Intensive Advanced

Python

Course Objectives

Advance knowledge and proficiency at using Python and the standard library

Explore data analytic frameworks

Examine advanced class features

Increase awareness and effectiveness using various third-party tools including Django

Initial Notes on Installation and Setup

- *We will need three tools for this course*

Python 3.10.x or later

(*https://www.python.org/downloads*)
During the install, look for the option to add it to your path

PyCharm Community Edition (free)

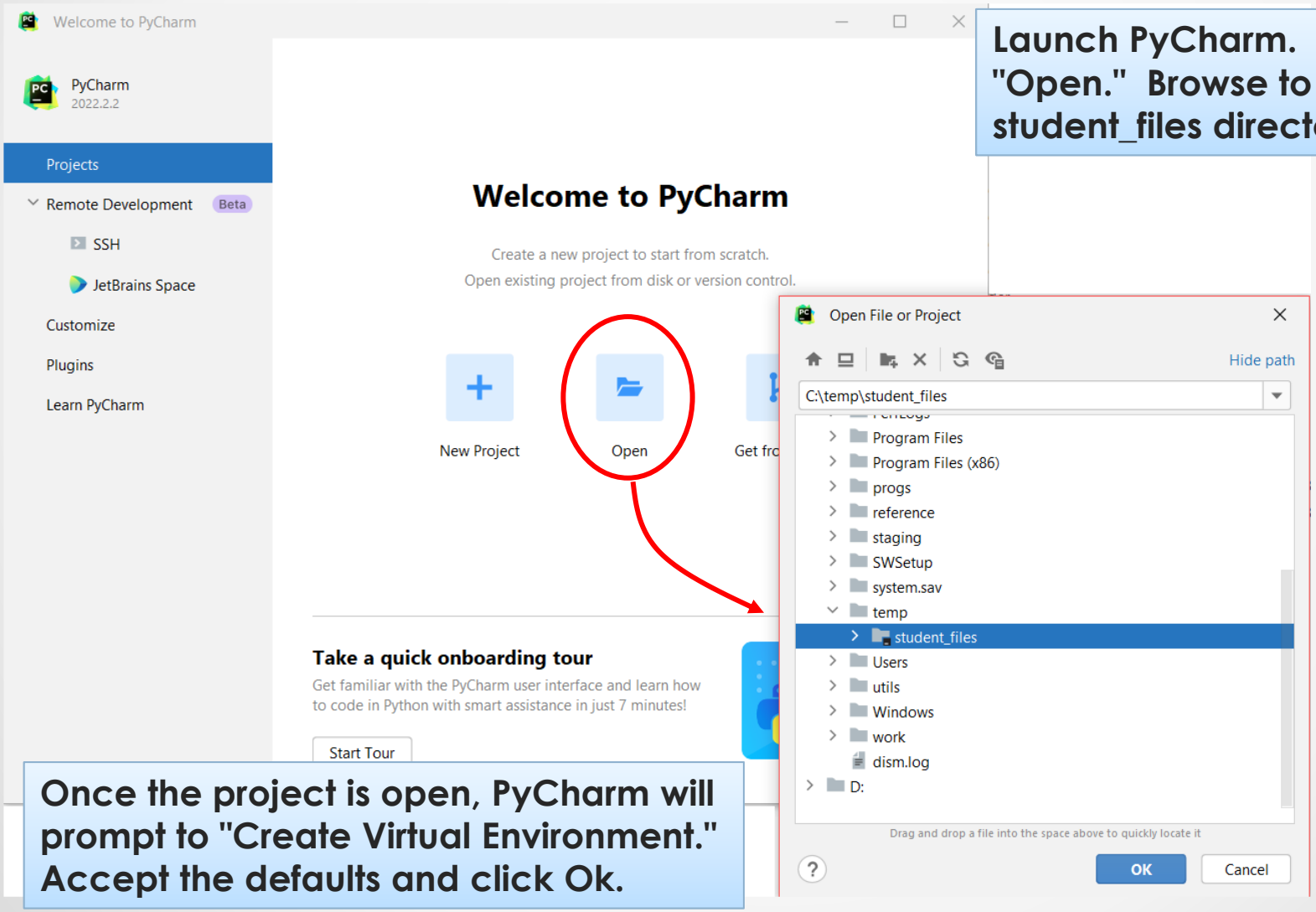
(*https://www.jetbrains.com/pycharm/*)

Student resources

A link (usually found in your email) to download the student files will be provided to you

Post-Installation Setup

- After installing each of the three previous items



Launch PyCharm. Select "Open." Browse to the student_files directory. Click Ok.

Once the project is open, PyCharm will prompt to "Create Virtual Environment." Accept the defaults and click Ok.

Course Agenda - Day 1

Day 1

Review of Intermediate Concepts

Automation and Standard Library Modules

Course Agenda - Day 2

Day 2

Automation *(continued)*

Advanced Class Design

Unittest and Mock Modules

Course Agenda - Day 3

Day 3

Introducing Data Analysis

- NumPy and Matplotlib
- Pandas Fundamentals
- Data Cleansing with Pandas

Course Agenda - Day 4

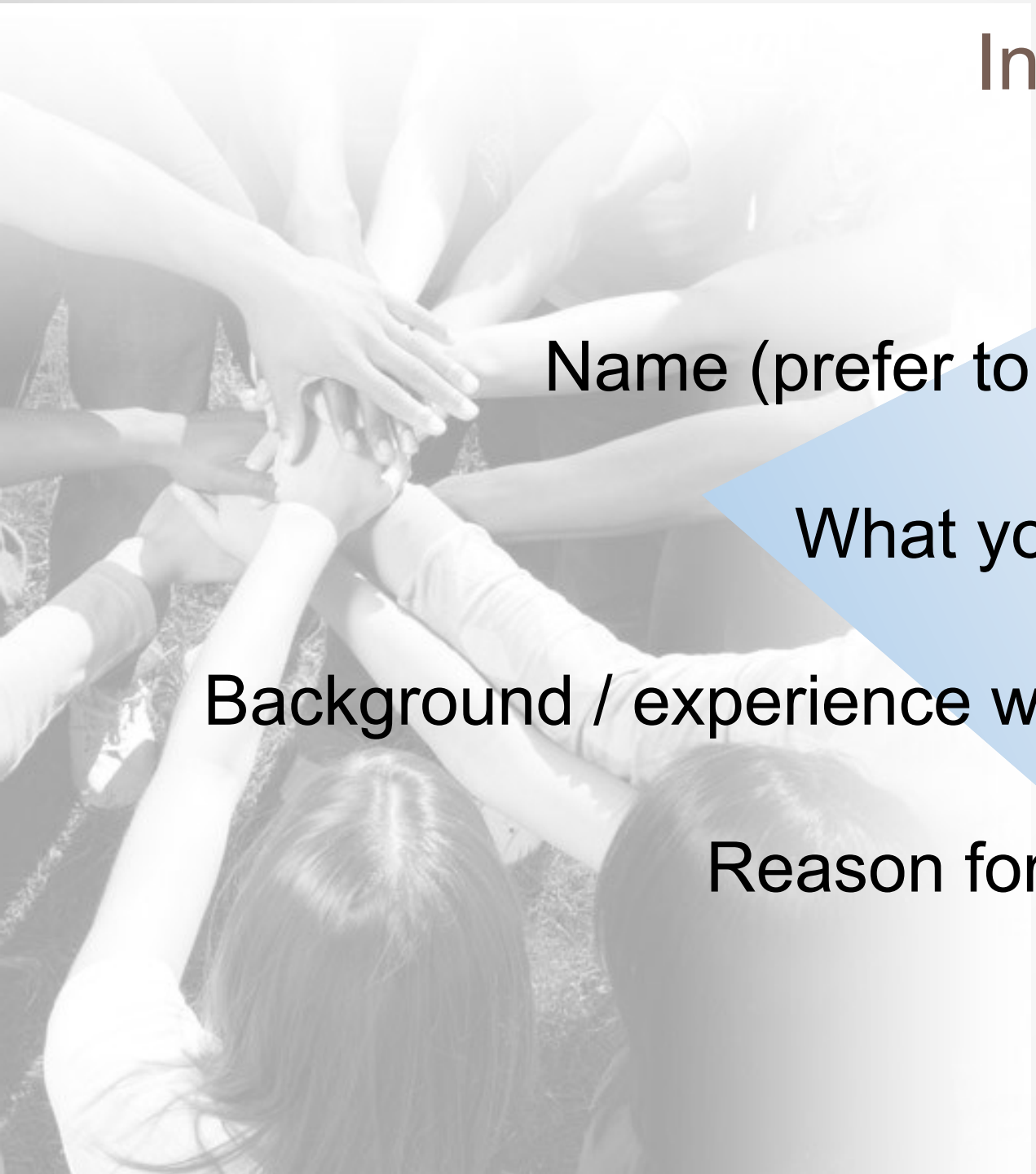
Day 4

Pandas *(continued)*

Django

Other Useful Third-Party Modules

Introductions



Name (prefer to be called)

What you work on

Background / experience with Python

Reason for attending

Typical Daily Schedule*

9:00	Start Day
10:10	Morning Break 1
11:20	Morning Break 2
12:30 – 1:30	Lunch
2:40	Afternoon Break 1
3:50	Afternoon Break 2
5:00	End of Day

** Your schedule may vary, timing is approximate*

Get the Most from Your Experience

Ask Questions

Chapter 1
Review of Intermediate Topics

Review of Prior Topics (and Some New Things Too!)

Chapter 1 - Overview

- Formatting Output
- Special Import Rules/Techniques
- String Buffering (StringIO)
- Generators
- Argparse
- Variable Scope Rules in Python
- Functional Programming Tools
- *Optional:* Decorators Review

Formatting Output: Unpacking

- The str type's format() method provides flexibility to render output values
  - * can be used to unpack iterables

```
winners = ('Bob', 'Sally', 'John')

print('The winners are: {0}, {1}, {2}'.format(winners[0],
                                              winners[1],
                                              winners[2]))



print('The winners are: {0}, {1}, {2}'.format(*winners))
```

* unpacks items of an iterable (such as a list) into individual elements

Formatting Output: Unpacking *(continued)*

Named tuples allow for naming of data fields

```
from collections import namedtuple
Winners = namedtuple('Winners', ['first', 'second', 'third'])
winners = Winners('Bob', 'Sally', 'John')
print('Top two winners: {0}, {1}'.format(winners.first,
                                         winners.second,
                                         winners.third))

print('Top two winners: {0}, {1}'.format(*winners))
```

More values than are needed can be provided to format()

```
winners = {'first': 'Bob', 'second': 'Sally', 'third': 'John'}
print('Top two winners: {first}, {second}'
                                .format(first=winners['first'],
                                        second=winners['second'],
                                        third=winners['third']))

print('Top two winners: {first}, {second}'.format(**winners))
```

** only works for dictionaries

format() Method Parameters

Zero padding
Field total width
Show grouping operator

{(target)(![rsa])(:((*fill*)[<>^=])([+-])(#)(0)(width)(,)(.precision)(type))}

index
or name
of field

conversion to
perform:
repr()
str()
ascii()

fill character
< right, > left
^ center
= pad after sign

+ show sign
- Only on negative
show number base

decimals to display
type can be 'f' 'e' 'd'
or '%' (e.g., .3f)

Py 3.6+ implementation note:
Large numbers may be written using underscores

```
distance = 240_000
print('Distance: {dist:_^+20,.3f} miles'.format(dist=distance))
print(f'Distance: {distance:_^+20,.3f} miles')
```

Distance to moon: ____+240,000.000____ miles

f-strings

- f-strings support the *format()* method syntax
  - f-strings allow evaluation of Python expressions
  - They are faster than the %-syntax and .format() method

```
first = 'bob'
last = 'Smith'
age = 37

def combine():
    return ' '.join([first, last])

print(f'{first.capitalize()} {last} {age} {combine()}')
```

f-strings Special Conditions

```
winners = Winners('Bob', 'Sally', 'John')
print(f'The winners are: {*winners}')
```

f-strings do not support the unpack operator (*)

```
winners = {'first': 'Bob', 'second': 'Sally', 'third': 'John'}
print(f'Top two winners: {winners.get("first")}
                         {winners["second"]}')
```

Top two winners: Bob Sally

f-strings support similar formatting syntax as format() as well as nested curly braces

```
distance = 240_000
field_width = 25
print(f'Earth to moon: {distance:_^+{field_width},.3f} miles')
```

Earth to moon: _____+240,000.000_____ miles

Use {{ }} to render *actual* curly braces within an f-string

Comparing Custom Data Storage Structures

namedtuple

```
from collections import namedtuple
Winners = namedtuple('Winners',
                     ['first', 'second', 'third'])
winners = Winners('Bob', 'Sally', 'John')

print(f'First place: {winners.first}')
```

NamedTuple (3.6+)

```
from typing import NamedTuple
Winners = NamedTuple('Winners', [('first', str),
                                 ('second', str),
                                 ('third', str)])
winners = Winners('Bob', 'Sally', 'John')

print(f'First place: {winners.first}')
```

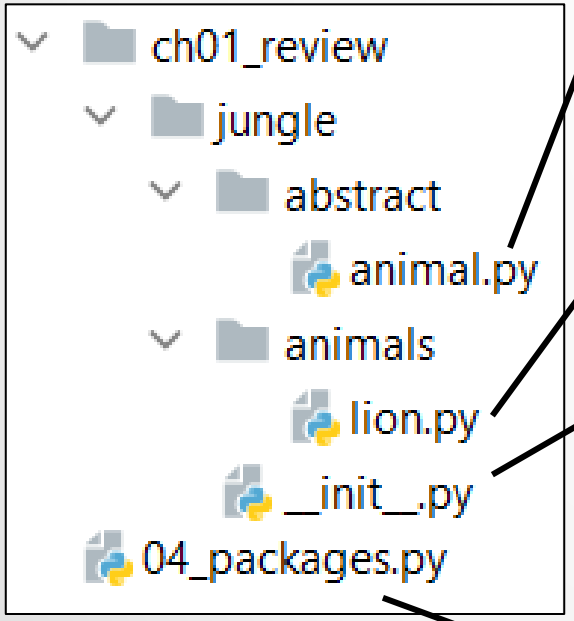
dataclass (3.7+)

```
from dataclasses import dataclass
@dataclass
class Winners:
    first: str
    second: str
    third: str
winners = Winners('Bob', 'Sally', 'John')
print(f'First place: {winners.first}')
```

First place: Bob

Packages and __init__.py

- When packages (even at intermediate levels) are imported that contain an *__init__.py*, it gets executed




```
class Animal:
    def noise(self):
        return ''
```

```
from jungle import Animal

class Lion(Animal):
    def noise(self):
        return 'roar!'
```

```
from .abstract.animal import Animal
from .animals.lion import Lion
```

Causes __init__.py to be executed

```
from ch01_review.jungle import Lion
print(Lion().noise())
```

Relative Imports

- While relative imports are generally discouraged, there are times when they can make sense
- Relative imports begin with a "dot" notation and must follow two important rules
  1. They must be used with the *from-import* technique

```
from .abstract.animal import Animal
```

  2. The file where they are used must be absolutely imported

04_packages.py

```
from .jungle import Lion
print(Lion().noise())
```

This fails as the file containing the relative imports must be absolutely imported

What Else Can Be in __init__.py?

- __init__.py can be used for several purposes
  - Import shortcuts (as demonstrated previously)
  - Docstrings — Place here to describe an entire package
  - __all__ — List of modules that can be imported when from *modname* import * is used
  - Version info

```
VERSION = (3, 1, 4, 'final', 0)
```

From Django 3.1.4 Django/__init__.py

  - Initialization code

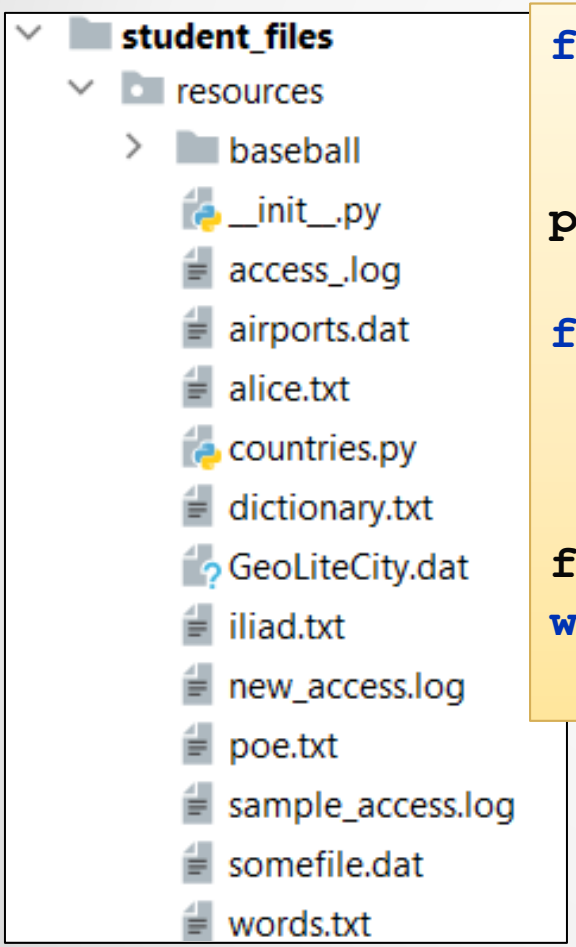
Dynamically Imported Modules

- Modules can be dynamically imported when not known before runtime

```
import importlib
import os

module_name = 'glob'
attribute = 'glob'


results = []
try:
    mod = importlib.import_module(module_name)
    func = getattr(mod, attribute)
    results = [os.path.basename(fn) for fn in func('./*.py')]
except (ModuleNotFoundError, AttributeError):
    pass

print(results)
```

Loading *(Package-Relative)* Resources *(Py3.7+)*

- It is possible to load resources (text/binary files) found within packages by specifying package names *not file paths*




```
from importlib.resources import contents,
                                is_resource,
                                open_text

package = 'resources'

for item in contents(package):
    if is_resource(package, item):
        print(f'{item} is a resource.')

fn = 'dictionary.txt'
with open_text(package, fn) as f:
    file_contents = f.read()
```

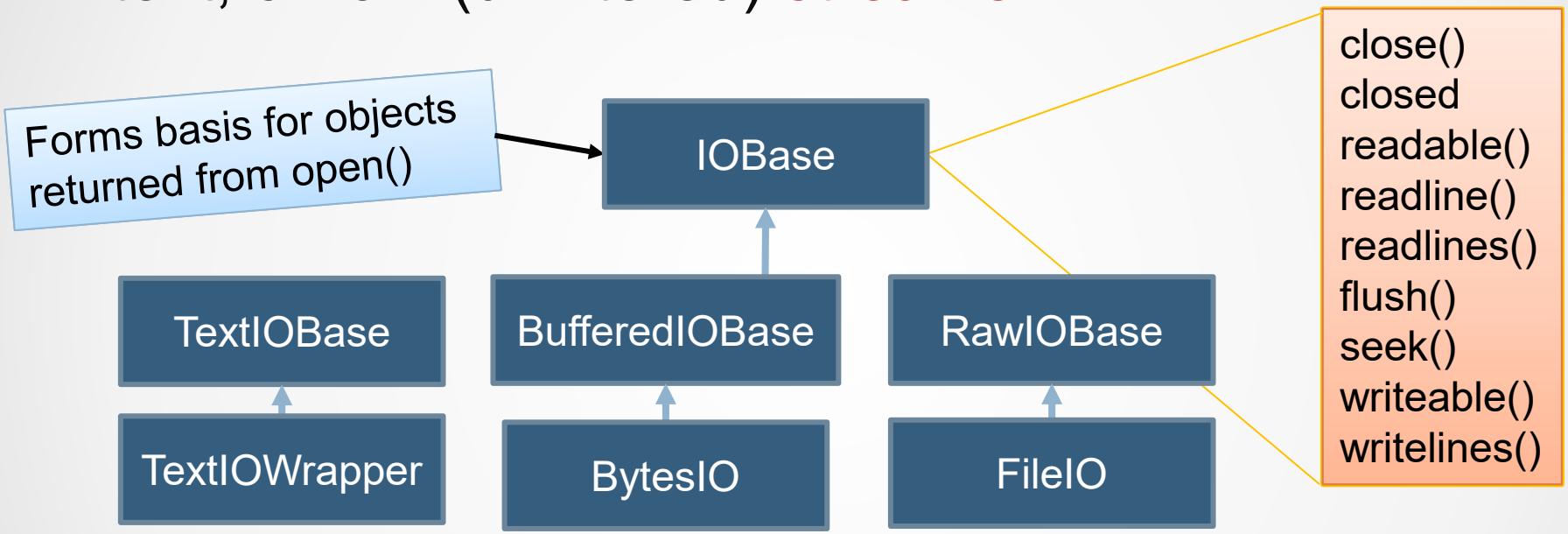
A package in our student files

open_text() accepts a *package* as a path

access_.log is a resource.
airports.dat is a resource.
alice.txt is a resource.
...
words.txt is a resource.
__init__.py is a resource.

The *io* Module

- The io module facilitates working with binary, text, or raw (unfiltered) streams




```
f = open('temp.py', encoding='utf-8')
print(type(f))
```

<class '_io.TextIOWrapper'>

StringIO

- Provides string buffering capabilities
  - Works similarly to a TextIOWrapper (file object)

```
import io

buffer = io.StringIO()
buffer.write('This is the initial buffer value.\n')
print('Added to buffer.', file=buffer)
buffer_results = buffer.getvalue()
buffer.close()

print(buffer_results)
```

file= points to the open stream to write (print) to

This is the initial buffer value.
Added to buffer.

```
with io.StringIO() as buffer:
    buffer.write('A with version\n')
    print('closes automatically.',
          file=buffer)
    print(buffer.getvalue())
```

StringIO is supported within a *with* control

Generators

- Provide lazy (on-demand only) retrieval of items
  - Designed to be iterated over
  - Invoking a generator *does not* begin executing it
  - Special yield keyword "pauses" the generator
  - "Picks up where it left off" when __next__() is called

```
for value in generator1():
    print(value)
```

```
def generator1():
    count = 1
    print(f'Generator paused. Count={count}')
    yield count

    count += 1
    print(f'Generator paused again. Count={count}')
    yield count

    count += 1
    print(f'Generator paused one last time. Count={count}')
    yield count
```

Using Generators

Abnegate v. (-ting) give up or renounce (a pleasure or right etc.)…

Abnegation n. Denial; renunciation of a doctrine…

Abnormal adj. Deviating from the norm; exceptional…

dictionary.txt

```
def read_definitions(filename='../resources/dictionary.txt'):
    try:
        with open(filename, encoding='utf-8') as in_val:
            for line in in_val:
                try:
                    word, definition = line.split(maxsplit=1)
                    yield word, definition
                except ValueError:
                    pass
    except IOError as err:
        print('Error reading from file.', err)

for idx, (word, definition) in enumerate(read_definitions(), 1):
    print(f'Word {idx}: {word:<20}{definition[:50]}')
```

```
Word 50: Abnegate            v. (-ting) give up or renounce (a pleasure or righ…
Word 51: Abnegation           n. Denial; renunciation of a doctrine.
Word 52: Abnormal            adj. Deviating from the norm; exceptional.
```

Working with *argparse*

The *argparse* standard library module can make it easier to define and parse command line arguments

```
from argparse import ArgumentParser
from itertools import islice

def get_args():
    parser = ArgumentParser()
    parser.add_argument('dir', default=os.getcwd(), nargs='?')
    parser.add_argument('filename')
    parser.add_argument('-c', '--count', default=20, type=int)
    parser.add_argument('-r', '--remove_header', default=False,
                        action='store_true', dest='header')
    return parser.parse_args()

args = get_args()

results = []
with open(os.path.join(args.dir, args.filename), encoding='utf-8') as f:
    if args.header:
        f.readline()
    for line in islice(f, args.count):
        results.append(line.strip().split(','))

for record in results:
    print(record)
```

Legal command line statements

python 08_argparse.py *foo.txt --count 10*
python 08_argparse.py *../resources foo.txt -r -c 5*
python 08_argparse.py *foo.txt --remove_header*

Your Turn! Task 1-1

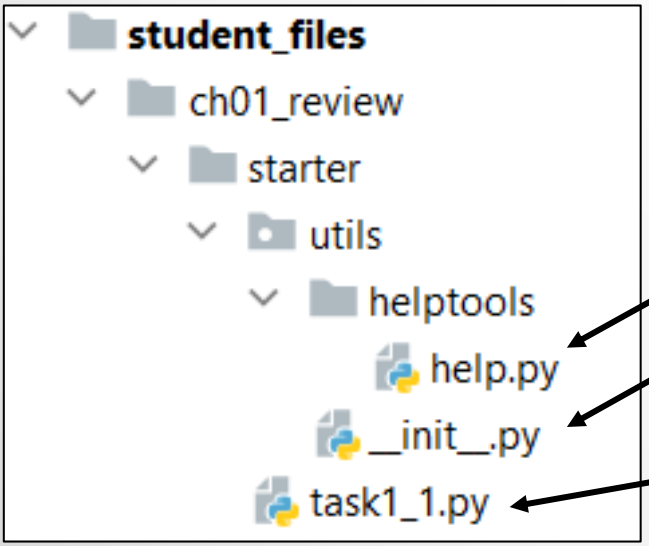
Hints are found in the source file. Step-by-step instructions are found in the back of the student manual

- *Create a utility that displays top-level module properties*
  - Do this by creating a generator that accepts the module name
  - Should read a module name and display the top-level attributes
  - The generator yields one module *attribute* at a time
  - Use a StringIO object to capture the dir command
  - Access the StringIO and *yield* retrieved attributes
  - Use argparse to capture command line input args
  - Create a shortcut import in the __init__.py

sample output

python task1_1.py sys

```
api_version
argv
base_exec_prefix
base_prefix
builtin_module_names
byteorder
call_tracing
callstats
copyright
displayhook
dllhandle
don't_write_bytecode
exc_info
excepthook
exec_prefix
executable
exit
```



Place generator in here

Create import shortcut here

Run program from here

Using *fire*

pip install fire

```
class FileReader:
    def __init__(self, filename, dir=os.getcwd(), count:int = 10,
                 header: bool = False):
        self.filename = filename
        self.dir = dir
        self.count = count
        self.header = header
        self.results = []

    def read(self):
        with open(os.path.join(self.dir, self.filename),
                  encoding='utf-8') as f:
            if self.header:
                f.readline()
            for line in islice(f, self.count):
                self.results.append(line.strip().split(','))

        return self.results
```

Fire quickly processes arguments calling functions, classes, or returning any desired values from the command line

Class method is automatically called

python 09_fire.py --filename somefile.txt --count 2 read

Args passed to __init__ should use -- notation

```
fire.Fire(FileReader)
```

Functions and Scope

- ***LEGB*** defines variable scope rules in Python

L look in the local function

E look at an enclosing function

G look to the global scope

B look for a built-in variable

```
width = 5

class Person:
    width = 10
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def display_name(self):
        width = 15
        print(self.name[:width])

p = Person('Ricardo Estebonacian', 39)
p.display_name()
print(p.name[:width])
```

Local scope

Ricardo Estebon

Global scope

Ricar

The global Keyword

- The *global* keyword indicates the desire to access <u>*and modify*</u> a global variable from within a function

```
width = 5

class Person:
    width = 10
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def display_name(self):
        global width
        width = 15
        print(self.name[:width])

p = Person('Ricardo Estebonacian', 39)
p.display_name()
print(p.name[:width])
```

Changes to width affect the global variable now

Local scope

Ricardo Estebon

Global scope

Ricardo Estebon

Non-locals

- *Nonlocal* allows nested functions to access <u>and modify</u> variables of the *enclosing* function

```
width = 5

class Person:
    width = 10

    def __init__(self, name: str = None, age: int = 0):
        self.name = name
        self.age = age

    def display_name(self):
        width = 15

        def display():
            nonlocal width
            width = 3
            print(self.name[:width])
        display()
        print(self.name[:width])

p = Person('Ricardo Estebonacian', 39)
p.display_name()
print(p.name[:width])
```

What happens if the nonlocal line is removed?

