- Good for counting, datasets

Normally, when you use an invalid key with a dictionary, it raises a `KeyError`. The `defaultdict` class in the collections module allows you to provide a default value, so there will never be a KeyError.

You provide a function that returns the default value. This is necessary because if you provided, say, an empty list as the default value, *every* element would have the *same* list as its value. Using the `list()` function guarantees a new, distinct, list every time. A lambda function can be handy for this, but is not required.

Example

defaultdict_ex.py

```
from collections import defaultdict

dd = defaultdict(lambda: 0)  # create default dict with function that returns 0

dd['spam'] = 10  # assign some values to the dict
dd['eggs'] = 22

print(dd['spam'])  # print values
print(dd['eggs'])
print(dd['foo'])  # missing key 'foo' invokes function and returns 0

print('-' * 60)

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

fruit_info = defaultdict(list)

for fruit in fruits:
    first_letter = fruit[0]
    fruit_info[first_letter].append(fruit)

for letter, fruits in sorted(fruit_info.items()):
    print(letter, fruits)
```

defaultdict_ex.py

```
10
22
0
------------------------------------------------------------
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
```

```
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
```

Counting with Counter

- Use `collections.Counter`
- Values default to 0
- Just increment

For ease in counting, the `collections` module provides a `Counter` object. This is a dictionary whose default value is zero, and that has some extra methods.

You can just increment the value for any key, whether it's been seen before or not. Even simpler, you can pass in any iterable when creating the `Counter` object, and it will count the items in the iterable.

```
from collections import Counter
c = Counter()
c['spam'] += 1
c['ham'] += 1
c['spam'] += 1
c['toast'] += 1
c['spam'] += 1
```

Whether you increment keys individually or pass in an iterable of keys, all the keys must be hashable, as with normal dictionary keys.

Example

count_with_counter.py

```
from collections import Counter

with open("../DATA/breakfast.txt") as breakfast_in:
    foods = [line.rstrip() for line in breakfast_in]  # create list of foods with EOL
removed from line

counts = Counter(foods)  # initialize Counter object with list of foods

for item, count in counts.items():  # iterate over results
    print(item, count)
```

count_with_counter.py

```
spam 34
eggs 3
sausage 4
bacon 6
pancakes 2
dosas 4
crumpets 1
```

Named Tuples

- From collections module
- Like tuple, but each element has a name
- tuple.name in addition to tuple[n]

A `namedtuple` is a tuple where each element has a name, and can be accessed via the name in addition to the normal access by index. Thus, if `p` were a named tuple representing a point, with fields `x` and `y` you could say `p.x` and `p.y`, or you could say `p[0]` and `p[1]`.

A named tuple is created with the `namedtuple` class in the `collections` module. You are essentially creating a new data type. Pass the name of the tuple followed by a string containing the individual field names separated by spaces. `namedtuple()` returns a new class which with you can create instances of the tuple.

You can convert a named tuple instance into a dictionary with the `_asdict()` method.

A named tuple is the closest thing Python has to a C struct.

Example

named_tuples.py

```
from collections import namedtuple
from pprint import pprint

Knight = namedtuple('Knight', 'name title color quest comment') # create named tuple
class with specified fields (could also provide fieldnames as iterable)

k = Knight('Bob', 'Sir', 'green', 'whirled peas', 'Who am i?') # create named tuple
instance (must specify all fields)

print(k.title, k.name) # can access fields by name...
print(k[1], k[0]) # ...or index
print()

knights = [] # initialize list for knight data
with open('../DATA/knights.txt') as knights_in:
    for raw_line in knights_in:
        # strip \n then split line into fields
        name, title, color, quest, comment = raw_line.rstrip().split(':')
        # create instance of Knight namedtuple
        knight = Knight(name, title, color, quest, comment)
        # add tuple to list
        knights.append(knight)

for knight in knights: # iterate over list of knights
    print(f'{knight.title} {knight.name}: {knight.color}')
print()
pprint(knights)
```

named_tuples.py

```
Sir Bob
Sir Bob

King Arthur: blue
Sir Galahad: red
Sir Lancelot: blue
Sir Robin: yellow
Sir Bedevere: red, no blue!
Sir Gawain: blue

[Knight(name='Arthur', title='King', color='blue', quest='The Grail', comment='King of
the Britons'),
```

```
 Knight(name='Galahad', title='Sir', color='red', quest='The Grail', comment="'I could
handle some more peril'"),
 Knight(name='Lancelot', title='Sir', color='blue', quest='The Grail', comment='"It\'s
too perilous!"'),
 Knight(name='Robin', title='Sir', color='yellow', quest='Not Sure', comment='He boldly
ran away'),
 Knight(name='Bedevere', title='Sir', color='red, no blue!', quest='The Grail',
comment='AARRRRRRRGGGGHH'),
 Knight(name='Gawain', title='Sir', color='blue', quest='The Grail', comment='none')]
```

Printing data structures

- Default representation of data structures is ugly
- `pprint` makes structures more readable
- Use `pprint.pprint()`
- Useful for debugging

When debugging data structures, the print command is not so helpful. A complex data structure is just printed out all jammed together, one element after another, and is hard to read.

The `pprint` (pretty print) function in the `pprint` module will analyze a structure and print it out with indenting, to make it much easier to read.

You can customize the output with some named parameters:

- **indent** (default 1) specifies how many spaces to indent nested structures
- **width** (default 80) constrains the width of the output
- **depth** (default unlimited) says how many levels to print – levels beyond depth are shown with '...'.

Example

pretty_printing.py

```
from pprint import pprint

struct = {  # nested data structure
    'part1': [
        ['a', 'b', 'c'], ['d', 'e', 'f']
    ],
    'part2': {
        'red': 55,
        'blue': [8, 98, -3],
        'purple': ['Chicago', 'New York', 'L.A.'],
    },
    'part3': ['g', 'h', 'i'],
}

print('Without pprint:')
print(struct)  # print normally
print()

print('With pprint:')
pprint(struct)  # pretty-print
print()

print('With pprint (depth=2):')
pprint(struct, depth=2)  # only print top two levels of structure
print()
```

pretty_printing.py

```
Without pprint:
{'part1': [['a', 'b', 'c'], ['d', 'e', 'f']], 'part2': {'red': 55, 'blue': [8, 98, -3],
'purple': ['Chicago', 'New York', 'L.A.']}, 'part3': ['g', 'h', 'i']}

With pprint:
{'part1': [['a', 'b', 'c'], ['d', 'e', 'f']],
 'part2': {'blue': [8, 98, -3],
           'purple': ['Chicago', 'New York', 'L.A.'],
           'red': 55},
 'part3': ['g', 'h', 'i']}

With pprint (depth=2):
{'part1': [[...], [...]],
 'part2': {'blue': [...], 'purple': [...], 'red': 55},
```

```
 'part3': ['g', 'h', 'i']}
```

Zipped archives

- import zipfile for zipped files
- Get a list of files
- Extract files

The `zipfile` module allows you to read and write to zipped archives. In either case you first create a zipfile object; specifying a mode of "w" if you want to create an archive, and a mode of "r" (or nothing) if you want to read an existing zip file.

Modules for gzipped and bzipped files: `gzip`, `bz2`

Example

zipfile_ex.py

```
from zipfile import ZipFile, ZIP_DEFLATED
import os.path

# reading & extracting from zip file
zip_in = ZipFile("../DATA/textfiles.zip")  # Open zip file for reading
print(zip_in.namelist())  # Print list of members in zip file
tyger_text = zip_in.read('tyger.txt').decode()  # Read (raw binary) data from member and
convert from bytes to string
print(tyger_text[:100], '\n')
zip_in.extract('parrot.txt')  # Extract member

# creating a zip file
file_names = ["parrot.txt", "tyger.txt", "knights.txt", "alice.txt", "poe_sonnet.txt",
"spam.txt"]
zip_out = ZipFile("example.zip", mode="w", compression=ZIP_DEFLATED)  # Create new zip
file
for file_name in file_names:
    file_path = os.path.join("../DATA", file_name)
    print("adding {}".format(file_path))
    zip_out.write(file_path, file_name)  # Add member to zip file
```

zipfile_ex.py

```
['fruit.txt', 'parrot.txt', 'tyger.txt', 'spam.txt']
           The Tyger

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand o

adding ../DATA/parrot.txt
adding ../DATA/tyger.txt
adding ../DATA/knights.txt
adding ../DATA/alice.txt
adding ../DATA/poe_sonnet.txt
adding ../DATA/spam.txt
```

Tar Archives

- Use the `tarfile` module
- Check for existence of tar archives
- Read and extract members from tar archives

To work with a ***tar*** archive, use the `tarfile` module. It can analyze a file to see whether it's a valid tar file, extract files from the archive, add files to the archive, and other tar chores.

The `is_tarfile()` function will check a tar file and return True if it is a valid tar file. This will work even if it is compressed.

For other actions, use the `open()` function to create a `TarFile` object. From this object you can list, add, or extract members, depending on how the tar file was opened.

A `TarFile` object is an iterable of `TarInfo` objects, which contain the details about each member. Use `getmembers()` to get a list of members as `TarInfo` objects. Use `extract()` to extract a file to disk. The first argument to extract() is the member name; the named parameter path specifies the destination directory (default '.').

Use extractall() to extract all members to the current directory, or a specified path. A list of members may be specified; it must be a subset of the list returned by getmembers().

To create a tar file, use tarfile.open() with a mode of w for an uncompressed archive. Use modes w:gz or w:bz2 to the mode for a compressed archive. Use the add() method to add a file.

Example

tarfile_ex.py

```
import tarfile
import os

for tar_file in ('pres.tar', 'NOT_A.tar', 'potus.tar.gz'):  # iterate over sample files
    filename = os.path.join('../DATA', tar_file)
    is_valid = tarfile.is_tarfile(filename)  # check to see if file is a tarfile
    text = 'IS' if is_valid else 'IS NOT'
    print("{} {} a tarfile".format(filename, text))
print()

with tarfile.open('../DATA/pres.tar') as tarfile_in:  # open tar file
    for member in tarfile_in:  # iterate over members
        print(member.name, member.size)  # access member data
    print()

with tarfile.open('../DATA/pres.tar') as tarfile_in:
    tarfile_in.extract('presidents.txt', path='../TEMP')  # extract member to local file

with tarfile.open('../DATA/potus.tar.gz') as tarfile_in:
    tarfile_in.extract('presidents.csv', path='../TEMP')  # extract member to local file

with tarfile.open('../TEMP/text_files.tar', 'w') as tarfile_out:  # open new tar archive
for writing
    tarfile_out.add('../DATA/parrot.txt')  # add member
    tarfile_out.add('../DATA/alice.txt')  # add member

with tarfile.open('../TEMP/more_text_files.tar.gz', 'w:gz') as tar_gz_write:  # open new
tar archive for writing; archive is compressed with gzip
    tar_gz_write.add('../DATA/parrot.txt')  # add member
    tar_gz_write.add('../DATA/alice.txt')  # add member
```

tarfile_ex.py

```
../DATA/pres.tar IS a tarfile
../DATA/NOT_A.tar IS NOT a tarfile
../DATA/potus.tar.gz IS a tarfile

presidents.csv 2583
presidents.txt 4509
```

Archives the easy way

- Use shutil.make_archive()
- Specify base name and format
- Can specify root dir (default: base name)

The **make_archive()** function in **shutil** makes it easy to create zip files and tar archives. To use it, specify the base name of the archive (the "spam" of "spam.zip") and the format. By default, it assumes the folder is the same name as the base name. The optional third argument is the folder to archive.

```
import shutil

shutil.make_archive('DATA', 'tgz')
```

Table 1. make_archive formats

Format code	Description	Extension
zip	Zip archive	zip
tar	Uncompressed `tar` archive	tar
gztar	gzipped tar archive	.tar.gz
bztar	bzipped2 tar archive	.tar.bz2
xztar	xzipped tar archive	.tar.xz

Serializing Data

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Serializing data means taking a data structure and transforming it so it can be written to a file or other destination, and later read back into the same data structure.

Python uses the `pickle` module for data serialization.

To create pickled data, use either `pickle.dump()` or `pickle.dumps()`. Both functions take a data structure as the first argument. dumps() returns the pickled data as a string. dump () writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use `pickle.load()`, which takes a file-like object that has been opened for writing in binary mode. This will return the original data structure that had been pickled.

Example

pickling.py

```
import pickle
from pprint import pprint

# some data structures
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'
]

values = [
    3/7, 1/9, 14.5
]

data = [  # list of data structures
    colors,
    airports,
    values,
]

with open('../TEMP/pickled_data.pic', 'wb') as pic_out:  # open pickle file for writing
in binary mode
    pickle.dump(data, pic_out)  # serialize data structures to pickle file

with open('../TEMP/pickled_data.pic', 'rb') as pic_in:  # open pickle file for reading in
binary mode
    pickled_data = pickle.load(pic_in)  # de-serialize pickle file back into data
structures

pprint(pickled_data)  # 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'},
 [0.42857142857142855, 0.1111111111111111, 14.5]]
```

Chapter 1 Exercises

Exercise 1-1 (count_ext.py)

Write a script which will count the number of distinct extensions in a file tree. It should take the initial directory as a command line argument, and then display the total number of files with each distinct file extension that it finds. Files with no extension should be skipped. Use a Counter object to do the counting.

HINT: Use `os.walk` to navigate the file tree.

Exercise 1-2 (pres_tuple.py, save_potus_info.py, read_potus_info.py)

Part A

Create a module with a namedtuple with the name `President`, with fields `term`, `firstname`, `lastname`, `birthplace`, `birthstate`, `party`

Part B

Using the namedtuple from {labnum}a, read the data from `presidents.csv` into a list of `President` objects.

Using the `pickle` module, save the list to a file named **potus.pic**.

Part C

Write a script to open **potus.pic**, and restore the data back into a list.

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

Exercise 1-3 (make_zip.py)

Write a script which creates a zip file containing **save_potus_info.py**, **read_potus_info.py**, and **potus.pic**.

Chapter 2: Container Classes

Objectives

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

Container classes

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

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

Builtin containers

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

Several container types are builtin.

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

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

Example

builtin_containers.py

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

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

builtin_containers.py

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

Containers in the standard library

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

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

These types are:

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

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

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

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

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

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

NOTE	Starting with Python 3.6, all dictionaries preserve order, making OrderedDicts obsolete. (Of course legacy code may still use them, and some existing modules in the stdlib use or return them).

Example

stdlib_containers.py

```
from collections import Counter, defaultdict, deque, namedtuple, OrderedDict

# Counter
with open('../DATA/words.txt') as words_in:
    all_words = [line[0] for line in words_in]
    word_counter = Counter(all_words)  # Count list of words by passing iterable of words
to Counter instance


print(word_counter.most_common(10))  # Counter.most_common() return n most common
occurrences
print('-' * 60)

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

fruit_by_first = defaultdict(list)  # Create default dict whose default value is a new
list object

for fruit in fruits:
    fruit_by_first[fruit[0]].append(fruit)  # Append fruit to dictionary element whose
key is the first letter and whose value is a list

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

# deque
d = deque()  # Create an empty deque
for c in 'abcdef':
    d.append(c)   # Append to the deque
print(d)
for c in 'ghijkl':
    d.appendleft(c)  # Prepend to the deque
print(d)
d.extend('mno')  # Extend the deque at the end one letter at a time
print(d)
d.extendleft('pqr')  # Extend the deque at the beginning one letter at a time
print(d)
print(d[9])
```

```
print(d.pop(), d.popleft())  # Pop from end, beginning
print(d)
print('-' * 60)


# namedtuple
President = namedtuple('President', 'first_name, last_name, party')  # Create named tuple
with specified fields
p = President('Theodore', 'Roosevelt', 'Republican')  # Create instance of named tuple
print(p, len(p))
print(p[0], p[1], p[-1])
print(p.first_name, p.last_name, p.party)  # Access tuple fields by name

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

stdlib_containers.py

```
[('s', 19030), ('c', 16277), ('p', 14600), ('a', 10485), ('r', 10199), ('d', 10189),
('m', 9689), ('b', 9325), ('t', 8782), ('e', 7086)]
------------------------------------------------------------
a ['apricot', 'apple']
b ['banana', 'blueberry']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
g ['guava', 'grape']
k ['kiwi']
l ['lemon', 'lime', 'lychee']
o ['orange']
p ['pomegranate', 'papaya', 'pear', 'persimmon', 'peach']
t ['tamarind']
w ['watermelon']
------------------------------------------------------------
deque(['a', 'b', 'c', 'd', 'e', 'f'])
deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f'])
deque(['l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o'])
deque(['r', 'q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm',
'n', 'o'])
a
o r
deque(['q', 'p', 'l', 'k', 'j', 'i', 'h', 'g', 'a', 'b', 'c', 'd', 'e', 'f', 'm', 'n'])
------------------------------------------------------------
President(first_name='Theodore', last_name='Roosevelt', party='Republican') 3
Theodore Roosevelt Republican
Theodore Roosevelt Republican
President(first_name='Abraham', last_name='Lincoln', party='Republican')
Abraham Lincoln
------------------------------------------------------------
```

The array module

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

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

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

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

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

Example

array_examples.py

```
from sys import getsizeof
from array import array
from random import randint

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

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

for datatype in 'i', 'h', 'L', 'Q', 'd':
    data_array = array(datatype, values)  # Create array object from values with various
datatypes
    print(f'Size of {datatype} array: {getsizeof(data_array)}  Contents:',
          data_array[:5], '...')  # Print size of array (will vary based on datatype)
    print()
```

array_examples.py

```
Size of integer list: 8856

Size of i array: 4064  Contents: array('i', [8535, 23787, 14424, 594, 7833]) ...

Size of h array: 2064  Contents: array('h', [8535, 23787, 14424, 594, 7833]) ...

Size of L array: 8064  Contents: array('L', [8535, 23787, 14424, 594, 7833]) ...

Size of Q array: 8064  Contents: array('Q', [8535, 23787, 14424, 594, 7833]) ...

