



TEKsystems Education Services

presents

Intensive Intermediate Python

Copyright

This subject matter contained herein is covered by a copyright owned by:

Copyright © 2024 Robert Gance, LLC

This document contains information that may be proprietary. The contents of this document may not be duplicated by any means without the written permission of TEKsystems.

TEKsystems, Inc. is an Allegis Group, Inc. company. Certain names, products, and services listed in this document are trademarks, registered trademarks, or service marks of their respective companies.

All rights reserved

20750 Civic Center Drive
Suite 400, Oakland Commons II
Southfield, MI 48076

800.294.9360

COURSE CODE IN1468 /1.1.2024, fixes: 3.18.2024



©2024 Robert Gance, LLC

ALL RIGHTS RESERVED

This course covers Intensive Intermediate Python

No part of this manual may be copied, photocopied, or reproduced in any form or by any means without permission in writing from the author—Robert Gance, LLC, all other trademarks, service marks, products or services are trademarks or registered trademarks of their respective holders.

This course and all materials supplied to the student are designed to familiarize the student with the operation of the software programs. THERE ARE NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, MADE WITH RESPECT TO THESE MATERIALS OR ANY OTHER INFORMATION PROVIDED TO THE STUDENT. ANY SIMILARITIES BETWEEN FICTITIOUS COMPANIES, THEIR DOMAIN NAMES, OR PERSONS WITH REAL COMPANIES OR PERSONS IS PURELY COINCIDENTAL AND IS NOT INTENDED TO PROMOTE, ENDORSE, OR REFER TO SUCH EXISTING COMPANIES OR PERSONS.

This version updated: 1/1/2024

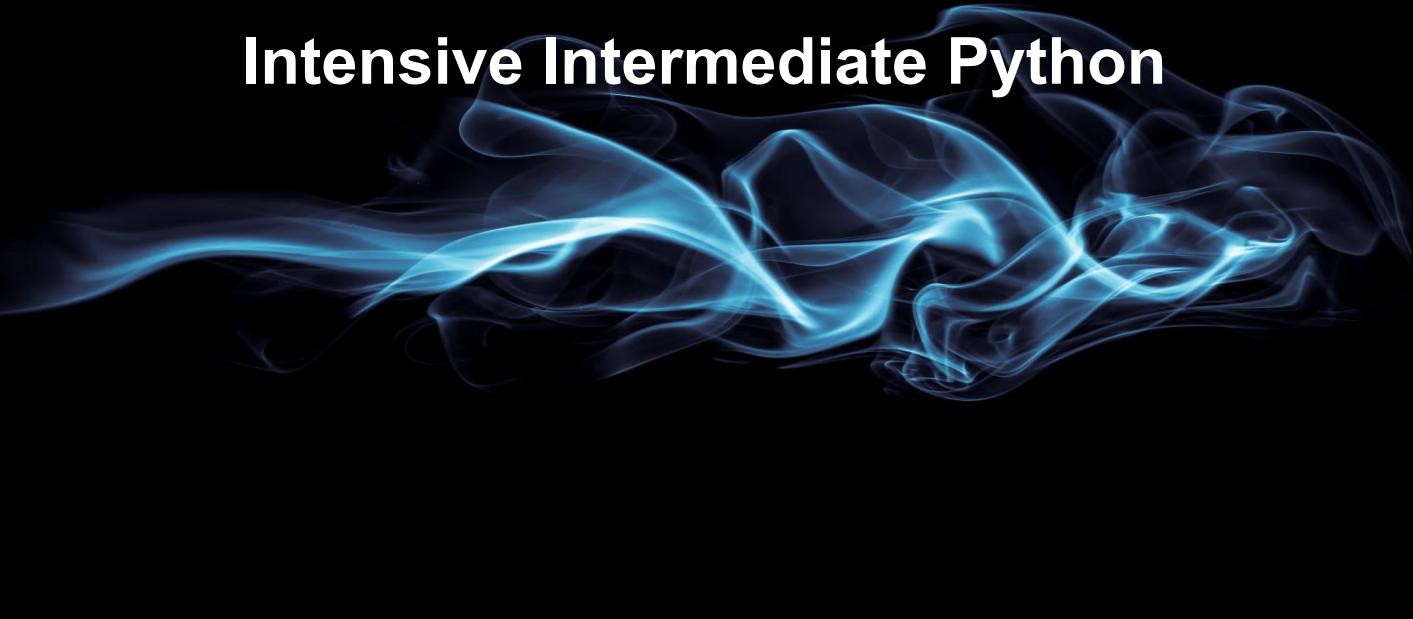
Notes

Chapters at a Glance

Chapter 1	Python Virtual Environments	16
Chapter 2	Language Fundamentals Review	28
Chapter 3	Advanced Iterating	60
Chapter 4	Working with Relational Databases	82
Chapter 5	More with Classes and Objects	105
Chapter 6	Python API Development	126
Chapter 7	Intro to Data Analysis	155
Chapter 8	Advanced Functions and Decorators	203
	Course Summary	236