Ric

Ric

Ricar

globals() and locals()

- The globals() dict provides access to any variables defined at the top-level of a module

shared.py

```
width = 5
print(globals()['width'])
```

```
5
```

globals()

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x000001DE4A533FA0>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'__file__': 'C:\\temp\\python\\student_files\\temp.py', '__cached__': None, 'width': 5}
```

locals() (not shown here) returns all variables on the current stack (i.e., where you are currently at when it is called)

Sharing Global Variables Across Modules

shared.py

```
width = 5
```

module1.py

```
import shared
import module2

print(shared.width)
```

module2.py

```
import shared

shared.width = 10
```

10

Functional Programming Tools

- Python provides several functions for FP

```
iter = map(function, iterable)
results = list(iter)
```

Returns an iterator

Returns a single object

```
from functools import reduce
result = reduce(function, sequence [, initial_val])
```

```
filterobj = filter(boolean_function, iterable)
```

Returns values only if Boolean function returns True

```
boolResult = any(iterable)
```

True if any one is true

```
boolResult = all(iterable)
```

True if all are true

FP Function Quick Ref

```
months = {'January': 31, 'February': 28, 'March': 31, ... }

results = []
for abbr in map(lambda mo: mo[:3], months):
    results.append(abbr)
print(results)
```

Map returns an iterator

['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

```
reduced = functools.reduce(lambda mo1, mo2: f'{mo1} {mo2}',
                                                    results)
print(reduced)
```

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

```
no_ber_months = filter(lambda mo: mo[-3:] != 'ber', months)
results = list(no_ber_months)
print(results)
```

['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August']

```
print(any([mo < 29 for mo in months.values()]))
print(all([mo > 29 for mo in months.values()]))
```

True

False

Mini-Task 1

- Given the following data structure, use functools.reduce() to return all overdue orders

```
orders = [
    {'id': 1005, 'name': 'Joseph Gonzales', 'status': 'paid',
     'amount': 327.08, 'email': 'jgonz201@gmail.com'},
    {'id': 1022, 'name': 'Tamarin Wilkins', 'status': 'overdue',
     'amount': 1137.81, 'email': 'gtw.wilkins@csudw.com'},
    {'id': 2154, 'name': 'Christine Coldwell', 'status': 'overdue',
     'amount': 65.33, 'email': 'coldwise@humazing.com'}
    ...
]
```

```
overdue_orders = {
    1022: 1137.81,
    2154: 65.33,
    ...
}
```

Use reduce(function, collection, initial_value) to return this structure
initial_value will be the starting structure, { } .

Chapter 1 - Summary

- Scope rules in Python are usually pretty simple: local variables get precedence over global variables
  - Control the manipulation of global variables using the *global* keyword
- Use StringIO for buffering string data
  - Useful for staging data in memory rather than to files
- Argparse takes the grunt work out of command line processing
  - Fire, provides a different approach toward using args
- Incorporate relative importing in special circumstances and __init__.py to improve simplify shortcuts
- Functional programming tools provide often-forgotten techniques for simplifying code algorithms

Chapter 2
Automation and Standard Library Tools

Further Coverage of Modules in the Standard Library

Chapter 2 - Overview

- File Management and Directory Access
- Email Notifications
- Creating PDFs
- Timeit
- Dictionaries and JSON
- Optional: Context Manager Utilities

File and Directory Access Modules

- The standard library provides several useful modules for working with files and directories
  - These can be useful tools for automation scripting
- Some of these modules include:
  - pathlib - cross-platform directory-based utilities
  - fnmatch - for matching file patterns and file names
  - shelve - persistence simplified
  - shutil - for moving/copying multiple files
  - filecmp - compares files and directories
  - zipfile - Creates, reads, extracts items from zip files

Path (from *pathlib* Module)

- Path serves as a wrapper around paths to allow for a more object-oriented style of use
  - Path objects have numerous convenience methods often removing the need to import the os module
  - Generally, favor the use of Path(*item*).method() over os.method(path)

```
import fnmatch
import pathlib

parent = pathlib.Path('..')
working_dir = 'ch02_automation'
working = parent / working_dir
# or working = pathlib.Path().cwd()

pattern = '*_contextmgr*.py'
for filepath in working.iterdir():
    if fnmatch.fnmatch(filepath, pattern):
        print(filepath)
```

Wraps the parent directory

Path to ch02_automation directory

Alternatively, use cwd() to get it

Provides an iterator to items in this dir

filepath is a Path() object

More with Path Objects

- The Path() object supports many methods

```
this_file = pathlib.Path(__file__)
this_file.absolute()
this_file.read_text()

with this_file.open() as f:
    print([item.strip() for item in itertools.islice(f, 7)])
this_file.exists()
this_file.is_file()
this_file.is_dir()

this_file.name
this_file.stem

this_file.parent
list(this_file.parents)

list(working.glob('*.dat'))
pathlib.Path().home()
```

Open, read entire file, close the file

Path objects representing files can be opened directly

Reads exactly 7 lines from file

True

False

01_pathlib_fnmatch_shelve.py

01_pathlib_fnmatch_shelve

Path('..ch02_automation')

[Path('..ch02_automation'), ..., Path('C:\')]

fnmatch

- fnmatch() finds files by using Unix-style (not regex) patterns

fnmatch() returns *True* or *False* if the pattern and filename match. It is useful due to its simplicity and due to user familiarity with Unix file matching syntax

```
..\ch02_automation\13_contextmgr.py
..\ch02_automation\14_contextmgr_closing.py
..\ch02_automation\15_contextmgr_function.py
..\ch02_automation\16_contextmgr_redirect.py
..\ch02_automation\17_contextmgr_decorator.py
```

```
import fnmatch
import pathlib

parent = pathlib.Path('..')
working_dir = 'ch02_automation'
working = parent / working_dir

pattern = '*_contextmgr*.py'
for filename in working.iterdir():
    if fnmatch.fnmatch(filename, pattern):
        print(filename)
```

Supported Patterns

*	match all
?	match any one character
[abc]	match any of these characters
[!abc]	match any character not listed here

shelve

- shelve provides persistent dictionary storage
  - shelve.open() returns a *shelve* object

As of this writing, shelve.open() didn't yet open Path() objects

```
shelve_data = working / 'user_prefs.dat'
with shelve.open(str(shelve_data)) as sh:
    sh['data_dir'] = pathlib.Path('..') / 'resources'
    sh['data_file'] = 'poe.txt'
```

Saves directory and file object into the persistent shelve

```
try:
    print(sh['data_file'])
except ValueError as err:
    print('-->', err)
```

--> invalid operation on closed shelf

Reopens shelve object, retrieves persisted data

```
with shelve.open(str(shelve_data)) as sh:
    file_location = sh['data_dir'] / sh['data_file']
    if file_location.exists():
        print(colorama.Fore.GREEN, file_location.read_text())
```

shutil

- shutil contains utilities to help with copying, moving, and removing collections of files
  - For individual file-related tasks, use the os module

```
shutil.copy(src, dest)          -   dest can be a directory
shutil.copy2(src, dest)         -   dest can be a directory
shutil.copyfile(src, dest)      -   dest must be a path+file
shutil.copytree()               -   copy an entire directory tree
shutil.move(src, dest)          -   moves an entire directory tree
shutil.rmtree(path)             -   removes a directory tree
```

	dst can be Directory?	Copies Permissions	Copies Metadata
copy()	Yes	Yes	No
copy2()	Yes	Yes	Yes
copyfile()	No	No	No

filecmp

- The filecmp module helps with comparing differences between files and directories
  - cmp(file1, file2, shallow=True) compares two files

```
filecmp.cmp('dir1/file1', 'dir2/file1')
filecmp.cmp('dir1/file1,'dir2/file1', shallow=True)
```

Compares two files' stats; returns True if they are the same

```
filecmp.cmp('dir1/file1,'dir2/file1', shallow=False)
```

Compares files *based on the contents* in addition to file metadata

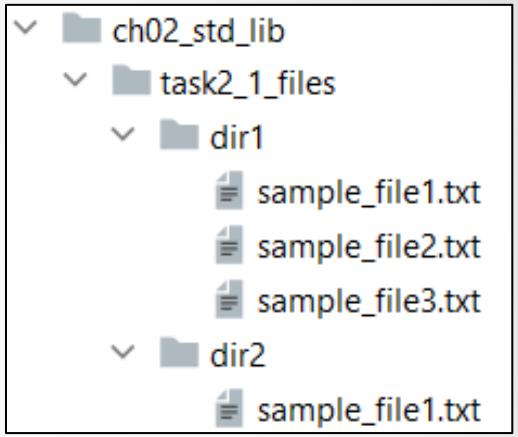
```
task2_1_files = Path('task2_1_files')

file1 = task2_1_files / 'dir1/sample_file1.txt'
file2 = task2_1_files / 'dir2/sample_file1.txt'

print(filecmp.cmp(file1, file2))
print(filecmp.cmp(file1, file2, shallow=False))
```

```
False
False
```

filecmp (cont.)

- dircmp() compares two directories according to specified attribute

```
filecmp.dircmp(dir1, dir2).attribute
```




```
dir1 = task2_1_files / 'dir1'
dir2 = task2_1_files / 'dir2'

results =
    filecmp.dircmp(dir1, dir2)
print('\nReport results:')
results.report()
```

```
Report results:
diff task2_1_files/dir1 task2_1_files/dir2
Only in task2_1_files/dir1 : ['sample_file2.txt',
                              'sample_file3.txt']
Differing files : ['sample_file1.txt']
```

```
print('Left only:  ', results.left_only)
```

```
Left only:  ['sample_file2.txt',
             'sample_file3.txt']
```

```
print('In Common:  ', results.common)
```

```
In Common: ['sample_file1.txt']
```

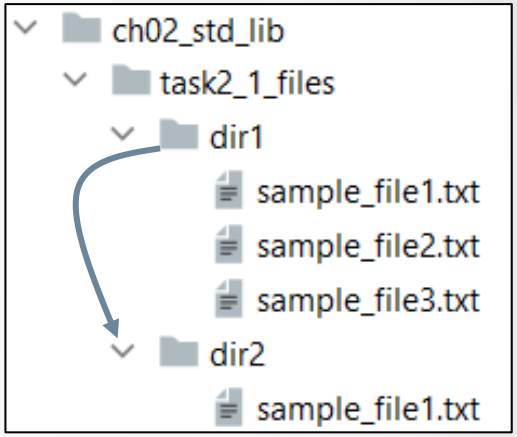
```
print('Right only: ', results.right_only)
```

```
Right only:  []
```

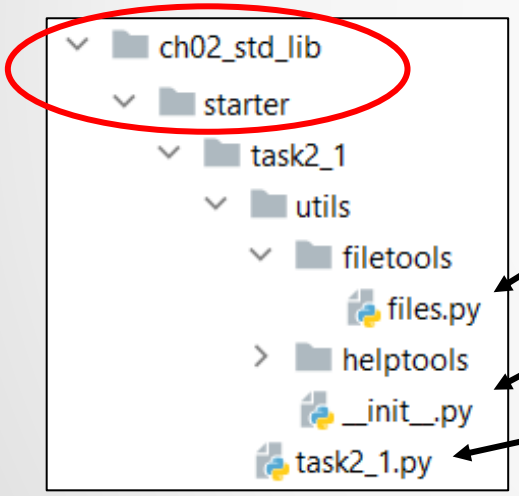
Your Turn! Task 2-1

- Use shutil, filecmp, and pathlib to copy the files in the *task2_1_files* directory from dir1 to dir2

Don't copy files that already exist in dir2



- Display a list of the files copied



Complete the copy_files function within files.py

Create a shortcut import in __init__.py

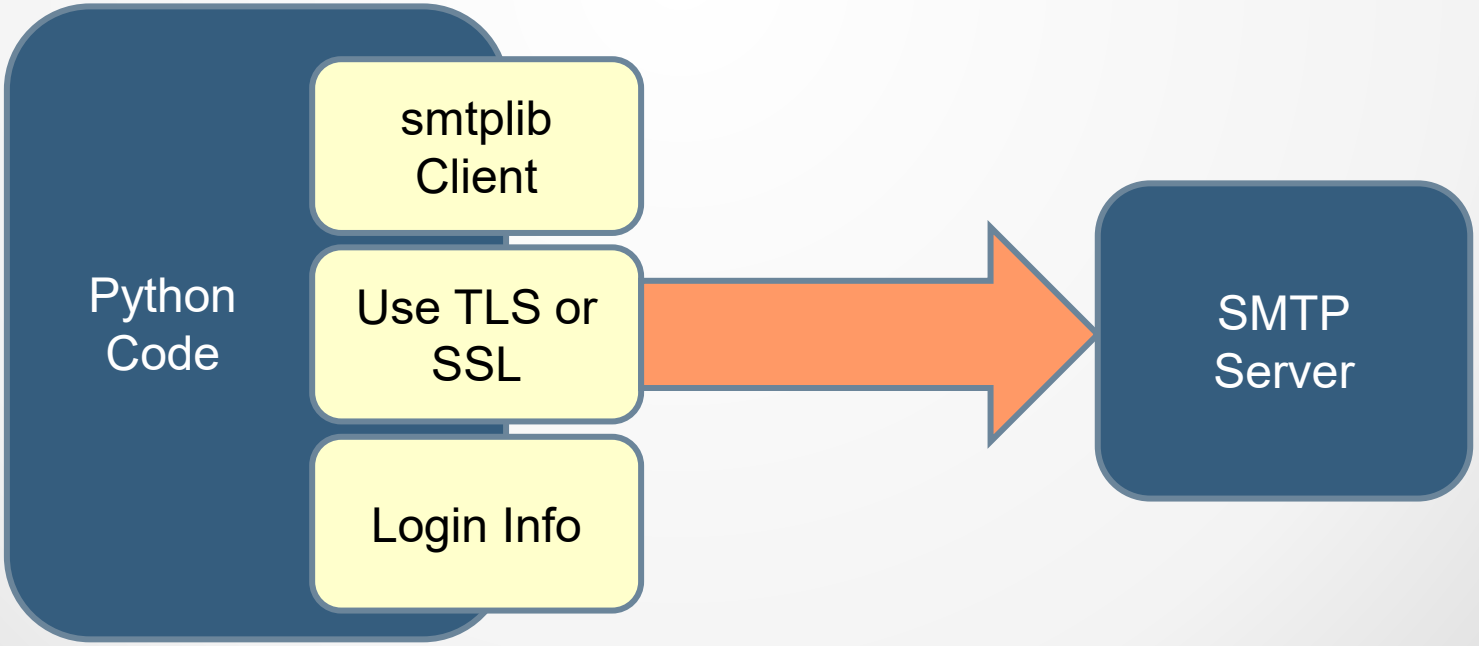
Invoke the copy_files() function from task2_1.py

Step-by-step instructions can be found in the back of the student manual

Optional, advanced (on your own):
Can you create a similar function, called *delete_common()* that deletes the common files from the destination? Hint: Use Path().unlink().

Email

- Sending emails via Python can be useful for automation tasks
  - It can often be tricky to gain access to your SMTP server
  - We'll explore creating an email using a "dummy" server



Email to a Dev SMTP Server

- Launch the server from a command line using:

```
python -m smtpd -c DebuggingServer -n localhost:1025
```

- Create and send an email using the email and smtplib modules:

```
from pathlib import Path
import smtplib
from email.message import EmailMessage

message_body = Path('../resources/sample_message.txt')
email_msg = EmailMessage()
email_msg.set_content(message_body.read_text())
email_msg['Subject'] = 'Sample Message'
email_msg['From'] = 'joe@example.com'
email_msg['To'] = 'dave@example.com'

server = smtplib.SMTP('localhost', 1025)
server.send_message(email_msg)
server.quit()
```

Construct the email

Create the client capable of sending the email

Send the email to the server

Server Email Received

- The SMTP dev server will receive and display your email message

```
C:\temp>python -m smtpd -c DebuggingServer -n localhost:1025
---------- MESSAGE FOLLOWS ----------
b'Content-Type: text/plain; charset="utf-8"'
b'Content-Transfer-Encoding: 7bit'
b'MIME-Version: 1.0'
b'Subject: Sample Message'
b'From: joe@example.com'
b'To: john@example.com'
b'X-Peer: ::1'
b''
b'Hello!'
b''
b'This is the body for our email message sample.  It will be sent'
b'from the example in ch02_automation.'
------------ END MESSAGE ------------
```

Sending Emails Through Your Server

- To send emails to <u>your</u> SMTP server, you will need to know

Are you using TLS or SSL?
Host name, port number, username, and password

```
import smtplib
from email.message import EmailMessage

message_body = Path('../resources/sample_message.txt')
email_msg = EmailMessage()
email_msg.set_content(message_body.read_text())

email_msg['Subject'] = 'Sample Message'
email_msg['From'] = 'joe@example.com'
email_msg['To'] = 'dave@example.com'
```

```
# if using TLS:
server = smtplib.SMTP(hostname, 587)
server.starttls()
```

```
# if using SSL:
server =
smtplib.SMTP_SSL(hostname, 486)
```

```
server.login(username, password)
server.send_message(email_msg)
server.quit()
```

Sending Email Attachments

```
from pathlib import Path
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.application import MIMEApplication

email_msg = MIMEMultipart()

email_msg['Subject'] = 'Email with attachment'
email_msg['From'] = 'joe@example.com'
email_msg['To'] = 'dave@example.com'

attachment = Path(__file__)
part = MIMEApplication(attachment.read_text(),
                       Name=attachment.name)
part['Content-Disposition'] =
                    f'attachment; filename="{attachment.name}"'
email_msg.attach(part)

server = smtplib.SMTP('localhost', 1025)
server.send_message(email_msg)
server.quit()
```

Create the email type that supports attaching documents to it

Open the file to attach and read it

Working with PDFs

- Numerous tools exist for scraping, modifying, and creating PDFs in Python

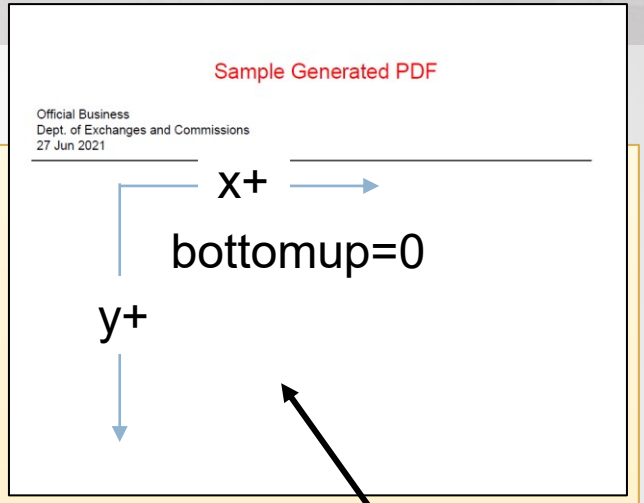
Tool	Description	Install
PyPDF2	Reading/Modifying PDFs	pip install PyPDF2
Tabula-py	Reads PDFs into CSV files	pip install tabula-py
Pdfminer	Extracts text from PDFs	pip install pdfminer
Pdfkit	Create PDFs from Webpages	pip install pdfkit
Camelot	Extracts tables from PDFs	pip install Camelot-py[cv]
ReportLab	Creates PDFs from scratch	pip install reportlab
Fpdf2	(was PyFPDF) Free PDF creator	pip install fpdf2
Pikepdf	PDF reader/writer	pip install pikepdf

  - These tools are all third party and must all be pip installed

ReportLab

To use ReportLab:
pip install reportlab or ***conda install reportlab***

- The following generates a PDF




```
from reportlab.lib.colors import red
from reportlab.lib.pagesizes import A4
from reportlab.lib.units import inch
from reportlab.pdfgen.canvas import Canvas

canvas = Canvas('sample.pdf', pagesize=A4,
                bottomup=0)

canvas.setFont('Helvetica', 20)

canvas.setFillColor(red)
canvas.drawCentredString(x=4.25*inch, y=1*inch,
                         text='Sample Generated PDF')
canvas.setFont('Helvetica', 12)
canvas.setFillColor(black)
canvas.drawString(30, 110,'Official Business')
canvas.drawString(30, 125,'Dept. of Exchanges and Commissions')
canvas.drawString(30, 140,'27 Jun 2021')
canvas.line(25, 150, 560, 150)
canvas.save()
```

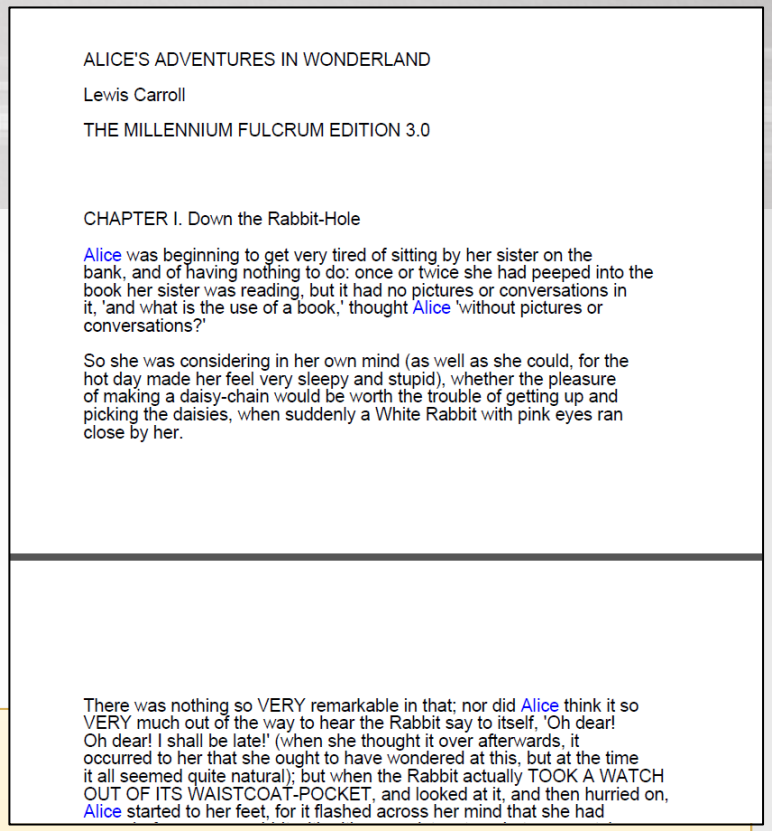
Canvas, the main object, represents the PDF document

Renders text

Saves the PDF

Rendering Text *(1 of 2)*

- Use the SimpleDocTemplate and Paragraph classes to render free flowing text
  - SimpleDocTemplate is best when all pages have the same look



ALICE'S ADVENTURES IN WONDERLAND

Lewis Carroll

THE MILLENNIUM FULCRUM EDITION 3.0

CHAPTER I. Down the Rabbit-Hole

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, 'and what is the use of a book,' thought Alice 'without pictures or conversations?'

So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid), whether the pleasure of making a daisy-chain would be worth the trouble of getting up and picking the daisies, when suddenly a White Rabbit with pink eyes ran close by her.

There was nothing so VERY remarkable in that; nor did Alice think it so VERY much out of the way to hear the Rabbit say to itself, 'Oh dear! Oh dear! I shall be late!' (when she thought it over afterwards, it occurred to her that she ought to have wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually TOOK A WATCH OUT OF ITS WAISTCOAT-POCKET, and looked at it, and then hurried on, Alice started to her feet, for it flashed across her mind that she had

```
from pathlib import Path

from reportlab.lib.colors import blue, black
from reportlab.lib.pagesizes import letter
from reportlab.platypus import SimpleDocTemplate, Paragraph,
                               Spacer

doc = SimpleDocTemplate('sample2.pdf', pagesize=letter,
   rightMargin=72, leftMargin=72, topMargin=72, bottomMargin=72)

doc_items = []
fontsize = 12
```

Start a PDF with a layout where each page is the same style

Rendering Text *(2 of 2)*

```
with Path('../resources/alice.txt').open() as f:
    for line in f:
        line = line.strip()
        if len(line) >= 1:
            if 'Alice' in line:
                parts = line.split('Alice')
                line = f'<font size="{fontsize}" color="{black}">
                        {parts[0]}</font><font size="{fontsize}"
                        color="{blue}">Alice</font>
                        <font size="{fontsize}" color="{black}">
                        {parts[1]}</font>'
            else:
                line = f'<font size="{fontsize}" color="{black}">
                         {line}</font>'
            doc_items.append(Paragraph(line))
        else:
            doc_items.append(Spacer(1, 12))

doc.build(doc_items)
```

Read line-by-line from the file, wrap a ***Paragraph*** object around it, add it to the list of items to be rendered.

Build the PDF

Rendering Tables *(1 of 2)*

- Render tables using the Table object and TableStyle to control the look and feel

name	city	country	IATA_FAA
Goroka	Goroka	Papua New Guinea	GKA
Madang	Madang	Papua New Guinea	MAG
Mount Hagen	Mount Hagen	Papua New Guinea	HGU
Nadzab	Nadzab	Papua New Guinea	LAE
Port Moresby Jacksons Intl	Port Moresby	Papua New Guinea	POM
Wewak Intl	Wewak	Papua New Guinea	WWK
Narsarsuaq	Narssarssuaq	Greenland	UAK
Nuuk	Godthaab	Greenland	GOH
Sondre Stromfjord	Sondrestrom	Greenland	SFJ
Thule Air Base	Thule	Greenland	THU
Akureyri	Akureyri	Iceland	AEY
Egilsstadir	Egilsstadir	Iceland	EGS
Hornafjordur	Hofn	Iceland	HFN
Husavik	Husavik	Iceland	HZK
Isafjordur	Isafjordur	Iceland	IFJ
Keflavik International Airport	Keflavik	Iceland	KEF
Patreksfjordur	Patreksfjordur	Iceland	PFJ

TableStyle([('STYLE', (startcol, startrow), (endcol, endrow), value, ...[optional_value])])

```
from reportlab.lib.colors import Color, blue, lightsalmon, dimgray
from reportlab.platypus import SimpleDocTemplate, Table, TableStyle

doc_items = []

doc = SimpleDocTemplate('sample3.pdf', pagesize=letter,
      rightMargin=72, leftMargin=72, topMargin=72, bottomMargin=72)

airports = []
```

Rendering Tables *(2 of 2)*

```
with Path('../resources/airports.dat').open(encoding='utf-8') as f:
    for count, line in f:
        fields = line.split(',')
        name, city, country, code = fields[1], fields[2],
                                    fields[3], fields[4]
        airports.append((name, city, country, code))

table = Table(airports, colWidths=(200, 150, 100, 60))
ts = TableStyle([('ALIGN', (0, 0), (-1, 0),'CENTER'),
                 ('TEXTCOLOR',(0, 0), (-1, 0), blue),
                 ('ALIGN', (0, 1), (-1, -1),'LEFT'),
                 ('GRID', (0, 0), (-1, -1), 1, dimgray),
                 ('BACKGROUND', (0, 0), (-1, 0), lightsalmon),
                 ('BACKGROUND', (0, 1), (-1, -1),
                                    Color(0.9, 0.95, 0.88))
])

table.setStyle(ts)
doc_items.append(table)
doc.build(doc_items)
```

These affect the header row

Optional: Your Turn! Task 2-2

- Generate a PDF, send it as an email attachment
- Work from *starter/task2_2/task2_2.py* and complete the remaining two methods
- Complete and then call the remaining two functions at the bottom of the source file

COMPLETED!

get_data()

generate_pdf()

send_email()

Use hints within the source file along with specific instructions for this exercise in the back of the student manual

psutil

```
pip install psutil
pip install --user psutil
pip3 install psutil
pip3.10 install psutil
```

- *psutil* can provide cross-platform information about memory and cpu utilization

```
import psutil

print('% utilization: ', psutil.cpu_percent(interval=2))
print('Physical CPU cores: ', psutil.cpu_count(False))
print('CPU Speed: ', psutil.cpu_freq())
print('Memory:', psutil.virtual_memory())
print('Partition info: ', psutil.disk_partitions())
print('Network devices: ', psutil.net_if_addrs())

net_connections = psutil.net_connections()
print('Network sockets: ', net_connections)

print('Users: ', psutil.users())
print('Process IDs: ', psutil.pids())
pid = os.getpid()
print('This process: ', pid)
p = psutil.Process(pid)
print(p, p.exe())
```

timeit

- timeit can measure time lapses in code
  - It can execute code snippets repeatedly to determine how long they take

The main snippet to execute

```
python -m timeit -n nTimes -r nRepeats 'stmt'
```

Execute the statement -n times

Repeat the -r sets of -n executions

```
python -m timeit -n 100 -r 5 'stmt'
```

This will execute the statement -n times and repeats that -n executions 5 overall sets

Using *timeit* from within a Module *(1 of 2)*

```
def sample_func1():
    s = 'This is our starting string'
    with StringIO() as buffer:
        for i in range(200000):
            print(s, file=buffer)
        s = buffer.getvalue()

    return len(s)
```

This builds a StringIO() object by appending s into the buffer 200000 times

```
def sample_func2():
    s = 'This is our starting string'
    for i in range(200000):
        s += 'This is our starting string'
    return len(s)
```

This builds a str object by appending strings together 200000 times

Using *timeit* from within a Module *(2 of 2)*