Size of d array: 8064  Contents: array('d', [8535.0, 23787.0, 14424.0, 594.0, 7833.0])
...
```

Table 2. array object type codes

Format	C Type	Python Type	Standard size	Notes
b	signed char	integer	1	(1),(3)
B	unsigned char	integer	1	(3)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2),(3)
Q	unsigned long long	integer	8	(2), (3)
n	ssize_t	integer		(4)
N	size_t	integer		(4)
f	float	float	4	(5)
d	double	float	8	(5)

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

Emulating builtin types

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

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

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

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

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

Example

container_abc.py

```
from collections.abc import Sized, Iterator  # Abstract base classes, used similarly to interfaces in Java or C#


class BadContainer(Sized):  # This class may not be instantiated without defining `len()`
    pass


class GoodContainer(Sized):
    def __len__(self):   # This class is fine, since `Sized` requires `len()` to be implemented
        return 42


try:
    bad = BadContainer()  # Instantiating `BadContainer` raises an error.
except TypeError as err:
    print(err)
else:
    print(bad)

print()

try:
    good = GoodContainer()  # Instantiating `GoodContainer` is fine
except TypeError as err:
    print(err)
else:
    print(good)
    print(len(good))  # Builtin function `len()` works with all objects that inherit from `Sized` (due to implementation of `len())`

print()


class MyIterator(Iterator):  # ABC `Iterator` provides abstract method `next`
    data = 'a', 'b', 'c'
    index = 0

    def __next__(self):  # Must be implemented for Iterators
        if self.index >= len(self.data):
            raise StopIteration
        else:
```

```
            return_val = self.data[self.index]
            self.index += 1
            return return_val


m = MyIterator()  # Create instance of `MyIterator`
for i in m:  # Iterate over the iterator instance
    print(i)
print()

print(hasattr(m, '__iter__'))  # Check to see if `m` is iterable
```

container_abc.py

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

<__main__.GoodContainer object at 0x7fc6c80cab80>
42

a
b
c

True
```

Table 3. Special Methods and Variables

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

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

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

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

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

Method or Variables	Description
`__reversed__(self)`	Called by reversed() builtin
`__contains__(self, object)`	Implements in operator

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

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

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

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

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

Method or Variables	Description
`__oct__(self)` `__hex__(self)`	Implement oct() and hex() builtins
`__index__(self)`	Implement operator.index()
`__coerce__(self, other)`	Implement "mixed-mode" numeric arithmetic.

Creating list-like containers

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

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

Example

multiindexlist.py

```
class MultiIndexList(list):  # Define new class that inherits from list

    def __getitem__(self, item):  # Redefine __getitem__ which implements []
        if isinstance(item, tuple):  # Check to see if index is tuple
            if len(item) == 0:
                raise ValueError("Tuple must be non-empty")
            else:
                tmp_list = []
                for index in item:
                    tmp_list.append(
                        super().__getitem__(index)  # Call list.__getitem__() for each
index in tuple
                    )
                return tmp_list
        else:
            return super().__getitem__(item)  # Call the normal __getitem__()


if __name__ == '__main__':
    m = MultiIndexList(
        'banana peach nectarine fig kiwi lemon lime'.split()
    )  # Initialize a MultiIndexList
    m.append('apple')  # Add an element (works like normal list)
    m.append('mango')
    print(m)

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

multiindexlist.py

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

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

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

Creating dict-like containers

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

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

Example

stringkeydict.py

```
class StringKeyDict(dict):  # Create class that inherits from dict
    def __setitem__(self, key, value):  # Overwrite how values are stored in the dict via
_DICT_[_KEY_] = _VALUE_
        if isinstance(key, str):   # Make sure key is a string
            super().__setitem__(key, value)  # Use dict's setitem to set value if it is
not a key
        else:
            raise TypeError("Keys must be strings not {}s".format(  # Raise error if non-
string key is used
                type(key).__name__
            ))


if __name__ == '__main__':
    d = StringKeyDict(a=10, b=20)   # Create and initialize StringKeyDict instance
    for k, v in [('c', 30), ('d', 40), (1, 50), (('a', 1), 60), (5.6, 201)]:
        try:
            print("Setting {} to {}".format(k, v), end=' ')
            d[k] = v   # Try to add various key/value pairs
        except TypeError as err:
            print(err)  # Error raised on non-string key
        else:
            print('SUCCESS')

    print()
    print(d)
```

stringkeydict.py

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

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

Free-form containers

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

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

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

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

Chapter 2 Exercises

Exercise 2-1 (maxlist.py)

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

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

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

Exercise 2-2 (strdict.py)

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

Chapter 3: Functional Tools

Objectives

- Conceptualize higher-order functions
- Learn what's in functools and itertools
- Transform data with map/reduce
- Chain multiple iterables together
- Implement function overloading with single dispatch
- Use iterative tools to work with iterators
- Chain multiple iterators together
- Compute combinations and permutations
- Zip multiple iterables with default values

Higher-order functions

- Functions that operate on (or return) functions

Higher-order functions operate on or return functions. The most traditional higher-order functions in other languages are `map()` and `reduce()`. These are used for functional programming, which prevents side effects (modifying data outside the function).

Some languages make it difficult or impossible to pass functions into other functions, but in Python, since functions are first-class objects, it is easy.

Higher-order functions may be nested, to have multiple transformations on data.

Lambda functions

- Inline anonymous functions
- Contain only parameters and return value
- Useful as predicates in higher-order functions

A **lambda** function is an inline anonymous function declaration. It evaluates as a function object, in the same way that `def function(⋯):` ... does.

A lambda declaration has only parameters and the return value. Blocks are not allowed, nor is the **return** keyword.

Lambdas are useful as predicates (callbacks) in higher-order functions.

Example

higher_order_functions.py

```
#

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


def process_list(alist, func):  # Define a function that accepts a list and a passed-in
function (AKA callback)
    new_list = []
    for item in alist:
        new_list.append(func(item))  # Call the callback function on one item of the
passed-in list
    return new_list


f1 = process_list(fruits, str.upper)  # Call process_list() with str.upper as the
callback
print(f1, "\n")

f2 = process_list(fruits, lambda s: s[0].upper())  # Call process_list() with a lambda
function as the callback
print(f2, "\n")

f3 = process_list(fruits, len)  # Call process_list() with the builtin function len() as
the callback()
print(f3, "\n")

total_length = sum(process_list(fruits, len))  # Pass the result of process_list() to the
builtin function sum() to sum all the values in the returned list

print(total_length, "\n")
```

higher_order_functions.py

```
['POMEGRANATE', 'CHERRY', 'APRICOT', 'DATE', 'APPLE', 'LEMON', 'KIWI', 'ORANGE', 'LIME',
'WATERMELON', 'GUAVA', 'PAPAYA', 'FIG', 'PEAR', 'BANANA', 'TAMARIND', 'PERSIMMON',
'ELDERBERRY', 'PEACH', 'BLUEBERRY', 'LYCHEE', 'GRAPE']

['P', 'C', 'A', 'D', 'A', 'L', 'K', 'O', 'L', 'W', 'G', 'P', 'F', 'P', 'B', 'T', 'P',
'E', 'P', 'B', 'L', 'G']

[11, 6, 7, 4, 5, 5, 4, 6, 4, 10, 5, 6, 3, 4, 6, 8, 9, 10, 5, 9, 6, 5]

138
```

The operator module

- Provides operators as functions
- Use rather than simple lambdas

The `operator` module provides functional versions of Python's standard operators. This saves the trouble of creating trivial lambda functions.

Instead of

```
lambda x, y: x + y
```

You can just use

```
operator.add
```

Both of these functions take two operators and add them, return the result.

Example

operator_module.py

```
from operator import add


a = 10
b = 15

print("a + b: {}".format(a + b))  # Add with add operator
print("add(a, b): {}".format(add(a, b)))  # Add with add function
```

operator_module.py

```
a + b: 25
add(a, b): 25
```

The functools module

- Included in standard library
- Supports higher-order programming
- Many standard higher-order functions

The `functools` module provides tools for higher-ordering programming, which means functions that operate on, or return functions (some do both). Functional programming avoids explicit loops.

`map()` and `reduce()` are two functions that are the basis of many functional algorithms. `map()` is a builtin function. `reduce()` *was* builtin in Python 2, but in Python 3 must be imported from `functools`.

NOTE Most of these functional algorithms can also be implemented with list comprehensions or generator expressions, which many people find easier to both read and write.

map()

- Applies function to every element of iterable
- Syntax
  - `map(function, iterable)`

map() returns a virtual list (i.e., a generator) created by applying a function to every element of an iterable.

`map(func, list)` returns an iterator like `[func(list[0]), func(list[1]), func(list[2], ⋯)]`

The first argument to `map()` is a function that takes one argument. Each element of the iterable is passed to the function, and the return value is added to the result generator.

Example

using_map.py

```
#

strings = ['wombat', 'koala', 'kookaburra', 'blue-ringed octopus']

result = [s.upper() for s in strings]  # Using a list comprehension, which is usually
simpler than map()
print(result)

result = list(map(str.upper, strings))  # Using map to copy list to upper case
print(result)

result = list(map(len, strings))  # Using map to get list of string lengths
print(result)
```

using_map.py

```
['WOMBAT', 'KOALA', 'KOOKABURRA', 'BLUE-RINGED OCTOPUS']
['WOMBAT', 'KOALA', 'KOOKABURRA', 'BLUE-RINGED OCTOPUS']
[6, 5, 10, 19]
```

reduce()

- Imported from `functools`
- Applies function to every element plus previous result

`reduce()` returns the value created by applying a function to every element of a list *and* the previous function result. `reduce()` must be imported from the `functools` module.

`reduce(func, list)` returns `func(list[n], func(list[n-1], func(list[2]⋯func(list[0]))))`

A third argument to `reduce()` provides the initial value. By default, the initial value is the first element of the list.

Other functions such as `sum()` or `str.join()` can be defined in terms of `map()` or `reduce()`.

The **mapreduce** approach to massively parallel processing, such as Hadoop, was inspired by map() and reduce().

Example

using_reduce.py

```
#
from operator import add, mul
from functools import reduce

values = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# sum()
result = reduce(add, values) # add values in list (initial value defaults to 0)
print("result is", result)

# sum() + 1000
result = reduce(add, values, 1000)  # add values in list (initial value is 1000)
print("result is", result)

# product
result = reduce(mul, values)  # multiply all values together (initial value is 1,
otherwise product would be 0)
print("result is", result)

strings = ['fi', 'fi', 'fo', 'fum']

# join
result = reduce(add, strings, "") # concatenate strings (initial value is empty string;
each string in iterable added to it)
print("result is", result)

# join + upper case
result = reduce(add, map(str.upper, strings), "")  # same, but make strings upper case
print("result is", result)
```

using_reduce.py

```
result is 550
result is 1550
result is 36288000000000000
result is fififofum
result is FIFIFOFUM
```

Partial functions

- Some arguments filled in
- Create desired signature

Partial functions are wrappers that have some arguments already filled in for the "real" function. This is especially useful when creating callback functions. It is also nice for creating customized functions that rely on functions from the standard library.

While you can create partial functions by hand, using closures, the `partial()` function (from the `functools` module) simplifies creating such a function.

The arguments to `partial()` are the function and one or more arguments. It returns a new function object, which will call the specified function and pass in the provided argument(s).

Example

partial_functions.py

```
#
import re

from functools import partial

count_by = partial(range, 0, 25)  # create partial function that "preloads" range() with
arguments 0 and 25

print((list(count_by(1))))  # call partial function with parameter, 0 and 25
automatically passed in
print((list(count_by(3))))  # call partial function with parameter, 0 and 25
automatically passed in
print((list(count_by(5))))  # call partial function with parameter, 0 and 25
automatically passed in
print()

has_a_number = partial(re.search, r'\d+')  # create partial function that embeds pattern
in re.search()

strings = [
    'abc', '123', 'abc123', 'turn it up to 11', 'blah blah'
]

for s in strings:
    print("{}:".format(s), end=' ')
    if has_a_number(s): # call re.search() with specified pattern
        print("YES")
    else:
        print("NO")
```

partial_functions.py

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24]
[0, 3, 6, 9, 12, 15, 18, 21, 24]
[0, 5, 10, 15, 20]

abc: NO
123: YES
abc123: YES
turn it up to 11: YES
blah blah: NO
```

Single dispatch

- Provides generic functions
- Emulates function/method overloading
- Functions registered by signature

The `singledispatch` class provides *generic functions*. A generic function is defined once with the `singledispatch()` decorator, then other functions may be registered to it.

To register a function, decorate it with `@original_function.register(type)`. Then, when the original function is passed an argument of type type as its first parameter, it will call the registered function instead. If no registered functions are found for the type, it will call the original function.

This can of course, be done manually by checking argument types, then calling other methods, but using `singledispatch` is much less cumbersome.

To check which function would be chosen for a given type, use the `dispatch()` method of the original function. To get a list of all registered functions, use the registry attribute.

The `singledispatch` module was added to the standard library beginning with Python 3.4.

Because it only works for the first parameter, `singledispatch` was not in the past useful for class methods, but starting with Python 3.8, the `functools.singledispatchmethod` class will provide single dispatch for methods.

Example

single_dispatch.py

```
from singledispatch import singledispatch
from io import TextIOWrapper

@singledispatch
def xopen(source, mode="r"):  # define generic function that is actually called
    source_type = type(source).__name__
    raise TypeError("Invalid arg: must be file, str, or bytes, not {}
".format(source_type))

@xopen.register(TextIOWrapper)  # define handler for TextIOWrapper type (normal text
files)
def _(fileobj):
    return fileobj

@xopen.register(str)  # define handler for string type
def _(str, mode="r"):
    return open(str, mode)


@xopen.register(bytes)  # define handler for bytes type
def _(bytes, mode="r"):
    return open(bytes.decode(), mode)


mary_in = open('../DATA/mary.txt')  # open a file and get a file object (type
TextIOWrapper)

for x in mary_in, '../DATA/mary.txt', b'../DATA/mary.txt', 52, ['a', 5]:
    try:
        file_in = xopen(x)  # call single dispatch function -- correct handler will
automatically be called
        result = file_in.read()
        print("Length: {}".format(len(result)))
    except TypeError as err:
        print('ERROR:', err)

print('-' * 60)
print(xopen.dispatch(str), "\n")  # show handler function for str

for arg_type, func in xopen.registry.items():
    print("{:30s} {}".format(str(arg_type), func))  # show functions for each registered
type
```

single_dispatch.py

```
Length: 108
Length: 108
Length: 108
ERROR: Invalid arg: must be file, str, or bytes, not int
ERROR: Invalid arg: must be file, str, or bytes, not list
------------------------------------------------------------
<function _ at 0x7f8678026ca0>

<class 'object'>               <function xopen at 0x7f86a8088040>
<class '_io.TextIOWrapper'>    <function _ at 0x7f8678026c10>
<class 'str'>                  <function _ at 0x7f8678026ca0>
<class 'bytes'>                <function _ at 0x7f8678026d30>
```

The itertools module

- Tools to help with iteration
- Provide many types of iterators

The `itertools` module provides many different iterators. Some of the tools work on existing iterators, while others create them from scratch.

These tools work well with functions from the operator module, as well as interacting with the functional tools described earlier.

Many of the constructors in `itertools` were inspired by Haskell, SML, and APL.

See **iterable_recipes.py** in the EXAMPLES folder for a group of utility functions that use many of the routines in this chapter.

Infinite iterators

- Iterate infinitely, or specified number of times
- Cycle over or repeat values

Infinite iterators return iterators that will iterate a specified number of times, or infinitely. These iterators are similar to generators; they do not keep all the values in memory.

`islice()` selects a slice of an iterator. You can specify an iterable and up to 3 slice arguments – start, stop, and increment, similar to the the slice arguments of a list. If just stop is provided, it stops the iterator after than many values.

`count()` is similar to `range()`. It provides a sequence of numbers with a specified increment. The big difference is that there is no end condition, so it will increment forever. You will need to to test the values and stop, or use islice().

`cycle()` loops over an iterable repeatedly, going back to the beginning each time the end is reached.

`repeat()` repeats a value infinitely, or a specified number of times.

Example

infinite_iterators.py

```
#
from itertools import islice, count, cycle, repeat

for i in count(0, 10):  # count by tens starting at 0 forever
    if i > 50:
        break  # without a check, will never stop
    print(i, end=' ')
print("\n")

for i in islice(count(0, 10), 6):  # saner, using islice to get just the first 6 results
    print(i, end=' ')
print("\n")

giant = ['fee', 'fi', 'fo', 'fum']

for i in islice(cycle(giant), 10):  # cycle over values in list forever (use islice to
stop)
    print(i, end=' ')
print("\n")

for i in repeat('tick', 10):  # repeat value 10 times (default is repeat forever)
    print(i, end=' ')
print("\n")
```

infinite_iterators.py

```
0 10 20 30 40 50

0 10 20 30 40 50

fee fi fo fum fee fi fo fum fee fi

tick tick tick tick tick tick tick tick tick tick
```

Extended iteration

- Extended iteration over multiple objects
- Truncated iteration over a list

Another group of iterator functions provides extended iteration.

`chain()` takes two or more iterables, and treats them as a single iterable.

To chain the elements of a single iterable together, use `chain.from_iterable()`.

`dropwhile()` skips leading elements of an iterable until some condition is reached. `takewhile()` stops iterating when some condition is reached.

Example

extended_iteration.py

```
#
from itertools import chain, takewhile, dropwhile

spam = ['alpha', 'beta', 'gamma']
ham = ['delta', 'epsilon', 'zeta']

for letter in chain(spam, ham):  # treat spam and ham as a single iterable
    print(letter, end=' ')
print("\n")

eggs = [spam, ham]

for letter in chain.from_iterable(eggs):  # treat all elements of eggs as a single
iterable
    print(letter, end=' ')
print("\n")

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

for fruit in takewhile(lambda f: len(f) > 4, fruits):  # iterate over elements of fruits
as long as length of current item > 4
    print(fruit, end=' ')
print("\n")

for fruit in takewhile(lambda f: f[0] != 'k', fruits):  # iterate over elements of
fruits as long as fruit does not start with 'k'
    print(fruit, end=' ')
print("\n")

values = [5, 18, 22, 31, 44, 57, 59, 61, 66, 70, 72, 78, 90, 99]

for value in dropwhile(lambda f: f < 50, values):  # skip over elements of values as long
as value is < 50, then iterate over all remaining elements
    print(value, end=' ')
print("\n")
```

extended_iteration.py

```
alpha beta gamma delta epsilon zeta

alpha beta gamma delta epsilon zeta

pomegranate cherry apricot

pomegranate cherry apricot date apple lemon

57 59 61 66 70 72 78 90 99
```

Grouping

- Groups consecutive elements by value
- Input must be sorted

The `groupby()` function groups consecutive elements of an iterable by value. As with sorted(), the value my be determined by a key function. This is similar to sort -u or the uniq commands in Linux.

groupby() returns an iterable of subgroups, which can then be converted to a list or iterated over. Each subgroup has a key, which is the common value, and an iterable of the values for that key.

For a more real-life example of grouping, see `group_dates_by_week.py` in the EXAMPLES folder.

Example

groupby_examples.py

```
from itertools import groupby

with open('../DATA/words.txt') as words_in:  # open file for reading
    all_words = (w.rstrip() for w in words_in)  # create generator of all words, stripped
of the trailing '

    g = groupby(all_words, key=lambda e: e[0])  # create a groupby() object where the key
is the first character in the word

    counts = {letter: len(list(wlist)) for letter, wlist in g}  # make a dictionary where
the key is the first character, and the value is the number of words that start with that
character; groupby groups all the words, then len() counts the number of words for that
character

sorted_letters = sorted(counts.items(), key=lambda e: e[1], reverse=True)  # sort the
counts dictionary by value (i.e., number of words, not the letter itself) into a list of
tuples
for letter, count in sorted_letters:  # loop over the list of tuples and print the letter
and its count
    print(letter, count)

print()
print("Total words counted:", sum(counts.values()))  # sum all the individual counts and
print the result
```

groupby_examples.py

```
s 19030
c 16277
p 14600
a 10485
r 10199
d 10189
m 9689
b 9325
t 8782
e 7086
i 6994
f 6798
h 6333
o 5858
g 5599
u 5095
l 5073
n 4411
w 3727
v 2741
k 1734
j 1378
q 827
y 552
z 543
x 137

Total words counted: 173462
```

Combinatoric generators

- Provides products, combinations and permutation
- Can specify max length of sub-sequences

Several functions provide products, combinations, and permutations.

product() return an iterator with the Cartesian product of two iterables.

combinations() returns the unique n-length combinations of an iterator.

permutations() returns all n-length sub-sequences of an iterator.

If the length is not specified, all sub-sequences are returned.

Example

combinations_permutations.py

```
#
from itertools import product, permutations, combinations

SUITS = 'CDHS'
RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

cards = product(SUITS, RANKS)  # Cartesian product (match every item in one list to every
item in the other list)
print(list(cards), '\n')

cards = [r + s for r, s in product(SUITS, RANKS)]  # reverse order and concatenate
elements using list comprehension
print(cards, '\n')

giant = ['fee', 'fi', 'fo', 'fum']

result = combinations(giant, 2)  # all distinct combinations of 4 items taken 2 at a time
print(list(result), "\n")

result = permutations(giant, 2)  # all distinct permutations of 4 items taken 2 at a time
print(list(result), "\n")
```

combinations_permutations.py

```
[('C', '2'), ('C', '3'), ('C', '4'), ('C', '5'), ('C', '6'), ('C', '7'), ('C', '8'),
('C', '9'), ('C', '10'), ('C', 'J'), ('C', 'Q'), ('C', 'K'), ('C', 'A'), ('D', '2'),
('D', '3'), ('D', '4'), ('D', '5'), ('D', '6'), ('D', '7'), ('D', '8'), ('D', '9'), ('D',
'10'), ('D', 'J'), ('D', 'Q'), ('D', 'K'), ('D', 'A'), ('H', '2'), ('H', '3'), ('H',
'4'), ('H', '5'), ('H', '6'), ('H', '7'), ('H', '8'), ('H', '9'), ('H', '10'), ('H',
'J'), ('H', 'Q'), ('H', 'K'), ('H', 'A'), ('S', '2'), ('S', '3'), ('S', '4'), ('S', '5'),
('S', '6'), ('S', '7'), ('S', '8'), ('S', '9'), ('S', '10'), ('S', 'J'), ('S', 'Q'),
('S', 'K'), ('S', 'A')]

['C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9', 'C10', 'CJ', 'CQ', 'CK', 'CA', 'D2',
'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'D10', 'DJ', 'DQ', 'DK', 'DA', 'H2', 'H3',
'H4', 'H5', 'H6', 'H7', 'H8', 'H9', 'H10', 'HJ', 'HQ', 'HK', 'HA', 'S2', 'S3', 'S4',
'S5', 'S6', 'S7', 'S8', 'S9', 'S10', 'SJ', 'SQ', 'SK', 'SA']

[('fee', 'fi'), ('fee', 'fo'), ('fee', 'fum'), ('fi', 'fo'), ('fi', 'fum'), ('fo',
'fum')]

[('fee', 'fi'), ('fee', 'fo'), ('fee', 'fum'), ('fi', 'fee'), ('fi', 'fo'), ('fi',
'fum'), ('fo', 'fee'), ('fo', 'fi'), ('fo', 'fum'), ('fum', 'fee'), ('fum', 'fi'),
('fum', 'fo')]
```

Chapter 3 Exercises

Exercise 3-1 (sum_of_values.py)

Read in the data from float_values.txt and print out the sum of all values. Do this with functional tools – there should be no explicit loops in your code.

TIP use reduce() + operator.add on the file object.

Exercise 3-2 (pres_by_state_func.py)

Using presidents.txt, print out a list of the number of presidents from each state.

TIP Use map() + lambda to split lines from presidents.txt on the 7th field, then use groupby() on that.

Exercise 3-3 (count_all_lines.py)

Count all of the lines in all the files specified on the command line without using any loops.

TIP use map() + chain.from_iterable() to create an iterable of all the lines, then use reduce to count them.

Chapter 4: Metaprogramming

Objectives

- Learn what metaprogramming means
- Access local and global variables by name
- Inspect the details of any object
- Use attribute functions to manipulate an object
- Design decorators for classes and functions
- Define classes with the type() function
- Create metaclasses

Metaprogramming

- Writing code that writes (or at least modifies) code
- Can simplify some kinds of programs
- Not as hard as you think!
- Considered deep magic in other languages

Metaprogramming is writing code that generates or modifies other code. It includes fetching, changing, or deleting attributes, and writing functions that return functions (AKA factories).

Metaprogramming is easier in Python than many other languages. Python provides explicit access to objects, even the parts that are hidden or restricted in other languages.

For instance, you can easily replace one method with another in a Python class, or even in an object instance. In Java, this would be deep magic requiring many lines of code.

globals() and locals()

- Contain all variables in a namespace
- globals() returns all global objects
- locals() returns all local variables

The **globals()** builtin function returns a dictionary of all global objects. The keys are the object names, and the values are the objects values. The dictionary is "live" — changes to the dictionary affect global variables.

The **locals()** builtin returns a dictionary of all objects in local scope.

Example

globals_locals.py

```
from pprint import pprint  # import prettyprint function

spam = 42  # global variable
ham = 'Smithfield'


def eggs(fruit):  # function parameters are local
    name = 'Lancelot'  # local variable
    idiom = 'swashbuckling'  # local variable
    print("Globals:")
    pprint(globals())  # globals() returns dict of all globals
    print()
    print("Locals:")
    pprint(locals())  # locals() returns dict of all locals


eggs('mango')
```

globals_locals.py

```
Globals:
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': '/Users/jstrick/curr/courses/python/common/examples/globals_locals.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7fb71008ccd0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'eggs': <function eggs at 0x7fb700048040>,
 'ham': 'Smithfield',
 'pprint': <function pprint at 0x7fb7100fb550>,
 'spam': 42}

Locals:
{'fruit': 'mango', 'idiom': 'swashbuckling', 'name': 'Lancelot'}
```

The inspect module

- Simplifies access to metadata
- Provides user-friendly functions for testing metadata

The `inspect` module provides user-friendly functions for accessing Python metadata.

Example

inspect_ex.py

```
import inspect
import geometry
from carddeck import CardDeck

deck = CardDeck("Leonard")

things = (
    geometry,
    geometry.circle_area,
    CardDeck,
    CardDeck.get_ranks,
    deck,
    deck.shuffle,
)

print("Name               Module?  Function?  Class?  Method?")
for thing in things:
    try:
        thing_name = thing.__name__
    except AttributeError:
        thing_name = type(thing).__name__ + " instance"
    print("{:18s} {!s:6s}   {!s:6s}     {!s:6s}  {!s:6s}".format(
        thing_name,
        inspect.ismodule(thing),  # test for module
        inspect.isfunction(thing),  # test for function
        inspect.isclass(thing),  # test for class
        inspect.ismethod(thing),
    ))


print()
def spam(p1, p2='a', *p3, p4, p5='b', **p6):  # define a function
    print(p1, p2, p3, p4, p5, p6)

# get argument specifications for a function
print("Function spec for Ham:", inspect.getfullargspec(spam))
print()

# get frame (function call stack) info
print("Current frame:", inspect.getframeinfo(inspect.currentframe()))
```

inspect_ex.py

```
Name               Module?  Function?  Class?  Method?
geometry           True     False      False   False
circle_area        False    True       False   False
CardDeck           False    False      True    False
get_ranks          False    False      False   True
CardDeck instance  False    False      False   False
shuffle            False    False      False   True

Function spec for Ham: FullArgSpec(args=['p1', 'p2'], varargs='p3', varkw='p6',
defaults=('a',), kwonlyargs=['p4', 'p5'], kwonlydefaults={'p5': 'b'}, annotations={})

Current frame:
Traceback(filename='/Users/jstrick/curr/courses/python/common/examples/inspect_ex.py',
lineno=40, function='<module>', code_context=['print("Current frame:",
inspect.getframeinfo(inspect.currentframe()))\n'], index=0)
```

Table 4. inspect module convenience functions

Function(s)	Description
ismodule(), isclass(), ismethod(), isfunction(), isgeneratorfunction(), isgenerator(), istraceback(), isframe(), iscode(), isbuiltin(), isroutine()	check object types
getmembers()	get members of an object that satisfy a given condition
getfile(), getsourcefile(), getsource()	find an object's source code
getdoc(), getcomments()	get documentation on an object
getmodule()	determine the module that an object came from
getclasstree()	arrange classes so as to represent their hierarchy
getargspec(), getargvalues()	get info about function arguments
formatargspec(), formatargvalues()	format an argument spec
getouterframes(), getinnerframes()	get info about frames
currentframe()	get the current stack frame
stack(), trace()	get info about frames on the stack or in a traceback

Working with attributes

- Objects are dictionaries of attributes
- Special functions can be used to access attributes
- Attributes specified as strings
- Syntax

```
getattr(object, attribute [,defaultvalue] )
hasattr(object, attribute)
setattr(object, attribute, value)
delattr(object, attribute)
```

All Python objects are essentially dictionaries of attributes. There are four special builtin functions for managing attributes. These may be used to programmatically access attributes when you have the name as a string.

getattr() returns the value of a specified attribute, or raises an error if the object does not have that attribute. `getattr(a,'spam')` is the same as `a.spam`. An optional third argument to getattr() provides a default value for nonexistent attributes (and does not raise an error).

hasattr() returns the value of a specified attribute, or None if the object does not have that attribute.

setattr() an attribute to a specified value.

delattr() deletes an attribute and its corresponding value.

Example

attributes.py

```
class Spam():

    def eggs(self, msg):  # create attribute
        print("eggs!", msg)


s = Spam()

s.eggs("fried")

print("hasattr()", hasattr(s, 'eggs'))  # check whether attribute exists

e = getattr(s, 'eggs')  # retrieve attribute
e("scrambled")


def toast(self, msg):
    print("toast!", msg)


setattr(Spam, 'eggs', toast)  # set (or overwrite) attribute

s.eggs("buttered!")

delattr(Spam, 'eggs')  # remove attribute

try:
    s.eggs("shirred")
except AttributeError as err:  # missing attribute raises error
    print(err)
```

attributes.py

```
eggs! fried
hasattr() True
eggs! scrambled
toast! buttered!
'Spam' object has no attribute 'eggs'
```

Adding instance methods

- Use setattr()
- Add instance method to class
- Add instance method to instance

Using `setattr()`, it is easy to add instance methods to classes. Just add a function object to the class. Because it is part of the class itself, it will automatically be bound to the instance. Remember that an instance method expects `self` as the first parameter. In fact, this is the meaning of a bound instance — it is "bound" to the instance, and therefore when called, it is passed the instance as the first parameter.

Once added, the method may be called from any existing *or new* instance of the class.

To add an instance method to an *instance* takes a little more effort. Because it's not being added to the class, it is not automatically bound. The function needs to know what instance it should be bound to. This can be accomplished with the `types.MethodType()` function.

Pass the function and the instance to `MethodType()`.

Example

adding_instance_methods.py

```
from types import MethodType

class Dog(): # Define Dog type
    pass

d1 = Dog()  # Create instance of Dog

def bark(self):  # Define (unbound) function
    print("Woof! woof!")

setattr(Dog, "bark", bark)  # Add function to class (which binds it as an instance
method)

d2 = Dog() # Define another instance of Dog

d1.bark()  # New function can be called from either instance
d2.bark()

def wag(self): # Create another unbound function
    print("Wagging...")

setattr(d1, "wag", MethodType(wag, d1))  # Add function to instance after passing it
through MethodType()

d1.wag()  # Call instance method
try:
    d2.wag()  # Instance method not available - only bound to d1
except AttributeError as err:
    print(err)
```

adding_instance_methods.py

```
Woof! woof!
Woof! woof!
Wagging...
'Dog' object has no attribute 'wag'
```

Callable classes

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

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

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

you can say

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

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

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

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

NOTE See `memorychecker.py` in the EXAMPLES folder for a real-life use of a callable class to make it easy to check the current memory footprint.

Example

callable_class.py

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

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

class HTMLWrapper():

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

    def __call__(self, text):  # Define function to be called when instance is called
        return '<{0}>{1}</{0}>'.format(self._tag, text)

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

    # callable class
    h1 = HTMLWrapper('h1')  # Create instance of "callable class"
    print(h1('spam'))  # Instance is callable -- essentially  h1.call('spam')
    div = HTMLWrapper('div')
    print(div('ham'))
    print(div('toast'))
    print(div('jam'))
```

callable_class.py

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

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

Decorators

- Classic design pattern
- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take parameters (but not required to)
- functools.wraps() preserves function's properties

In Python, many decorators are provided by the standard library, such as property() or classmethod()

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the @app.route() decorator in Flask maps a URL to a view function.

As another example, **unittest** provides decorators to skip tests. A very common decorator is **@property**, which converts a class method into a property object.

A decorator can be any *callable*, which means it can be a normal function, a class method, or a class which implements the __call__() method (AKA callable class, as discussed earlier).

A simple decorator expects the item being decorated as its parameter, and returns a replacement. Typically, the replacement is a new function, but there is no restriction on what is returned. If the decorator itself needs parameters, then the decorator returns a wrapper function that expects the item being decorated, and then returns the replacement.

Table 5. Decorators in the standard library

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

Applying decorators

- Use @ symbol
- Applied to *next* item only
- Multiple decorators OK

The **@** sign is used to apply a decorator to a function or class. A decorator only applies to the next definition in the script.

The most important thing to know about the decorators is the following syntax:

```
@spam
def ham():
    pass
```

is exactly the same as

```
ham = spam(ham)
```

and

```
@spam(a, b, c)
def ham():
    pass
```

is exactly the same as

```
ham = spam(a, b, c)(ham)
```

Once you understand this, then creating decorators is just a matter of writing functions or classes and having them return the appropriate thing.

Table 6. Implementing Decorators

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

Trivial Decorator

- Decorator can return anything
- Not very useful, usually

A decorator does not have to be elaborate. It can return anything, though typically decorators return the same type of object they are decorating.

In this example, the decorator returns the integer value 42. This is not particularly useful, but illustrates that the decorator always replaces the object being decorated with *something*.

Example

deco_trivial.py

```
def void(thing_being_decorated):
    return 42  # replace function with 42

name = "Guido"
x = void(name)

@void  # decorate hello() function
def hello():
    print("Hello, world")

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

print(hello, type(hello)) # hello is now the integer 42, not a function
print(howdy, type(howdy)) # hello is now the integer 42, not a function
print(x, type(x))
```

deco_trivial.py

```
42 <class 'int'>
42 <class 'int'>
42 <class 'int'>
```

Decorator functions

- Provide a wrapper around a function
- Purposes
  - Add functionality
  - Register
  - ??? (open-ended)
- Optional parameters

A decorator function acts as a wrapper around some object (usually function or class). It allows you to add features to a function without changing the function itself. For instance, the @property, @classmethod, and @staticmethod decorators are used in classes.

A simple decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code. More complex decorators expect parameters to the decorator itself. In this case the decorator returns a function that expects the original function, and returns the replacement function.

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

The **wraps** decorator from the functools module in the standard library should be used with the function that returns the replacement function. This makes sure the replacement function keeps the same properties (especially the name) as the original (target) function. Otherwise, the replacement function keeps all of its own attributes.

Example

deco_debug.py

```
from functools import wraps


def debugger(old_func):  # decorator function -- expects decorated (original) function as
a parameter

    @wraps(old_func)  # @wraps preserves name of original function after decoration
    def new_func(*args, **kwargs):  # replacement function; takes generic parameters
        print("*" * 40)  # new functionality added by decorator
        print("** function", old_func.__name__, "**")  # new functionality added by
decorator

        if args:  # new functionality added by decorator
            print("\targs are ", args)
        if kwargs:  # new functionality added by decorator
            print("\tkwargs are ", kwargs)

        print("*" * 40)  # new functionality added by decorator

        return old_func(*args, **kwargs)  # call the original function

    return new_func  # return the new function object


@debugger  # apply the decorator to a function
def hello(greeting, whom='world'):
    print("{}, {}".format(greeting, whom))


hello('hello', 'world')  # call new function
print()

hello('hi', 'Earth')
print()

hello('greetings')
```

deco_debug.py

```
**************************************
** function hello **
    args are  ('hello', 'world')
**************************************
hello, world

**************************************
** function hello **
    args are  ('hi', 'Earth')
**************************************
hi, Earth

**************************************
** function hello **
    args are  ('greetings',)
**************************************
greetings, world
```

Decorator Classes

- Same purpose as decorator functions
- Two ways to implement
  - No parameters
  - Expects parameters
- Decorator can keep state

A class can also be used to implement a decorator. The advantage of using a class for a decorator is that a class can keep state, so that the replacement function can update information stored at the class level.

Implementation depends on whether the decorator needs parameters.

If the decorator does *not* need parameters, the class must implement two methods: __init__() is passed the original function, and can perform any setup needed. The __call__ method *replaces* the original function. In other word, after the function is decorated, calling the function is the same as calling *CLASS*.__call__.

If the decorator *does* need parameters, __init__() is passed the parameters, and __call__() is passed the original function, and must *return* the replacement function.

A good use for a decorator class is to log how many times a function has been called, or even keep track of the arguments it is called with (see example for this).

Example

deco_debug_class.py

```
class debugger():  # class implementing decorator

    function_calls = []

    def __init__(self, func):  # original function passed into decorator's constructor
        self._func = func

    def __call__(self, *args, **kwargs):  # __call__() is replacement function

        # print("*" * 40)  # add useful features to original function
        # print("function {}()".format(self._func.__name__))  # add useful features to
original function
        # print("\targs are ", args)  # add useful features to original function
        # print("\tkwargs are ", kwargs)  # add useful features to original function
        #
        # print("*" * 40)  # add useful features to original function

        self.function_calls.append(  # add function name and arguments to saved list
            (self._func.__name__, args, kwargs)
        )

        result = self._func(*args, **kwargs)  # call the original function
        return result # return result of calling original function

    @classmethod
    def get_calls(cls): # define method to get saved function call information
        return cls.function_calls

@debugger  # apply debugger to function
def hello(greeting, whom="world"):
    print("{}, {}".format(greeting, whom))

@debugger  # apply debugger to function
def bark(bark_word, *, repeat=2):
    print("{0}! ".format(bark_word) * repeat)

hello('hello', 'world')  # call replacement function
print()

hello('hi', 'Earth')
print()

hello('greetings')
```

```
bark("woof", repeat=3)
bark("yip", repeat=4)
bark("arf")

hello('hey', 'girl')

print('-' * 60)

for i, info in enumerate(debugger.get_calls(), 1):  # display function call info from
class
    print("{:2d}. {:10s} {!s:20s} {!s:20s}".format(i, info[0], info[1], info[2]))
```

deco_debug_class.py