```
print('Before StringIO: ', psutil.virtual_memory())

timeit.repeat(stmt='sample_func1()', globals=globals(),
              number=10, repeat=3)

print('After StringIO: ', psutil.virtual_memory())
```

Before StringIO: svmem(total=68478214144, available=59049201664,
percent=13.8, used=**9429012480**, free=59
[0.5648092, 0.595277, 0.5686495]
After StringIO: svmem(total=68478214144, available=5905589043
percent=13.8, used=**9422323712**, free=590

timeit results

Difference is memory used: 6688768

```
print('Before str append: ', psutil.virtual_memory())

timeit.repeat(stmt='sample_func2()', globals=globals(),
              number=10, repeat=3)

print('After str append: ', psutil.virtual_memory())
```

Before str append: svmem(total=68478214144, available=59055890432,
percent=13.8, used=**9422323712**, free=
[7.510370, 7.583362, 7.739652]
After str append: svmem(total=68478214144, available=5908260
percent=13.7, used=**9395609600**, free=

timeit results

Difference is memory used: 26714112

Comparing Techniques *(1 of 2)*

- Python provides numerous ways to solve problems

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

Technique 1

Consider an algorithm for finding the most common value in data

```
occurs = {}
for val in data:
    if val not in occurs.keys():
        occurs[val] = 1
    else:
        occurs[val] += 1
most_common = sorted(occurs.items(),
                     key=lambda x: x[1],
                     reverse=True)
print(f'(Technique 1, std. dict) Most: {most_common[0][0]}
        with {most_common[0][1]} occurrences.')
```

Creates a list of tuples ordered by values

(Technique 1, std. dict) Most: 5 with 4 occurrences.

Comparing Techniques *(2 of 2)*

Technique 2

```
from collections import defaultdict
occurs = defaultdict(int)
for val in data:
    occurs[val] += 1
most_common = sorted(occurs.items(), reverse=True
                     key=lambda pair: pair[1])
print(f'(Technique 2, defaultdict) Most: \
        {most_common[0][0]} with {most_common[0][1]}
        occurrences.')
```

```
(Technique 2, defaultdict) Most: 5 with 4 occurrences.
```

Technique 3

```
from collections import Counter
most = Counter(data).most_common(1)[0]
print(f'(Technique 3, Counter) Most: {most[0]} with \
        {most[1]} occurrences.')
```

```
(Technique 3, Counter) Most: 5 with 4 occurrences.
```

Mini-Task 2: Which Technique is Best?

- Using what we've learned about the *timeit* module, determine which, if any, of the three counting techniques previously shown performs best
- Use timeit.repeat() to test each technique
- Work from *ch02_automation/mini-task2.py*
  - Since the functions to test are already imported, use globals=globals() within timeit.repeat() to ensure they can be seen

Context Managers

- When using 'with', __enter__() and __exit__() are called

__enter__ is always called at the beginning
Return value becomes the optional 'as' object
__exit__ is called *no matter what happens*

```
class CtxMgr(object):
    def __enter__(self):
        print('in enter')
        return 'foo'

    def __exit__(self, typ, value, traceback):
        print('in exit')
        return True

with CtxMgr() as obj:
    print(obj)
```

Suppress exception

in enter
foo
in exit

Creating Custom Context Managers

- The following example demonstrates the use of the *with* control and a ThreadPool class to illustrate how to customize a context manager
- The example:
  - Manages tasks placed onto a *queue*
  - Retrieves arguments for the task from this queue
  - Manages 10 threads (by default, configurable)
  - Implements a custom context manager

Advanced Context Managers *(1 of 3)*

```
class WorkerThread(Thread):
    def __init__(self, task, req_queue, results_queue,
                 name='WorkerThread', sentinel='TERMINATE',
                 logging_on=False):
        super().__init__(name=name)
        self.task = task
        self.req_queue = req_queue
        self.results_queue = results_queue
        self.sentinel = sentinel
        self.level = logging.INFO if logging_on else logging.NOTSET

    def run(self):
        while True:
            try:
                args = self.req_queue.get()
                if args[0] == self.sentinel:
                    break
                results = self.task(*args)
                self.results_queue.put(results)
                self.req_queue.task_done()
            except Empty:
                continue
```

Log output can be toggled

The task (job) is passed into the thread

Blocks until a job is available

Each thread:
- obtains a task from the queue,
- executes it if it is not the sentinel value,
- places the results on the results queue,
- marks the task as done

Advanced Context Managers *(2 of 3)*

```
pool = ThreadPool(get_data)

with pool as req_queue:
    for url in tasks:
        req_queue.put((url,))

print('Results:')
for result in pool.get_results():
    print(result)
```

get_data is the work to be performed

ThreadPool is instantiated. get_data (the work to be performed) is passed into the constructor

ThreadPool is a context manager. Its __enter__ returns the job queue. URLs are placed on the job queue

```
def get_data(url):
    try:
        text = requests.get(url).text
        soup = BeautifulSoup(text, 'html.parser')
        result = soup.title.text
    except requests.exceptions.ConnectionError as err:
        result = err.args[0]
    return result
```

Our work to be performed

Advanced Context Managers *(3 of 3)*

```
class ThreadPool(object):
    def __init__(self, task, Worker=WorkerThread,
                 num_threads = 10, logging_on=False):
        self.req_queue = Queue()
        self.results_queue = Queue()
        ...

    def __enter__(self):
        for i in range(self.num_threads):
            name = f'Thread {i}'
            self.thr_pool.append(self.Worker(self.task,
                                 self.req_queue, self.results_queue,
                                 name, self.sentinel, self.logging_on))

        for th in self.thr_pool:
            th.start()

        return self.req_queue

    def __exit__(self, typ, val, tb):
        for th in self.thr_pool:
            self.req_queue.put((self.sentinel,))

        for th in self.thr_pool:
            th.join()
```

Create the job and result queues

Instantiate and start the threads

Place a "shutdown" message on queue so each thread gets one

Wait for each thread to finish

Contextlib Module

- The contextlib module provides additional tools for creating context managers
  - closing
  - @contextmanager
  - redirect_stdout *(as of Py3.4)*
  - ContextDecorator *(as of Py3.2)*

contextlib.closing *(1 of 2)*

- Objects with a close() method can be "adapted" to use a with control by wrapping them in closing()

```
from contextlib import closing

class Cache:
    def __init__(self, func):
        self.cache = {}
        self.func = func
        print(f'Beginning caching of {self.func.__name__}')

    def __call__(self, *args, **kwargs):
        if args not in self.cache:
            ret = self.func(*args, **kwargs)
            self.cache[args] = ret
        else:
            print(f'hitting cache with...{args}')
        return self.cache[args]

    def close(self):
        print('Clearing cache...')
        self.cache.clear()
```

contextlib.closing() provides an __enter__ and __exit__
__exit__ simply invokes the wrapped object's close() method

Objects wrapped by closing() **MUST** have a close() method

contextlib.closing *(2 of 2)*

Our add() function will draw from the cache when arguments are re-used

```
@Cache
def add(a, b):
    return a + b

print(add(5, 3))
print(add(3, 5))
print(add(5, 3))
add.close()

with closing(add) as add:
    print(add(5, 3))
    print(add(3, 5))
    print(add(5, 3))
```

```
8
8
hitting cache with...(5, 3)
8
Clearing cache...
```

Clears the cache

Will automatically clear the cache when the with control ends

```
8
8
hitting cache with...(5, 3)
8
Clearing cache...
```

contextlib.contextmanager

- @contextmanager decorator converts a generator into context manager

```
@contextlib.contextmanager
def elapsed(name=None):
    timer_name = name
    begin = time.time()
    try:
        yield timer_name
    except Exception as err:
        print(err)
    finally:
        end = time.time()
        print(f'elapsed time: {end - begin:.3f} seconds')

with elapsed():
    entries = open('../resources/access_.log').read()
    # raise Exception('An error occurred.')
```

Stuff *before* the *yield* is equivalent to __enter__

The yielded value shows up in the 'as' clause

Stuff *after* the *yield* is equivalent to __exit__

elapsed time: 0.060 seconds

contextlib.redirect_stdout

- redirect_stdout() can send output to another location, such as a file or in-memory buffer

```
import contextlib
import io

sbuf = io.StringIO()
with contextlib.redirect_stdout(sbuf):
    print('This goes into the StringIO buffer.')
    print('So does this.')

print(f'Result:\n{sbuf.getvalue()}')
```

This technique, although limited to Py3.4+, can allow temporarily redirecting content to a file

contextlib.ContextDecorator

- Allows a function to be decorated by a context manager

```
from contextlib import ContextDecorator

class ContextMgr(ContextDecorator):
    def __enter__(self):
        print('entering')
        return self

    def __exit__(self, typ, value, traceback):
        print('exiting')
        return False

@ContextMgr
def f():
    print('the work')

f()
```

```
entering
the work
exiting
```

Arrow

pip install arrow

- Arrow allows for creating and formatting dates and times

```
import arrow

utc = arrow.utcnow()
my_now = arrow.now()
my_now_pacific = arrow.now('US/Pacific')

print(utc)
print(my_now)
print(my_now_pacific)

print(my_now.format())
print(my_now.format('MMM/DD/YY'))
print(my_now.humanize())

print(my_now.year)
print(my_now.day)
print(my_now.month)

some_date = arrow.get('2020-05-08 22:05:48', 'YYYY-MM-DD HH:mm:ss')
print(some_date.format('MMM/DD/YY'))
print(some_date.humanize())

extract_date = arrow.get('I marched in November, 2021.', 'MMMM, YYYY')
print(extract_date.format('MMMM, YYYY'))
```

```
2018-04-13T22:05:48.220782+00:00
2018-04-13T16:05:48.220782-06:00
2018-04-13T15:05:48.314534-07:00
```

```
2018-04-13 16:49:51-06:00
Apr/13/18
just now
```

```
2018
13
4
```

```
May/08/20
In 25 months
```

```
November, 2021
```

tqdm ("taqadum")

pip install tqdm

- tqdm easily adds a progress bar to any iterable

```
import pandas as pd
import tqdm
avgs = []

batting = pd.read_csv('../resources/baseball/Batting.csv',
                      usecols=('yearID', 'AB', 'H'))

batting = batting.loc[(batting['yearID'] >= 1957) &
                      (batting['AB'] >= 502)]

for idx, row in tqdm.tqdm(batting.iterrows(), total=batting.shape[0]):
    avg = row['H'] / row['AB']
    avgs.append(avg)
```

Simply wrap the iterable in a tqdm.tqdm() call

```
28%|                    | 1427/5013 [00:22<00:56, 64.02it/s]
```

Chapter 2 - Summary

- Numerous useful modules in the standard library can aid in automating environments
- As you advance your Python skills, these modules will becoming more and more familiar to you
  - contextlib provides utilities for customizing the with control
  - pathlib, fnmatch, shutil, filecmp assist with searching, copying, and comparing files
  - timeit, psutil helps measure performance of code

Chapter 3
Advanced Classes

Advanced Features of Python's Classes

Chapter 3 - Overview

- Modeling Classes
- Descriptors
- Inheritance
- Abstract Classes
- Callable Classes
- Design Patterns

Creating Classes

Py 3 does not require (object), but Py 2 new-style classes do (see next slide)

```
class Race(object):
    def __init__(self, name: str = '', distance: float = 0.0,
                 units: str = 'km', typ: str = '',
                 location: str = '', size: int = 0):
        self.name = name
        self.distance = distance
        self.units = units
        self.type = typ
        self.location = location
        self.size = size

    def __str__(self):
        return f'{self.name} ({self.distance} {self.units})'

    def __repr__(self):
        return f'({self.name}) {type(self)}'
```

```
race1 = Race('BolderBOULDER', 10, 'km')
race2 = Race('Chicago Marathon', 26.2, 'mi',
             location='Chicago, IL', typ='running')
print(race1)
print([race1, race2])
```

New-style vs Old-style Classes

- Python 2.2 introduced "new-style" classes

```
class Person(object): pass
```

New-style class

  - Mostly backward compatible with classic classes

```
class Person: pass
```

Old-style (classic) class

  - Main differences between new and classic classes:
    - Introduction of __slots__
    - Descriptors: staticmethods, classmethods, properties
    - Modified multiple inheritance search procedure

Note: All Python 3 classes are automatically new-style classes

Static vs Class Methods

```
from importlib.resources import open_text

class Webster:
    @staticmethod
    def read_data(package, fn):
        results = {}
        try:
            with open_text(package, fn, encoding='utf-8') as f:
                for line in f:
                    try:
                        word, definition = line.split(maxsplit=1)
                        results[word] = definition
                    except ValueError:
                        pass
        except IOError as err:
            print('Error reading file.', err, file=sys.stderr)

        return results

webster = Webster.read_data('resources', 'dictionary.txt')
print(webster['Precocious'])
```

Methods decorated with **@staticmethod** do not require a self argument

Access static methods using Classname.methodname()

Static vs Class Methods *(continued)*

```
class Webster2:
    package = 'resources'
    fn = 'dictionary.txt'

    @classmethod
    def read_data(cls):
        results = {}
        try:
            with open_text(cls.package, cls.fn) as f:
                for line in f:
                    try:
                        word, definition = line.split(maxsplit=1)
                        results[word] = definition
                    except ValueError:
                        pass
        except IOError as err:
            print('Error reading file.', err, file=sys.stderr)

        return results


webster = Webster2.read_data()
print(webster['Precipitous'])
```

Instead of a static method, a class method may be more appropriate if values from the class will be accessed.

__slots__

- New-style classes (and all Py3 classes) can define the set of attributes that instances have

```
class Race(object):

    __slots__ = ('name', 'distance', 'units',
                 'type', 'location', 'size')

    def __init__(self, name: str = '', ...):
        ...(constructor definition same as before)...


race1 = Race('BolderBOULDER', 10, 'km')
race1.foo = 'hello world'
print(race1.__dict__)
```

Prevents the addition of new attributes

Both of these statements generate an AttributeError!

Main intention of __slots__ is to create classes that are not dict-based, but rather descriptor-based

Notes about using __slots__

- Advantages:
  - When __slots__ is used, the instance is not implemented with a dictionary backing
    - Therefore, *it has no __dict__ attribute*
  - Instances tend to consume less memory
  - When apps have large numbers of instances, __slots__ can be beneficial on memory allocation

- Disadvantages:
  - Loss of flexibility of objects
  - Problems with inheritance hierarchies

Properties

- New-style classes (and all Py 3 classes) support properties *(Python's version of getters & setters)*

```
class Race(object):

    def __init__(self, name='', distance=0.0, units='km', typ='',
                 location='', size=0):
        self.name = name
        self.distance = distance
        ...other properties not shown...

    @property
    def distance(self):
        return self._distance

    @distance.setter
    def distance(self, distance):
        if distance < 0:
            distance = 0
        self._distance = distance

race1 = Race('BolderBOULDER', -10, 'km')
print(race1)
```

Invokes the setter

Will be filtered by the setter

What are all these get-set methods?

__get__
__getattr__
__getattribute__
__getitem___

__set__
__setattr__
__setitem___

__del__
__delattr__
__delete__
__delitem__

What are all these methods for?...

__getitem__, __setitem__, __delitem__

```
class TopMovies:
    def __init__(self, movies=None):
        if movies is None:
            self.movies = [
                ('Avatar', 2787965087, 2009),
                ('Titanic', 2186772302, 1997),
                ('Jurassic World', 1651382496, 2015),
                ('The Avengers', 1519557910, 2012),
                ('Furious 7', 1511726205, 2015)]

    def __len__(self):
        return len(self.movies)

    def __getitem__(self, key):
        return self.movies[key]

    def __setitem__(self, key, value):
        self.movies[key] = value

    def __delitem__(self, key):
        del self.movies[key]

    def __iter__(self):
        return iter(self.movies)
```

These allow an object to be accessed using *index* and *slice* notation

```
tm = TopMovies()
del tm[3]
for movie in tm:
    print(movie)

print(tm[0])
print(tm[-2:])
```

__getattribute__

- __getattribute__ is invoked *before all* access to attributes

```
class Foo(object):
    def __init__(self):
        self.bar = 10
        self.baz = 20

    def __getattribute__(self, attr):
        if attr == 'bar':
            return 'hello'

        return object.__getattribute__(self, attr)

f = Foo()
print(f.bar, f.baz)
```

hello 20

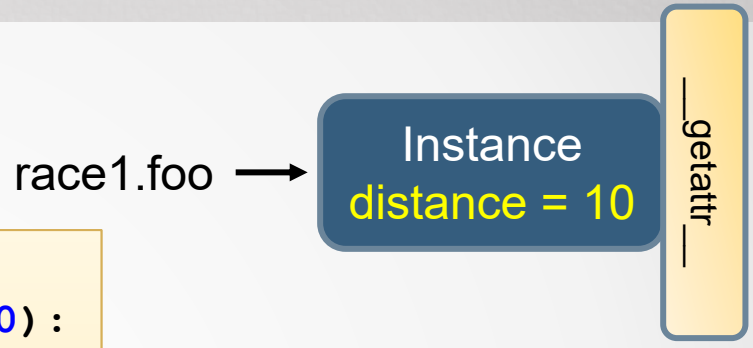
f.bar

__getattribute__

Instance
bar = 10

__getattr__, __setattr__

- __getattr__ provides *fallback* access to attributes




```
class Race(object):
    def __init__(self, name='', distance=0.0):
        self.name = name
        self._distance = distance

    ...distance property not shown...

    def __getattr__(self, attr):
        print(f'getattr called for: {attr}')
        try:
            retval = self.__dict__[attr]
        except KeyError:
            retval = None
        return retval
```

```
race1 = Race('BolderBOULDER', 10)
print(race1)
print(race1.foo)
```

foo attribute does not exist, so __getattr__ will be called

Introducing Descriptors

Descriptors provide a way to *customize behavior* when an attribute is referenced

```
class FullName:
    def __get__(self, instance, owner):
        return ' '.join([instance.first, instance.last])

    def __set__(self, instance, new_name):
        names = new_name.split(' ')
        if len(names) == 1:
            first = ''
            last = new_name
        else:
            first = names[0]
            last = ' '.join(names[1:])
        instance.first = first
        instance.last = last
```

```
class Person:

    full_name = FullName()

    def __init__(self, first='', last=''):
        self.first = first
        self.last = last
```

```
p1 = Person('John', 'Smith')
p2 = Person('Sally','Jones')
p3 = Person('Fred', 'Davis')
print(p1.full_name)
p2.full_name = 'Sally Van Wilder'
print(p2.full_name)
p3.full_name = 'Freddy Davis'
print(p3.full_name)
```

Calls descriptor's __set__

Calls descriptor's __get__

Using Descriptors *(1 of 2)*

```
class Convert:
    conversions = {'mi': 1.0, 'km': 0.621, 'm': 0.000621, 'ft': 0.0001894}

    def __init__(self):
        self.data = WeakKeyDictionary()

    def __get__(self, instance, owner):
        return self.data.get(instance, 0)

    def __set__(self, instance, value):
        convert_value = self.conversions.get(instance.units.lower(), 0)
        self.data[instance] = value * convert_value
        instance.units = 'mi'
```

```
class Race:
    distance = Convert()

    def __init__(self, name='', distance=0.0, units='mi'):
        self.name = name
        self.units = units
        self.distance = distance

    def __str__(self):
        return f'{self.name}: {self.distance:.1f} {self.units}'
```

Place descriptors at the class level

__get__ and __set__ invoked automatically when descriptor are the same name as attributes

Using Descriptors *(2 of 2)*

```
race1 = Race('Chicago Marathon', 26.2)
print(race1)

race2 = Race('BolderBOULDER', 10, 'km')
print(race2)

race3 = Race('Hot Chocolate 5k', 16404, 'ft')
print(race3)

race4 = Race('Carlsbad 5000', 5000, 'm')
print(race4)

race5 = Race('Alien Sprint', 100, 'blips')
print(race5)
```

Output:

```
Chicago Marathon: 26.2 mi
BolderBOULDER: 6.2 mi
Hot Chocolate 5k: 3.1 mi
Carlsbad 5000: 3.1 mi
Alien Sprint: 0 mi
```

If you remove the following line:

```
distance = Convert()
```

the code would still be functional,
but would output the following:

```
Chicago Marathon: 26.2 mi
BolderBOULDER: 10.0 km
Hot Chocolate 5k: 16404.0 ft
Carlsbad 5000: 5000.0 m
Alien Sprint: 100.0 blips
```

Inheritance

```
class Event(object):
    def __init__(self, name='', location='', date=None):
        self.name = name
        self.location = location
        self.date = date

    def __str__(self):
        return f'{self.name}'
```

To ensure base class constructors and methods are executed, invoke them explicitly

```
class Race(Event):
    def __init__(self, name='', location='', date=None,
                 distance=0.0, units='km', typ=''):
        Event.__init__(self, name, location, date)
        self.distance = distance
        self.units = units
        self.type = typ

    def __str__(self):
        return f'{Event.__str__(self)} ({self.distance} {self.units})'
```

```
evt1 = Event('Group Trash Pickup', 'Downtown')
print(evt1)
evt2 = Race('BolderBOULDER', 'Bolder, CO', date(2016, 5, 30), 10,
            'km', 'Running')
print(evt2)
```

Prefer super()

```
class Event(object):
    def __init__(self, name='', location='', date=None):
        self.name = name
        self.location = location
        self.date = date

    def __str__(self):
        return f'{self.name}'
```

super() should not include the self parameter when used

```
class Race(Event):
    def __init__(self, name='', location='', date=None,
                 distance=0.0, units='km', typ=''):
        super().__init__(name, location, date)
        self.distance = distance
        self.units = units
        self.type = typ

    def __str__(self):
        return f'{super().__str__()} ({self.distance} {self.units})'
```

```
evt1 = Event('Group Trash Pickup', 'Downtown')
print(evt1)
evt2 = Race('BolderBOULDER', 'Bolder, CO', date(2016, 5, 30), 10,
            'km', 'Running')
print(evt2)
```

Abstract Classes

- The abc module helps create abstract base classes
  - The abstract base class *cannot be instantiated*
  - The child class inherits methods from the abstract base class
  - Should implement the unimplemented base class methods

Python 3.4+

```
import abc

class AbstractBase(abc.ABC):

    def __init__(self, item):
        self.item = item
        self.do_action()

    @abc.abstractmethod
    def do_action(self):

        pass
```

Abstract Classes Example

Subclasses must define this method

```
class TextMessage(abc.ABC):
    def __init__(self, msg):
        self.msg = msg

    @abc.abstractmethod
    def get_text(self):
        pass
```

```
value = 'Meeting: 3pm.'
msg1 = HtmlMessage(value)
msg2 = JsonMessage(value)
print(msg1.get_text())
print(msg2.get_text())
msg3 = TextMessage('No Meeting.')
```

```
class HtmlMessage(TextMessage):
    def __init__(self, msg):
        super().__init__(msg)

    def get_text(self):
        return f'<span>{self.msg}</span>'
```

Your meeting is at 3pm.
{ "text": "Your meeting is at 3pm." }
TypeError: Can't instantiate abstract class

```
class JsonMessage(TextMessage):
    def __init__(self, msg):
        super().__init__(msg)

    def get_text(self):
        return f'{{ "text": "{self.msg}" }}</span>'
```

For the Exercise: Using *requests*

- requests (third-party) has become a preferred tool for making HTTP requests

pip install requests

```
import requests

r = requests.get('http://www.yahoo.com')
print(r.url)
print(r.status_code)
print(r.headers)
print(r.request.headers)

print(r.text)

print(r.json())
```

Returns a string of the response

Converts a JSON response to a dict

For the Exercise: Jinja2

```
from jinja2 import Environment, FileSystemLoader
from jinja2.exceptions import TemplateError, TemplateNotFound
env = Environment(loader=FileSystemLoader('./tmpl'))

races = [
    Race('BolderBOULDER', 10000, 'm'),
    ...
]

try:
    tmpl = env.get_template('example_tmpl.jinja')
    print(tmpl.render(data=races))
except (TemplateNotFound, TemplateError) as e:
    print('Error: {0} {1}'.format(type(e), e))
```

example_tmpl.jinja

```
{% for item in data %}
    {{item.name}} - {{item.distance}} {{item.units}}
{% endfor %}
```

Your Turn! - Task 3-1: *Movie Query (1 of 3)*

- Create a script to query a remote website that returns JSON-based movie information
- Work from the ch03_adv_classes/starter folder

Note: if no internet access is available, uncomment the **import mocklab** line in task3_1.py

```
Enter a movie phrase: titanic
 1 - Titanic
 2 - Titanic
 ...
 12 - Titanic Disaster
 13 - Titanic 2000
 14 - Titanic: The Complete Story
 15 - Asia's Titanic
 16 - Saving the Titanic
 17 - Titanic 2
 18 - Words of the Titanic
 19 - Titanic Arrogance
 20 - Beyond Titanic
Enter number for details: 1
Title:                Titanic
Released:             1997-11-18
Runtime:              194
Budget:               $200,000,000
Earnings:             $1,845,034,188
Tagline:              Nothing on Earth
could come between them.
```

Your Turn! - Task 3-1: *Movie Query (2 of 3)*

- Sample search results:

http://api.themoviedb.org/3/search/movie?api_key={key}&query=avengers


```
{
    "page":1,  "total_results":40,  "total_pages":2,
    "results":[{
        "vote_count":10876,  "id":24428,
        "title":"The Avengers",
        "original_title":"The Avengers",
        "overview":"When an unexpected enemy emerges ...",
        "release_date":"2012-04-25", ...
     }, ...there will be additional results also
     ]
}
```


- Sample detailed title results:

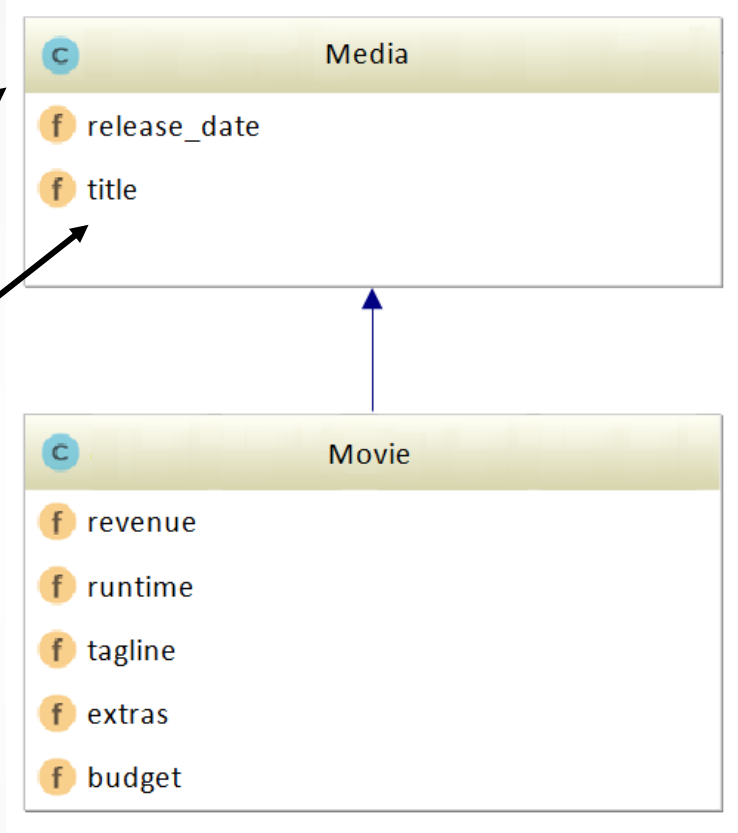
https://api.themoviedb.org/3/movie/{id}?api_key={key}


```
{
    "runtime":143, "title":"The Avengers", "budget":220000000,
    "id":24428,
    "overview":"When an unexpected enemy emerges...",
    "release_date":"2012-04-25",
    "revenue":1519557910,
    "tagline":"Some assembly required.",
}
```

Your Turn! - Task 3-1: *Movie Query (3 of 3)*

- Create classes to contain movie data as follows:

abstract class



- Create a *descriptor* for the *title* property to make it read only

- Prompt for a movie search phrase, display a list of results
- Prompt to select a title which then retrieves specific details
- Use the provided Jinja2 template for rendering details
- Edit starter/movie/**movie.py** and starter/**task3_1.py**

Creating Callable Classes

- To make a class instance directly callable, implement the __call__ magic method

```
import datetime

class Today(object):
    def __init__(self, fmt='%b %d, %Y'):
        self.fmt = fmt

    def __call__(self):
        return datetime.date.today().strftime(self.fmt)


t = Today()
print(t())
```

May 13, 2022

Callable classes can make the user think an object is a simple function, while the creator gains the benefits of entire classes

Operator Overloading

- To overload an operator, you must override one of the magic methods of that class:

Magic Method	Description
__init__	Constructor
__del__	Destructor
__str__	toString() equivalent
__add__	+ operator
__iter__	Object's iterator
__len__	Length of object
__cmp__	How to compare the object
__eq__	Is object equal to
__hash__	Object's hash value
__lt__	Less than (also: gt, ge, le, eq, ne)

Operator Overloading Example *(1 of 2)*