```
hello, world

hi, Earth

greetings, world
woof! woof! woof!
yip! yip! yip! yip!
arf! arf!
hey, girl
------------------------------------------------------------
 1. hello      ('hello', 'world')   {}
 2. hello      ('hi', 'Earth')      {}
 3. hello      ('greetings',)       {}
 4. bark       ('woof',)            {'repeat': 3}
 5. bark       ('yip',)             {'repeat': 4}
 6. bark       ('arf',)             {}
 7. hello      ('hey', 'girl')      {}
```

Decorator parameters

- Decorator functions require two nested functions
- Method __call__() returns replacement function in classes

A decorator can be passed parameters. This requires a little extra work.

For decorators implemented as functions, the decorator itself is passed the parameters; it contains a nested function that is passed the decorated function (the target), and it returns the replacement function.

For decorators implemented as classes, init is passed the parameters, __call__() is passed the decorated function (the target), and __call__ returns the replacement function.

There are many combinations of decorators (8 total, to be exact). This is because decorators can be implemented as either functions or classes, they may take parameters, or not, and they can decorate either functions or classes. For an example of all 8 approaches, see the file **decorama.py** in the EXAMPLES folder.

Example

deco_params.py

```
#

from functools import wraps  # wrapper to preserve properties of original function


def multiply(multiplier): # actual decorator -- receives decorator parameters

    def deco(old_func): # "inner decorator" -- receives function being decorated

        @wraps(old_func)  # retain name, etc. of original function
        def new_func(*args, **kwargs): # replacement function -- this is called instead
of original
            result = old_func(*args, **kwargs) # call original function and get return
value
            return result * multiplier # multiple result of original function by
multiplier

        return new_func # deco() returns new_function

    return deco # multiply returns deco



@multiply(4)
def spam():
    return 5


@multiply(10)
def ham():
    return 8

a = spam()
b = ham()
print(a, b)
```

deco_params.py

```
20 80
```

Creating classes at runtime

- Use the **type()** function
- Provide dictionary of attributes

A class can be created programmatically, without the use of the class statement. The syntax is

```
type("name", (base_class, ···), {attributes})
```

The first argument is the name of the class, the second is a tuple of base classes (use object if you are not inheriting from a specific class), and the third is a dictionary of the class's attributes.

NOTE Instead of type, any other *metaclass* can be used.

Example

creating_classes.py

```
def function_1(self):  # create method (not inside a class -- could be a lambda)
    print("Hello from f1()")


def function_2(self):  # create method (not inside a class -- could be a lambda)
    print("Hello from f2()")


NewClass = type("NewClass", (), {  # create class using type() -- parameters are class
name, base classes, dictionary of attributes
    'hello1': function_1,
    'hello2': function_2,
    'color': 'red',
    'state': 'Ohio',
})

n1 = NewClass()  # create instance of new class

n1.hello1()  # call instance method
n1.hello2()
print(n1.color)  # access class data
print()

SubClass = type("SubClass", (NewClass,), {'fruit': 'banana'})  # create subclass of first
class
s1 = SubClass()  # create instance of subclass
s1.hello1()  # call method on subclass
print(s1.color)  # access class data
print(s1.fruit)
```

creating_classes.py

```
Hello from f1()
Hello from f2()
red

Hello from f1()
red
banana
```

Monkey Patching

- Modify existing class or object
- Useful for enabling/disabling behavior
- Can cause problems

"Monkey patching" refers to technique of changing the behavior of an object by adding, replacing, or deleting attributes from outside the object's class definition.

It can be used for:

- Replacing methods, attributes, or functions
- Modifying a third-party object for which you do not have access
- Adding behavior to objects in memory

If you are not careful when creating monkey patches, some hard-to-debug problems can arise

- If the object being patched changes after a software upgrade, the monkey patch can fail in unexpected ways.
- Conflicts may occur if two different modules monkey-patch the same object.
- Users of a monkey-patched object may not realize which behavior is original and which comes from the monkey patch.

Monkey patching defeats object encapsulation, and so should be used sparingly.

Decorators are a convenient way to monkey-patch a class. The decorator can just add a method to the decorated class.

Example

meta_monkey.py

```
class Spam():  # create normal class

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

    def eggs(self):  # add normal method
        print("Good morning, {}. Here are your delicious fried eggs.".format(self._name,
))


s = Spam('Mrs. Higgenbotham')  # create instance of class
s.eggs()  # call method


def scrambled(self):  # define new method outside of class
    print("Hello, {}. Enjoy your scrambled eggs".format(self._name, ))


setattr(Spam, "eggs", scrambled)  # monkey patch the class with the new method

s.eggs()  # call the monkey-patched method from the instance
```

meta_monkey.py

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

Do you need a Metaclass?

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

Before we cover the details of metaclasses, a disclaimer: you will probably never need to use a metaclass. When you think you might need a metaclass, consider using inheritance or a class decorator. However, metaclasses may be a more elegant approach to certain kinds of tasks, such as registering classes when they are defined.

There are two use cases where metaclasses are always an appropriate solution, because they must be done before the class is created:

- Modifying the class name
- Modifying the the list of base classes.

Several popular frameworks use metaclasses, Django in particular. In Django they are used for models, forms, form fields, form widgets, and admin media.

Remember that metaclasses can be a more elegant way to accomplish things that can also be done with inheritance, composition, decorators, and other techniques that are less "magic".

About metaclasses

- Metaclass:Class::Class:Object

Just as a class is used to create an instance, a *metaclass* is used to create a class.

The primary reason for a metaclass is to provide extra functionality at *class creation* time, not *instance creation* time. Just as a class can share state and actions across many instances, a metaclass can share (or provide) data and state across many *classes*.

The metaclass might modify the list of base classes, or register the class for later retrieval.

The builtin metaclass that Python provides is **type**.

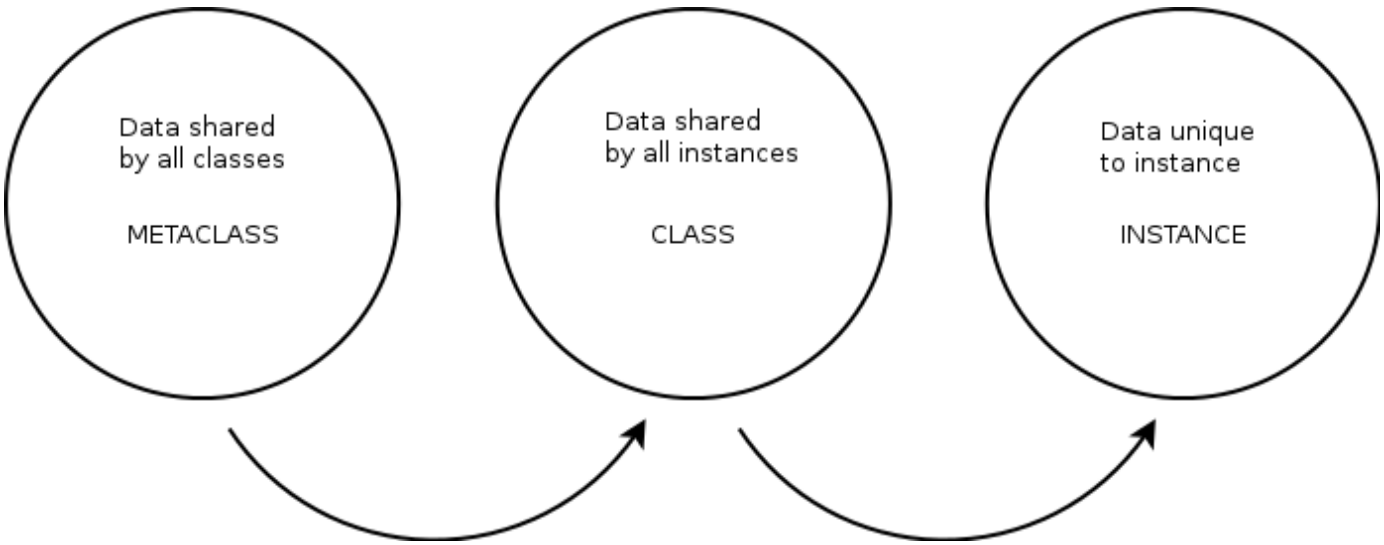
As we saw earlier ,you can create a class from a metaclass by passing in the new class's name, a tuple of base classes (which can be empty), and a dictionary of class attributes (which also can be empty).

```
class Spam(Ham):
    id = 1
```

is exactly equivalent to

```
Spam = type('Spam', (Ham,), {"id": 1})
```

Replacing "type" with the name of any other metaclass works the same.



Mechanics of a metaclass

- Like normal class
- *Should* implement __new__
- *Can* implement
  - __init__
  - __prepare__
  - __call__

To create a metaclass, define a normal class. Most metaclasses implement the **__new__** method. This method is called with the type, name, base classes, and attribute dictionary (if any) of the new class. It should return a new class, typically using **super().__new__()**, which is very similar to how normal classes create instances. This is one place you can modify the class being created. You can add or change attributes, methods, or properties.

For instance, the Django framework uses metaclasses for Models. When you create an instance of a Model, the metaclass code automatically creates methods for the fields in the model. This is called "declarative programming", and is also used in SqlAlchemy's declarative model, in a way pretty similar to Django.

When you execute the following code:

```
class SomeClass(metaclass=SomeMeta):
    pass
```

META(name, bases, attrs) is executed, where META is the metaclass (normally **type()**). Then,

1. The __prepare__ method of the metaclass is called
2. The __new__ method of the metaclass is called
3. The __init__ method of the metaclass is called.

Next, after the following code runs:

```
obj = SomeClass()
```

SomeMeta.call() is called. It returns whatever SomeMeta.__new__() returned.

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

Example

metaclass_generic.py

```
class Meta(type):

    def __prepare__(class_name, bases):
        """
        "Prepare" the new class. Here you can update the base classes.

        :param name: Name of new class as a string
        :param bases: Tuple of base classes
        :return: Dictionary that initializes the namespace for the new class (must be a
dict)
        """
        print("in metaclass (class={}) __prepare__()".format(class_name), end=' ==> ')
        print("params: name={}, bases={}".format(class_name, bases))
        return {'animal': 'wombat', 'id': 100}

    def __new__(metatype, name, bases, attrs):
        """
        Create the new class. Called after __prepare__(). Note this is only called when
classes

        :param metatype: The metaclass itself
        :param name: The name of the class being created
        :param bases: bases of class being created (may be empty)
        :param attrs: Initial attributes of the class being created
        :return:
        """
        print("in metaclass (class={}) __new__()".format(name), end=' ==> ')
        print("params: type={} name={} bases={} attrs={}".format(metatype, name, bases,
attrs))
        return super().__new__(metatype, name, bases, attrs)

    def __init__(cls, *args):
        """

        :param cls: The class being created (compare with 'self' in normal class)
        :param args: Any arguments to the class
        """
        print("in metaclass (class={}) __init__()".format(cls.__name__), end=' ==> ')
        print("params: cls={}, args={}".format(cls, args))

        super().__init__(cls)

    def __call__(self, *args, **kwargs):
```

```
        """
        Function called when the metaclass is called, as in NewClass = Meta(...)

        :param args:
        :param args:
        :param kwargs:
        :return:
        """
        print("in metaclass (class={})__call__()".format(self.__name__))


class MyBase():
    pass

print('=' * 60)

class A(MyBase, metaclass=Meta):
    id = 5

    def __init__(self):
        print("In class A __init__()")

print('=' * 60)

class B(MyBase, metaclass=Meta):
    animal = 'wombat'

    def __init__(self):
        print("In class B __init__()")


print('-' * 60)
m1 = A()
print('-' * 60)
m2 = B()
print('-' * 60)
m3 = A()
print('-' * 60)
m4 = B()
print('-' * 60)
print("animal: {} id: {}".format(A.animal, B.id))
```

metaclass_generic.py

```
============================================================
in metaclass (class=A) __prepare__() ==> params: name=A, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=A) __new__() ==> params: type=<class '__main__.Meta'> name=A
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 5, '__module__':
'__main__', '__qualname__': 'A', '__init__': <function A.__init__ at 0x7fea1805b670>}
in metaclass (class=A) __init__() ==> params: cls=<class '__main__.A'>, args=('A',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 5, '__module__': '__main__',
'__qualname__': 'A', '__init__': <function A.__init__ at 0x7fea1805b670>})
============================================================
in metaclass (class=B) __prepare__() ==> params: name=B, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=B) __new__() ==> params: type=<class '__main__.Meta'> name=B
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 100, '__module__':
'__main__', '__qualname__': 'B', '__init__': <function B.__init__ at 0x7fea1805b700>}
in metaclass (class=B) __init__() ==> params: cls=<class '__main__.B'>, args=('B',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 100, '__module__': '__main__',
'__qualname__': 'B', '__init__': <function B.__init__ at 0x7fea1805b700>})
------------------------------------------------------------
in metaclass (class=A)__call__()
------------------------------------------------------------
in metaclass (class=B)__call__()
------------------------------------------------------------
in metaclass (class=A)__call__()
------------------------------------------------------------
in metaclass (class=B)__call__()
------------------------------------------------------------
animal: wombat id: 100
```

Singleton with a metaclass

- Classic example
- Simple to implement
- Works with inheritance

One of the classic use cases for a metaclass in Python is to create a *singleton* class. A singleton is a class that only has one actual instance, no matter how many times it is instantiated. Singletons are used for loggers, config data, and database connections, for instance.

To create a single, implement a metaclass by defining a class that inherits from **type**. The class should have a class-level dictionary to store each class's instance. When a new instance of a class is created, check to see if that class already has an instance. If it does not, call __call__ to create the new instance, and add the instance to the dictionary.

In either case, then return the instance where the key is the class object.

Example

metaclass_singleton.py

```
class Singleton(type): # use type as base class of a metaclas
    _instances = {}  # dictionary to keep track of instances

    def __new__(typ, *junk):
        # print("__new__()")
        return super().__new__(typ, *junk)

    def __call__(cls, *args, **kwargs):  # __call__ is passed the new class plus its
parameters
        # print("__call__()")
        if cls not in cls._instances:    # check to see if the new class has already been
instantiated
            cls._instances[cls] = super().__call__(*args, **kwargs)  # if not, create the
(single) class instance and add to dictionary

        return cls._instances[cls]   # return the (single) class instance


class ThingA(metaclass=Singleton):   # Define two different classes which use Singleton
    def __init__(self, value):
        self.value = value


class ThingB(metaclass=Singleton):   # Define two different classes which use Singleton
    def __init__(self, value):
        self.value = value


ta1 = ThingA(1)  # Create instances of ThingA and ThingB
ta2 = ThingA(2)
ta3 = ThingA(3)

tb1 = ThingB(4)
tb2 = ThingB(5)
tb3 = ThingB(6)

for thing in ta1, ta2, ta3, tb1, tb2, tb3:
    print(type(thing).__name__, id(thing), thing.value)  # Print the type, name, and ID
of each thing -- only one instance is ever created for each class
```

metaclass_singleton.py

```
ThingA 140304369003248 1
ThingA 140304369003248 1
ThingA 140304369003248 1
ThingB 140304369003008 4
ThingB 140304369003008 4
ThingB 140304369003008 4
```

Chapter 4 Exercises

Exercise 4-1 (pres_attr.py)

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

Exercise 4-2 (pres_monkey.py, pres_monkey_amb.py)

Monkey-patch the President class to add a method get_full_name which returns a single string consisting of the first name and the last name, separated by a space.

> **TIP** Instead of a method, make full_name a property.

Exercise 4-3 (sillystring.py)

Without using the **class** statement, create a class named SillyString, which is initialized with any string. Include an instance method called every_other which returns every other character of the string.

Instantiate your string and print the result of calling the **every_other()** method. Your test code should look like this:

```
ss = SillyString('this is a test')
print(ss.every_other())
```

It should output

```
ti sats
```

Exercise 4-4 (doubledeco.py)

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

Exercise 4-5 (word_actions.py)

Write a decorator, implemented as a class, to register functions that will process a list of words. The decorated functions will take one parameter — a string — and return the modified string.

The decorator itself takes two parameters — minimum length and maximum length. The class will store the min/max lengths as the key, and the functions as values, as class data.

The class will also provide a method named **process_words**, which will open **DATA/words.txt** and read it line by line. Each line contains a word.

For every registered function, if the length of the current word is within the min/max lengths, call all the functions whose key is that min/max pair.

In other words, if the registry key is (5, 8), and the value is [func1, func2], when the current word is within range, call func1(*w*) and func2(*w*), where *w* is the current word.

Example of class usage:

```
word_select = WordSelect()  # create callable instance

@word_select(16, 18) # register function for length 16-18, inclusive
def make_upper(s):
    return s.upper()

word_select.process_words()  # loop over words, call functions if selected
```

Suggested functions to decorate:

- make the word upper-case
- put stars before or around the word
- reverse the word

Remember all the decorated functions take one argument, which is one of the strings in the word list, and return the modified word.

Chapter 5: Generators and Other Iterables

Objectives

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

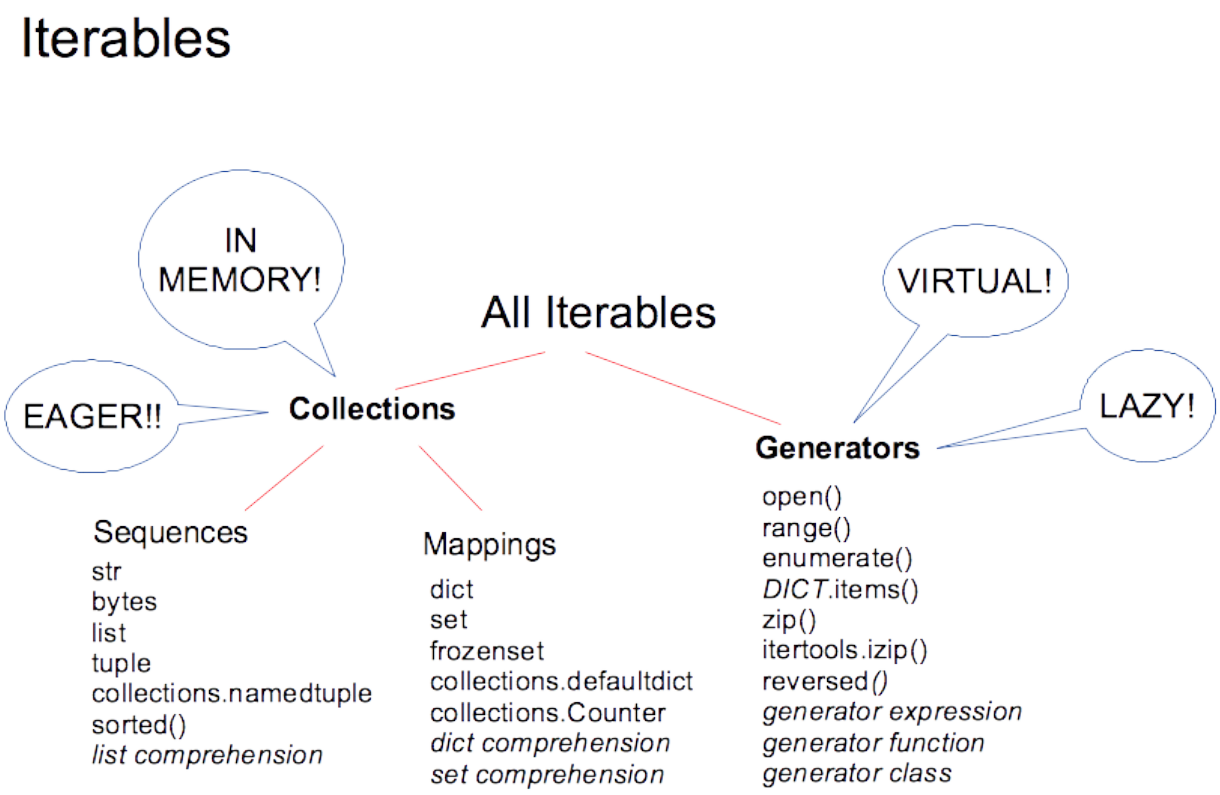
Iterables

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

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

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

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



Unpacking function arguments

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

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

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

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

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

Example

param_unpacking.py

```
from datetime import date

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

for dt in dates:
    d = date(*dt)  # instead of date(dt[0], dt[1], dt[2])
    print(d)

print()

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

sort_opts = {
    'key': lambda e: e.lower(),
    'reverse': True,
}  # config info in dictionary

sorted_fruits = sorted(fruits, **sort_opts)  # dictionary converted to named parameters
print(sorted_fruits)
```

param_unpacking.py

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

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

Iterable unpacking

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

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

```
var1, ··· = iterable
```

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

Example

iterable_unpacking.py

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

x, y, z = values  # unpack values (which is an iterable) into individual variables

print(x, y, z)
print()

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

for row in people:
    first_name, last_name, _ = row  # unpack row into variables
    print(first_name, last_name)
print()

for first_name, last_name, _ in people:  # a for loop unpacks if there is more than one
variable
    print(first_name, last_name)
print()

# extended unpacking
values = ['a', 'b', 'c', 'd', 'e', 'f']
x, y, *z = values
print(x, y, z)

x, *y, z = values
print(x, y, z)

*x, y, z = values
print(x, y, z)
```

iterable_unpacking.py

```
a b c

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

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

a b ['c', 'd', 'e', 'f']
a ['b', 'c', 'd', 'e'] f
['a', 'b', 'c', 'd'] e f
```

Extended iterable unpacking

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

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

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

NOTE Extended variable unpacking is not available in Python 2.