```
from functools import total_ordering

@total_ordering
class Race:
    distance = Convert()

    def __init__(self, name='', distance=0.0, units='km'):
        self.name = name
        self.units = units
        self.distance = distance

    def __lt__(self, other):
        return self.distance < other.distance

    def __str__(self):
        return f'{self.name} ({self.distance} {self.units})'
```

Providing one comparison magic method implements the other methods (lt, gt, le, ge)

Operator Overloading Example *(2 of 2)*

```
races = [
    Race('BolderBOULDER', 10000, 'm'),
    Race('Chicago Marathon', 26.2, 'mi'),
    Race('Carlsbad 5000', 5, 'km'),
    Race('Country Music Half Marathon', 13.1, 'mi'),
    Race('Capitol Mile', 5280, 'ft'),
    Race('Peachtree Road Race', 10, 'km'),
]

print(races[5] >= races[0])
races.sort()
for race in races:
    print(race)
```

Because of @total_ordering, custom sorting now occurs

```
True
Capitol Mile (1.000032 mi)
Carlsbad 5000 (3.105 mi)
BolderBOULDER (6.21 mi)
Peachtree Road Race (6.21 mi)
Country Music Half Marathon (13.1 mi)
Chicago Marathon (26.2 mi)
```

Mini-Task 3

- Revisit the Movie class from Task 3-1
- Overload the __len__() magic method returning the movie's runtime
- Work from your completed starter/movie/movie.py file
- Test your solution by running the already completed mini_task3.py
  - If you didn't finish Task3-1, you can change the Movie import to use the *solution* folder instead

Singleton Pattern

- Singletons in Python are <u>best</u> implemented using a *module*
  - Modules are only loaded once

If you *must* use a class to implement a singleton, the solution below works

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is not None:
            return cls._instance
        cls._instance = object.__new__(cls)
        cls._instance.args = list(args)
        cls._instance.kwargs = kwargs
        return cls._instance
```

Does one object exist yet? If so, return it

Otherwise, create it

```
s1 = Singleton(10)
s1.args[0] += 1
s2 = Singleton()
print(s2.args[0])
```

11

Creates the object

Second object is the same as the first

Overloaded Constructor Pattern

An approach for simulating overloaded constructors is to use @classmethod

```
class Race(object):
    distance = Convert()

    def __init__(self, name, distance=0.0, units='km'):
        self.name = name
        self.units = units
        self.distance = distance

    @classmethod
    def race10k(cls, name):
        return cls(name, 10, 'km')

    @classmethod
    def race_marathon(cls, name):
        return cls(name, 26.2, 'mi')
```

cls() invokes the __init__()

```
race1 = Race('Carlsbad 5000', 5, 'km')
race2 = Race.race10k('BolderBOULDER')
race3 = Race.race_marathon('Chicago Marathon')
```

Carlsbad 5000 (3.105 mi)
BolderBOULDER (6.21 mi)
Chicago Marathon (26.2 mi)

Factory Pattern

```
import abc
class Message(abc.ABC):
    def __init__(self, msg=''):
        self.msg = msg
    @abc.abstractmethod
    def get_output(self):
        pass
```

Base Class with undefined *get_output()* method

```
class Text(Message):
    def __init__(self, msg):
        super().__init__(msg)

    def get_output(self):
        return self.msg

class Html(Message):
    def __init__(self, msg):
        super().__init__(msg)

    def get_output(self):
        return f'<span>{self.msg}</span>'

class Json(Message):
    def __init__(self, msg):
        super().__init__(msg)

    def get_output(self):
        return f'{{ "msg": "{self.msg}" }}'
```

Subclasses define their own *get_output()* methods

Using the Factory Pattern

A mapping is used to define the type of class to instantiate

```
class MessageFactory:
    message_formats = {'text': Text, 'html': Html, 'json': Json}

    @classmethod
    def create_msg(cls, msg_type, msg):
        return cls.message_formats.get(msg_type, Text)(msg)
```

Instantiation

```
msg = MessageFactory.create_msg('html', 'Sample HTML msg.')
print(type(msg), msg.get_output())
```

<class Html> Sample HTML msg.

```
msg = MessageFactory.create_msg('json', 'Sample JSON msg.')
print(type(msg), msg.get_output())
```

<class Json> { "msg": "Sample JSON msg." }

```
msg = MessageFactory.create_msg('text', 'A simple text msg.')
print(type(msg), msg.get_output())
```

<class Text> A simple text msg.

Chapter 3 - Summary

- To assist with the numerous magic methods discussed in this chapter, use this handy reference:

https://minhhh.github.io/posts/a-guide-to-pythons-magic-methods

- Python's classes, when implemented with proper magic methods can be very effective and useful tools in your code development arsenal

Chapter 4
Unit Testing and Mock Objects

Testing Code with Mock Objects

Chapter 4 - Overview

- Unit Testing
- Mock Objects
- Monkey Patching

Unit Testing Practices

- Unit tests isolate pieces of code to verify if they work as expected
- Test case methods (TCM) should
  - Test one criterion only (have one assert per test)
  - Use easily understood names for TCMs
  - Be short, quick, and simple
  - Tests should not touch any external resources (databases, network, file system, etc.)
  - Tests should only fail when a bug is introduced into the code being tested, not when
  - Use mock objects for more complex code

unittest

- The unittest module (based on the classic Junit from Java) comes packaged within the Python Standard Library
- Unit test methods perform assertions comparing expected vs. actual behavior
- Some of the unittest methods include

Method	Checks for
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(a)	a == True
assertFalse(a)	a == False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(a)	A is None
assertIsNotNone(a)	a is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	A not in b
assertIsInstance(a, b)	isintance(a, b)
assertIsNotInstance(a, b)	not isinstance(a, b)

Example Using unittest (1 of 3)

- As an example of using the unittest module, consider a test for the following class

sample.py

```
class Operation:
    def __init__(self, value: float):
        self.value = value

    def add(self, second):
        return self.value + second

    def subtract(self, second):
        return self.value - self.value

    def multiply(self, second):
        return self.value * second
```

This is purposely an error!

Example Using unittest (2 of 3)

```
class TestOperation(unittest.TestCase):

    def setUp(self):
        self.operation = Operation(5)

    def test_add(self):
        second = 5
        actual = self.operation.add(second)
        expected = 10
        self.assertEqual(expected, actual, 'add() not correct')

    def test_subtract(self):
        second = 4
        actual = self.operation.subtract(second)
        expected = 1
        self.assertEqual(expected, actual, 'subtract() not correct')

    def test_multiply(self):
        second = 3
        actual = self.operation.multiply(second)
        expected = 15
        self.assertEqual(expected, actual, 'multiply() not correct')
```

setup() is executed once before each test_XXX() method

Running the Tests

- Run the previous test module several ways:
  - Add the following code:

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

  - To run the test from a command line

```
python -m unittest <test_filename> -v
```

  - Or in PyCharm, right-click the class, select *Run Unittests...*

unittest.mock

- Python3 provides the unittest.mock submodule
  - Use mock when testing functions that involve complex objects (like a request object) or touch external systems
- unittest.mock module items

Mock	-	class providing behavior that replaces external influences
MagicMock	-	same as Mock but provides stubs for magic methods
@patch	-	decorates a function applying a mock object to resources used within the function. It "monkey patches" items within the test function
mock_open	-	A helper to create a mock which replaces the open() function

The @patch() Decorator

- patch() is used to replace objects with something else
- It is typically used as a decorator
  - It can also be used as a context manager or using .start() and .stop methods

```
patch(target, new)
```

If patch() is used as a *decorator*, then *new is omitted*. If *new* is omitted, a MagicMock object will replace the target

target	-	The object to be replaced

new	-	The object to insert instead

Understanding How patch() Works

sample.py

```
def my_function():

    return 'welcome to my_function!'
```

We want to mock (replace) this function with something else

```
import unittest.mock as mock
import ch04_unittest.sample as samp


@mock.patch('ch04_unittest.sample.my_function')
def invoke_me(mock_obj):

    mock_obj.return_value = 'my_function has been mocked!'

    result = samp.my_function()
    print(result)



invoke_me()
```

This is what will be replaced

mock_obj is what replaces it

my_function has been mocked!

my_function has been replaced with an object that simulates the return value for our original my_function()

How Does Monkey Patching Work? (1 of 2)

- Applying patch() can assist with testing, fixing bugs, and more

```
patch_items = {}

def patch(module, attr, newitem):
    default = object()
    old = getattr(module, attr, default)
    if old is not default:
        patch_items[(module, attr)] = old
    setattr(module, attr, newitem)

def unpatch(module, attr):
    default = object()
    olditem =
      getattr(module, attr, default)
    if olditem is default:
        return olditem
    old = patch_items[(module, attr)]
    setattr(module, attr, old)
```

This function accepts an attribute from a *module* and replaces it with *newitem*

This function restores the originally replaced function back

How Does Monkey Patching Work? (2 of 2)

```
def get_files(dirname):
    return
os.listdir(dirname)
```

```
def listdir_patch(dirname):
    return ['no files found']

patch(os, 'listdir', listdir_patch)
print(get_files('.'))

unpatch(os, 'listdir')
print(get_files('.'))

patch(sys, 'path', 'hello!')
print(sys.path)

unpatch(sys, 'path')
print(sys.path)
```

Replaces os.listdir()

Restores os.listdir()

['no files found']

['01_singleton.py', '02_overloaded_constructor.py', ...]

Replaces sys.path with 'hello'

hello

Restores original sys.path

['c:\\python310', 'c:\\python310\lib', ...]

@patch with *side_effect*

- @patch() replaces specified objects with something else
  - side_effect can be used to point to a function

sample.py

We want to unit test get_files() but patch os.listdir()

```
import os
def get_files(directory):
    return os.listdir(directory)
```

```
import unittest
import unittest.mock as mock
from ch04_unittest.sample import get_files

def os_listdir_mock(dir):
    return ['a', 'b', 'c', dir]

class TestGetFiles(unittest.TestCase):
    @mock.patch('os.listdir', side_effect=os_listdir_mock)
    def test_get_files(self, os_listdir):
        actual = get_files('foo')
        expected = ['a', 'b', 'c', 'foo']
        self.assertSequenceEqual(expected, actual)
```

We monkey patch the new function in place of os.listdir() calls

@patch with *return value*

sample.py

```
import os
def get_files(directory):
    return os.listdir(directory)
```

Instead of providing a mock function, in many cases a *return value* from that function can be simulated

```
import unittest
import unittest.mock as mock

from ch04_unittest.sample import get_files

class TestGetFiles(unittest.TestCase):
    @mock.patch('os.listdir', return_value=['a', 'b', 'c'])
    def test_get_files(self, os_listdir):
        actual = get_files('foo')
        expected = ['a', 'b', 'c']
        self.assertSequenceEqual(expected, actual,
                                 'different return from os.listdir')
        self.assertTrue(os_listdir.call_count == 1)
        self.assertTrue(os_listdir.called)
```

os_listdir is a MagicMock object

The mock object provides additional information as well

@patch-ing Multiple Items

sample.py

```
def only_files(directory):
    if os.path.isdir(directory):
        return [f for f in os.listdir(directory)
                                    if os.path.isfile(f)]
```

01_mocking_3.py

```
def os_path_isfile_mock(f):
    return f != 'not file'

class TestOnlyFiles(unittest.TestCase):
    @mock.patch('os.path.isfile', return_value=True)
    @mock.patch('os.path.isdir', return_value=True)
    @mock.patch('os.listdir', return_value=['a', 'b', 'c'])
    def test_only_files_normal(self, os_listdir, os_path_isdir,
                                                os_path_isfile):
        actual = only_files('foo')
        expected = ['a', 'b', 'c']
        self.assertSequenceEqual(expected, actual)
```

Your Turn! Task 4-1 *(Optional)*

- Apply unit tests to the *Movie App*
  - Complete the unit tests that have been started for you within starter/test/testapp.py and test_movie.py
  - The refactored application has been moved into functions within app.py
  - *Run test/suite.py* which is designed to run your test cases or if using PyCharm, you may run the test files individually
  - Create tests to verify the following functions work

    1) Movie's read-only descriptor (within test_movie.py),
    2) render_movie,
    3) movie_search(),
    4) detailed_search()

    (within test_app.py)

Chapter 4 - Summary

- Unit tests should be fast and only fail when a bug is introduced into source code
- Testing external resources is not the goal of a unit test
  - The mock module provides a MagicMock object that can "simulate" those external resources
- There is no need to test Python standard library or third-party code; therefore, it can be mocked
- Use the patch decorator to "replace" third-party and standard library code

Chapter 5
Data Analysis and NumPy

Introducing Data Analysis Tools within Python

Chapter 5 - Overview

IPython and Jupyter Notebooks

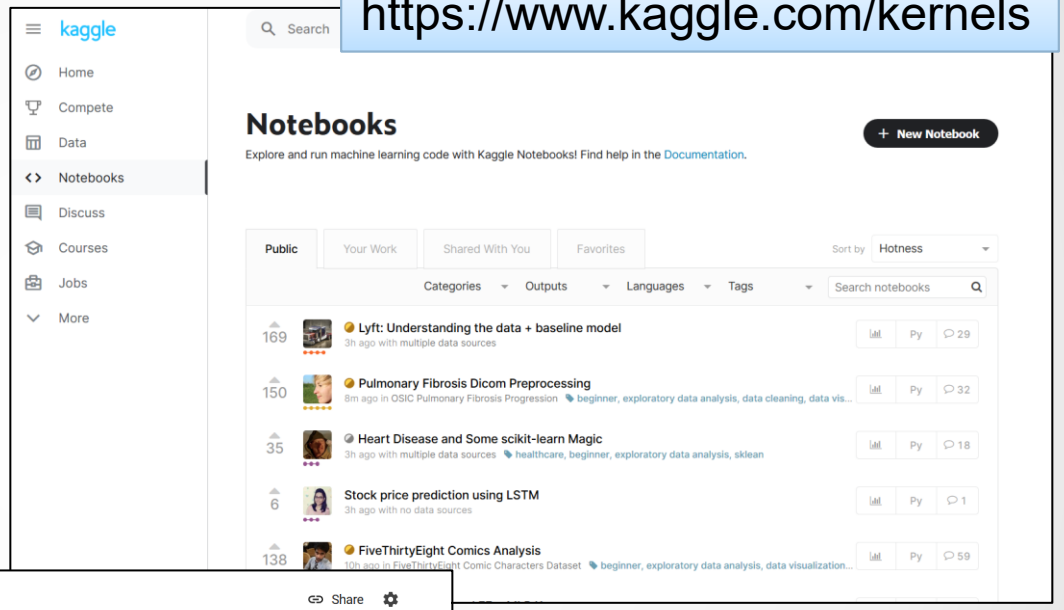
NumPy

Matplotlib

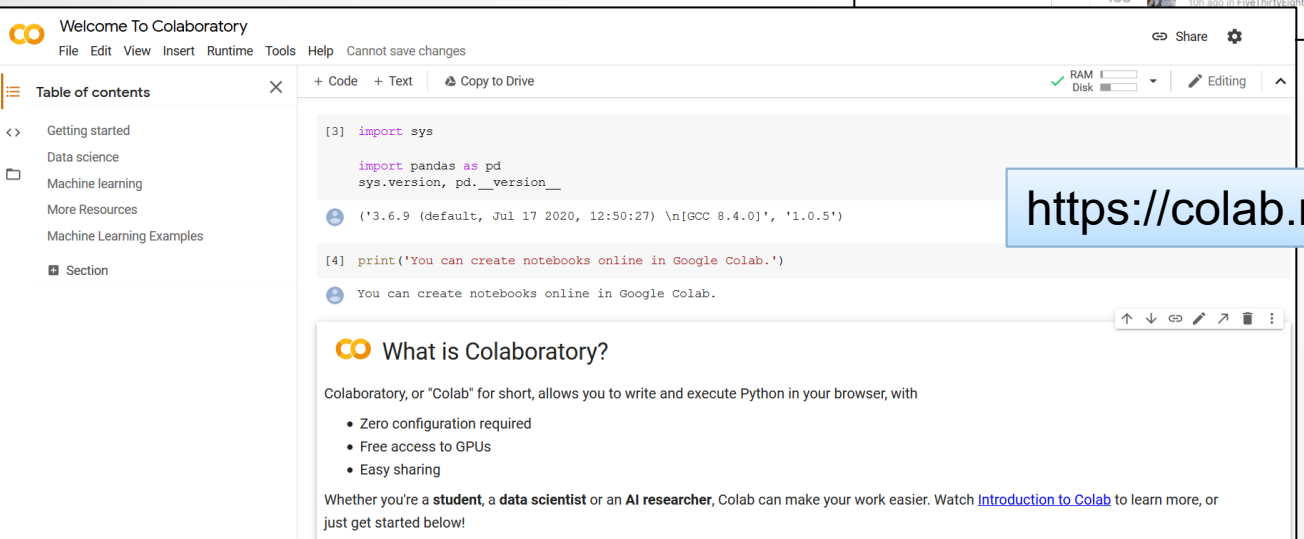
IPython and Jupyter

- IPython is a Python-based interactive computing environment
  - Jupyter Notebook provides a web-based frontend for it
  - You can explore public notebooks and even create your own online

https://www.kaggle.com/kernels



https://colab.research.google.com



Launching Jupyter

- To use Jupyter, it must be installed
  - Some distributions, such as Anaconda, already provide it

```
pip install ipython
```

```
pip install jupyter
```

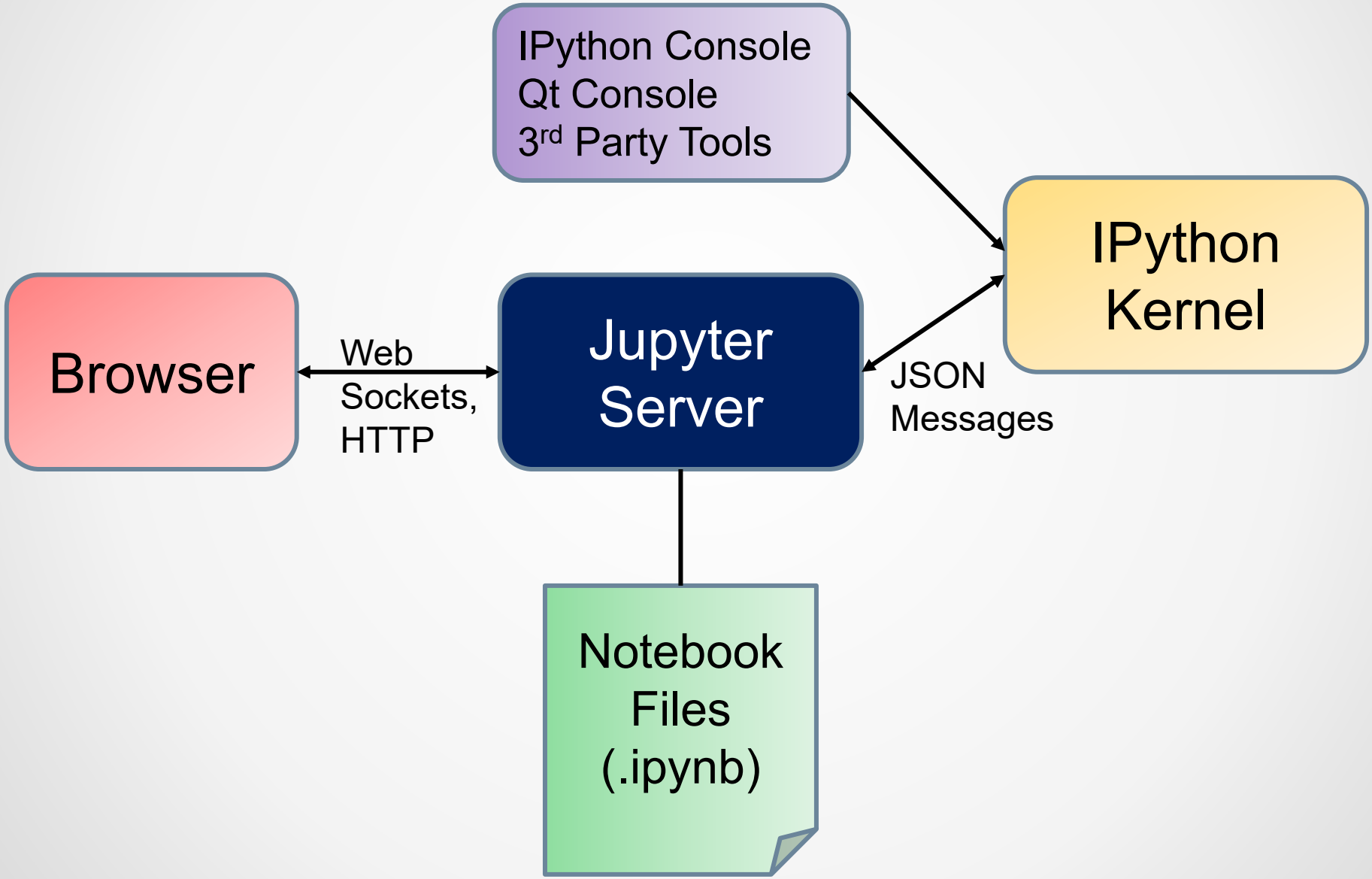
  - Launch a notebook (from a desired home directory)

```
jupyter notebook
```

```
c:\temp\notebooks>jupyter notebook
[W 14:12:24.283 NotebookApp] Unrecognized JSON config file version, assuming ver
sion 1
[I 14:12:25.132 NotebookApp] [nb_conda_kernels] enabled, 1 kernels found
[I 14:12:25.576 NotebookApp] [nb_anacondacloud] enabled
[I 14:12:25.581 NotebookApp] [nb_conda] enabled
[I 14:12:25.647 NotebookApp] \u2713 nbpresent HTML export ENABLED
[W 14:12:25.648 NotebookApp] \u2717 nbpresent PDF export DISABLED: No module nam
ed 'nbbrowserpdf'
[I 14:12:25.767 NotebookApp] Serving notebooks from local directory: c:\temp\not
ebooks
[I 14:12:25.768 NotebookApp] 0 active kernels
[I 14:12:25.768 NotebookApp] The Jupyter Notebook is runni
t:8888/
[I 14:12:25.768 NotebookApp] Use Control-C to stop this se
 kernels (twice to skip confirmation).
```

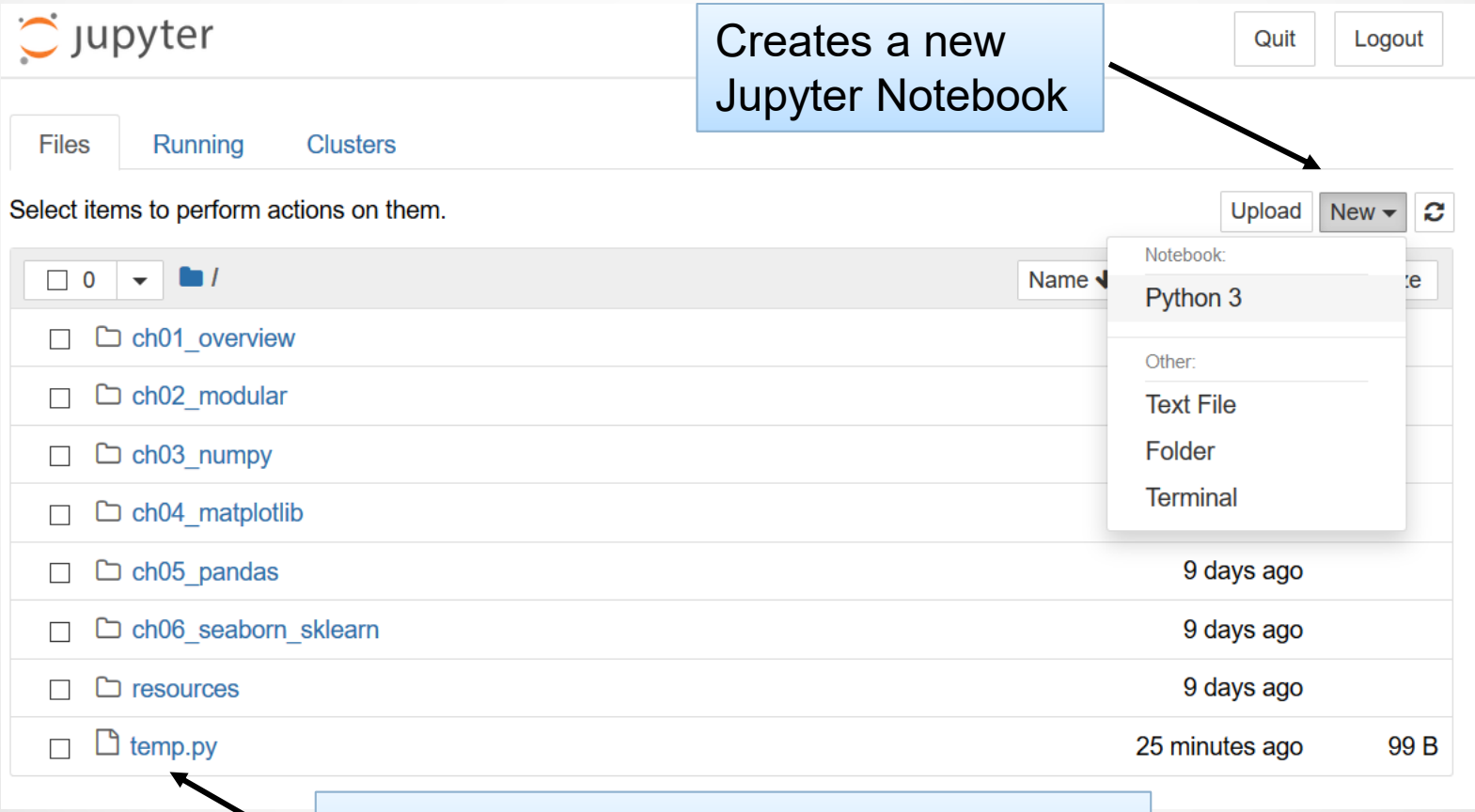
Once the server launches, it can be accessed via the browser at: **localhost:8888** (by default)

IPython Jupyter Relationship



Creating a New Notebook

- The dashboard appears when Jupyter first starts
- Create a new notebook as shown:



Creates a new Jupyter Notebook

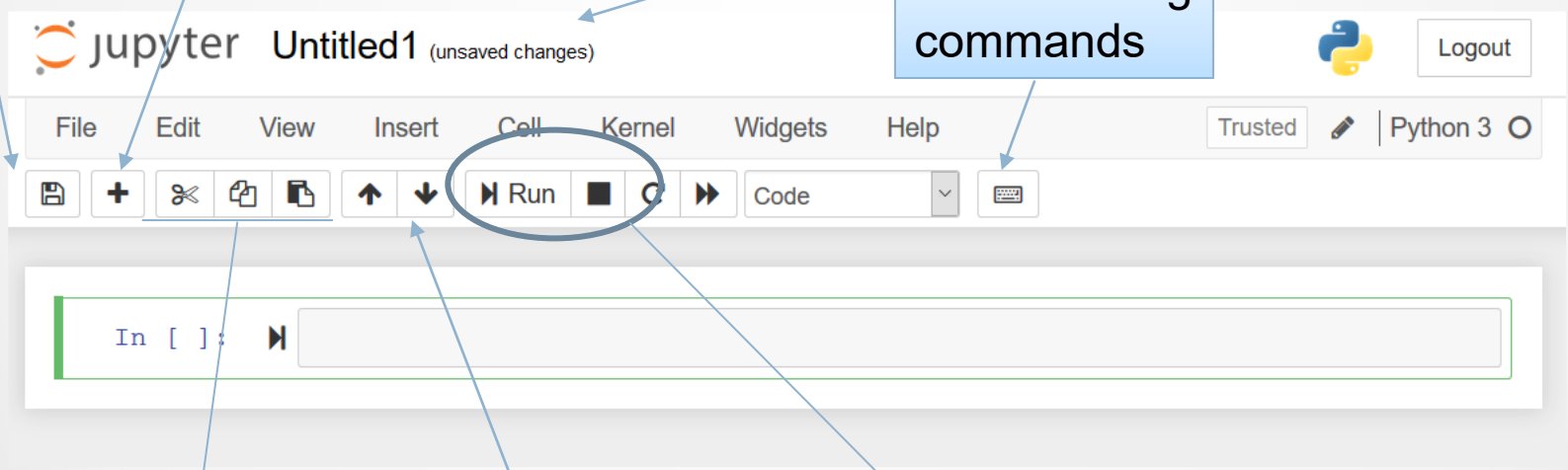
Items shown are based on the directory where you start the Jupyter server