Example

extended_iterable_unpacking.py

```
values = ['a', 'b', 'c', 'd', 'e']  # values has 6 elements

x, y, *z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

x, *y, z = values  # {splat} takes all extra elements from iterable
print("x: {}    y: {}    z: {}".format(x, y, z))
print()

*x, y, z = values  # {splat} takes all extra elements from iterable
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:  # name gets all but the last field
    print(name)
print()
```

extended_iterable_unpacking.py

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

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

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

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

What exactly is an iterable?

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

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

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

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

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

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

List comprehensions

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

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

A loop like this:

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

can be rewritten as

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

A conditional `if` may be added:

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

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

Common uses:

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

Example

list_comprehensions.py

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

ufruits = [fruit.upper() for fruit in fruits]        # Simple transformation of all
elements
afruits = [fruit.title() for fruit in fruits if fruit.startswith('a')]  # Transformation
of selected elements only

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

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

doubles = [v * 2 for v in values]   # Any kind of data is OK

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

nums = [x for x in values if isinstance(x, int)]   # Select only integers from list
print(nums, '\n')

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

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

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

deck = [(rank, suit) for suit in suits for rank in ranks]   # More than one for is OK

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

list_comprehensions.py

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

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

[2, 42, 18, 92]

>gronk< >pulaba< >floog<

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

...

Generators

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

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

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

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

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

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

There are three ways to create generators in Python:

- generator expression
- generator function
- generator class

Generator Expressions

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

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

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

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

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

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

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

Example

generator_expressions.py

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

ufruits = (fruit.upper() for fruit in fruits)  # These are all exactly like the list
comprehension example, but return generators rather than lists
afruits = (fruit.title() for fruit in fruits if fruit.startswith('a'))

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

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

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

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

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

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

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

generator_expressions.py

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

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

2 42 18 92

>gronk< >pulaba< >floog<

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

Generator functions

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

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

Example

sieve_generator.py

```
def next_prime(limit):
    flags = set()  # initialize empty set (to be used for "is-prime" flags

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j)  # add non-prime elements to set
        yield i  # execution stops here until next value is requested by for-in loop


np = next_prime(200)  # next_prime() returns a generator object
for prime in np:  # iterate over yielded primes
    print(prime, end=' ')
```

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

```
def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            yield line.rstrip('\n\r')  # 'yield' causes this function to return a
generator object

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

line_trimmer.py

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

Coroutines

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

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

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

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

This can be useful for async-like coding.

```
...
   ham = yield spam
...

...
   next(p)
   p.send('foo')
   print(next(p))
...
```

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

Example

coroutine_example.py

```
def coroutine():  # Define a coroutine function
    in_value = ''
    while True:
        in_value = yield in_value.upper()  # Yield gets the result of send() AND provides
the next value
        print('in_value:', in_value)


c = coroutine()  # Create instance of coroutine

print(c)  # c is a coroutine object (also a generator)
print(next(c))  # next() will get the next value and prime the coroutine. You _must_ call
next before calling send()
print()

for letter in "alpha", "beta", 'gamma':
    print("out_value:", c.send(letter))  # Send a value to the coroutine
print()


def faux_range():  # define a generator
    i = 0
    while i < 4:
        yield i
        i += 1


def spam():
    yield from faux_range()  # Define generator; yield from _delegates_ to another
generator


s = spam()  # Create instance of generator

for x in s:  # Looping through a spam() instance effictively loops over faux_range()
    print(x)
```

coroutine_example.py

```
<generator object coroutine at 0x7fcd900d2890>


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

0
1
2
3
```

Generator classes

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

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

Such a class must implement two special methods:

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

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

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

Example

trimmedfile.py

```
class TrimmedFile:
    def __init__(self, file_name):  # constructor is passed file name
        self._file_in = open(file_name)

    def __iter__(self):  # A generator must implement iter(), which must return an
iterator. Typically it returns self, as the generator _is_ the iterator
        return self

    def __next__(self):  # next() returns the next value of the generator
        line = self._file_in.readline()
        if line == '':
            raise StopIteration  # Raise StopIteration when there are no more values to
generate
        else:
            return line.rstrip('\n\r')  # The actual work of this generator -- remove the
newline from the line


if __name__ == '__main__':
    trimmed = TrimmedFile('../DATA/mary.txt')  # To use the generator, create an instance
and iterator over it.
    for line in trimmed:
        print(line)
```

trimmedfile.py

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

Chapter 5 Exercises

Exercise 5-1 (pres_upper.py)

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

Exercise 5-2 (pres_upper_gen.py)

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

Exercise 5-3 (pres_gen.py)

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

Exercise 5-4 (fauxrange.py)

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

Chapter 6: PyParsing

Objectives

- Understand parsing
- Build generic parsers
- Add action functions to parsers
- Retrieve parsed data

Text Parsing Tools

- Standard tool: regular expressions
- REs are scary!
- REs are verbose
- Enter **pyparsing**

> Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.
>
> — Jamie Zawinski, 1997

The author of this course has been using regular expressions successfully for over two decades, and has done a ton of useful work with them. However, they can be cryptic and tricky. Here is a regular expression to parse a string like “Ja. 15, 2014" or “Au. 27, 1990":

```
\b([A-Z][a-z]{2})\.?\s+(\d+),\s+(\d{4})
```

If you want to retrieve the month, day, and year by name rather than numeric index, it would look like this:

```
\b(?P<MONTH>[A-Z][a-z]{2})\.?\s+(?P<DAY>\d+),\s+(?P<YEAR>\d{4})
```

As it turns out, there are other ways to parse text. You could create a tailor-made parser than iterated over the text character-by-character, with logic for finding your target. This is tedious, error-prone, time-consuming, and no one does it.

You could use string methods such as **split()**, **startswith()**, **endswith()**, etc., to grab pieces and then analyze them. This is likewise tedious, error-prone, and time-consuming, but some people do take this route because they are afraid of regular expressions.

A better option is the **pyparsing** module. This chapter describes how to apply pyparsing to everyday text parsing tasks.

About pyparsing

- All purpose text parser
- Similar to lex/yacc
- Many builtin types
- Not just for *ML

Pyparsing (https://pyparsing.wikispaces.com/) is a Python module for creating text parsers. It was developed by Paul McGuire. Install with pip for most versions of Python.

It has many builtin types, and you can construct **parsers** for any text sequences, including nested text. Regular expressions are not good with nested text, such as **XML**, **HTML**, or Python source code.

First, create a grammar to specify what should be matched. Then, you call a parsing function from the grammar and it returns text tokens while automatically skipping over white space. Pyparsing provides many functions for specifying what should be matched, how items should repeat, and more.

NOTE The Anaconda Python bundle includes pyparsing.

Defining a Grammar

The first step in using Pyparsing is to define a grammar. A grammar defines exactly what the target text can contain. The best way to do this is in a "top-down" manner, specifying the entire target, then refining what each component means until you get down to literal characters. While this step is optional, it really helps when actually defining the parser.

The usual notation for grammars is called Backus-Naur Form, or BNF for short. You don't have to worry about following the rules exactly with BNF, but it is convenient for describing how things fit together.

The basic form is

```
symbol ::= expression
```

This means that symbol is composed of the parts specified in the expression. For instance, a person's name can be specified as

```
name ::= first-name [middle-name] last-name
first_name ::= alpha | first_name alpha #one or more alphas
last_name ::=alphas+  #shorter way to express one or more alphas
alpha = 'a' | 'b' | 'c' etc
```

This means that a name consists of a first name, an optional middle name (brackets indicate optional components), and a last name. A first name consists of one or more alphabetic characters, as does a last name. The plus sign means "one or more".The pipe symbol means "or". Pyparsing has predefined symbols for letters, digits, and other common sets of characters.

There are more details, but you get the idea.

TIP If you are interested in the gory details of BNF, see http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

NOTE The grammar is not directly used by pyparsing, but is a way for you to formalize your parser.

A Simple Parser

To use pyparsing, you need to import it. For convenience, you can import all the names from pyparsing:

```
from pyparsing import *
```

Let's start with a US Social Security Number. The format is DDD-DD-DDDD, where D is any digit. Using BNF, the grammar for an SSN is

```
ssn ::= num+ '-' num+ '-' num+
num ::= '0' | '1' | '2' etc
```

In pyparsing, you can represent a fixed number of digits with the expression

```
Word(nums, exact=N)
```

A *Word* is a sequence of characters surrounded by white space or punctuation.

Since a dash is a literal character (a "primitive", in the parsing world), you don't need to do anything special with it. However, You can give the literal character a name, to help make your parser readable.

Thus, you have:

```
dash = '-'
ssn = Word(nums, exact=3) + dash + Word(nums, exact=2) + dash + Word(nums, exact=4)
```

To use the parser, call **parseString()** on the parser object:

```
target = '123-45-6789'
result = ssn.parseString(target)
print result
```

This gives the result

```
['123', '-', '45', '-', '6789']
```

For most applications, you would want the entire SSN as a single string, so you can add the Combine() function, which glues together all of the tokens from the parser passed to it as a single token.

```
ssn = Combine(Word(nums, exact=3) + dash + Word(nums, exact=2) + dash + Word(nums,
exact=4))
```

So now the result is

```
['123-45-6789']
```

NOTE Standard Python wisdom says 'from module import *' is a bad idea, but sometimes the convenience outweighs the risk.

Example

pp_parse_ssn.py

```
from pyparsing import *

'''  # Topdown reference grammar
grammar:
ssn = nums+ dash nums+ dash nums+
dash = '-'
nums = '0' | '1' | '2' etc etc
'''

dash = Suppress('-')  # Define dash for readability

ssn_parser = Combine(
    Word(nums, exact=3)  # Match exactly 3 digits
    + dash
    + Word(nums, exact=2)
    + dash
    + Word(nums, exact=4)
)   # Parser combines numbers and dashes
ssn_parser('ssn')
ssn_parser.setName("ssn")

input_string = """
    xxx 215-72-8314 yyy
    102-46-6919 zzz 182-19-2201
"""

for matches, start, stop in ssn_parser.scanString(input_string):
    print(matches, start, stop)
```

pp_parse_ssn.py

```
['215728314'] 9 20
['102466919'] 29 40
['182192201'] 45 56
```

Table 7. Builtin Tokens

Token	Description
And	Ordered sequence of required tokens (may be constructed with +)
CaselessKeyword	Like Keyword, but ignore case (result will be same case as match string)
CaselessLiteral	Like Literal, but ignore case (result will be same case as match string)
CharsNotIn	Match characters not in a given set
Combine	Fuse components together
Dict	A scanner for tables
Each	Require components in any order
Empty	Match empty content
FollowedBy	Adding lookahead constraints
Forward	The parser placeholder
GoToColumn	Advance to a specified position in the line
Group	Group repeated items into a list
Keyword	Match a literal string not adjacent to specified context (ie, standalone)
LineEnd	Match end of line
LineStart	Match start of line
Literal	Match a specific string
MatchFirst	Try multiple matches in a given order
NoMatch	A parser that never matches
NotAny	General lookahead condition
OneOrMore	Repeat a pattern one or more times
Optional	Match an optional pattern
Or	Parse one of a set of alternatives
ParseResults	Result returned from a match
QuotedString	Match a delimited string
Regex	Match a regular expression
SkipTo	Search ahead for a pattern
StringEnd	Match the end of the text
StringStart	Match the start of the text
Suppress	Omit matched text from the result
Upcase	Uppercase the result

Token	Description
White	Match whitespace
Word	Match characters from a specified set
WordEnd	Match only at the end of a word
WordStart	Match only at the start of a word
ZeroOrMore	Match any number of repetitions including none

Parsing functions

- parseString
- scanString
- searchString
- transformString

There are four functions that you can call from a parser to do the actual parsing.

parseString parses input text from the beginning of the target string; it ignores extra trailing text.

scanString looks through input text and generates matches.

searchString is like scanString, but returns a list of tokens.

transformString is like scanString, but specifies replacements for tokens

Accessing tokens

- Name tokens for each parser
- Options
  - Assign when parser is defined
  - Set with **setResultsName()**

To make it a little easier to access individual tokens, you can provide names for the tokens, either with the **setResultsName()** function, or by just calling the parser with the name as its argument, which can be done when the parser is defined. Assigning names to tokens is the preferred approach.

Let's say you had a configuration file that looked like this:

```
city=Atlanta
state=Georgia
population=5522942
```

To parse a string in the format "KEY=VALUE", there are 3 components: the key, the equals sign, and the value. When parsing such an expression, you don't really need the equals sign in the results. The Suppress() function will parse a token without putting it in the results.

In the code, the names 'section', 'keylist', 'key', and 'value' (and others) are attributes of the parser results.

Example

pp_parse_ini.py

```
from pyparsing import *

'''
    inifile ::= section+
    section ::= section_tag + section_data
    section_tag ::= '[' alphanums+ ']'
    section_data ::= key_value_pair+
    key_value_pair ::= key '=' value
    key ::= alphanums+
    value ::= chars+
'''

value = Word(' \t' + printables, excludeChars='=')('value')  # A value can be any
character, plus space or tab, _except_ '='
key = Word(alphanums)('key')  # A key can be any alphanumeric character (but not '=')
key_value_pair = Group(key + Suppress('=') + value)  # A key/value pair is key + '=' +
value
section_data = Group(OneOrMore(key_value_pair))('keylist')  # Section data is one or more
key/value pairs
start_tag = Suppress('[')   # Section start tag is '['
end_tag = Suppress(']')   # Section end tag is ']'
section_tag = start_tag + Word(alphanums)('section') + end_tag  # Section tag is start
tag + alphanumerics + end tag
section = Group(section_tag + section_data + Suppress(Optional(White())))  # Section is
section tag + section data
ini_file = OneOrMore(section)  # Complete file is one or more sections

with open('../DATA/application.ini') as ini_in:
    contents = ini_in.read()  # Read contents into a string

for section in ini_file.parseString(contents):  # Parse the string
    print(section.section)  # Grab the section name
    for key, value in section.keylist:   # Iterate over key/value pairs in that section
        print('\t{:10s} {}'.format(key, value))
```

pp_parse_ini.py

```
App
    Vendor     ActiveState
    Name       Komodo Edit
    Version    8.5.4
    BuildID    14424
    Copyright  Copyright (c) 1999 - 2014 ActiveState
    ID         {b1042fb5-9e9c-11db-b107-000d935d3368}
Gecko
    MinVersion 24.0
    MaxVersion 24.0
Shell
    Icon       chrome/icons/default/komodo
XRE
    EnableProfileMigrator 1
    EnableExtensionManager 1
```

Case Study: A URL decomposed

URLs are, of course, frequently used in everyday life. Without them, the Internet would be a vast mishmash. Oh, wait. It already is.

Well, without URLs, the vast mishmash would have no directional signage. In this section, you'll learn how to parse a *complete* URL.

For details on the syntax of a URL, see http://en.wikipedia.org/wiki/URI_scheme#Generic_syntax. A URL consists of the following segments:

- scheme (AKA protocol)
- user information
- host name/address
- port number
- path
- query
- fragment

Most of those segments are optional. Only the scheme and host name/address are required.

Spaces are not allowed in URLs and are translated to a plus sign.

Many punctuation characters are not allowed as well, and are encoded as a percent sign, plus their ASCII value as a two-character hex number.

The allowed set of characters is letters, digits, and underscores, plus the following: -. ~ % +.

The URL parser grammar

An informal top-down BN grammar for parsing URLs:

```
url ::= scheme '://' [userinfo] host [port] [path] [query] [fragment]
scheme ::= http | https | ftp | file
userinfo ::= url_chars+ ':' url_chars+ '@'
host ::= alphanums | host (. + alphanums)
port ::= ':' nums
path ::= url_chars+
query ::= '?' + query_pairs
query_pairs ::= query_pairs | (query_pairs '&' query_pair)
query_pair = url_chars+ '=' url_chars+
fragment = '#' + url_chars
url_chars = alphanums + '-_.~%+'
```

Building a parser for the grammar

Using the grammar, start at the bottom. The smallest components must be defined first. Work your way up, combining components, until you have completely defined the target text:

NOTE Your informal grammar is top-down, but you implement your pyparsing grammar bottom-up.

Example

pp_url_parser.py