Notebook Components

- Notebooks are documents that contain live code, images, video, text, graphs, widgets and more
  - Notebooks can be run from within the browser using different kernels (programming languages)
  - Create .ipynb files on the local file system
- Notebooks are made up of cells
- Two most common cells:
  - Markdown cells (narrative LaTeX-based text)
  - Code cells (cells containing code from a language)

Notebook Cells Toolbar

- The Notebook menu and toolbar provide the ability to modify, execute, and save Notebooks



Save and checkpoint

Insert new cell below

Notebook name, click to modify

List of editing commands

Cut, copy, paste cell

Move cells up/down

Run cell, interrupt kernel, restart kernel

Notebook Editor Modes

- Command mode

`ESC` key on a cell initiates this mode

```
In [ ]:
```

  - In this mode, a cell responds to special commands (see the list of editing commands)

- Edit mode

`Enter` key on a cell initiates this mode

```
In [ ]: x = 10
```

  - In this mode, a cell can be typed into (like an editor)

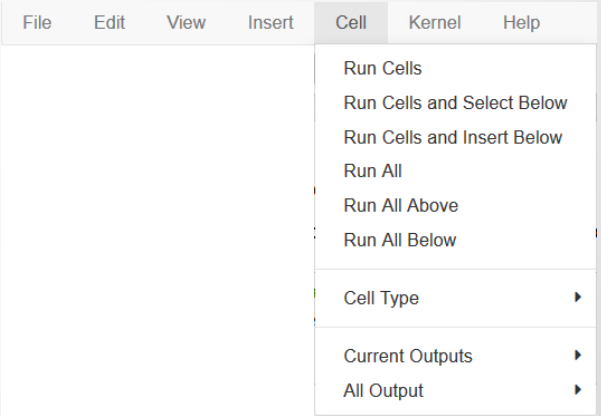
Clicking in the appropriate areas will also cause the cell to go into Edit or Command Mode

Running Notebooks

- Pressing **`Shift-Enter`** or clicking while the selected cell is in command mode will run the cell

```
In [1]: import sys
        print(sys.version)

        3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.
        1900 64 bit (AMD64)]

In [ ]:
```

- To run all cells, use the menu item



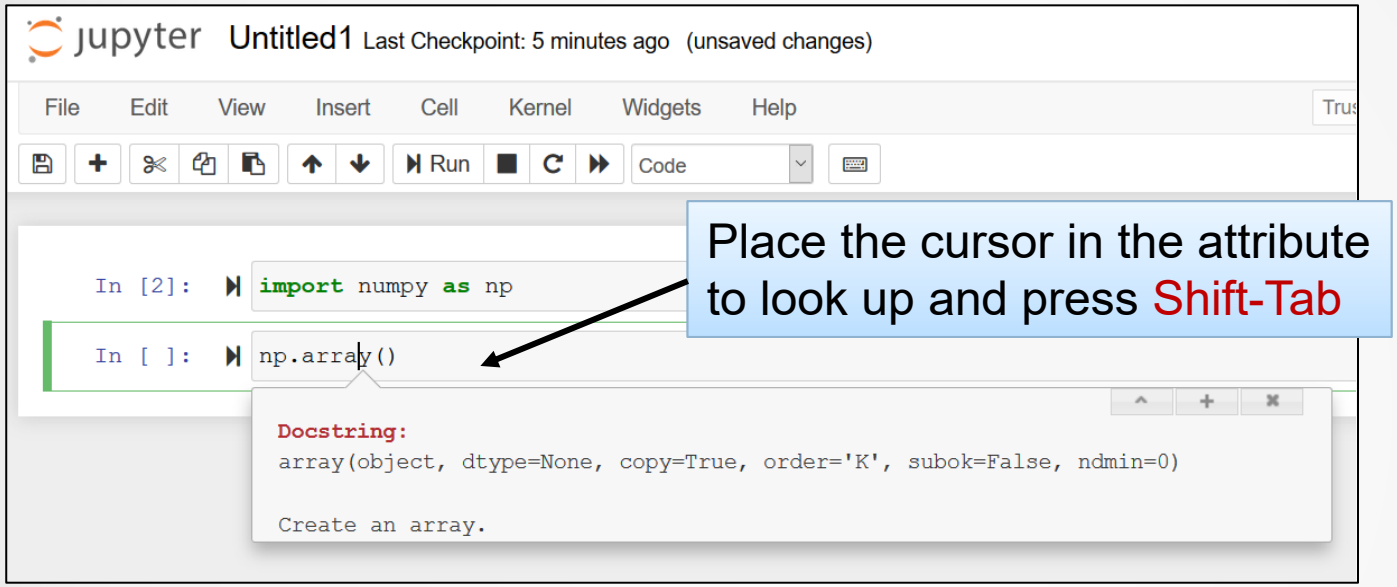
Markdown Notation

- Markdown cells allow for rendering *formatted text* into notebooks
  - Markdown notation include:

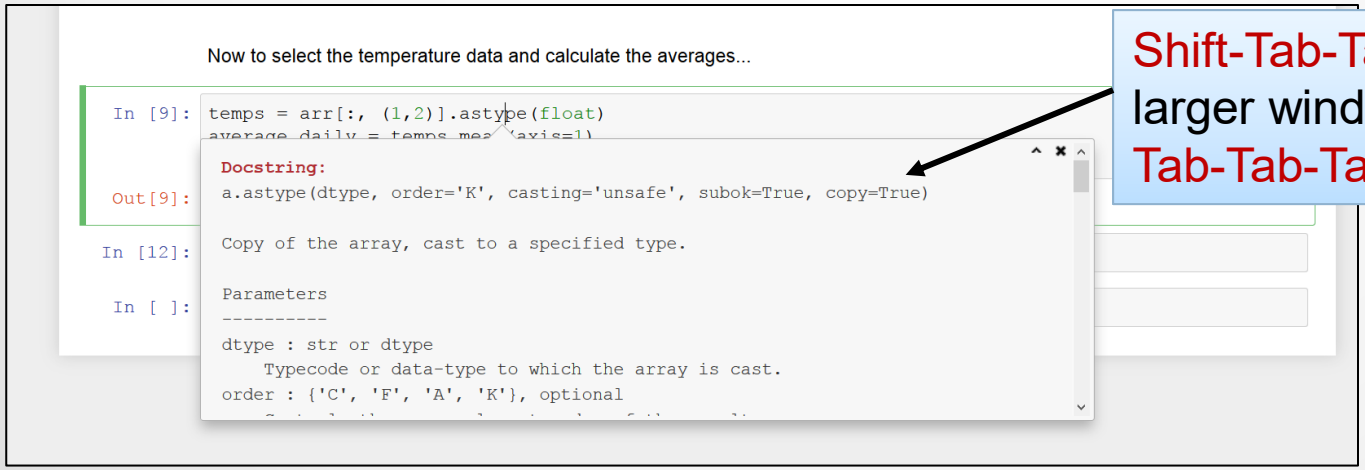
H1	#	Headers	Lists	1. Ordered lists
H2	##			+ Increments the ordered list
H3	###			
H4	####			* Unordered lists
H5	#####			
H6	########			
Some Italic Text		Italic Text	Links	![Some link](http://url)
Some BoldText		Bold Text	Latex	$inline_expr$ $$expr_on_own_line$$

Code Assistance within Jupyter

- Jupyter supports viewing code assistance



Place the cursor in the attribute to look up and press Shift-Tab



Shift-Tab-Tab will create a larger window, while Shift-Tab-Tab-Tab-Tab docks it

Reference: A Few Useful Shortcuts

Action	Type Sequence
Insert a Cell Below	ESC, B
Insert a Cell Above	ESC, A
Switch Cell to Markdown	ESC, M, Enter
Cut a Cell, Paste a Cell	ESC then X, ESC then V
Delete a Cell	D then D
Split a Cell in Two (at the cursor)	Ctrl-Shift-Minus

Your Turn! - Task 5-1: Using Jupyter

- Start your Jupyter Server (as discussed) within your student_files directory
  - Create a new Notebook

- In a Notebook cell, create a Python function to read the **`resources/population_data.csv`** file in any way you wish
  - Convert the year and population to int types
  - Return the data as a *list of tuples*
  - *Don't include the header line*
    (hint: f.readline() will consume one line)

- In another cell, *call the function* and display the data

What is NumPy?

- NumPy is a Python-based framework providing multi-dimensional array creation and manipulation tools

- Friends of NumPy: there is a core set of tools that commonly appear alongside NumPy:
  - SciPy - Mathematical and scientific modules
  - Matplotlib - Data plotting and visualization
  - IPython - Python interactive engine
  - Pandas - Data manipulation and acquisition
  - Scikit-image - Image processing tools
  - Scikit-learn - Provides machine learning algorithms
  - Many others

The NumPy Array

Stands for n-dimensional array

- NumPy's main object is the array (ndarray type)
  - Array dimensions in numpy are called axes
  - The number of axes is its rank

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

axes = 1 or rank = 1
Length (of axis 0): 5

```
arr2 = np.array([ [1, 2, 3, 4, 5],
                  [6, 7, 8, 9, 10] ])
```

axes = 2 or rank = 2
Length of axis 0: 2
Length of axis 1: 5

```
arr3 = np.array([ [1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12] ])
```

axes = 2 or rank = 2
Length (of axis 0): 4
Length (of axis 1): 3
Shape: (4, 3)

Array Terms (Definitions)

- An ndArray (NumPy array type) has several attributes:
  - ndarray.ndim — number of axes or dimensions or rank
  - ndarray.shape — length of array in each dimension
    For m rows and n columns shape=(m, n)
  - ndarray.size — total number of elements
  - ndarray.strides — number of bytes to get to the next element in that dimension

```
arr3 = np.array([ [1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12] ])
print('Shape: {0}, Size:{1}, Axes:{2}, Types:{3}, Strides:{4}'
    .format(arr3.shape, arr3.size, arr3.ndim, arr3.dtype,
            arr3.strides ))
```

Shape: (4, 3), Size: 12, Axes: 2, Types: int32, Strides: (12, 4)

Array Values and *dtype*

- NumPy arrays are *homogeneous*

```
import numpy as np

arr4 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9],
                 [10, 11, 12]])

arr5 = np.array([[1, 2, 3],
                 [4, 5., 6],
                 [7, 8, 9],
                 [10, 11, 12]])

arr6 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9],
                 [10, 11, 12]], dtype=float)

print(arr4.dtype, arr5.dtype, arr6.dtype)
```

The entire NumPy array must be the same data type (dtype)

The dtype will be set to the "largest" value within the array

Array value types can be set using the dtype argument

int32 float64 float64

np.astype()

- astype() can be used to change the array types

```
arr10 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])


print(arr10.dtype)                          # int32
arr_new = arr10.astype(np.float64)          # new array created
print(arr_new.dtype)                        # float64
```

Creating Arrays

- Several functions exist for creating NumPy arrays:

```
arr8 = np.zeros((2,2))
```

```
[
   [ 0.  0.]
   [ 0.  0.]
]
```

```
arr9 = np.ones((2,2))
```

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

```
arr10 = np.full((2,2), 6)
```

```
[
   [ 6.  6.]
   [ 6.  6.]
]
```

```
arr11 = np.eye(2)
```

```
[
   [ 1.  0.]
   [ 0.  1.]
]
```

start, end, total_items

```
arr12 = np.linspace(0, 20, 5)
```

```
[ 0.  5.  10.  15.  20.]
```

end *is* included

start, end, step_by

```
arr13 = np.arange(0, 15, 3)
```

```
[ 0 3 6 9 12]
```

end *not* included

Accessing Cells

- Access arrays using a single pair of square brackets regardless of the number of dimensions

```
arr15 = np.array([ [1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9],
                   [10, 11, 12] ])
```

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

`arr15[3][1]`

Standard Python notation works

`arr15[3, 1]`

Single brackets are preferred

`arr15[-3, -3]`

	-3	-2	-1
-4	1	2	3
-3	4	5	6
-2	7	8	9
-1	10	11	12

`arr15[-1]`

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

`arr15[2:, 1:]`

This notation is consistent across several Python data science frameworks such as Pandas

Additional Selecting and Modifying

	0	1	2
0	1	8	3
1	4	8	6
2	7	8	9
3	10	8	12

Slicing can modify multiple values at once

```
arr15[:, 1] = 8
```

Axis 0 (rows)

Axis 1 (columns)

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

Select one or more columns at once

```
arr15[:, [0, 2]]    or
arr15[:, (0, 2)]
```

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

Select one or more rows at once (with slicing)

```
arr15[1:3, :]
```

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

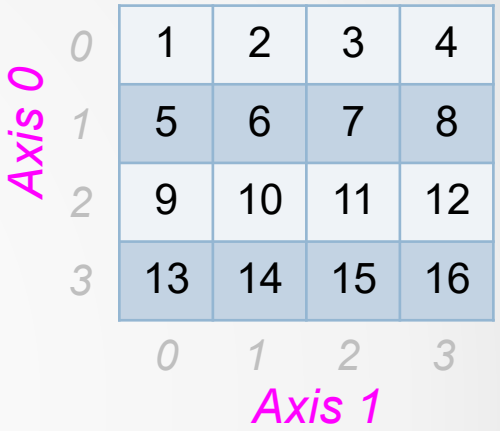
Selecting using the step-by option in slicing

```
arr15[1::2, 0::2]
```

Reshaping Arrays

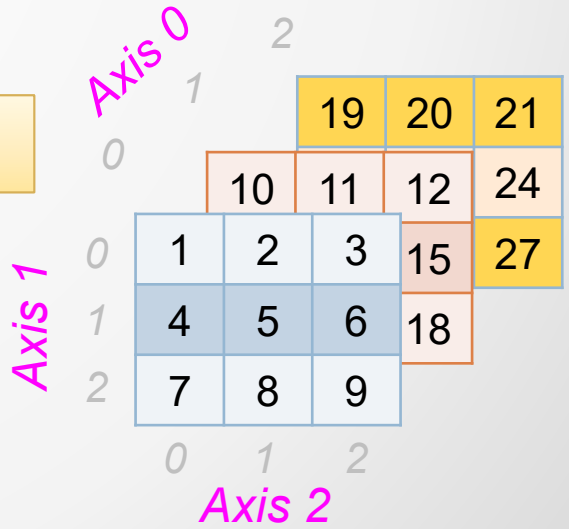
- ndarray.reshape(shape) - can reshape an array's dimensions

```
arr16 = np.arange(1, 17).reshape((4, 4))
```




```
arr17 = np.arange(1, 28).reshape((3, 3, 3))
```

New shape



NumPy Functions *(1 of 2)*

- NumPy ships with dozens of helpful functions:

abs()
absolute()
accumulate()
add()
all()
allclose()
alltrue()
angle()
any()
append()
arange()
arccos()
arccosh()
arcsin()
arcsinh()
arctan()
arctan2()
arctanh()
argmax()
argmin()
argsort()
array()
arrayrange()
array_split()
asarray()
asanyarray()
asmatrix()
astype()
atleast_1d()
atleast_2d()
atleast_3d()
average()
beta()
binary_repr()
bincount()
binomial()
bitwise_and()
bitwise_or()
bitwise_xor()
bmat()
broadcast()
bytes()
c_[]
cast[]()
ceil()
choose()
clip()
column_stack()
compress()
concatenate()
conj()
conjugate()
copy()
corrcoef()
cos()
cov()
cross()
cumprod()
cumsum()
delete()
det()
diag()
diagflat()
diagonal()
diff()
digitize()
dot()
dsplit()
dstack()
dtype()
empty()
empty_like()
eye()
fft()
fftfreq()
fftshift()
fill()
finfo()
fix()
flat
flatten()
fliplr()
flipud()
floor()
fromarrays()
frombuffer()
fromfile()
fromfunction()
fromiter()
generic
gumbel()
histogram()
hsplit()
hstack()
hypot()
identity()
ifft()
imag
index_exp[]
indices()
inf
inner()
insert()
inv()
iscomplex()
iscomplexobj()
item()
ix_()
lexsort()
linspace()
loadtxt()
logical_and()
logical_not()
logical_or()

NumPy Functions *(2 of 2)*

logical_xor()
logspace()
lstsq()
mat()
matrix()
max()
maximum()
mean()
median()
mgrid[]
min()
minimum()
multiply()
nan
ndenumerate()
ndim
ndindex()
newaxis
nonzero()
ogrid()
ones()
ones_like()
outer()
permutation()
piecewise()
pinv()
poisson()
poly1d()
polyfit()
prod()
ptp()
put()
putmask()
r_[]
rand()
randint()
randn()
random_sample()
ranf()
ravel()
real
recarray()
reduce()
repeat()
reshape()
resize()
rollaxis()
round()
rot90()
s_[]
sample()
savetxt()
searchsorted()
seed()
select()
shape
shuffle()
slice()
solve()
sometrue()
sort()
split()
squeeze()
std()
sum()
svd()
swapaxes()
T
take()
tensordot()
tile()
tofile()
tolist()
trace()
transpose()
tri()
tril()
trim_zeros()
triu()
typeDict()
uniform()
unique()
unique1d()
vander()
var()
vdot()
vectorize()
view()
vonmises()
vsplit()
vstack()
weibull()
where()
zeros()
zeros_like()

Aggregation Functions

- NumPy provides functions for performing tasks on <u>all</u> selected elements within the array

```
arr31 = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])
```

```
print(np.sum(arr31))                # 78
print(arr31.sum())                  # 78

print(np.mean(arr31[:, 0]))         # 5.5
print(arr31[:, 0].mean())           # 5.5

print(np.prod(arr31[:2, :2]))       # 40
print(arr31[:2, :2].prod())         # 40

print(arr31.prod(axis=1))           # [    6   120   504 1320]
```

Sums all elements

Average of column 0

Product of upper left 4 nums

Product of each row

Selections Using Booleans

- Selections can be made using a NumPy array of Booleans

	0	1	2	3
0	1	8	3	4
1	1	8	3	4
2	1	8	3	4
3	1	8	3	4

```
arr[:, np.array([True, False, True, False])]
```

Where a True occurs, the item is selected

- When made more dynamic, this achieves a filtering (masking) effect

```
arr21 = np.arange(1, 7)
```

[1 2 3 4 5 6]

```
arr21[ arr21 <= 3 ])
```

[1 2 3]

This is the mask. It evaluates all array elements and yields: [True True True False False False]

When plugged back into the array, only values where the mask is *True* will be seen

Using Masking

- Selecting columns based on a criterion:

```
arr22 = np.arange(11, 29, 2).reshape(3, 3)
```

```
[ [11 13 15]
  [17 19 21]
  [23 25 27] ]
```

```
arr22[:, arr22[0] > 12]
```

```
[ [13 15]
  [19 21]
  [25 27] ]
```

Mask: selects values from row 0 that are > 12.
This yields: [False True True]

It is plugged back into the "columns" (axis 1) place to yield only the desired columns

- Selecting rows based on a criterion:

```
arr22[arr22[:, 2] > 15]
```

```
[ [17 19 21]
  [23 25 27] ]
```

Mask: selects values from the third column that are > 15.
This yields: [False True True]

It is plugged back into the "rows" (axis 0) place to yield desired rows

Your Turn! - Task 5-2: Using NumPy

- Continue from Task 5-1, this time load the population data returned from the function into a NumPy array
  - Note: use dtype=object since values aren't homogeneous

- Determine the following:
  1. What is the shape, size, and number of dimensions of the array?
  2. What is the average population?
  3. How many records are from 2020? (Yes, we can see it, but how can NumPy determine this)
  4. Display all records with a population above 350,000

Sorting Based on Values

- Use NumPy *argsort()* method to sort a specific column

```
arr23 = np.roll(np.arange(1, 10).reshape((3, 3)), 2)
```

```
[[8 9 1]
 [2 3 4]
 [5 6 7]]
```

Specifies the first column to sort on

```
sort_by_col0 = arr[arr[:, 0].argsort()]
```

```
[[2 3 4]
 [5 6 7]
 [8 9 1]]
```

Take all rows & first column, return the indices of the sorted column, plug them back into the array

```
sort_by_col2 = arr[arr[:, 2].argsort()]
```

```
[[8 9 1]
 [2 3 4]
 [5 6 7]]
```

Sorts on the third column

Your Turn! - Task 5-3: Sort the Data

- Continue from Task 5-2
- Sort the data by population
- Display the name of the city with the highest population (only the city name, not the record)

Data Visualization and Python

- Numerous frameworks exist for creating 2D and 3D charts:

• Matplotlib	de facto standard for 2D Python visualization
• Pandas	provides data structures and wrappers for Matplotlib API
• Seaborn	wrapper for Matplotlib providing simpler code to generate charts
• Ggplot	tool that works with pandas data-frames and provides layered charts
• Bokeh	Python charting tool allowing for various level of control

Introducing Matplotlib

- Matplotlib is the Python de facto standard API for creating 2D graphs
  - Integrates with NumPy arrays
  - Supports numerous chart types

Bar	Pie	Line	Streamplots
Line	Log	Scatter	Ellipses
Scatter	Bar	Polar	Table Demo
			...and more...

- Common module import conventions

```
import matplotlib as mpl
```

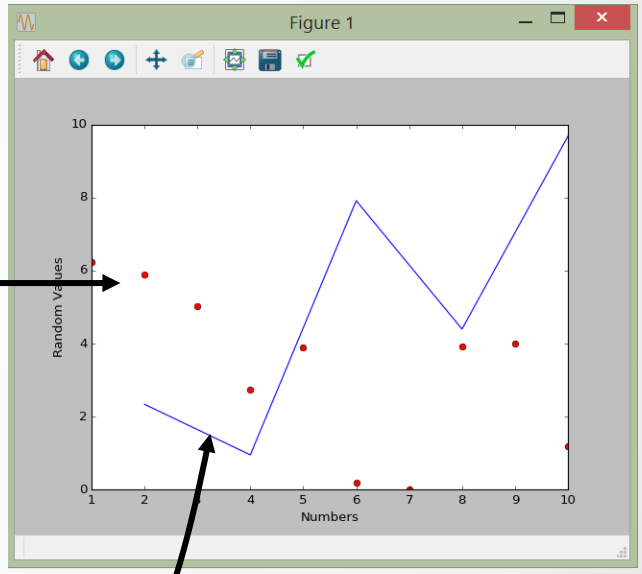
```
import matplotlib.pyplot as plt
```

We will refer to this module as plt from now on

The **pyplot** submodule contains the primary functionality for plotting

A First Plot

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.arange(1, 11)
y1 = 10 * np.random.rand(1, 10).squeeze()
plt.ylabel('Random Values')
plt.xlabel('Numbers')
plt.plot(x1, y1, 'ro')

x2 = np.arange(2, 11, 2)
y2 = 10 * np.random.rand(1, 5).squeeze()
plt.plot(x2, y2)

plt.show()
```



Use this (outside of Jupyter) to cause the rendering to appear

The Plot Method

- The plot() method supports numerous arguments: `plt.plot(x, y, style)`

An array of data on the x-axis

An array of data on the y-axis

The style is a string that controls the color and markers

Chart Styles

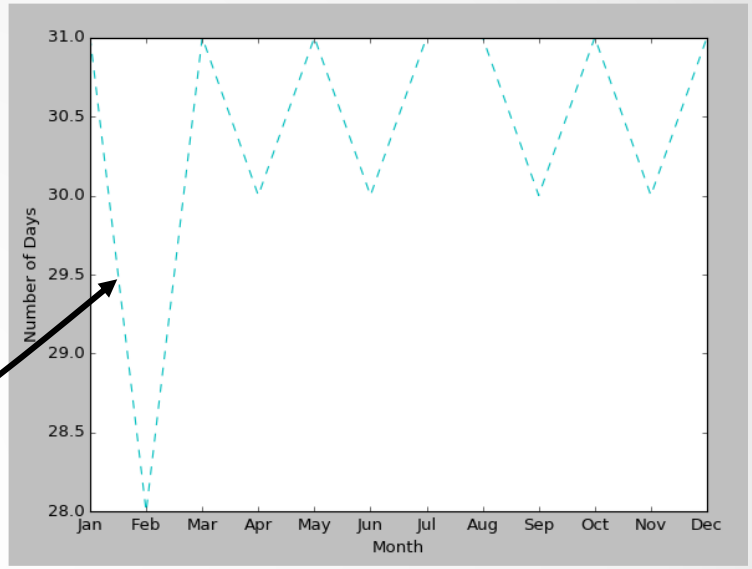
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

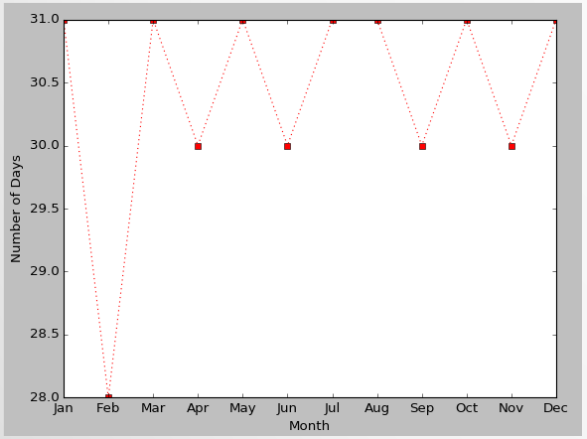
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker

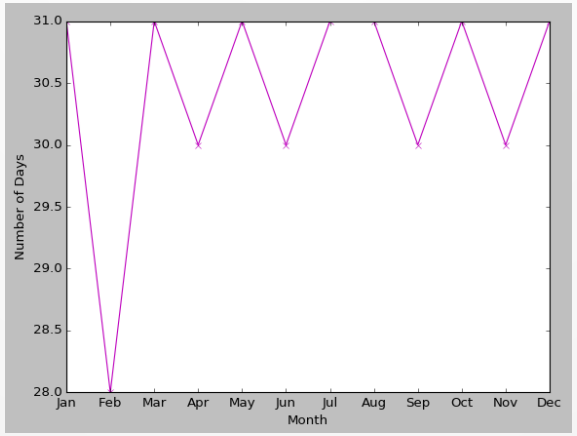
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

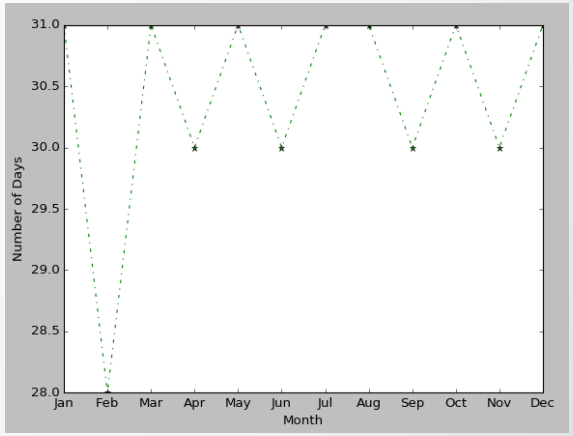
Plot Style Variations

```
x_ticks = ['Jan', 'Feb', ..., 'Dec']
x = np.arange(1, 13)
y = [31, 28, 31, 30, ..., 31]

plt.xlabel('Month')
plt.ylabel('Number of Days')
plt.xticks(x, x_ticks)
plt.plot(x, y, 'c--')
plt.show()
```










```
plt.plot(x, y, 'rs:')
```

```
plt.plot(x, y, 'mx-')
```

```
plt.plot(x, y, 'g*-.')
```

Line Style Properties

- In addition to the previous slide, here are some other line styles that may be used:

```
plt.plot(xdata, ydata, other_properties)
```

color, c - any '#hex_value', or single letter ('r'),
linestyle, ls - 'solid', 'dashed', 'dotted', '-', '--', '-.', '.', or other valid symbol
linewidth, lw - float value for the width
alpha - 0.0 (transparent) to 1.0 (opaque)
label - text placed on the line
fillstyle - 'full', 'left', 'right', 'bottom', 'top', 'none'
marker - any marker style described on an earlier slide (e.g. 'v' or 'o')
drawstyle - 'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'
markersize, ms - float value for the size of the markers
markerfacecolor - a color for the markers
markeredgecolor - a color for the edges of the markers
markeredgewidth - width of the line that draws the markers

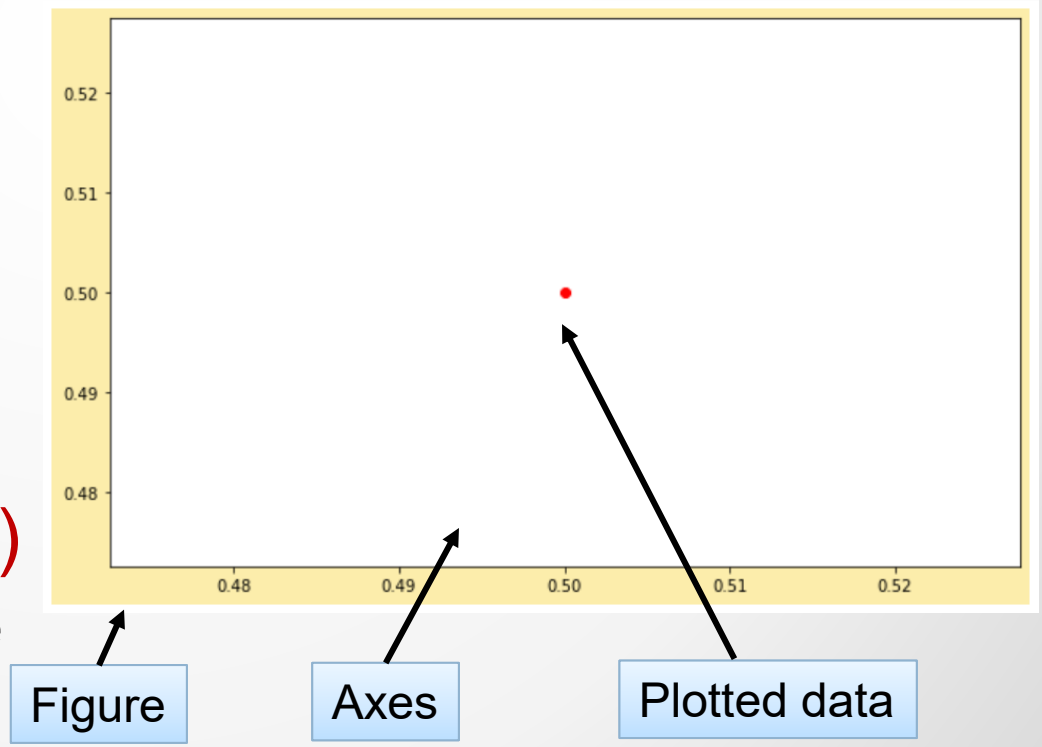
Figures and Axes

- For additional control over the rendering, create a Figure and Axes object

```
figure = plt.figure(figsize=(10, 6), facecolor='#fcedab')
ax = figure.add_axes((0.1, 0.1, 0.8, 0.8));
ax.plot(0.5, 0.5, 'ro');
```

- Use plt.figure() to create the drawing canvas

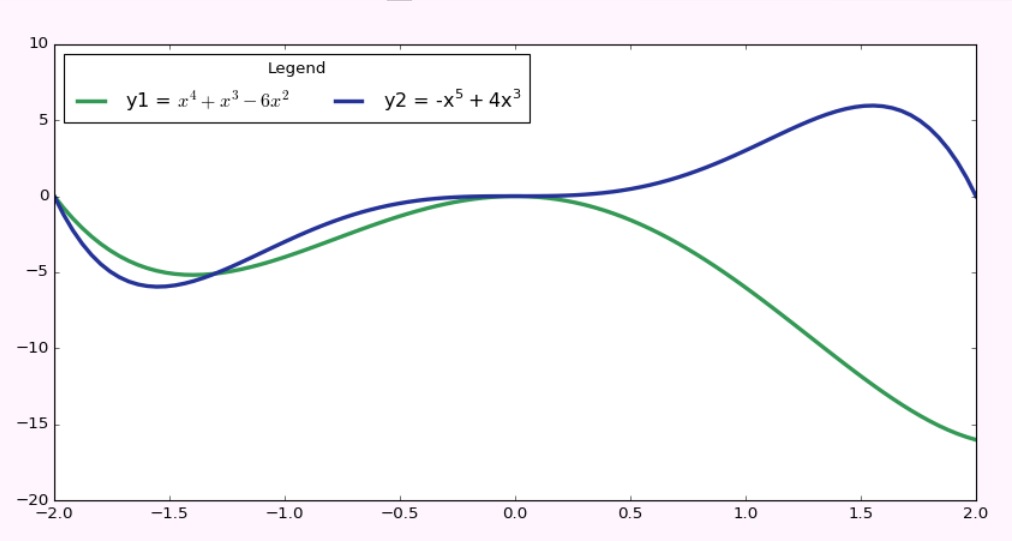
- Once a figure object exists, call add_axes() to create a coordinate system on the figure



Legends

- Use plt.legend() to customize a legend:

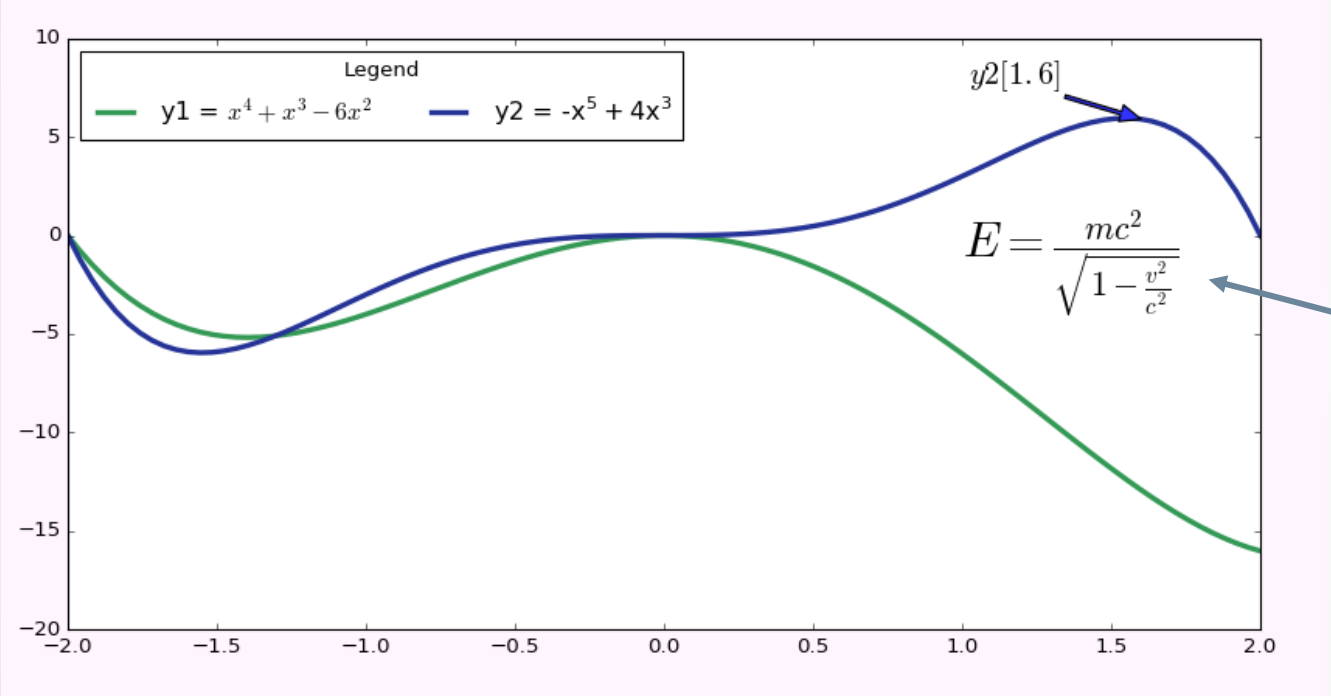
Matplotlib supports LaTeX, a special markup for displaying numerical values.




```
x = np.linspace(-2, 2, 100)
y1 = x**4 - x**3 - 6*x**2
y2 = -x**5 + 4*x**3

figure = plt.figure(figsize=(12, 6), facecolor='#fff5ff')

sub1 = figure.add_axes((0.1, 0.1, 0.8, 0.8))
sub1.plot(x, y1, color='#339955', lw=3.0, ls='solid')
sub1.plot(x, y2, color='#253397', lw=3.0, ls='solid')
sub1.legend([r'y1 = $x^4 + x^3 - 6x^2$',
             r'y2 = $\mathdefault{-x^5 + 4x^3}$'],
            loc='best', title='Legend', ncol=2)
plt.margins(0)
```

Annotations

- Annotations add documentation to visuals




```
sub1.annotate(text=r'$E = \frac{mc^2}{\sqrt{1-\frac{v^2}{c^2}}}$',
              xy=(0.7, 0.6), xycoords='figure fraction',
              fontsize=28)
```

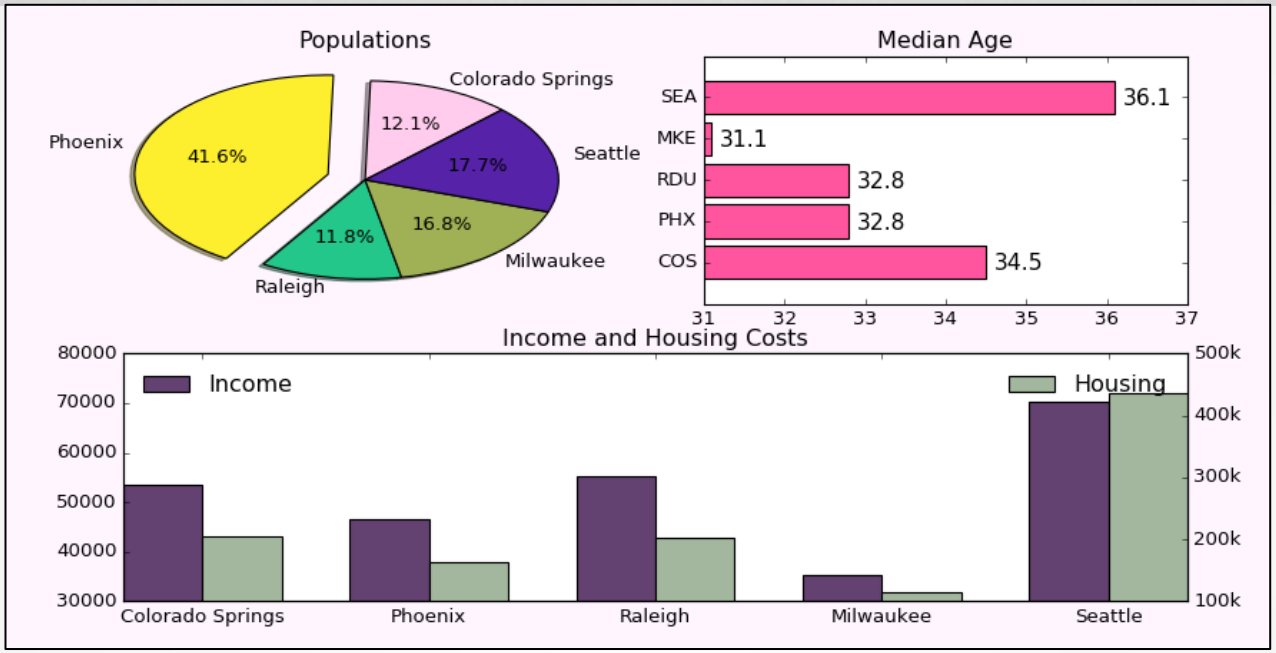
savefig

- Save figures using figure.savefig():

```
figure.savefig(fname, dpi=None, facecolor='w',
               edgecolor='w', format=None,
               orientation='portrait')
```

```
figure.savefig('legendary.png')
```

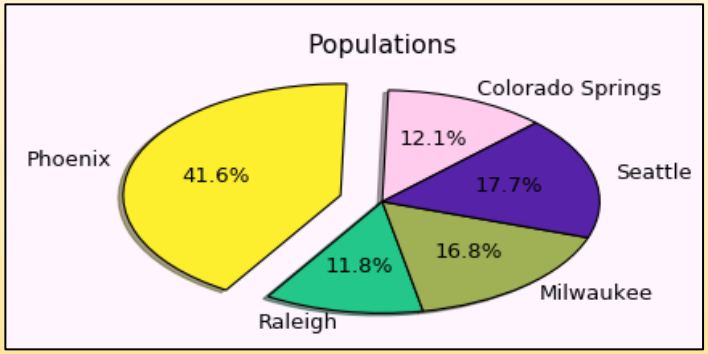
Other Chart Examples *(1 of 3)*



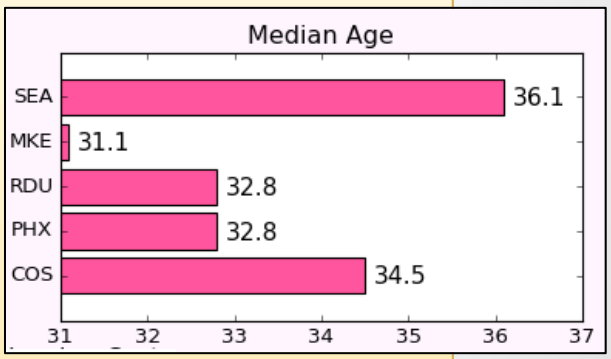

```
figure = plt.figure(figsize=(12, 6), facecolor='#fff5ff')

cities = ['Colorado Springs', 'Phoenix', 'Raleigh',
          'Milwaukee', 'Seattle']
cities_abbr = ['COS', 'PHX', 'RDU', 'MKE', 'SEA']
elevations = [6172, 1132, 437, 723, 429]
populations = [431000, 1488000, 423000, 599000, 634000]
median_age = [34.5, 32.8, 32.8, 31.1, 36.1]
median_income = [53550, 46601, 55170, 35186, 70172]
median_house = [205600, 162300, 202800, 113900, 436600]
```

Other Chart Examples *(2 of 3)*

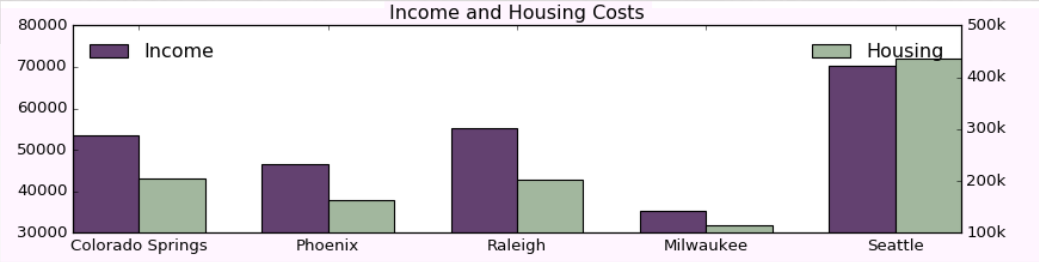
```
sub1 = figure.add_subplot(221)
colors = ['#ffccee', '#fdef2e', '#23c78a', '#a0b055', '#5623a8']
sub1.pie(populations, labels=cities,
         colors=colors,
         explode=[0, 0.2, 0, 0, 0],
         autopct='%1.1f%%',
         shadow=True, startangle=45)
sub1.set_title('Populations')
```




```
sub2 = figure.add_subplot(222)
y_ticks = np.arange(len(cities))
sub2.barh(y_ticks, median_age, height=0.8,
          align='center', color='#ff559e')
sub2.set_yticks(y_ticks)
sub2.set_xlim((31, 37))
sub2.set_yticklabels(cities_abbr)
sub2.set_title('Median Age')
for loc, age in enumerate(median_age):
    sub2.annotate(str(age), xy=(median_age[loc], loc),
                  xycoords='data', xytext=(+5, -5),
                  textcoords='offset points', fontsize=14)
```



Other Chart Examples *(3 of 3)*



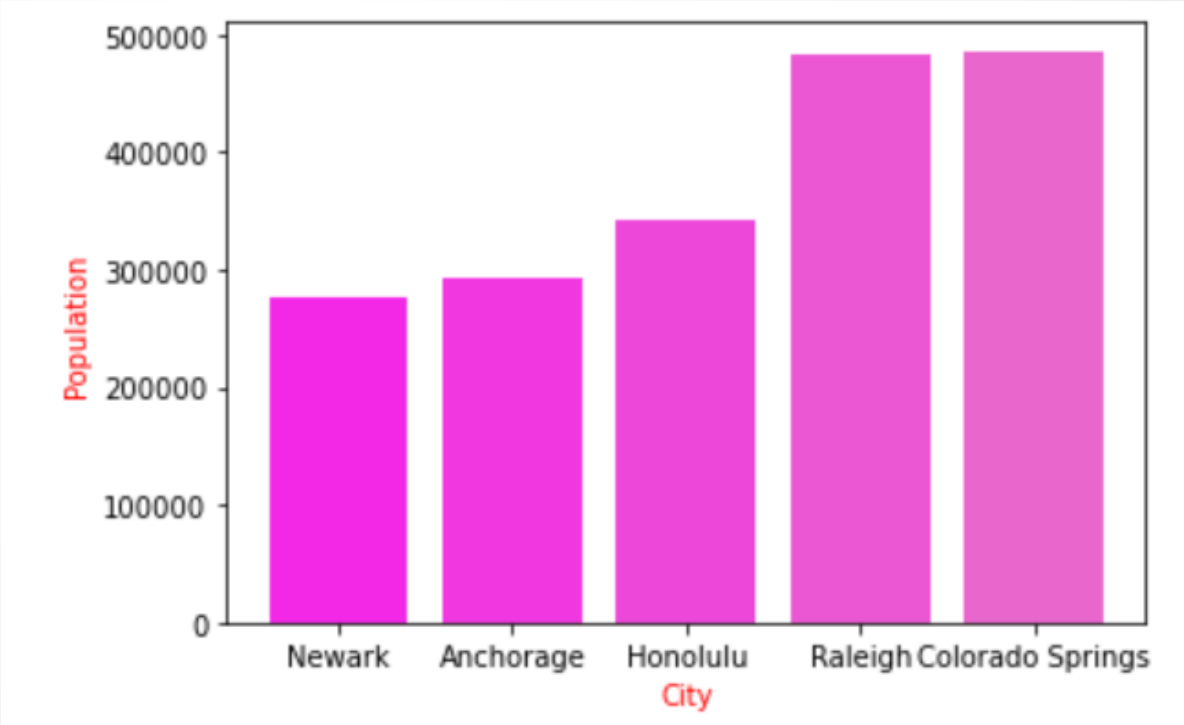
Grouped Bar, Twin Axis

```
sub3 = figure.add_subplot(2, 2, (3, 4))
bar_width = 0.35
x_ticks = np.arange(len(cities))
sub3.bar(x_ticks, median_income, width=bar_width, color='#634170')
sub3.set_xticks(x_ticks + bar_width)
sub3.set_xticklabels(cities)
sub3.set_ylim(30000, 80000)
sub3.set_title('Income and Housing Costs')
sub3.legend(['Income'], loc='upper left', frameon=False)

sub3b = sub3.twinx()
sub3b.bar(x_ticks + bar_width, median_house,
          width=bar_width, color='#a2b79e')
sub3b.set_ylim(100000, 500000)
sub3b.set_yticks(np.arange(100000, 600000, 100000))
sub3b.set_yticklabels(['100k', '200k', '300k', '400k', '500k'])
sub3b.legend(['Housing'], loc='upper right', frameon=False)
```

Creates axis on other side of x

Your Turn! Task 5-4

- Continue working from Task 5-3
- Add an import for matplotlib.pyplot to your Notebook
- Create a bar plot of cities vs. population
  - Hint: use plt.bar()
  - Use x= and height= for the parameters
  - Select the cities column and population column respectively



Chapter 5 - Summary

- NumPy provides a powerful array type, called ndarray, that can be n-dimensions deep
- Its numerous operations provide the capability to manipulate, transform, and reshape data
- The NumPy array is used as a foundational data structure with other frameworks
  - It integrates nicely with other tools such as Matplotlib and Pandas (each discussed in later chapters)
- In a similar vein, Matplotlib is the foundational data visualization framework within Python

Chapter 6
Pandas

Working with the Python Pandas Tool

Chapter 6 - Overview

- Introducing Pandas
- Series
- DataFrames
- Reading from Files
- Data Preprocessing Techniques

Introducing Pandas

- Pandas is a data analysis library for Python and part of the SciPy suite

- Features include:
  - APIs for reading data into data structures
  - Data merging, reshaping and pivoting of data
  - Split-apply-combine operations

- Data structures include:
  - Series
  - DataFrame
  - Panel

Panel is deprecated as of version 0.20

Common import syntax

We will refer to this module as pd in further examples

```
import pandas as pd
```

Pandas Series

- A Pandas Series is a *one-dimensional ndarray*
  - Series have an index (like a dictionary) called labels
  - Many ndarray methods are overridden to better support Series operations

s = pd.Series(data, index, dtype)

```
import pandas as pd

ser1 = pd.Series([212, 32, 0, -273])
print(ser1)
```

Output:
```
0    212
1     32
2      0
3   -273
dtype: int64
```

Creating Series

Series can be created several different ways

```
ser2 = pd.Series(name='City Elevations',
                 index=['Colorado Springs', 'Phoenix',
                        'Raleigh', 'Milwaukee', 'Seattle'],
                 data=[6172, 1132, 437, 723, 429])

print(ser2)
```

```
Colorado Springs    6172
Phoenix             1132
Raleigh             437
Milwaukee           723
Seattle             429
Name: City Elevations, dtype: int64
```

```
city_elevations = {'Denver': 5883, 'Austin': 632,
       'Philadelphia': 21, 'Billings': 3649, 'Anchorage': 144}
ser3 = pd.Series(city_elevations, name='City Elevations')

print(ser3)
```

```
Anchorage       144
Austin          632
Billings        3649
Denver          5883
Philadelphia    21
Name: City Elevations, dtype: int64
```

Series Data

- Series allow access to values using methods similar to Python dictionaries:

```
ser4 = pd.Series([218, 15, 619, 13, 1295],
                 index=['Mobile', 'San Diego', 'Chicago',
                        'New York City', 'Oklahoma City'],
                 name='City Elevations')
```

```
ser4['Mobile']                          # 218
ser4.get('New York City')               # 13
ser4['Albuquerque']                     # KeyError
ser4.get('Albuquerque')                 # None
ser4.get('Albuquerque', -1)             # -1
ser4[['Chicago', 'Oklahoma City']]      # new series
ser4.Chicago                            # 619

ser4[1:3]                               # San Diego 15
                                          Chicago 619
ser4.name                               # City Elevations
ser4.median()                           # 218.0
```

Accessing Series Data

- *series*.describe() returns a *new summary Series* of the original Series:

```
ser4.describe()
```

```
count          5.000000
mean         432.000000
std          541.983395
min           13.000000
25%           15.000000
50%          218.000000
75%          619.000000
max         1295.000000
Name: City Elevations, dtype: float64
<class 'pandas.core.series.Series'>
```

```
ser4.describe()['mean']
```

```
432.0
```

```
ser4.index
```

```
Index(['Mobile', 'San Diego', 'Chicago', 'New
York City', 'Oklahoma City'], dtype='object')
```

```
ser4.values
```

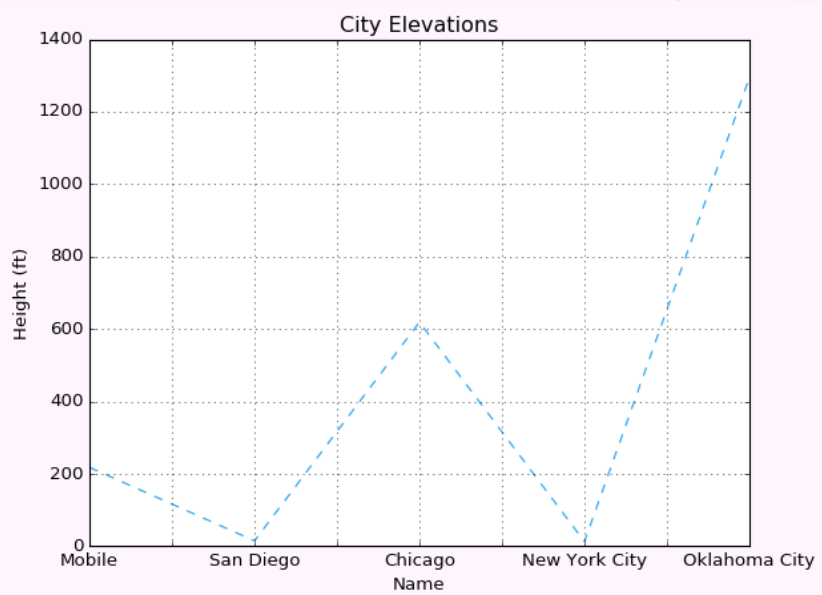
```
[ 218   15  619   13 1295]
```

Plotting Series

- Pandas' data structures support Matplotlib data plotting arguments

```
import matplotlib.pyplot as plt
import pandas as pd

ser4 = pd.Series([218, 15, 619, 13, 1295],
                 index=['Mobile', 'San Diego', 'Chicago',
                        'New York City', 'Oklahoma City'],
                 name='City Elevations')
figure = plt.figure()
axis = figure.add_subplot(111)
axis.set_title('City Elevations')
axis.set_xlabel('Name')
axis.set_ylabel('Height (ft)')

ser4.plot(style='--', kind='line',
          ax=axis, color='#24a1f2',
          grid=True)

plt.show()
```

Not needed in Jupyter



DataFrames

- Pandas DataFrames are two-dimensional data structures containing labels on each axis

pd.DataFrame(data, index, columns, dtype)

  - Conceptually similar to a spreadsheet
  - DataFrames support an index (rows labels) and columns (column labels)

```
df5 = pd.DataFrame(np.arange(16).reshape(4, 4),
                   index=['row0', 'row1', 'row2', 'row3'],
                   columns=['col0', 'col1', 'col2', 'col3'])
```

index

columns

	col0	col1	col2	col3
row0	0	1	2	3
row1	4	5	6	7
row2	8	9	10	11
row3	12	13	14	15

values

Creating DataFrames

- DataFrames can be created several ways:

```
df1 = pd.DataFrame([['Colorado Springs', 'Phoenix', 'Raleigh',
                     'Milwaukee', 'Seattle'],
                    [6172, 1132, 437, 723, 429]] )

print(df1)
```

Positional-based index and columns

	0	1	2	3	4
0	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
1	6172	1132	437	723	429

```
df1.index = ['City', 'Elevation']
print(df1)
```

Label-based indexing

	0	1	2	3	4
City	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
Elevation	6172	1132	437	723	429

Creating DataFrames Using a Dict of Lists

```
import pandas as pd

data = {
    'elevation': [6172, 1132, 437, 723, 429],
    'median_age': [34.5, 32.8, 32.8, 31.1, 36.1],
    'median_income': [53550, 46601, 55170, 35186, 70172],
}

df2 = pd.DataFrame(data, index=['Colorado Springs', 'Phoenix',
                                'Raleigh', 'Milwaukee', 'Seattle'])
```

A dict of lists can be used to create a DataFrame

```
print(df2)
```

	elevation	median_age	median_income
Colorado Springs	6172	34.5	53550
Phoenix	1132	32.8	46601
Raleigh	437	32.8	55170
Milwaukee	723	31.1	35186
Seattle	429	36.1	70172

Creating DataFrames Using ndarrays

```
data = [
    [6172, 1132, 437, 723, 429],
    [34.5, 32.8, 32.8, 31.1, 36.1],
    [53550, 46601, 55170, 35186, 70172]
]

columns = ['elevation', 'median_age', 'median_income']
index = ['Colorado Springs', 'Phoenix', 'Raleigh',
         'Milwaukee', 'Seattle']

arr = np.array(data, dtype=int)
df3 = pd.DataFrame(arr.transpose(), index=index, columns=columns)

print(df3)
```

A NumPy array can be used to create a DataFrame

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

Working with DataFrames

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

```
df4.to_numpy()
```

Returns a NumPy Array of just the DataFrame data

```
array([[ 6172,    34, 53550],
       [ 1132,    32, 46601],
       [  437,    32, 55170],
       [  723,    31, 35186],
       [  429,    36, 70172]])
```

```
df4.describe()
```

Provides summary statistics information

	elevation	median_age	median_income
count	5.00000	5.0	5.000000
mean	1778.60000	33.0	52135.800000
std	2472.63307	2.0	12791.018067
min	429.00000	31.0	35186.000000
25%	437.00000	32.0	46601.000000
50%	723.00000	32.0	53550.000000
75%	1132.00000	34.0	55170.000000
max	6172.00000	36.0	70172.000000

Working with DataFrames *(continued)*

```
df4.info(verbose=True)
```

Provides metadata info about a DataFrame

```
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, Colorado Springs to Seattle
Data columns (total 3 columns):
elevation        5 non-null int32
median_age       5 non-null int32
median_income    5 non-null int32
dtypes: int32(3)
memory usage: 100.0+ bytes
```

```
df4.T
```

Returns the transpose (view) of the array

	Colorado Springs	Phoenix	Raleigh	Milwaukee	Seattle
elevation	6172	1132	437	723	429
median_age	34	32	32	31	36
median_income	53550	46601	55170	35186	70172

Accessing DataFrames

- Similar rules apply to DataFrames when accessing *columns*

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

Get *elevation* column, returns a Series

```
df4['elevation']
df4.get('elevation')
```

```
Colorado Springs    6172
Phoenix             1132
Raleigh              437
Milwaukee            723
Seattle              429
Name: elevation, dtype: int32
```

Selecting a column yields a Series

Accessing Data with loc[] and iloc[]

- loc[*label*] and iloc[*index*] provide better consistency for selecting data from DataFrames