```
from pyparsing import *

# http://bob:secret@some.host.com:1234/more/path?p1=v1&p2=v2#junk

'''
URL grammar
   url ::= scheme '://' [userinfo] host [port] [path] [query] [fragment]
   scheme ::= http | https | ftp | file
   userinfo ::= url_chars+ ':' url_chars+ '@'
   host ::= alphanums | host (. + alphanums)
   port ::= ':' nums
   path ::= url_chars+
   query ::= '?' + query_pairs
   query_pairs ::= query_pairs | (query_pairs '&' query_pair)
   query_pair = url_chars+ '=' url_chars+
   fragment = '#' + url_chars
   url_chars = alphanums + '-_.~%+'
'''  # Top-down grammar definition

url_chars = alphanums + '-_.~%+'  # Define legal characters in a URL

fragment  = Combine((Suppress('#') + Word(url_chars)))('fragment')  # Fragment is
trailing information such as a tag ID

scheme = oneOf('http https ftp file')('scheme')   # Scheme is the type of URL
(technically the type of URI) e.g. https
host = Combine(delimitedList(Word(url_chars), '.'))('host')  # Host is the registered
name or IP address of the resource
port = Suppress(':') + Word(nums)('port')   # Port is optional network port of the
resource  e.g. :1234
user_info = (   # User info is optional username/password e.g. USER:PASSWORD@
    Word(url_chars)('username')    # User name contains any legal url characters
    + Suppress(':')                # Skip the ':'
    + Word(url_chars)('password')  # Password contains any legal url characters
    + Suppress('@')                # Skip the '@'
)
query_pair = Group(Word(url_chars) + Suppress('=') + Word(url_chars))  # Query pair is a
key/value pair separated by '='
query = Group(Suppress('?') + delimitedList(query_pair, '&'))('query')  # Query is list
of query pairs separated by ''

path = Combine(
    Suppress('/')
```

```
    + OneOrMore(~query + Word(url_chars + '/'))
)('path')  # Path contains sections delimited by '/'

url_parser = (
    scheme
    + Suppress('://')
    + Optional(user_info)
    + host
    + Optional(port)
    + Optional(path)
    + Optional(query)
    + Optional(fragment)
)   # URL parser combines all of the above in the proper order
```

Example

pp_parse_urls.py

```
from pp_url_parser import url_parser  # import the parser from a module

test_urls = [
  'http://www.notarealsite.com',
  'http://www.notarealsite.com/',
  'http://www.notarealsite.com:1234/',
  'http://bob:%243cr3t@www.notarealsite.com:1234/',
  'http://www.notarealsite.com/presidents',
  'http://www.notarealsite.com/presidents/byterm?term=26&name=Roosevelt',
  'http://www.notarealsite.com/presidents/26',
  'http://www.notarealsite.com/us/indiana/gary/population',
  'ftp://ftp.info.com/downloads',
  'http://www.notarealsite.com#moose',

'http://bob:s3cr3t@www.notarealsite.com:8080/presidents/byterm?term=26&name=Roosevelt#bio
',
]

fmt = '{:10s} {}'

for test_url in test_urls:
  print("URL:", test_url)
  tokens = url_parser.parseString(test_url)  # parse the string from the beginning

  print(tokens, '\n')
  print(fmt.format("Scheme:", tokens.scheme))  # access tokens by name
  print(fmt.format("User name:", tokens.username))
  print(fmt.format("Password:", tokens.password))
  print(fmt.format("Host:", tokens.host))
  print(fmt.format("Port:", tokens.port))
  print(fmt.format("Path:", tokens.path))
  print("Query:")
  for key, value in tokens.query:
    print("\t{} ==> {}".format(key, value))
  print(fmt.format('Fragment:', tokens.fragment))
  print('-' * 60, '\n')
```

pp_parse_urls.py

```
URL:
http://bob:s3cr3t@www.notarealsite.com:8080/presidents/byterm?term=26&name=Roosevelt#bio
['http', 'bob', 's3cr3t', 'www.notarealsite.com', '8080', 'presidents/byterm', [['term',
```

```
'26'], ['name', 'Roosevelt']], 'bio']

Scheme:    http
User name: bob
Password:  s3cr3t
Host:      www.notarealsite.com
Port:      8080
Path:      presidents/byterm
Query:
    term ==> 26
    name ==> Roosevelt
Fragment:  bio
--------------------------------------------------------------
```

...

Taking Action

- Associate action with parser object
- Called on text when it is parsed.
- use **setParseAction()**

Any parser (including the individual parsers that make up the “main” parser) can have an action associated with it. When the parser is used, it calls the function with a list of the scanned tokens.

If the function returns a list of tokens, it replaces the original tokens. If it returns ‘None’, the tokens are not modified.

This can be used to convert numeric strings into actual numbers, to clean up and normalize names, or to completely replace or delete tokens.

Example

pp_fist_full_of_parsers.py

```
from pyparsing import *

def upper_case_it(tokens):  # A function to uppercase every string in a list of strings
    return [t.upper() for t in tokens]

prefix = 'A Fist Full of '  # Fixed-text prefix of what the script is looking for

fist_contents = Word(alphas + ' ')  # Text can be letters or spaces (should fine-tune to
require at least letter)

fist_contents.setParseAction(upper_case_it)  # Call upper_case_it() when targe text found

title_parser = Combine(prefix + fist_contents)  # Complete title is "A Fist Full of" +
more text

titles = (
    'A Fist Full of Dollars',
    'A Fist Full of Spaghetti',
    'A Fist Full of Wombats',
    'A Fist Full of Jelly Beans',
    'A Fist Full of Clint Eastwood Movies',
)

for title in titles:
    print(title_parser.parseString(title)[0])  # Parse the titles of of the string
```

pp_fist_full_of_parsers.py

```
A Fist Full of DOLLARS
A Fist Full of SPAGHETTI
A Fist Full of WOMBATS
A Fist Full of JELLY BEANS
A Fist Full of CLINT EASTWOOD MOVIES
```

Chapter 6 Exercises

Exercise 6-1 (parse_email_addresses.py)

An email address looks like `name@domain`

More specifically, it consists of one or more alphanumeric sequences, separated by dots, then an at sign ('@'), then one or more alphanumeric sequences separated by dots.

Parse the emails in the file **DATA/correspondence.txt** and print them out one per line.

> **NOTE** For simplicity, this lab is glossing over some of the details of valid email addresses, including valid domains, punctuation characters in names, etc.

Chapter 7: 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

- Concurrency
- 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 or *concurrency*.

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

```
from threading import Thread, Lock
import random
import time

STDOUT_LOCK = Lock()

def my_task(num):  # function to run in each thread
    time.sleep(random.randint(1, 3))
    with STDOUT_LOCK:
        print("Hello from thread {}".format(num))


for i in range(16):
    t = Thread(target=my_task, args=(i,))  # create thread
    t.start()  # launch thread

print("Done.")  # "Done" is printed immediately -- the threads are "in the background"
```

thr_noclass.py

```
Done.
Hello from thread 0
Hello from thread 7
Hello from thread 12
Hello from thread 13
Hello from thread 14
Hello from thread 3
Hello from thread 2
Hello from thread 9
Hello from thread 15
Hello from thread 10
Hello from thread 6
Hello from thread 1
Hello from thread 4
Hello from thread 8
Hello from thread 11
Hello from thread 5
```

Creating a thread class

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

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

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

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

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

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

Example

thr_simple.py

```
from threading import Thread, Lock
import random
import time

STDOUT_LOCK = Lock()

class SimpleThread(Thread):
    def __init__(self, num):
        super().__init__()  # call base class constructor -- REQUIRED
        self._threadnum = num

    def run(self):  # the function that does the work in the thread
        time.sleep(random.randint(1, 3))
        with STDOUT_LOCK:
            print("Hello from thread {}".format(self._threadnum))


for i in range(16):
    t = SimpleThread(i)  # create the thread
    t.start()  # launch the thread
```

thr_simple.py

```
Hello from thread 3
Hello from thread 5
Hello from thread 7
Hello from thread 4
Hello from thread 0
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 12
Hello from thread 13
Hello from thread 14
Hello from thread 15
Hello from thread 1
Hello from thread 9
Hello from thread 10
Hello from thread 11
```

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

```
import threading  # see multiprocessing.dummy.Pool for the easier way
import random
import time

WORDS = 'apple banana mango peach papaya cherry lemon watermelon fig elderberry'.split()

MAX_SLEEP_TIME = 3
WORD_LIST = []  # the threads will append words to this list
WORD_LIST_LOCK = threading.Lock()  # generic locks
STDOUT_LOCK = threading.Lock()  # generic locks

class SimpleThread(threading.Thread):
    def __init__(self, num, word):  # thread constructor
        super().__init__()  # be sure to call parent constructor
        self._word = word
        self._num = num

    def run(self):  # function invoked by each thread
        time.sleep(random.randint(1, MAX_SLEEP_TIME))

        with STDOUT_LOCK:  # acquire lock and release when finished
            print("Hello from thread {} ({})".format(self._num, self._word))

        with WORD_LIST_LOCK:  # acquire lock and release when finished
            WORD_LIST.append(self._word.upper())

all_threads = []  # make list ("pool") of threads (but see Pool later in chapter)
for i, random_word in enumerate(WORDS, 1):
    t = SimpleThread(i, random_word)  # create thread
    all_threads.append(t)  # add thread to "pool"
    t.start()  # start thread

print("All threads launched...")

for t in all_threads:
    t.join()  # wait for thread to finish

print(WORD_LIST)
```

thr_locking.py

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

Using queues

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

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

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

Example

thr_queue.py

```
import random
import queue
from threading import Thread, Lock as tlock
import time

NUM_ITEMS = 30000
POOL_SIZE = 128

word_queue = queue.Queue(0)  # initialize empty queue

shared_list = []
shared_list_lock = tlock()  # create locks
stdout_lock = tlock()  # create locks


class RandomWord():  # define callable class to generate words
    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):  # worker thread

    def __init__(self, name):  # thread constructor
        Thread.__init__(self)
        self.name = name

    def run(self):  # function invoked by thread
        while True:
            try:
                s1 = word_queue.get(block=False)  # get next item from thread
                s2 = s1.upper() + '-' + s1.upper()
                with shared_list_lock:  # acquire lock, then release when done
                    shared_list.append(s2)

            except queue.Empty:  # when queue is empty, it raises Empty exception
                break


# fill the queue
random_word = RandomWord()
for i in range(NUM_ITEMS):
    w = random_word()
    word_queue.put(w)

start_time = time.ctime()

# populate the threadpool
pool = []
for i in range(POOL_SIZE):
    worker_name = "Worker {:c}".format(i + 65)
    w = Worker(worker_name)  # add thread to pool
    w.start()  # launch the thread
    pool.append(w)

for t in pool:
    t.join()  # wait for thread to finish

end_time = time.ctime()

print(shared_list[:20])

print(start_time)
print(end_time)
```

thr_queue.py

```
['CONFORMITIES-CONFORMITIES', 'RECTIFIABILITY-RECTIFIABILITY', 'TRIAMCINOLONE-
TRIAMCINOLONE', 'COOFS-COOFS', 'LECHAYIMS-LECHAYIMS', 'DEXIES-DEXIES', 'EXILE-EXILE',
'SNAPPING-SNAPPING', 'SPINNEY-SPINNEY', 'SATYRIDS-SATYRIDS', 'REDACTION-REDACTION',
'CHICOS-CHICOS', 'MANGANOUS-MANGANOUS', 'TERSELY-TERSELY', 'NIDGET-NIDGET', 'LLAMA-
LLAMA', 'MANDARINS-MANDARINS', 'GYROFREQUENCIES-GYROFREQUENCIES', 'TONER-TONER',
'SOKEMAN-SOKEMAN']
Sat Apr  1 09:12:52 2023
Sat Apr  1 09:12:53 2023
```

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

```
import sys
import random
from multiprocessing import Manager, Lock, Process, Queue, freeze_support
from queue import Empty
import time

NUM_ITEMS = 25000  # set some constants
POOL_SIZE = 64
```

```
class RandomWord():  # callable class to provide random words
    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):  # will be called when you call an instance of the class
        return self._words[random.randrange(0, self._num_words)]


class Worker(Process):  # worker class -- inherits from Process

    def __init__(self, name, queue, lock, result):  # initialize worker process
        Process.__init__(self)
        self.queue = queue
        self.result = result
        self.lock = lock
        self.name = name

    def run(self):  # do some work -- will be called when process starts
        while True:
            try:
                word = self.queue.get(block=False)  # get data from the queue
                word = word.upper()  # modify data
                with self.lock:
                    self.result.append(word)  # add to shared result

            except Empty:  # quit when there is no more data in the queue
                break


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

    word_queue = Queue()  # create empty Queue object

    manager = Manager()  # create manager for shared data
    shared_result = manager.list()  # create list-like object to be shared across all
processes
    result_lock = Lock()  # create locks

    random_word = RandomWord()  # create callable RandomWord instance
    for i in range(NUM_ITEMS):
        w = random_word()
        word_queue.put(w)  # fill the queue

    start_time = time.ctime()
```

```
    pool = []  # create empty list to hold processes
    for i in range(POOL_SIZE):  # populate the process pool
        worker_name = "Worker {:03d}".format(i)
        w = Worker(worker_name, word_queue, result_lock, shared_result)  # create worker
process
        #
        w.start()  # actually start the process -- note: in Windows, should only call
X.start() from main(), and may not work inside an IDE
        pool.append(w)  # add process to pool

    for t in pool:
        t.join()  # wait for each queue to finish

    end_time = time.ctime()

    print((shared_result[-50:]))  # print last 50 entries in shared result
    print(len(shared_result))
    print(start_time)
    print(end_time)
```

multi_processing.py

```
['LANTHANUM', 'REEXPORT', 'EMACIATE', 'INOCULATIONS', 'SHIPWRECKING', 'PERFIDIES',
'RETROFLEXIONS', 'WHIRRED', 'CRENELLING', 'GUILLOCHE', 'SUFFOCATIVE', 'HALLMARK',
'CRESS', 'ADAMANCIES', 'METOPONS', 'ERGODICITY', 'GINGELEYS', 'UNOBSERVED', 'PISCINAS',
'ELASTASES', 'SILURIDS', 'REGISTRANTS', 'TRICHOMONIASIS', 'ALOOF', 'MERCURATES',
'SPATIAL', 'HUSSARS', 'STRAITLACEDNESSES', 'YOKES', 'CATACLYSMICALLY', 'LIBERALIZES',
'WHISHTED', 'RAMPAGES', 'LUPINES', 'PALM', 'DEMURRALS', 'EXPRESSIVITY', 'GALACTOSAMINE',
'UNOXYGENATED', 'HIPBONES', 'MISHIT', 'CAMOMILE', 'COALED', 'RELINQUISHMENT', 'TUCHUN',
'CONCRETENESSES', 'EYELET', 'BILHARZIAL', 'DART', 'UTOPISTIC']
25000
Sat Apr  1 09:12:53 2023
Sat Apr  1 09:13:01 2023
```

Using pools

- Provided by `multiprocessing` and `multiprocessing.dummy`
- 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` class 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 of results in the same order as the original data.

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

Example

proc_pool.py

```
import random
from multiprocessing import Pool

POOL_SIZE = 32  # number of processes

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in]  # read word file into a list, stripping off


random.shuffle(WORDS)  # randomize word list


def my_task(word):  # actual task
    return word.upper()


if __name__ == '__main__':
    ppool = Pool(POOL_SIZE)  # create pool of POOL_SIZE processes

    WORD_LIST = ppool.map(my_task, WORDS)  # pass wordlist to pool and get results; map
assigns values from input list to processes as needed

    print(WORD_LIST[:20])  # print last 20 words

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

proc_pool.py

```
['TIDINGS', 'SKETCHBOOKS', 'STYLELESSNESSES', 'WICKERS', 'NONGRAMMATICAL',
'HONORABLENESSES', 'COTRANSFERS', 'TREADS', 'PLEBES', 'ENDOSULFANS', 'MAXIMISE',
'NONRIVAL', 'ABRIDGING', 'PROCARYOTE', 'DRYWALLS', 'WILDFOWLINGS', 'JANITORS', 'TINGED',
'SQUISHIEST', 'OVERFOCUS']
Processed 173462 words.
```

Example

thr_pool.py

```
from multiprocessing.dummy import Pool # get the thread pool object

POOL_SIZE = 32 # set # of threads to create

with open('../DATA/words.txt') as words_in:
    WORDS = [w.strip() for w in words_in] # get list of 175K words

def my_task(word):  # function to apply to each element
    return word.upper()

thread_pool = Pool(POOL_SIZE) # create pool

word_list = thread_pool.map(my_task, WORDS) # map elements across all threads

print(word_list[:20])

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

thr_pool.py

```
['AA', 'AAH', 'AAHED', 'AAHING', 'AAHS', 'AAL', 'AALII', 'AALIIS', 'AALS', 'AARDVARK',
'AARDVARKS', 'AARDWOLF', 'AARDWOLVES', 'AARGH', 'AARRGH', 'AARRGHH', 'AAS', 'AASVOGEL',
'AASVOGELS', 'AB']
Processed 173462 words.
```

Example

thr_pool_mw.py

```
from multiprocessing.dummy import Pool  # .dummy has Pool for threads
import requests
import time

POOL_SIZE = 8

BASE_URL = 'https://www.dictionaryapi.com/api/v3/references/collegiate/json/'  # base url
of site to access

with open('dictionaryapikey.txt') as api_key_in:
    API_KEY = api_key_in.read().rstrip()  # get credentials

SEARCH_TERMS = [  # terms to search for; each thread will search some of these terms
    'wombat', 'pine marten', 'python', 'pearl',
    'sea', 'formula', 'translation', 'common',
    'business', 'frog', 'muntin', 'automobile',
    'green', 'connect','vial', 'battery', 'computer',
    'sing', 'park', 'ladle', 'ram', 'dog', 'scalpel',
    'emulsion', 'noodle', 'combo', 'battery'
]
def main():
    for function in get_data_threaded, get_data_serial:
        start_time = time.perf_counter()
        results = function()
        for search_term, result in zip(SEARCH_TERMS, results):  # iterate over results,
mapping them to search terms
            print("{}:".format(search_term.upper()), end=" ")
            if result:
                print(result)
            else:
                print("** no results **")
        total_time = time.perf_counter() - start_time
        print("{} took {:.2f} seconds\n".format(function.__name__, total_time))

def fetch_data(term):  # function invoked by each thread for each item in list passed to
map()
    try:
        response = requests.get(
            BASE_URL + term,
            params={'key': API_KEY},
        )  # make the request to the site
    except requests.HTTPError as err:
        print(err)
```

```
        return []
    else:
        data = response.json()  # convert JSON to Python structure
        parts_of_speech = []
        for entry in data: # loop over entries matching search terms
            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))  # return list of parsed entries matching
search term


def get_data_threaded():
    p = Pool(POOL_SIZE)  # create pool of POOL_SIZE threads
    return p.map(fetch_data, SEARCH_TERMS)  # launch threads, collect results

def get_data_serial():
    return [fetch_data(w) for w in SEARCH_TERMS]


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

...