```
df5 = pd.DataFrame(np.arange(16).reshape(4, 4),
        index=['row0', 'row1', 'row2', 'row3'],
        columns=['col0', 'col1', 'col2', 'col3'])
```

	col0	col1	col2	col3
row0	0	1	2	3
row1	4	5	6	7
row2	8	9	10	11
row3	12	13	14	15

```
df5.iloc[1:3]
```

	col0	col1	col2	col3
row1	4	5	6	7
row2	8	9	10	11

```
df5.loc['row1']
```

```
col0    4
col1    5
col2    6
col3    7
Name: row1, dtype: int32
```

```
df5.iloc[2:, 2:]
```

	col2	col3
row2	10	11
row3	14	15

```
df5.loc['row2':, 'col2':]
```

	col2	col3
row2	10	11
row3	14	15

Index (and Column) Sorting

- Data can be "reordered" using pd.sort_index()

df4

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Phoenix	1132	32	46601
Raleigh	437	32	55170
Milwaukee	723	31	35186
Seattle	429	36	70172

```
df6 = df4.sort_index()
```

df6

	elevation	median_age	median_income
Colorado Springs	6172	34	53550
Milwaukee	723	31	35186
Phoenix	1132	32	46601
Raleigh	437	32	55170
Seattle	429	36	70172

Reorders the index, df4 is unchanged

Use df4.sort_index(inplace=True) to change df4 directly

```
df7 = df4.sort_values(by='median_age')
```

Sorts the DataFrame on the median_age column

Use ascending=False to sort from largest to smallest

Pandas read_csv()

- pd.read_csv() reads data files into a DataFrame
  - Useful arguments include:
    - delimiter=',' — data item separator character
    - skiprows=None — how many rows to skip at the beginning
    - encoding='utf-8' — file encoding type
    - nrows — number of rows to read
    - header — which row to use for the column headers

Doesn't consider the first line as the header now

```
contacts = pd.read_csv('contacts.dat', header=None, dtype='str')
contacts.columns = ['name', 'address', 'state', 'zip',
            'area_code', 'phone', 'email', 'company', 'position']
```

Boolean Indexing (Masking) in Pandas

- Data values that match the given Boolean criterion will be seen in the resulting data structure
  - Example: *Find only records where BOTH the high and low temps are above 51 degrees*

```
sat_temps = pd.DataFrame(data=[(78, 50), (82, 52), (83, 53)],
                         index=['Colorado Springs', 'Canon City', 'Pueblo'],
                         columns=['High', 'Low'])
```

	High	Low
Colorado Springs	78	50
Canon City	82	52
Pueblo	83	53

```
sat_temps.loc[(sat_temps['High'] >= 51) & (sat_temps['Low'] >= 51)]
```

	High	Low
Canon City	82	52
Pueblo	83	53

head() and tail()

- Use df.head(n=5) or df.tail(n=5) to display the first n or last n rows of a DataFrame

```
import pandas as pd
contacts = pd.read_csv('contacts2.dat', header=None, dtype='str')
contacts.columns = ['name', 'address', 'state', 'zip', 'area_code',
                    'phone', 'email', 'company', 'position']

print(contacts.head(3))
```

Displays the first 3 rows of the DataFrame

```
Bob Green, 4517 Elm St. Riverside, NJ,08075,301,356-8921,bob@abc.com, ...
Violet Smith, 220 E. Main Ave Philadelphia, PA,09119,202,421-9008, ...
John Brown, 231 Oak Blvd. Black Hills, SD,82101,719,303-1219, ...
```

Your Turn! - Task 6-1: Using Pandas

- Create a new notebook
- Import Pandas
- Load *population_data.csv* via Pandas' read_csv() method into a DataFrame
  - Use the file's headers as column names

1. Determine the shape
2. Determine the average population using the describe() function
3. Use masking to display only cities whose population is greater than 350,000

Data Cleansing with Pandas

- Data Cleansing is the process of correcting or removing irregular values from a dataset
- This involves numerous tasks including:
  - Exploration (Detection)
  - Examine the data for completeness (e.g., missing values)
  - Verify constraints (regular expressions, type, ranges, cross-field constraints)
  - Fix anomalies (remove and/or replace)
  - Remove observations containing missing values
  - Impute values
  - Remove duplicates
  - Text formatting
  - Detect and handle outliers

For data modeling, also:
- Encode data
  - Convert continuous values to categorical
  - Cut and bin continuous values
  - Perform type conversions
- Normalize / Standardize column values

Our Sample Dataset

- For the following discussion, we'll use a known dataset, titanic.csv

- We'll work from ch06_pandas/Data Cleansing.ipynb

Titanic Dataset

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

Chapter 6 - Summary

- Pandas *Series and DataFrames* provide numerous features beyond what NumPy can normally provide
- Pandas' numerous methods make the task of data preprocessing much easier
- These methods enable data cleansing on a scale that is largely unmatched within Python or many other languages for that matter

Chapter 7
Introducing Django

Testing Code with Mock Objects

Chapter 7 - Overview

- Django Features
- Creating Applications
- Creating Models
- Establishing a Database
- Templating
- Ajax

Django Features

The web framework for perfectionists with deadlines.

https://www.djangoproject.com/

- Full-featured web application framework
  - Model View Controller Architecture
  - Templating, Form Handling, Authentication
  - Database Integration
  - Object-Relation Mapping
  - Built-in Admin Interface
  - Internationalization
  - URL routing
  - Security Protections

Version Info

4.1	Aug 2022 (some breaking changes)
4.0	Dec 2021
3.2	Apr 2021
3.1	Aug 2020
3.0	Dec 2019
2.0	Dec 2017
1.11	Apr 2017 Py 3.6 support
1.10	Aug 2016
1.9	Dec 2015
1.8	Apr 2015 Multiple Templating Engines
1.7	Sep 2014 Db Migrations
1.5	Feb 2013 Python 3 support
1.2	May 2010
1.0	Sep 2008
0.9	Nov 2005

Documentation URL

https://docs.djangoproject.com/en/4.1/

Install and Test Django

- Django is third-party and must be installed

```
[sudo] pip install Django
```

- With Django installed, open a new terminal (command) window and type:

```
> django-admin --version
4.1.3
```

Note: You may need to check if Django installed into <PYTHONHOME>/Scripts (Windows) or <PYTHONHOME>/Libs/site_packages/Django/bin (OS X and Linux)

Creating a Project: django-admin.py

- django-admin.py is the starting tool used
- To create a project, issue the command

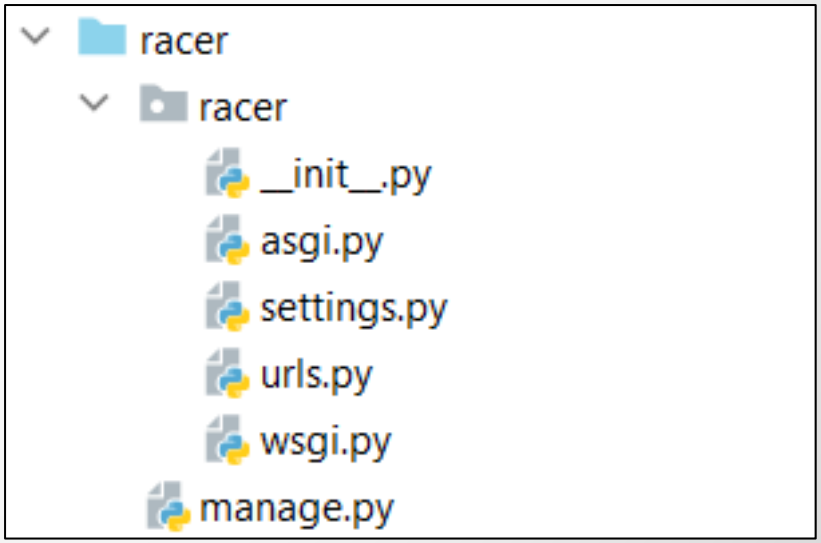
Execute from the directory where the project root folder should be created

```
django-admin startproject <project_name>
```

Our Task 7-1 performs the following command from within the student_files directory:

```
> django-admin startproject racer
```

The following project is generated



django-admin.py and manage.py

- django-admin.py is installed with Django initially
  - Used to create projects

```
django-admin <command> [options]
```

- manage.py is a wrapper around *django-admin.py*
  - Most commands are executed using this utility
  - It ensures your project is on sys.path before executing

```
python manage.py <command> [options]
```

Many manage.py commands

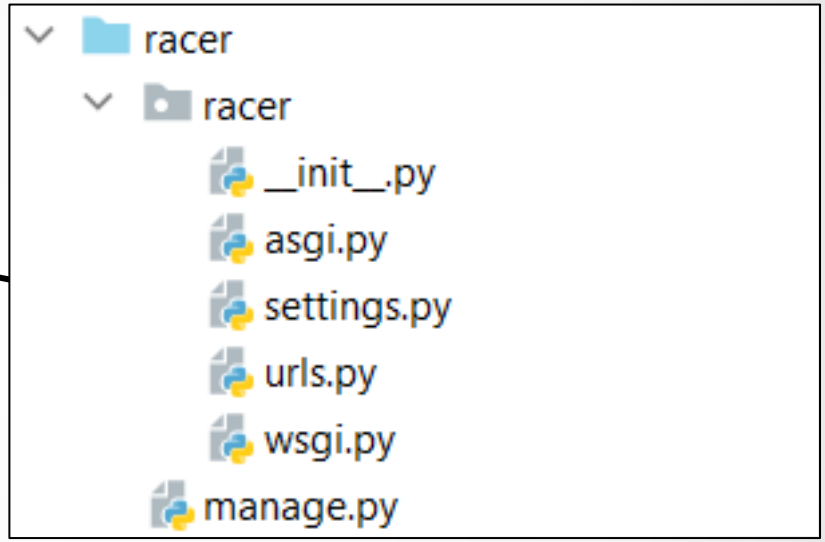
changepassword	shell
check	showmigrations
clearsessions	sql
compilemessages	sqlall
createcachetable	sqlclear
createsuperuser	sqlcustom
dbshell	sqldropindexes
diffsettings	sqlflush
dumpdata	sqlindexes
findstatic	sqlmigrate
flush	sqlsequencereset
Inspectdb	squashmigrations
loaddata	startapp
makemessages	startproject
makemigrations	syncdb
migrate	test
runfcgi	testserver
runserver	validate

Type django-admin at a command prompt to get this list

Establishing a Database

- Django works with numerous databases
  - Officially: MySQL, SQLite, Oracle, PostgreSQL, MariaDB

settings.py is the main config file within a Django project



  - The following section exists within settings.py

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

Django Backend Support

- Django provides "official" and "unofficial" support for numerous backends:

Officially supported Databases:
PostgreSQL
MySQL
Oracle
SQLite
MariaDB

Unofficially supported Databases (using 3rd party drivers):
Google App Engine
MongoDB
Cassandra
Redis
SimpleDB (AWS)
ElasticSearch
IBM DB2
MS SQL Server
ODBC
Firebird
SAP SQL Anywhere

Django database documentation:
https://docs.djangoproject.com/en/4.1/ref/databases/

Example Setting Up Other Backends

- Configure *settings.py* to define which database(s) you wish to use:

Can also be:
django.db.backends.oracle
django.db.backends.mysql
django.db.backends.sqlite3

settings.py

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "mydatabase",
        "USER": "username",
        "PASSWORD": "password",
        "HOST": "127.0.0.1",
        "PORT": "5432",
    }
}
```

Creating Apps

- Django *projects* are made up of one or more *apps*
  - Apps perform specific tasks

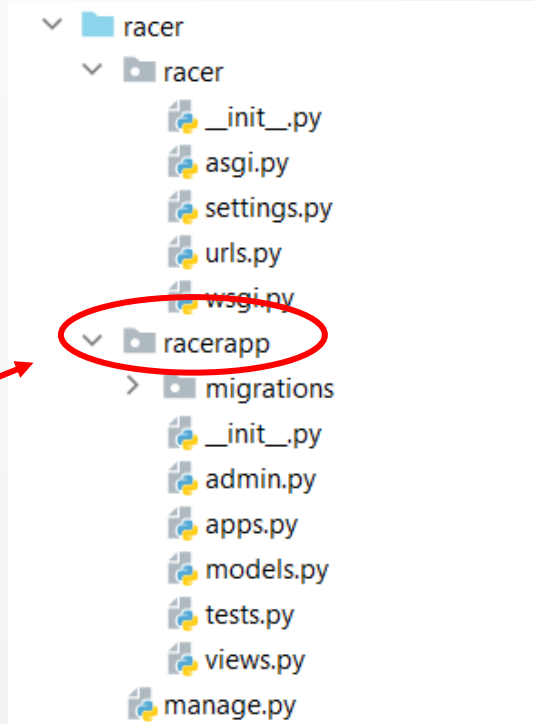
Projects can be viewed as a module containing settings. Apps are holders of models (yet to be discussed)

- To create an app issue the command

```
python manage.py startapp <app_name>
```

The following command was executed from our racer project directory:

```
> python manage.py startapp racerapp
```



Adding Apps to Settings

- After *startapp* creates an app, it should be added to the INSTALLED_APPS in settings.py

python manage.py startapp racerapp

settings.py

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "racerapp"
]
```

We added the *racerapp* name to the installed apps section of *settings.py*

Models

- Models are used to define database tables and synchronize data between the app and DB
  - Models inherit from models.Model and fields are declared at the class level

racerapp/models.py

```
from django.db import models


# Create your models here.
class Participant(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=100)
    address = models.CharField(max_length=255)
    age = models.IntegerField()

    race = models.ForeignKey('Race', on_delete=models.CASCADE, )

    def __str__(self):
        return f'{self.name} ({self.age})'
```

Integrating with the Database

- First build the tables used to manage the app and authentications

The following command was executed:

```
> python manage.py migrate
```

```
Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles
  Apply all migrations: sessions, auth, contenttypes, admin
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
  Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
    Applying sessions.0001_initial... OK
```

Defining and Creating Model Tables

- Creating tables is a two-step process
  1. Examine the models and create SQL with the *makemigrations* command

```
> python manage.py makemigrations racerapp
```

```
Migrations for 'racerapp':
  racerapp\migrations\0001_initial.py:
    - Create model Race
    - Create model Participant
```

  2. Execute the SQL creating the tables with *migrate*

```
> python manage.py migrate racerapp
```

```
Operations to perform:
  Apply all migrations: racerapp
Running migrations:
  Applying racerapp.0001_initial... OK
```

Checking Models and the Shell

- An interactive shell works with the database directly

This launches an interactive shell within your project root

The following command launches the shell:

```
> python manage.py shell
```

Within the shell, a Race and Participant can be persisted:

```
In [1]: import racerapp
```

```
In [2]: from racerapp.models import Race, Participant
```

```
In [3]: r = Race(name='Chicago Marathon', location='Chicago, IL',
                 distance=26.2)
```

```
In [4]: r.save()
```

```
In [5]: p = Participant(name='John Smith', address='123 Main St.',
                        age=25, race=r)
```

```
In [6]: p.save()
```

Using the Admin Features

- admin.py provides the ability to launch a web-based administrative interface to edit models and manage users

racerapp/admin.py

We added these to the existing file

```
from django.contrib import admin
from racerapp.models import Race, Participant

# Register your models here.
admin.site.register(Race)
admin.site.register(Participant)
```

- To use the tool, first create a superuser:

```
> python manage.py createsuperuser
```

```
Username:  admin
Email address:  admin@fake.com
Password:  admin_password
Password (again):  admin_password
Superuser created successfully.
```

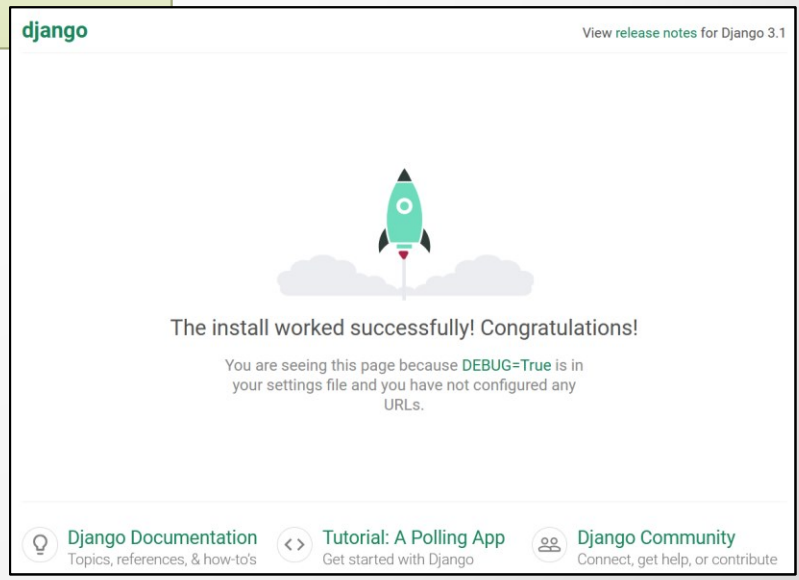
Running the Server

- To test out the admin interface, first run the server from the project root directory

```
> python manage.py runserver
```

```
Performing system checks...

System check identified no issues (0 silenced).

Django version 4.1.3, using settings 'racer.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

- Navigate to localhost:8000 to view the default page



Web-based Admin Interface

- Navigating to localhost:8000/admin yields the Django administrative interface

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

RACERAPP		
Participants	+ Add	Change
Races		

Recent actions

My actions

None available

After logging in, Race and Participant objects can be viewed and created

Add race

Name: BolderBoulder

Location: Boulder, CO

Distance: 6.2

Save and add another | Save and continue editing | SAVE

Creating Django Views

- Views are functions that *accept a request* as a parameter and *return a response*
  - To create a view, open the <app>/views.py file

racerapp/views.py

```
from django.shortcuts import render
from django.http import HttpResponse
from django.core import serializers

from racerapp.models import Race, Participant


def list_races(request):
    races = Race.objects.values()
    return render(request, 'list.html', {'races': races})
```

This function (a view) is called for the URL: *http://localhost:8000/races*

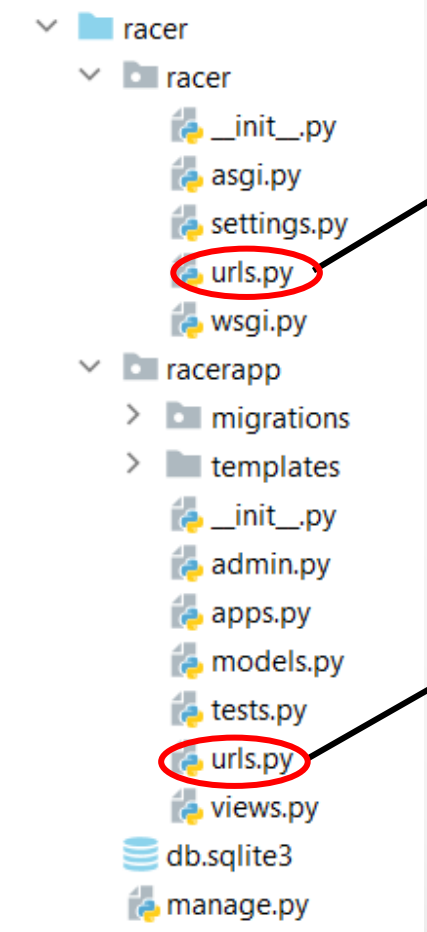
But where is this defined?

This retrieves the races from the database

This forwards the request and races to a template called *list.html*

Linking urls.py Files

- To get to view functions, define mappings in urls.py



racer/urls.py

```
from django.contrib import admin
from django.urls import include, path

import racerapp.urls

urlpatterns = [
    path("admin/", admin.site.urls),
    path("races/", include(racerapp.urls))
]
```

This routes *to* the app

racerapp/urls.py

```
from django.urls import re_path

from racerapp import views

urlpatterns = [
    re_path(r'^$', views.list_races,
                                name='race_list')
]
```

This routes *within* the app

Points to the view on prev slide

How URLs Map

- URLs first map to the urls.py in racer and then to the urls.py in racerapp

racerapp/urls.py

```
from django.urls import re_path
from racerapp import views

urlpatterns = [
    re_path(r'^$', views.list_races, name='race_list'),
]
```

A regex that is matched against the incoming URL *after* the match in racer/urls.py

The function in views.py to be invoked

An optional name to help distinguish this mapping

Rendering a Template

- Views can render templates via a *render()* call

```
def list_races(request):
    races = Race.objects.values()
    return render(request, 'list.html', {'races': races})
```

racerapp/templates/list.html

```
<html lang="en">
    <head>
        <title>Race List</title>
    </head>
    <body>
        <h1>Race List</h1>
        <ul>
            {% for race in races %}
                <li>
                    <a href="/races/register/{{ race.name }}/">
                        {{ race.name }}
                    </a>
                </li>
            {% endfor %}
        </ul>
    </body>
</html>
```

This is Django templating syntax

Race List

Chicago Marathon
BolderBoulder

Regex Named Groups

- More complex mappings can use *regex named groups*

http://localhost:8000/races/register/Chicago Marathon

racerapp/urls.py

```
from django.urls import re_path
from racerapp import views

urlpatterns = [
    re_path(r'register/(?P<racename>[^/]+)', views.register),
    re_path(r'^$', views.list_races, name='race_list')
]
```

We added this new mapping to urls.py

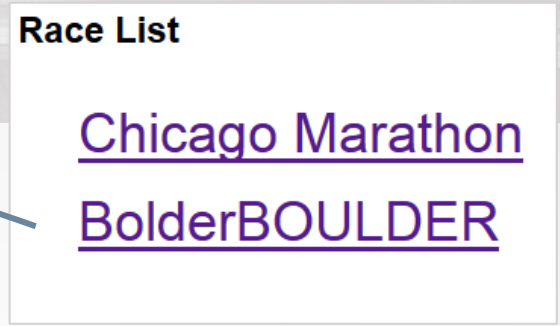
racerapp/views.py

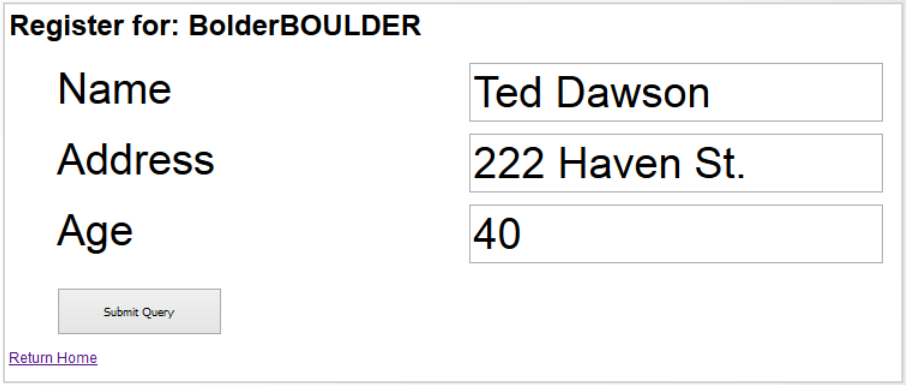
```
def register(request, racename=''):
    return render(request, 'register.html',
                  {"racename": racename})
```

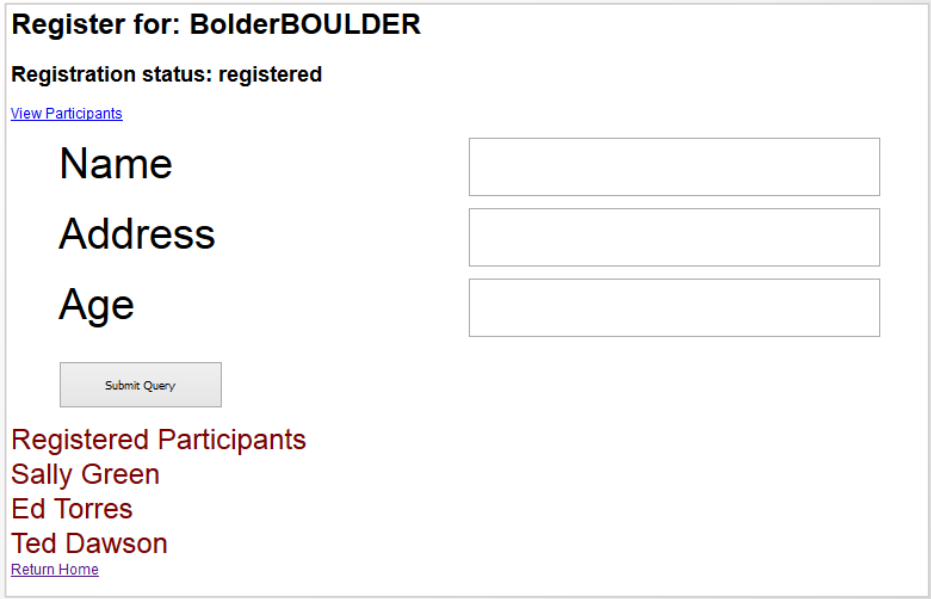
Your Turn! Task 7-1

- Build the Racer Django application
- Follow the steps within the workbook

Locate the instructions for this exercise in the back of the student manual







Chapter 7 - Summary

- Django has quite a bit of depth to it, making its learning curve somewhat steep
- It has several advantages over other frameworks, however
  - Most features are all integrated into the tool
  - Built-in templating, admin, and ORM, security, and more
- Django can be used in cloud environments such as Google App Engine or AWS

Chapter 8
Packaging For Distribution

Packaging to Improve Reusability

Chapter 8 - Overview

- Packaging Python Apps
- Other Mentionable Tools
  - Arrow
  - tqdm
  - Scrapy

Packaging Code for Distribution

- Python code can be packaged and made available for others

Module = Single (.py) Python files
Package = Multiple files grouped together as a unit

- *Python Package Index*, or *PyPI* ("pie-p-eye") is a public repository for sharing distributions

- Packaged tools may also be kept private and shared via an internally available share or server

But how are packages made "pip installable" by others?...