Alternatives to `Pool.map()`

- map elements of iterable to multiple task function arguments
- map elements asynchronously
- apply task function to a single value
- apply task function to a single value asynchronously

There are some alternative methods to `Pool.map()`. These apply functions to the data in different patterns, and can be run asynchrously as well.

Table 8. Pool methods

method	multiple args	concurrent	blocks until done	results ordered
`map`	no	yes	yes	yes
`apply`	yes	no	yes	no
`map_async`	no	yes	no	yes
`apply_async`	yes	yes	no	no

.

Alternatives to threading and multiprocessing

- `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.

Async

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.

Twisted

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.

NOTE

See the files named `consume_omdb*.py` and `omdblib.py` in EXAMPLES for examples comparing single-threaded, multi-threaded, multi-processing, and async versions of the same program. There are also examples using `concurrent.futures`, an alternate interface for creating thread or process fpools.

Chapter 7 Exercises

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

Exercise 7-1 (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 7-2 (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 7-3 (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.

NOTE No answer is provided for this exercise.

Chapter 8: Design Patterns

Objectives

- Examine important OOP principles
- Understand why design patterns are useful
- Learn common design patterns
- Implement design patterns in Python

Coupling and cohesion

- Coupling: interdependence of two components
- Cohesion: how well components fit together
- Strive for looser coupling but tighter cohesion

Coupling and **cohesion** are two terms often used when studying design patterns. They refer to the way in which components interact.

Coupling describes how much (or how little) two components depend on each other. The more dependence, the greater the complexity, and the more that one has to change when the other changes. Components are tightly coupled if they share variables, or exchange information that alters their behavior.

Thus, one primary design goal for software components is looser coupling. This means that components can interact, but are independent of each other. An example of this is illustrated by the Strategy pattern.

Cohesion describes how well components fit together. In the ideal world, each component should implement just one function or provide just one data value. The more responsibilities a component has, the less cohesive it is.

Interfaces

- Do not exist as such in Python
- Also known as abstract classes (but not in Java)
- Can be implemented with the abc module

An important software design rule is to program to the **interface**, not the object.

Code which expects a particular object type is tightly coupled to that type, and any changes to the object type will require changes to the code.

For example, rather than building a logger that writes to a text file, it is better to build a logger that writes to an object that acts like, but is not necessarily, a text file. In some languages, we would say that the logger uses the text file interface, but Python does not directly support interfaces.

Being a dynamic language, Python uses "duck" typing, which means that any object providing the appropriate methods can be substituted for the expected object.

Thus, our logger can write to any object that implements a text-file-like interface, in the informal sense.

This is the Pythonic way, and works well, especially if components are well documented.

If you want to ensure that a class implements a particular set of methods, you can use the **abc** module. It allows you to create abstract base classes. These classes may not be directly instantiated, and if abstract methods are not overwritten, an exception is raised upon instantiation.

Whenever you want to enforce that "interface", add the abstract class to the base class list. This is possible because Python supports multiple inheritance.

Example

abstract_base_classes.py

```
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):  # metaclasses control how classes are created; ABCMeta
adds restrictions to classes that inherit from Animal

    @abstractmethod   # when decorated with @abstractmethod, speak() becomes an abstract
method
    def speak(self):
        pass

class Dog(Animal):  # Inherit from abstract base class Animal
    def speak(self):   # speak() must be implemented
        print("woof! woof!")

class Cat(Animal):  # Inherit from abstract base class Animal
    def speak(self):  # speak() must be implemented
        print("Meow meow meow")

class Duck(Animal): # Inherit from abstract base class Animal
    pass  # Duck does not implement speak()

d = Dog()
d.speak()

c = Cat()
c.speak()

try:
    d = Duck()  # Duck throws a TypeError if instantiated
    d.speak()
except TypeError as err:
    print(err)
```

abstract_base_classes.py

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

What are design patterns?

- Reusable solution pattern
- Not a design or specification
- Not an implementation
- A description of how to solve a problem
- A formalized best practice

A **design pattern** is basically a "way of doing something". Design patterns grow from observing the best ways to solve problems.

A design pattern is not a specification for a particular software node; it is a description of how to do something. It is more than an algorithm, because algorithms only describe logic. Design patterns describe the big-picture details of how to implement software for common use cases.

Design patterns became well-known in the computer science world after the publication of the book Design Patterns: Elements of Reusable Object-Oriented Software, published in 1994. The four authors are typically referred to as the "Gang of Four", and the book itself as the "GoF".

Design patterns are associated with object-oriented software, but are not restricted to OOP. The GoF book is pretty dry reading, and the examples are in Smalltalk and C++. For a newer take on design patterns, the following books are excellent:

- Mastering Python Design patterns
- Learning Python Design Patterns
- Head First Design Patterns (book examples are in Java)

NOTE find details for these books in Appendix A – Bibliography

Why use design patterns?

- Tried and true
- Vocabulary for discussing solutions
- Provide structure
- Higher-level than packages of actual code

Design patterns provide workable approaches to common coding problems. They help developers avoid mistakes, because the patterns have been discovered and refined by the actual experience of other programmers.

Another advantage is that design patterns provide a vocabulary for discussing the architecture of a group of related modules (e.g., classes or packages). One programmer can say "I think an Observer would work great for publishing events from our stock market widget", and another programmers will know what that means. It becomes a shorthand for standard approaches.

Remember that design patterns are design patterns. They operate at a higher level than packages and frameworks, which frequently contain specific implementations of patterns.

A real bonus when programming in Python is that many design patterns are already built into the language. Examples of these are Singletons (AKA modules) and Decorators.

Types of patterns

- Four categories of patterns (or so)
  - Creational
  - Structural
  - Behavioral
  - Concurrency

There are four general categories of patterns, organized by how they are used within a project.

Creational patterns are used for software that creates objects and other entities. Well-known creational patterns include Abstract Factory and Singleton.

Structural patterns are used to create relationships between software components. Common structural patterns are Adapter, Proxy, and Flyweight.

Behavioral patterns are used to interact between components. Common behavioral patterns are Observer, Command, and Visitor.

Concurrency patterns are used to help coordinate between components used in multiprogramming. Examples are Double-checked locking, Object Pool, and Active object.

These are not the only ways to organize design patterns. Others have created more categories, or organized them in a totally different hierarchy.

Table 9. Creational Patterns

Pattern	Description
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Factory method	Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses (dependency injection[15]).
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
Multiton	Ensure a class has only named instances, and provide a global point of access to them.
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalization of connection pool and thread pool patterns.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Resource acquisition	Ensure that resources are properly released by tying them to the lifespan of suitable objects.
Singleton	Ensure a class has only one instance, and provide a global point of access to it

Table 10. Structural Patterns

Pattern	Description
Adapter or Wrapper or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to sub-classing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of similar objects efficiently.
Front Controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Twin	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.

Table 11. Behavioral Patterns

Pattern	Description
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Null object	Avoid null references by providing a default object.
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Servant	Define common functionality for a group of classes.
Specification	Recombinable business logic in a Boolean fashion.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Table 12. Concurrency Patterns

Pattern	Description
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
Balking	Only execute an action on an object when the object is in a particular state.
Binding properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.[19]
Block chain	Decentralized way to store data and agree on ways of processing it in a Merkle tree, optionally using Digital signature for any individual contributions.
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.
Event-based asynchronous	Addresses problems with the asynchronous pattern that occur in multi-threaded programs.[20]
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
Join	Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the use of threads and locks, this is a high level programming model.
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[21]
Messaging design pattern (MDP)	Allows the interchange of information (i.e. messages) between components and applications.
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.
Read-write lock	Allows concurrent read access to an object, but requires exclusive access for write operations.
Scheduler	Explicitly control when threads may execute single-threaded code.
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.
Thread-specific storage	Static or "global" memory local to a thread.

Singleton

- Ensures only one instance is created
- Singleton class requires metaprogramming in Python
- Variation called the Borg
- A Python module is already a singleton

The Singleton pattern is used to provide global access to a resource, when it is expensive, or incorrect, to have multiple copies available. The singleton can also control concurrent access.

Typical use cases for the singleton are loggers, spoolers, etc., as well as database or network connections.

There are three standard ways to implement Single classes in Python: using a module (no classes involved), a "classic" singleton, or a Borg.

Module as Singleton

- Modules are only loaded once
- Global variables in module are available after import

Since a python module is already, by design, a singleton. Rather than creating a Singleton class, you can just put data and methods in a normal module, and the module can be imported from anywhere in the application. The data and methods will only exist in one place, and there will never be more than one instance. This is useful for simple cases, but it is limited in some ways.

Example

designpatterns/singletons/singletonmodule/dbconn.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# only one connection and one cursor will be created

import sqlite3

db_connection = sqlite3.connect(':memory:')

db_cursor = db_connection.cursor()
```

Example

designpatterns/singletons/singletonmodule/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from dbconn import db_cursor
from builddb import build_database

build_database()

db_cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonmodule/singleton_main.py

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

Classic Singleton

- Override *new*
- *init* is still called for each instance

To create a classic singleton, override the *new* method of a class. This method is normally not used in creating classes, because the *init*() takes the role of constructor, but *new*() is the "real" constructor, in that it returns the actual new object when an instance is requested.

One thing to be careful about is that *init*() is still called for each instance, so put a test in *init*() to make sure you're only initializing the instance once.

Example

designpatterns/singletons/singletonclassic/databaseconnection.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# only one connection and one cursor will be created

import sqlite3

class DatabaseConnection(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(DatabaseConnection, cls).__new__(cls)
        return cls.instance

    def __init__(self):
        # IMPORTANT! One-time setup
        if not hasattr(self, '_db_connection'):
            self._db_connection = sqlite3.connect(':memory:')
            self._db_cursor = self._db_connection.cursor()

    @property
    def cursor(self):
        return self._db_cursor
```

Example

designpatterns/singletons/singletonclassic/builddb.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# even though we are importing DatabaseConnection, it does not
# create a new connection
from databaseconnection import DatabaseConnection

db_conn = DatabaseConnection()  # get the instance

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'),
]

def build_database():
    db_conn.cursor.execute('''
        create table computer_people (
            first_name varchar(16),
            last_name varchar(16),
            known_for varchar(16)
        );
    ''')

    insert_query = '''
        insert into computer_people
        (first_name, last_name, known_for)
        values (?, ?, ?);
    '''

    db_conn.cursor.executemany(insert_query, PEOPLE)
```

Example

designpatterns/singletons/singletonclassic/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from databaseconnection import DatabaseConnection

from builddb import build_database

build_database()

db_conn = DatabaseConnection()

db_conn.cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_conn.cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonclassic/singleton_main.py

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

The Borg

- Singleton that creates multiple instances
- BUT, each instance shares same state with all others
- Allows inheritance from singleton
- AKA monostate

If you don't need to inherit from a Singleton, then the classic implementation is good. However, when you need to inherit, you can use the Borg implementation, named for the alien race on Star Trek TNG.

In the Borg, each instance is a distinct object, but all of the instances share the same state.

This is done by sharing data dictionaries at the class level. The *new*() method returns a new instance object, but replaces the default dictionary with the shared one. Thus, all Borg instances really share the same set of attributes.

Example

designpatterns/singletons/singletonborg/databaseconnection.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

import sqlite3

class DatabaseConnection(object):
    _shared = {}

    def __new__(cls, *args, **kwargs):
        instance = super(DatabaseConnection, cls).__new__(cls, *args, **kwargs)

        if not '_db_connection' in cls._shared:
            db_conn = sqlite3.connect(':memory:')
            db_cursor = db_conn.cursor()

            cls._shared['_db_connection'] = db_conn
            cls._shared['_db_cursor'] = db_cursor

        instance.__dict__ = cls._shared


        return instance

    @property
    def cursor(self):
        return self._db_cursor
```

Example

designpatterns/singletons/singletonborg/builddb.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

# even though we are importing DatabaseConnection, it does not
# create a new connection
from databaseconnection import DatabaseConnection

db_conn = DatabaseConnection()  # get the instance

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'),
]

def build_database():
    db_conn.cursor.execute('''
        create table computer_people (
            first_name varchar(16),
            last_name varchar(16),
            known_for varchar(16)
        );
    ''')

    insert_query = '''
        insert into computer_people
        (first_name, last_name, known_for)
        values (?, ?, ?);
    '''

    db_conn.cursor.executemany(insert_query, PEOPLE)
```

Example

designpatterns/singletons/singletonborg/singleton_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from databaseconnection import DatabaseConnection

from builddb import build_database

build_database()

db_conn = DatabaseConnection()

db_conn.cursor.execute(
    '''
        select first_name, last_name
        from computer_people
    '''
)

for row in db_conn.cursor.fetchall():
    print(' '.join(row))
```

designpatterns/singletons/singletonborg/singleton_main.py

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

Strategy

- Encapsulate algorithms to make them interchangeable
- Isolate the things that vary from the things that don't
- Provides an interface for behavior
- Encourages loose coupling

The Strategy pattern is used when you have behaviors that may vary, but the use of the behavior stays the same in a client.

The client may want to sort, but we may want to be able to change the sort algorithm independently, from quicksort to mergesort to Timsort. Strategy decouples the sorting algorithm from the component that wants to sort.

The client gets and instance of the algorithm component, which is typically a class written to an API (an interface, in some languages). As long as the class provides the correct API, the client doesn't care about the algorithm's details.

Another example is saving a file from a word processor. The interface can select a class that saves in native format, XML, PDF, plain text, or translates to Klingon. All it has to do is get an instance of NativeWriter, XMLWriter, PDFWriter, PlainTextWriter, or KlingonWriter. No matter what kind of writer, it just passes the current document to the writer's write() method. Of course in this case the writer component will need to be able to understand the in-memory document format.

Example

designpatterns/strategies/simuduck/flylib.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from abc import ABCMeta, abstractmethod

class FlyBase(object, metaclass=ABCMeta):
    @abstractmethod
    def fly(self):
        pass

class FlyWithWings(FlyBase):
    def fly(self):
        print("Soaring on my wings")

class Flightless(FlyBase):
    def fly(self):
        print("Gee, I can't fly")

if __name__ == '__main__':
    fww = FlyWithWings()
    fww.fly()

    fl = Flightless()
    fl.fly()
```

Example

designpatterns/strategies/simuduck/quacklib.py

```
#!/usr/bin/env python
# (c)2015 John Strickler
from abc import ABCMeta, abstractmethod

class QuackBase(object, metaclass=ABCMeta):
    @abstractmethod
    def quack(self):
        pass

class Quack(QuackBase):
    def quack(self):
        print('"Quack, quack"')

class Squeak(QuackBase):
    def quack(self):
        print('"Squeaky-squeaky"')

class MuteQuack(QuackBase):
    def quack(self):
        print("<I can't make any sounds>")

if __name__ == '__main__':
    q = Quack()
    q.quack()

    s = Squeak()
    s.quack()

    m = MuteQuack()
    m.quack()
```

Example

designpatterns/strategies/simuduck/duck.py

```
#!/usr/bin/env python
# (c)2015 John Strickler
import flylib
import quacklib

class Duck(object):

    def __init__(
            self,
            duck_type,
            quack=None,
            fly=None
    ):
        self._duck_type = duck_type
        self._quack_behavior = quacklib.Quack() if quack is None else quack
        self._fly_behavior = flylib.FlyWithWings() if fly is None else fly


    def swim(self):
        print("Swimming")

    def display(self):
        print("I am a", self._duck_type)

    def set_quack(self, quack):
        if not issubclass(quack, quacklib.QuackBase):
            raise TypeError("Invalid quack")
        self._quack_behavior = quack

    def set_fly(self, fly):
        if not issubclass(fly, flylib.FlyBase):
            raise TypeError("Invalid flight type")
        self._fly_behavior = fly

    def quack(self):
        self._quack_behavior.quack()

    def fly(self):
        self._fly_behavior.fly()
```

Example

designpatterns/strategies/simuduck/simuduck_main.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

from duck import Duck
from quacklib import MuteQuack, Squeak
from flylib import Flightless

d1 = Duck('Mallard')
d2 = Duck('Robot', quack=MuteQuack())
d3 = Duck('Rubber Duckie', quack=Squeak(), fly=Flightless())

for d in d1, d2, d3:
    d.display()
    d.quack()
    d.fly()
    print('-' * 60)
```

designpatterns/strategies/simuduck/simuduck_main.py

```
I am a Mallard
"Quack, quack"
Soaring on my wings
------------------------------------------------------------
I am a Robot
<I can't make any sounds>
Soaring on my wings
------------------------------------------------------------
I am a Rubber Duckie
"Squeaky-squeaky"
Gee, I can't fly
------------------------------------------------------------
```

Decorators

- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take parameters (but not required to)
- functools.wraps() preserves function's properties

In Python, decorators are a special case. Not only are decorators provided by the standard library, such as property() or classmethod(), but they have a special syntax. The @ sign is used to apply a decorator to a function or class.

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the @app.route() decorator in Flask maps a URL to a view function.

Table 13. Decorators in the standard library

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

Using decorators

- Provide a wrapper around a function or class
- Syntax

```
@decorator
def function():
    function_body
```

A decorator is a function or class that acts as a wrapper around a function or class. It allows you to modify the target without changing the target itself.

The @property, @classmethod, and @staticmethod decorators were described in the chapter on OOP.

Table 14. Decorator implementations

Implemented as (wrapper)	Decorates (target)	Takes params	Implementation [1]	Example as function call (without @)
function	function	No	Decorator returns replacement	target = decorator(target)
function	function	Yes	Decorator returns wrapper function Wrapper returns replacement function	target = decorator(params)(target)
class	function	No	__call__ IS replacement function	target = decorator(target)
class	function	Yes	__call__ RETURNS replacement function	target = decorator(params)(target)
function	class	No	Decorator modifies and returns original class	target = decorator(target)
function	class	Yes	Decorator returns wrapper Wrapper modifies and returns original class	target = decorator(params)(target)
class	class	No	__new__ returns original class	target = decorator(target)
class	class	Yes	__call__ returns original class	target = decorator(params)(target)

[1]In all cases, target is decorated with @decorator or @decorator(params). The example in italics is equivalent to the @ syntax, but is not normally used.

NOTE See the file **EXAMPLES/decorators/decorama.py** for examples of the above 8 different decorator implementations.

Creating decorator functions

- Decorator function gets original function
- Use functools.wraps

A decorator function is passed the target object (function or class), and returns a replacement object.

A simple decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code.

A decorated function, like

```
@mydecorator
def myfunction():
    pass
```

is really called like this

```
myfunction = mydecorator(myfunction)
```

The new function should be defined with generic arguments so it can handle the original function's arguments.

The replacement function should be decorated with functools wraps. This makes sure the replacement function keeps the name (and other attributes) of the original function.

Example

designpatterns/decorators/deco_print_name.py

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

from functools import wraps

def print_name( old_func ):

    @wraps(old_func)
    def new_func( *args, **kwargs ):
        # added functionality
        print("==> Calling function {0}".format(old_func.__name__))
        return old_func( *args, **kwargs )  # call the 'real' function

    return new_func   # return the new function object


@print_name
def hello():
    print("Hello!")