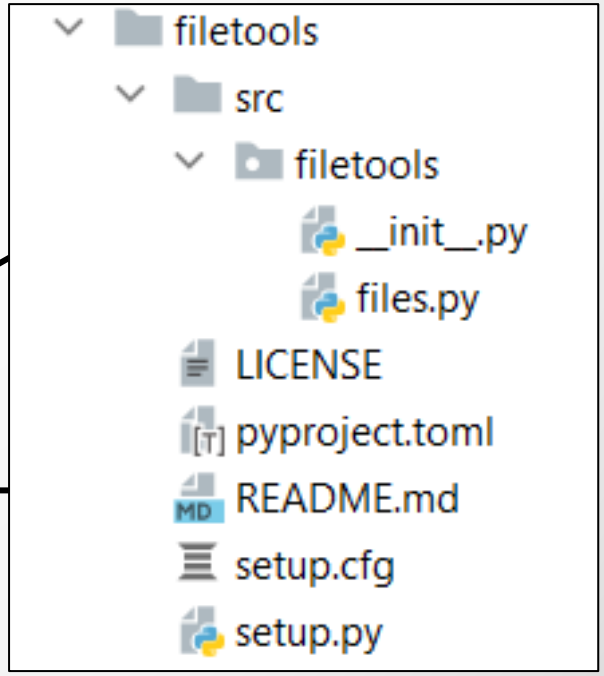
Packaging: Project Structure

- Python's common distribution format is called a wheel (.whl) file
  - The older format was an egg file (.egg)
  - Wheels only contain Python (.py) files, no compiled (.pyc) files or C extensions

- To create this format, begin with an appropriate project structure:

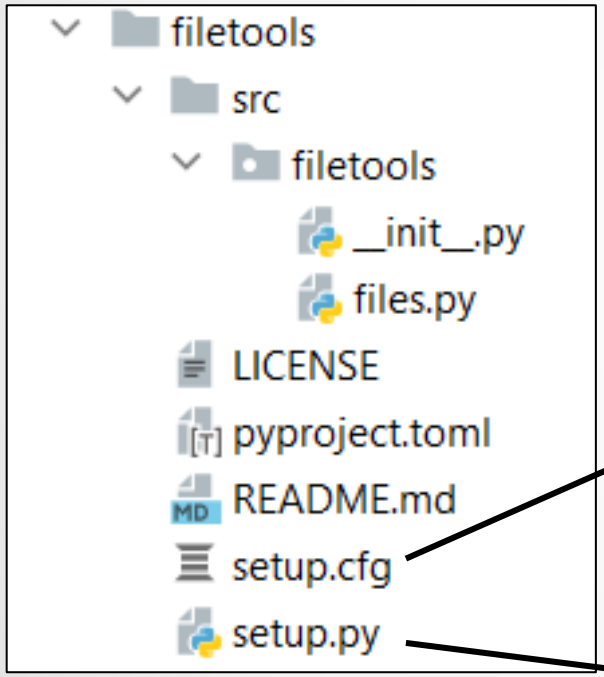
The desired package to distribute

Description of project (supports markdown)



Create the setup.cfg

- Before building the distribution, *setup.cfg* can define relevant packaging attributes



setup.cfg

```
[metadata]
name=filetools
author=IN1469
version=attr: filetools.__version__

[options]
python_requires = >= 3.6
package_dir=
    =src
packages=find:

[options.packages.find]

where=src
```

setup.py

```
from setuptools import setup

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

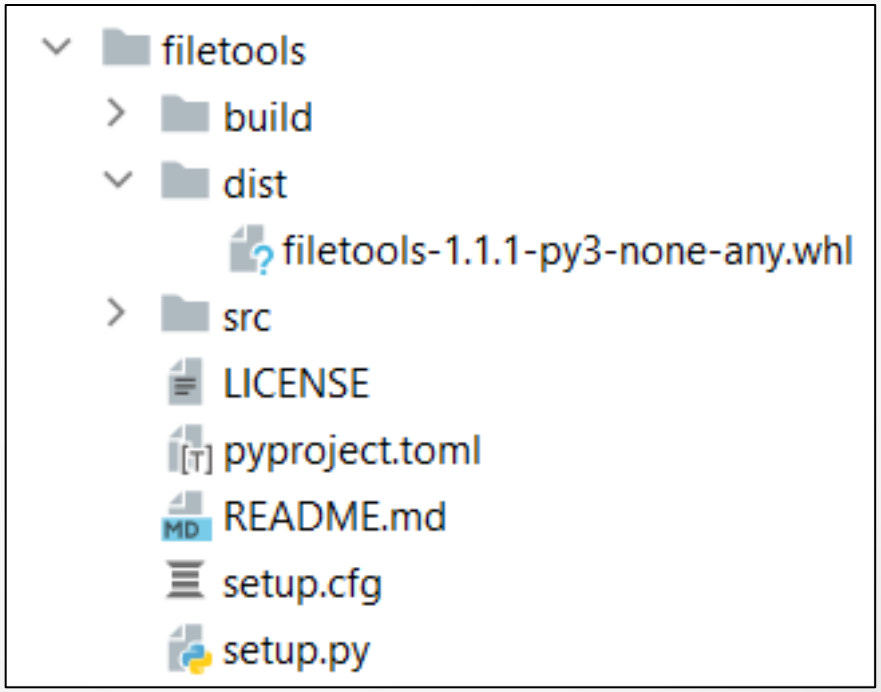
With setup.cfg present, setup.py becomes a simple Python file

Building the Distribution

- To create a wheel file, use the command

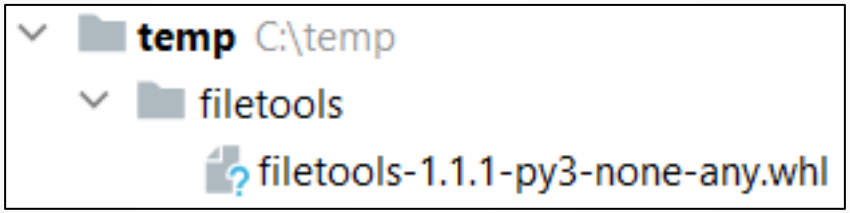
python setup.py bdist_wheel

In order to generate a wheel file, the third-party package, called *wheel*, must be pip installed
pip install wheel
(our requirements.txt already ensured this)



Using the Distribution (Internally)

- The new package can be installed remotely by deploying it to an accessible HTTP server
  1. A folder with the same name as our distribution (*filetools*) is created and contains the new wheel file



  2. We ran an demo HTTP server at this location (c:\temp)

This creates a simple web server in the temp directory running on port 7100

```
C:\temp> python -m http.server 7100
```

  3. Now others may use pip to retrieve the package

```
pip install --index-url=http://127.0.0.1:7100/  filetools
```

Uploading Distributions

- Private solutions can FTP or manually place the built file on the local server
  (in the appropriate folder name)

- For public solutions distribution uploads, should first install twine
  - Must have a PyPI account
  - Register the project
  - Upload distribution using twine: `twine upload dist/*`

Your Turn! Task 8-1 *(1 of 2)*

Build, Install, Uninstall the *filetools* package found in the student_files/ch08_packaging folder

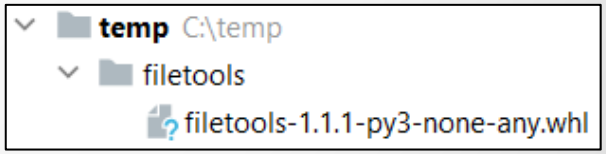
All instructions for this task are presented on this and the next slide

1. Open a terminal, cd to ch08_packaging/filetools and execute:

Wheel must be pip installed, but our build-out should have it

```
> python setup.py bdist_wheel
```

2. In a temp location, such as c:\temp, create a subdirectory called *filetools,* copy the newly created whl file into this folder



Note: In the terminal you use, be sure the Python virtual environment created for this course is activated!

Your Turn! Task 8-1 *(2 of 2)*

On Windows, to open another command window, simply type "start" + Enter before performing step 3

(continued)

3. Run an HTTP server from this temp directory:

```
C:\temp> python -m http.server 7100
```

4. In the command window, execute the pip command:

```
pip install --index-url=http://127.0.0.1:7100/  filetools
```

5. In PyCharm, within temp.py, test the new package:

temp.py

```
from filetools import copy_files
results = copy_files('ch02_automation/task2_1_files/dir1',
                     'ch02_automation/task2_1_files/dir2')
print(results)
```

6. Uninstall the package

```
pip uninstall filetools
```

Chapter 8 - Summary

- This journey ends with a reminder that Python has many amazing tools waiting to be utilized and that the overall journey, the one to learn Python, never end
- Explore sites like PyPI, StackOverflow, and others to learn and discover more
- Python continues to evolve and so must your skills!

Course Summary

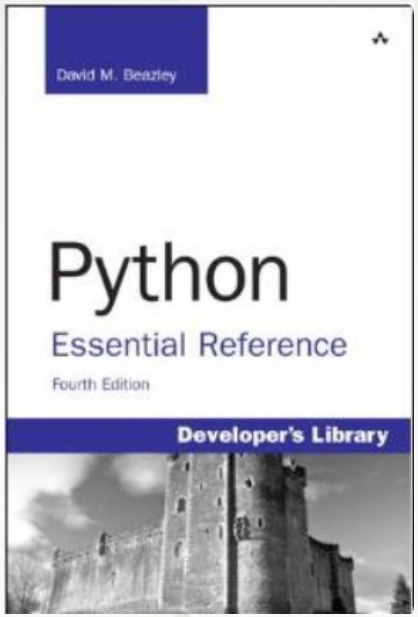
What Did We Learn?

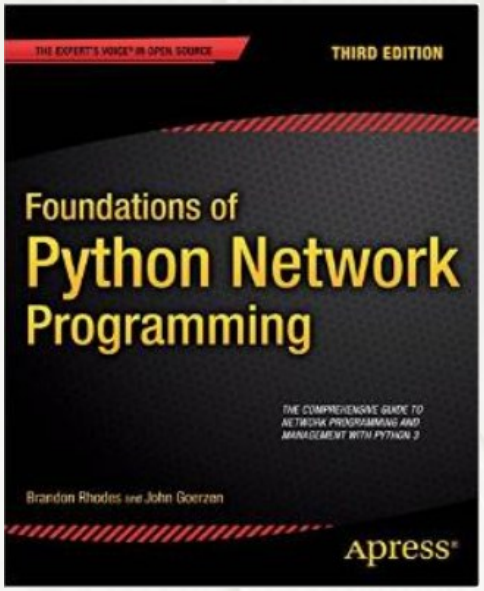
Review of Intermediate Topics
Generators
Dataclasses and NamedTuples
io and StringIO
Advanced Decorators
Timeit and Performance Measuring
Adv Context Managers
New-style, Old-style Classes
Inheritance
Multiple Inheritance
Slots
Descriptor Protocol
Callable Classes
Singletons
Factories
NumPy ndarray and Operations

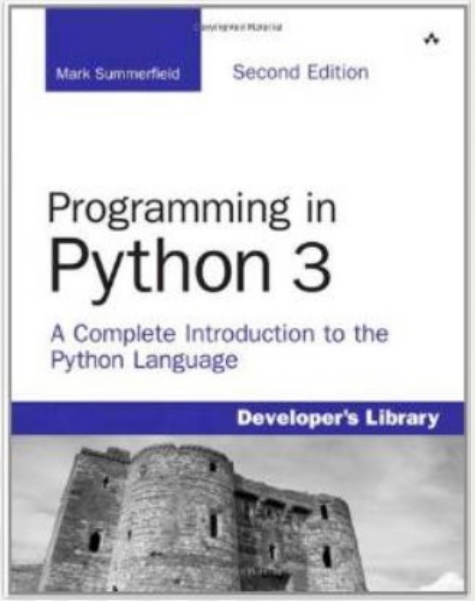
Unittest Module
Using Mock Objects
Monkey Patching when Testing
Matplotlib 2D Plotting
Pandas Series and DataFrames
Data Cleansing
Boolean Indexing
Loading Data into DataFrames
Working with Django
Django Models, Views
Django DB Migrations
Templating
Optionally (appendix):
Python GUIs
Using Tkinter
Creating Widgets

Resources

Books







Online Resources

http://pymotw.com/3/

http://docs.python-guide.org/en/latest/

http://learnpythonthehardway.org/book/

http://effbot.org/zone/librarybook-index.htm

http://planetpython.org/

Evaluations

- Please take the time to fill out an evaluation
- All evaluations are read and considered

Questions?

Appendix A

Python GUI Systems

Exploring Python GUIs

Appendix A - Overview

GUI Overview

Tkinter

Classes and GUIs

Python GUI Intro

- Python provides several options when developing Graphical User Interface (GUI) applications
  - Tkinter
    - Part of the Python Standard Library, typically no install needed
    - Python-based interface to the Tk GUI framework
    - No special license needed
  - PyQt (PyQt4, PyQt5, PyQt6)

    pip install PyQt5 | pip install PyQt6
    - A Python-based interface to the Qt C++ GUI framework
    - Support for Windows, OS X, Linux, Android, iOS
    - Can use Qt Designer, a GUI Drag-and-drop editor, to build the app and generate Python code from it
    - Dual license: commercial and GPLv3
  - Other GUI systems include: Kivy, wxPython, PySide2

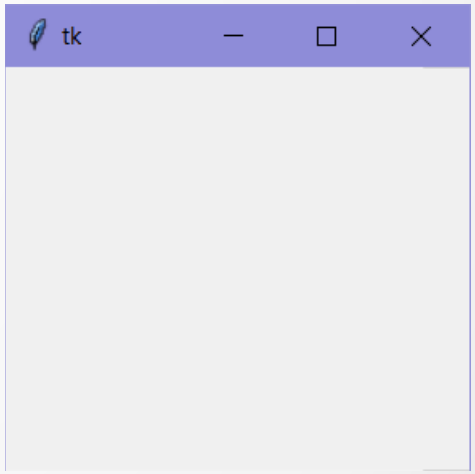
Tkinter Intro

- Tkinter creates *widgets* and places them into *windows*

- The main window is created from the Tk class

Try it now!

```
import tkinter as tk

window = tk.Tk()
window.mainloop()
```



Starts the window object's event loop which will handle user interactions such as button or key presses

Creates a top level widget. A Tcl interpreter is started (behind the scenes)

Tkinter Widgets

- There are numerous widgets within Tkinter
  - Frame — Intermediate widget containers to help with layout
  - Label — Widget containing text (or an image) to be placed
  - LabelFrame — A grouping of Labels
  - Button — Clickable interactive widget
  - Checkbutton — A toggle-type widget
  - Radiobutton — A selectable choice between two or more items
  - Entry — Single line input field
  - Text — Multiline input field
  - PanedWindow — asdf
  - Listbox — List of selectable items
  - Menubutton — Menu widgets
  - Scrollbar — Scrollbar widget
  - Scale — A slider widget
  - Spinbox — Arrows to increment/decrement values

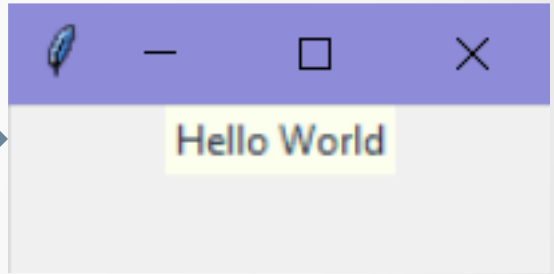
A submodule, called **ttk** contains these and a few additional widgets including: *Combobox*, *Progressbar*, *Notebook*, *Separator*, *Sizegrip*, and *Treeview*
(from tkinter import ttk)

Adding Widgets

- Before the mainloop(), widgets can be added to our window

Continue by trying it now!

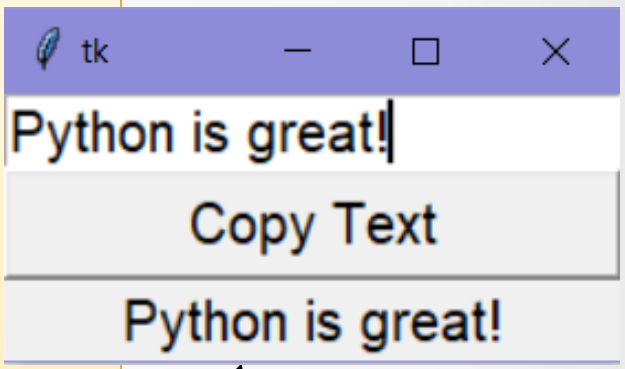
```
import tkinter as tk

root = tk.Tk()

label = tk.Label(text='Hello World',
                 foreground='#223344',
                 background='#fcfeee')
label.pack()

root.mainloop()
```



Event Handling

- Events and handlers can be created to enhance

```
def copy_text():
    label.configure(text=entry.get())

window = tk.Tk()

entry = tk.Entry(font=16)
entry.pack(fill='x')

button = tk.Button(text='Copy Text', font=16,
                   command=copy_text)
button.pack(fill='x')

label = tk.Label(text='', font=16)
label.pack(fill='x')



window.mainloop()
```



Layouts

- Widgets are placed into the view using one of three layout managers:
  - pack() — Tk sizes the window as small as it can while still displaying all the widgets
  - grid() — Widgets are laid out using a table of rows and columns
  - place() — Widgets are placed at specified locations

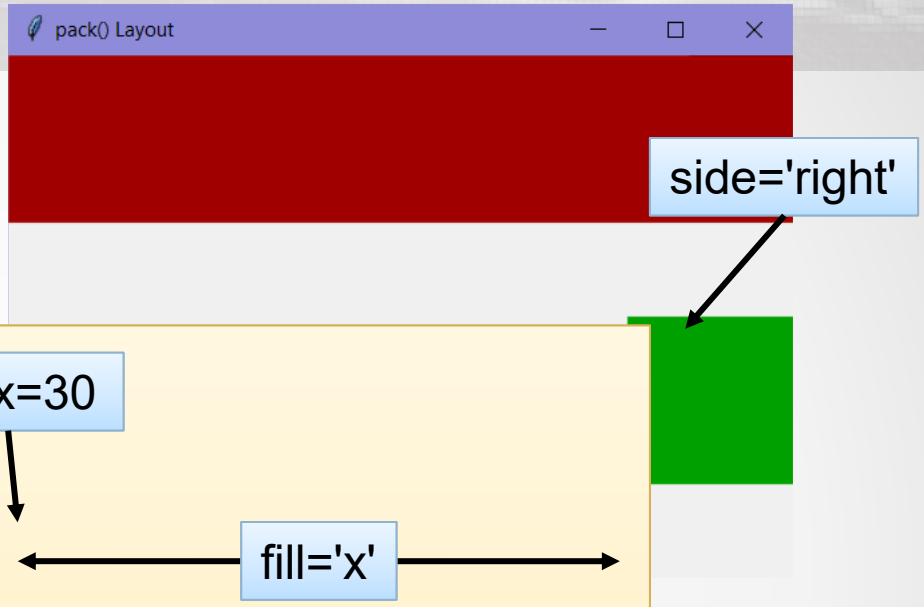
Using the pack() Layout

Frame widgets
100x100

```
import tkinter as tk

root = tk.Tk()
root.title('pack() Layout')
root.geometry('400x300')

frame1 = tk.Frame(width=100, height=100, bg='#a00000')
frame1.pack()

frame2 = tk.Frame(width=100, height=100, bg='#00a000')
frame2.pack()

frame3 = tk.Frame(width=100, height=100, bg='#0000a0')
frame3.pack()

root.mainloop()
```

pack() stacks widgets on top of each other in order they are packed. Remaining space is left empty.

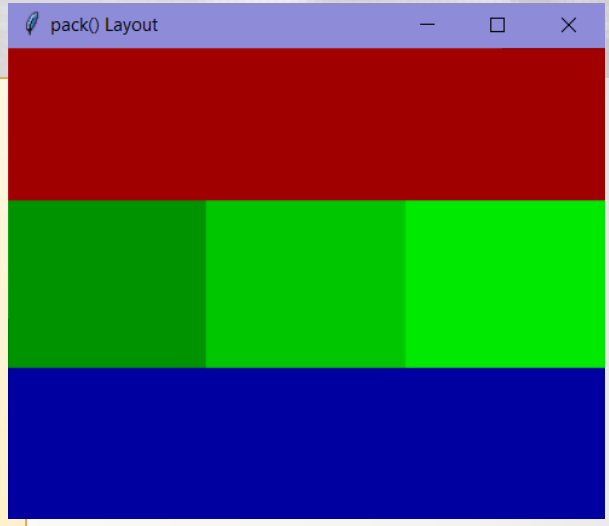
pack() Options

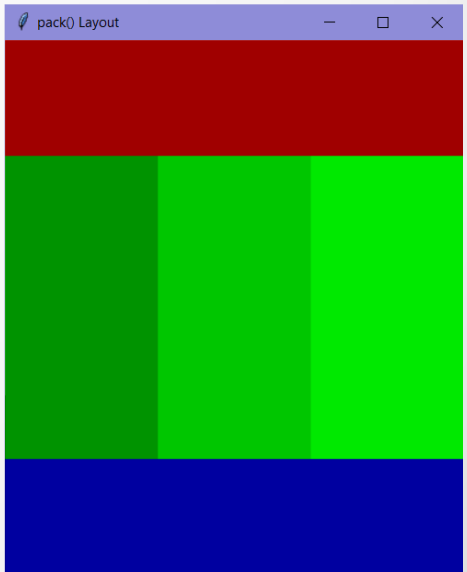



```
import tkinter as tk

root = tk.Tk()
root.title('pack() Layout')
root.geometry('400x300')

frame1 = tk.Frame(height=100, bg='#a00000')
frame1.pack(fill='x')

frame2 = tk.Frame(width=100, height=100, bg='#00a000')
frame2.pack(side='right')

frame3 = tk.Frame(height=50, bg='#0000a0')
frame3.pack(padx=30, fill='x', side='bottom')


root.mainloop()
```

More pack() Options

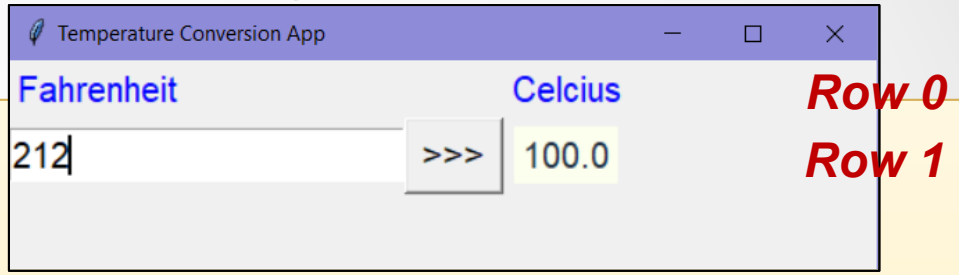
```
root = tk.Tk()
root.title('pack() Layout')
root.geometry('400x300')

frame1 = tk.Frame(height=100, bg='#a00000')
frame1.pack(fill='x', side='top')

frame2 = tk.Frame(height=100, bg='#0000a0')
frame2.pack(fill='x', side='bottom')

frame3 = tk.Frame(bg='#009300')
frame3.pack(expand=True, side='left')

frame4 = tk.Frame(bg='#00c600')
frame4.pack(expand=True, side='left',
            fill='both')

frame5 = tk.Frame(bg='#00e900')
frame5.pack(expand=True, side='left',
            fill='both')

root.mainloop()
```





Using the grid() Layout *(1 of 2)*

- A grid layout allows for defining rows and columns




```
import tkinter as tk

root = tk.Tk()
root.title('Temperature Conversion App')
root.geometry('500x120')

# row 0 definitions
lbl_fahr = tk.Label(font=16, fg='blue', text='Fahrenheit')
lbl_fahr.grid(row=0, column=0, ipadx=2, ipady=2, sticky='W')

lbl_celcius = tk.Label(font=16, fg='blue', text='Celcius')
lbl_celcius.grid(row=0, column=2, ipadx=2, ipady=2, sticky='W')

# row 1 definitions
entry = tk.Entry(font=16)
entry.configure(text='0')
entry.grid(row=1, column=0, ipadx=2, ipady=2)
```

Using the grid() Layout *(2 of 2)*

```
...continued from previous...
def enter_key(event):
    if event.keycode == 13:
        convert()

def convert():
    try:
        value = int(entry.get())
        conversion = 5.0 * (value - 32.0) / 9.0
        lbl_output.configure(text=str(conversion))
    except ValueError:
        lbl_output.configure(text='Must be a number!')
```

Calls *convert()* when the button is pressed

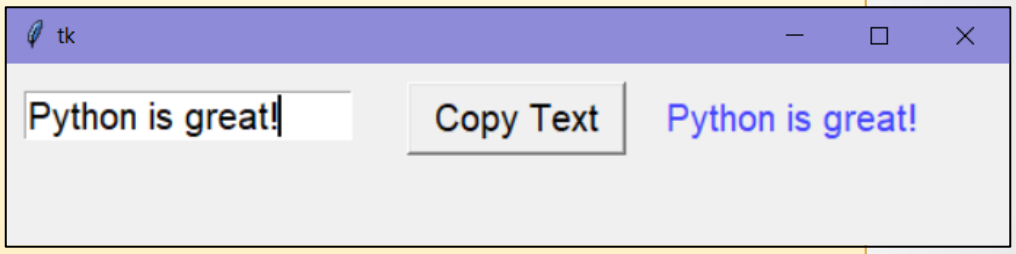
```
button = tk.Button(font=16, text='>>>', command=convert)
button.grid(row=1, column=1, ipadx=2, ipady=2)

lbl_output = tk.Label(master=root, font=16, text='0',
                      foreground='#223344', background='#fcfeee')
lbl_output.grid(row=1, column=2, ipadx=2, ipady=2)

root.bind('<Key>', enter_key)
root.mainloop()
```

bind() assigns our root window to handle <Key> presses and calls the enter_key() event handler

Using the place() Layout

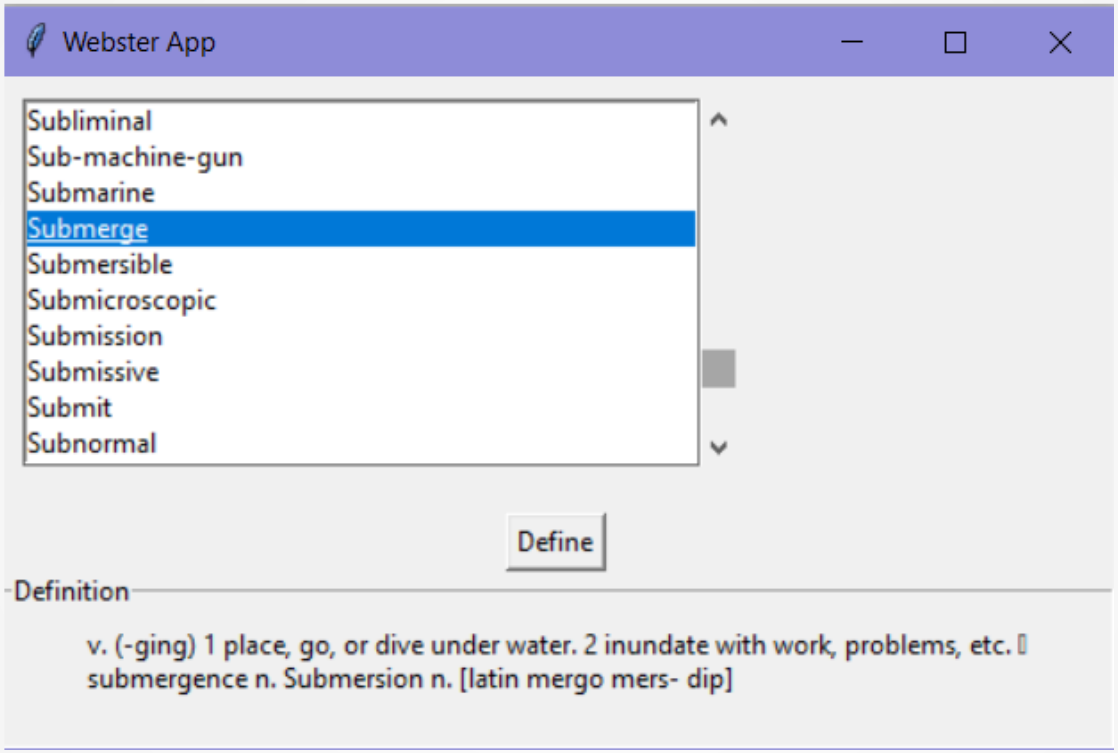
- *place()* allows for specifying x= and y= values to place widgets exactly where you want them
  - The origin (0, 0) is the upper left

```
def copy_text():
    label.configure(text=entry.get())

root = tk.Tk()
root.geometry('550x100')


entry = tk.Entry(width=16, font=16)
entry.place(x=10, y=15)

button = tk.Button(text='Copy Text', font=16, width=10,
                   command=copy_text)
button.place(x=220, y=10)

label = tk.Label(text='', font=16, width=16, fg='#4444ff')
label.place(x=340, y=15)
```



Your Turn! - Task 9-1

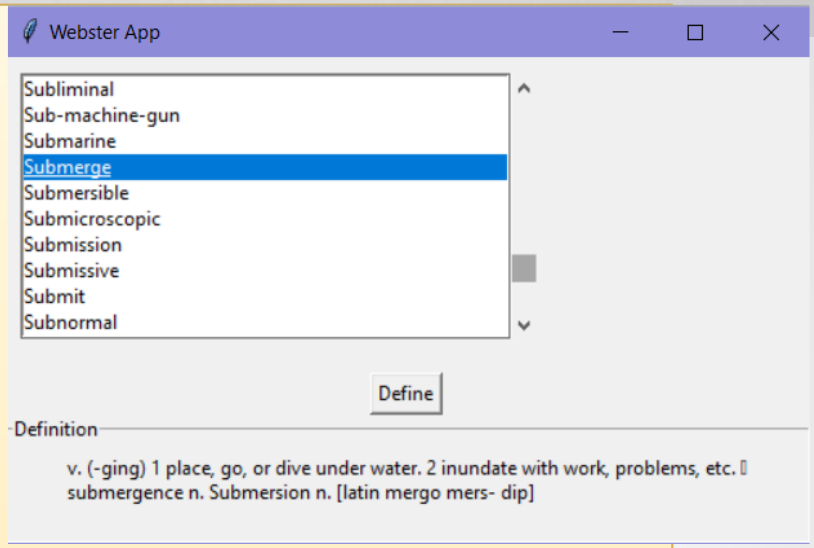
- Creating a GUI with Tkinter



- Create the "Webster App" shown above by following the instructions in the student workbook and working from the *task9_1_starter.py*

Organizing Code into Classes

```
class WordFrame(tk.Frame):
    ...

class DefinitionFrame(tk.LabelFrame):
    ...

class WebsterApp:
    def __init__(self, parent):
        parent.title('Webster App')
        parent.geometry('500x300')
        self.webster = self.load_data()

        self.word_frame = WordFrame(parent, self.webster)
        self.definition_frame = DefinitionFrame(parent)
        self.button = tk.Button(text='Define',
                                command=self.get_selection)
        self.button.pack(side='bottom')

    def load_data(self):
        ...

    def get_selection(self):
        ...
```



Sometimes it is easier to manage solutions using classes

Appendix A - Summary

- Tkinter is an easy-to-use tool that creates GUIs
  - Its major disadvantage is a lack of creative, flexible widgets
  - It also doesn't have a native OS look-and-feel
- Tkinter widgets integrate nicely into Python code, however
- Though discussions have focused on Tkinter, PyQt is also a popular Python GUI option and deserves its own consideration