@print_name
def goodbye():
    print("Goodbye!")

hello()
goodbye()
hello()
goodbye()
```

designpatterns/decorators/deco_print_name.py

```
==> Calling function hello
Hello!
==> Calling function goodbye
Goodbye!
==> Calling function hello
Hello!
==> Calling function goodbye
Goodbye!
```

Decorators with arguments

If the decorator itself needs arguments, then things get a little bit trickier. The decorator function gets the arguments, and returns a wrapper function that gets the original function (the one being decorated), and the wrapper function then returns the replacement function.

That is,

```
@mydecorator(param)
def myfunction():
    pass
```

is really called like this

```
myfunction = mydecorator(param)(myfunction)
```

Example

designpatterns/decorators/deco_print_name_with_label.py

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

from functools import wraps

def print_name(label):

    def wrapper(old_func):

        @wraps(old_func)
        def new_func( *args, **kwargs ):
            # added functionality
            print("{0}: function {1}".format(
                label,
                old_func.__name__
            ))
            return old_func( *args, **kwargs )  # call the 'real' function

        return new_func   # return the new function object
    return wrapper

@print_name('HELLO')
def hello():
    print("Hello!")

@print_name('HELLO')
def howdy():
    print("Howdy!")

@print_name('GOODBYE')
def goodbye():
    print("Goodbye!")

@print_name('GOODBYE')
def solong():
    print("So long!")


hello()
howdy()
goodbye()
solong()
```

designpatterns/decorators/deco_print_name_with_label.py

```
HELLO: function hello
Hello!
HELLO: function howdy
Howdy!
GOODBYE: function goodbye
Goodbye!
GOODBYE: function solong
So long!
```

Adapters

- Help make two incompatible interfaces compatible
- Prevent making changes to either interface
- Work with objects after implementation
- Adapt one interface to the other

The Adapter pattern is used to make two incompatible interfaces compatible. This is needed when you are not able to change the software on one side or the other. For instance, you're using a third-party library that you can't change. Your software was written for an older version of the library, and when they update the library, suddenly your code is broken. While in the ideal world, you could update the actual component that needs the library, an Adapter can let your existing code work with the updated library.

It can also be used for data conversions. Let's say you're getting data from a web service, but the units are kilometers and you need them in miles. You can create an adapter that connects to the original service, and converts the data before returning it to the requesting component.

An Adapter takes an instance of the caller, and uses the information in the instance to make the appropriate calls to the callee.

In the Java library, the IO readers and writers are adapters.

A Proxy is similar to an Adapter, but the Proxy presents the same interface as the component it represents, while an Adapter does not.

Example

designpatterns/adapters/cat_adapter.py

```
#!/usr/bin/env python
# (c)2015 John Strickler

class Cat(object):
    def meow(self):
        print("Meow!!")

class Dog(object):
    def bark(self):
        print("Arf arf!!")

class CatAdapter(Dog):
    def __init__(self, cat):
        self._cat = cat

    def bark(self):
        self._cat.meow()

Garfield = Cat()

dog = CatAdapter(Garfield)
dog.bark()
```

designpatterns/adapters/cat_adapter.py

```
Meow!!
```

Abstract Factory

- Create families of objects
- Meta-factory (factory of factories)
- Provide various objects that share API
- Alias: Kit
- Use cases: – System should be independent of how products are created – System configured with one of multiple families of products – Enforce constraints on family of related products – Create abstract classes for product family

Participating objects: Abstract Factory, Factory, Abstract Product, Product, Client

An Abstract Factory is a creational pattern that is useful when you want to decouple the creation of classes from an application (or other component).

An Abstract Factory can be considered a meta-factory in that it typically returns a specific factory which creates the desired object. The client code passes a parameter to the Abstract Factory to tell it which factory to return.

While a Factory uses inheritance, an Abstract Factory uses composition.

The official GoF definition of the Abstract Factory is:

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”

Builder

- Separate construction from representation
- Creates object composed of multiple parts
- Object not complete until all parts created
- Same builder can create multiple representations
- Use cases
  - Product creation algorithm should be independent of parts and assembly
  - Construction process should allow different representations of product

Participating objects: Builder Mixin, Builder, Director, Product

A Builder is a creational pattern that separates the construction of a product from its representation. A builder object creates the multiple parts of the target object. A director object controls the building of an object by using an instance of a builder object.

The representation of the product can vary, but the process remains the same.

Example

builder_pattern.py

```
#
from abc import ABCMeta, abstractmethod


class Director(object):

    def construct(self, builder):
        '''
        Builder uses multiple steps
        :param builder: A Builder object
        :return:
        '''
        builder.build_part_A()
        builder.build_part_B()

class BuilderBase(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def build_part_A(self): pass

    @abstractmethod
    def build_part_B(self): pass

    @abstractmethod
    def get_result(self): pass

class Product(object):
    def __init__(self, name):
        self._name = name
        self._parts = []

    def add(self, part):
        self._parts.append(part)

    def show(self):
        print("{} Parts -------".format(self._name))
        for part in self._parts:
            print(part)
        print()

class Builder1(BuilderBase):
    def __init__(self):
```

```
        self._product = Product("Product 1")

    def build_part_A(self):
        self._product.add("PartA")

    def build_part_B(self):
        self._product.add("PartB")

    def get_result(self):
        return self._product


class Builder2(BuilderBase):
    def __init__(self):
        self._product = Product("Product 2")

    def build_part_A(self):
        self._product.add("PartX")

    def build_part_B(self):
        self._product.add("PartY")

    def get_result(self):
        return self._product

if __name__ == '__main__':
    director = Director()
    builder1 = Builder1()
    builder2 = Builder2()

    director.construct(builder1)
    product1 = builder1.get_result()
    product1.show()

    director.construct(builder2)
    product2 = builder2.get_result()
    product2.show()
```

builder_pattern.py

```
Product 1 Parts -------
PartA
PartB

Product 2 Parts -------
PartX
PartY
```

Facade

- Simplified front end to a system
- Hides the details of a complex component
- Implemented as class or function
- Use cases – Simple interface for complex subsystem – Decouple subsystem from other subsystems – Create layered subsystems

Participating objects: Facade, subsystem classes

A Facade is a design pattern that creates a simplified interface for a complex component. It essentially hides the details of a system.

For instance, the requests module is a Facade for the urllib.* packages. requests makes proxies, authentication, and sending data much easier than doing everything individually.

Another name for a Facade could be "wrapper". It "wraps" some other component. One downside to a Facade is that it might not provide access to all the features available in the target component. A Facade provides a new interface to the component.

Example

csv_facade.py

```
#
"""
Demo of the Facade design pattern
"""
import csv

PRESIDENTS_FILE = '../DATA/presidents.csv'

class CSVUpper():
    """
    A facade for a CSV file
    """
    def __init__(self, csv_file):
        self._file_in = open(csv_file)
        self._rdr = csv.reader(self._file_in)

    def __iter__(self):
        return self

    def __next__(self):
        row = next(self._rdr)
        if row:
            return [r.upper() for r in row]
        else:
            raise StopIteration()

    def __del__(self):
        self._file_in.close()

if __name__ == '__main__':
    presidents = CSVUpper(PRESIDENTS_FILE)
    for p in presidents:
        print(p)
```

csv_facade.py

```
['1', 'GEORGE', 'WASHINGTON', 'WESTMORELAND COUNTY', 'VIRGINIA', 'NO PARTY']
['2', 'JOHN', 'ADAMS', 'BRAINTREE, NORFOLK', 'MASSACHUSETTS', 'FEDERALIST']
['3', 'THOMAS', 'JEFFERSON', 'ALBERMARLE COUNTY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['4', 'JAMES', 'MADISON', 'PORT CONWAY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['5', 'JAMES', 'MONROE', 'WESTMORELAND COUNTY', 'VIRGINIA', 'DEMOCRATIC - REPUBLICAN']
['6', 'JOHN QUINCY', 'ADAMS', 'BRAINTREE, NORFOLK', 'MASSACHUSETTS', 'DEMOCRATIC -
REPUBLICAN']
['7', 'ANDREW', 'JACKSON', 'WAXHAW', 'SOUTH CAROLINA', 'DEMOCRATIC']
['8', 'MARTIN', 'VAN BUREN', 'KINDERHOOK', 'NEW YORK', 'DEMOCRATIC']
['9', 'WILLIAM HENRY', 'HARRISON', 'BERKELEY', 'VIRGINIA', 'WHIG']
['10', 'JOHN', 'TYLER', 'CHARLES CITY COUNTY', 'VIRGINIA', 'WHIG']
['11', 'JAMES KNOX', 'POLK', 'MECKLENBURG COUNTY', 'NORTH CAROLINA', 'DEMOCRATIC']
['12', 'ZACHARY', 'TAYLOR', 'ORANGE COUNTY', 'VIRGINIA', 'WHIG']
['13', 'MILLARD', 'FILLMORE', 'CAYUGA COUNTY', 'NEW YORK', 'WHIG']
['14', 'FRANKLIN', 'PIERCE', 'HILLSBORO', 'NEW HAMPSHIRE', 'DEMOCRATIC']
['15', 'JAMES', 'BUCHANAN', 'COVE GAP', 'PENNSYLVANIA', 'DEMOCRATIC']
['16', 'ABRAHAM', 'LINCOLN', 'HODGENVILLE, HARDIN COUNTY', 'KENTUCKY', 'REPUBLICAN']
['17', 'ANDREW', 'JOHNSON', 'RALEIGH', 'NORTH CAROLINA', 'REPUBLICAN']
['18', 'ULYSSES SIMPSON', 'GRANT', 'POINT PLEASANT', 'OHIO', 'REPUBLICAN']
['19', 'RUTHERFORD BIRCHARD', 'HAYES', 'DELAWARE', 'OHIO', 'REPUBLICAN']
['20', 'JAMES ABRAM', 'GARFIELD', 'ORANGE, CUYAHOGA COUNTY', 'OHIO', 'REPUBLICAN']
```

Example

remote_cmd_facade.py

```
import paramiko
REMOTE_HOST = 'localhost'
REMOTE_USER = 'python'
REMOTE_PASSWORD = 'l0lz'

class RemoteConnection():
    def __init__(self, host, user, password):
        self._host = host
        self._user = user
        self._password = password
        self._connect()

    def _connect(self):
        self._ssh = paramiko.SSHClient()
        self._ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        self._ssh.connect(self._host, username=self._user, password=self._password)

    def run_command(self, command_line):
        stdin, stdout, stderr = self._ssh.exec_command(command_line)
        stdout_text = stdout.read()
        stderr_text = stderr.read()
        return stdout_text, stderr_text

    def __del__(self):
        self._ssh.close()

if __name__ == '__main__':
    rc = RemoteConnection(REMOTE_HOST, REMOTE_USER, REMOTE_PASSWORD)

    stdout, stderr = rc.run_command('whoami')
    print(stdout.decode(), '\n')

    stdout, stderr = rc.run_command('grep root /etc/passwd')
    print("STDOUT:", stdout.decode())
    print("STDERR:", stderr.decode())

    print('-' * 60)

    stdout, stderr = rc.run_command('grep root /etc/pizza')
    print("STDOUT:", stdout.decode())
    print("STDERR:", stderr.decode())
```

remote_cmd_facade.py

```
python


STDOUT: root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_cvmsroot:*:212:212:CVMS Root:/var/empty:/usr/bin/false

STDERR:
------------------------------------------------------------
STDOUT:
STDERR: grep: /etc/pizza: No such file or directory
```

Flyweight

- Many small objects that share state
- E.g. characters in a word processor
- Use cases
  - Applications that need a large number of objects
  - Number of objects would drive up storage costs
  - Object state can be easily made external
  - Factor out duplicate state among many objects
  - Application doesn’t require unique objects

Participating objects: Flyweight Mixin, Flyweight, Unshared Flyweight, Flyweight Factory, Client

While it is useful to have objects at a fundamental level of granularity, it can be expensive in terms of memory usage, and possibly performance.

As an example, consider a word processor. It would be nice to represent individual characters as objects, but it would take large amounts of memory if there were separate objects for every character in a large document.

The Flyweight pattern allows you to create many instances of an objects, but instances can share state. In the word processor example, all instances of the letter "m" would really be the same object. Any context-dependent information is calculated by or passed in from the clients (i.e., objects using the FlyWeight objects.

There are several ways to implement a Flyweight in Python. In all cases, there is a repository of instances, and instances are only created via a factory. In the example, a decorator is used to turn a class into a Flyweight, and the class itself is the factory via the *call*method.

Each time an instance is needed, it is either drawn from the dictionary of instances, or created and added to the dictionary. The dictionary acts as a cache of the state of all instances.

NOTE	For another example of using the FlyWeight decorator, see bigcat.py in the EXAMPLES folder

Command

- Encapsulate an operation
- Decoupled from the invoker
- Commands can be grouped
- Use cases – Parameter objects by task – Queue and execute tasks at different times – Support undo operations – Log changes to a system for replay – Transaction-based systems

Participating objects Command Mixin, Command, Client, Invoker, Receiver

The command pattern encapsulates an operation (or request). The code that executes the command is known as the invoker, and does not have to know the implementation details of the command object; it only needs to know the API, which can be inherited from an abstract base class to make sure the command object implements the necessary methods.

Individual command objects may be queued, and then executed in order at a specified time.

Example

command_pattern.py

```
#
from abc import ABCMeta, abstractmethod

class Command(metaclass=ABCMeta):
    '''Command "interface"'''

    @abstractmethod
    def execute(self): pass


class Fan:

    def start_rotate(self):
        print("Fan is rotating")

    def stop_rotate(self):
        print("Fan is not rotating")


class Light:

    def turn_on(self):
```

```
        print("Light is on")


    def turn_off(self):
        print("Light is off")


class Switch:

    def __init__(self, on, off):
        self.on_command = on
        self.off_command = off

    def on(self):
        self.on_command.execute()

    def off(self):
        self.off_command.execute()


class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()


class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()


class FanOnCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
        self.fan.start_rotate()

class FanOffCommand(Command):
    def __init__(self, fan):
        self.fan = fan

    def execute(self):
```

```
        self.fan.stop_rotate()

if __name__ == '__main__':
    light = Light()
    light_on_command = LightOnCommand(light)
    light_off_command = LightOffCommand(light)

    light_switch = Switch(light_on_command, light_off_command)
    light_switch.on()
    light_switch.off()

    fan = Fan()
    fan_on_command = FanOnCommand(fan)
    fan_off_command = FanOffCommand(fan)

    fan_switch = Switch(fan_on_command, fan_off_command)
    fan_switch.on()
    fan_switch.off()
```

command_pattern.py

```
Light is on
Light is off
Fan is rotating
Fan is not rotating
```

Mediator

- Controls interaction among related objects
- Similar to facade, but two-way
- Objects don't interact directly
- Use cases – Organize complex interdependence of objects – Reuse an object that interacts with other objects – Configure multi-object behavior

Participants

```
Mediator Mixin, Mediator, Colleague Classes
```

A mediator encapsulates interaction among related classes. Mediators are similar to facades, but are two-way, where a facade is generally one-way. A mediator knows the behavior of multiple objects, so that every object doesn't have to know about every other object. It reduces the coupling within a system that has many objects.

Chapter 8 Exercises

Exercise 8-1 (deco_timer.py)

Create a decorator named functiontimer that can be applied to a function and reports the number of seconds (fractional) it takes for the function to execute. You can get the time from the time.time() function. (i.e., import time, and call the time() function from it.)

Just subtract the start time from the end time and report it. You don't need any further calculations.

Write some functions and test your decorator on them.

Appendix A: Python Bibliography

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

Title	Author	Publisher
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, 3rd. 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		
Python Testing Cookbook	Greg L. Turnquist	Packt Publishing
Learning Python Testing	Daniel Arbuckle	Packt Publishing

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

Index

@

__call__(, 123
__init__(, 123
__iter__, 156
__new__(, 123
__next__, 156
__prepare__(, 123

A

abc module, 213
asynchronous communication, 184
asyncio, 209
attributes, 94

B

builtin types
    emulating, 42

C

C struct, 17
callable, 101
callbacks, 58
`chain()`, 77
`chain.from_iterable()`, 77
class
    defining at runtime, 115
cohesion, 212
collections, 133
`collections.Counter`, 15
collections.ordereddict, 11
combinations(), 82
concurrency, 184
`concurrent.futures`, 209
Container classes, 33
copy.deepcopy, 8
coroutine, 153
`count()`, 75
`Counter`, 15
Counter, 35
Coupling, 212
`cycle()`, 75

D

decorator class, 109
decorator function, 106
decorator parameters, 113
decorators, 101
decorators in the standard library, 102, 238
deep copying, 8
defaultdict, 35
delattr(), 94
deque, 35
design pattern, 215
`dict`, 34
dictionary
    custom, 51
Django framework, 123
`dropwhile()`, 77

E

Element, 54

F

for, 133
for loop, 143
functools.wraps, 106

G

generator, 133, 151
generator class, 147, 156
generator expression, 147, 148
generator expressions, 63
generator function, 147
generator.send(), 153
generators
    how to create, 147
generators, 147
getattr(), 94
GIL, 186
globals(), 87
`groupby()`, 80

H

hasattr(), 94

Higher-order functions, 57

I

`inspect` module, 90
interface, 213
`islice()`, 75
iterable, 133

L

lambda function, 58
`list`, 34
list, 48
list comprehension, 144
list comprehensions, 63
lists
    custom, 50
locals(), 87

M

make_archive, 27
map(, 64
`map()`, 57, 63
mapreduce, 65
metaclass, 122, 123
metaclasses, 121
metaprogramming, 86
monkey patches, 118
multiprocessing, 203
multiprocessing, 184
    Manager, 200
multiprocessing module, 200
multiprocessing.dummy, 203
multiprocessing.dummy.Pool, 203
multiprocessing.Pool, 203
multiprogramming, 184
    alternatives to, 209

N

`namedtuple`, 17
namedtuple, 35

O

`operator`, 61
OrderedDict, 35
ordereddict, 11

P

parsers, 162
parseString, 165, 169
`partial()`, 68
permutations(), 82
pipelines, 153
`pprint`, 20
`pprint.pprint()`, 20
product(), 82
pyparsing, 161
pyparsing, 162

R

`reduce()`, 57, 63, 65
regular expressions, 161
`repeat()`, 75

S

scanString, 169
searchString, 169
`set`, 34
setattr(), 94
setResultsName{lparen}{rparen}, 170
shallow copying, 8
shutil, 27
`singledispatch`, 71
special methods, 42
StopIteration, 156
`struct`, 39

T

`takewhile()`, 77
thread, 185
thread class
    creating, 190
threading, 184
threading module, 187
threading.Thread, 187
threads
    debugging, 199
    locks, 192
    queue, 195
    simple, 188
    variable sharing, 192

transformString, 169
`tuple`, 34
Twisted, 209
type, 122
type(), 115

U

unpacking function arguments, 134

Y

yield, 151