Notes

Intensive Intermediate Python



Course Objectives

- Reinforce Python fundamentals focusing on improving language understanding
- Deepen knowledge of the Python object model
- Explore additional core APIs to include iterators, generators, comprehensions, and more
- Introduce popular data analytics libraries
- Build scripts using numerous standard library and third-party modules

Course Agenda - Day 1

Day 1

Python Virtual Environments

Language Fundamentals Review

Course Agenda - Day 2

Day 2

Advanced Iterating Concepts

Working with Relational Databases

Course Agenda - Day 3

Day 3

OOP and Classes

Python API Development

Course Agenda - Day 4

Day 4

Intro to Data Analytics

Decorators

Introductions

Name (prefer to be called)



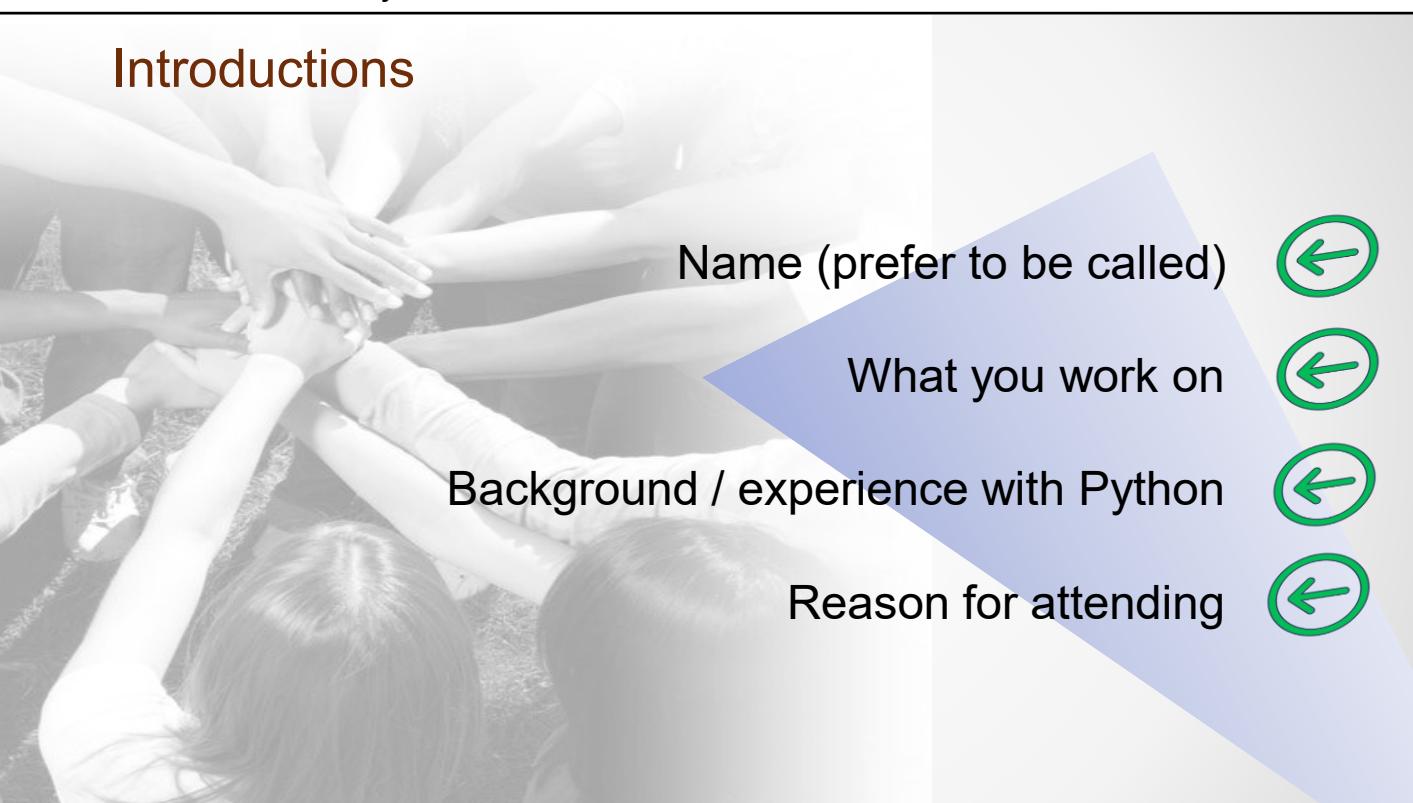
What you work on



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

Python Virtual Environments



Creating and Integrating Virtual Environments

Chapter 1 - Overview

Virtual Environment Tools

venv

Poetry

Other Tools



Python Virtual Environments

- Python virtual environment and package management tools are numerous *and perhaps a little confusing*
 - *virtualenv*
 - *pyvenv*
 - *venv*
 - *pyenv*
 - *Pipenv*
 - *Others*
- A virtual environment *allows for managing different versions of packages* without conflicting with the primary Python installation

Don't confuse Python virtual environments with OS virtualization tools such as VMWare, VirtualBox, Docker, etc.



Tools to Manage Virtual Environments

- Python provides the ability to create separate installations (called virtual environments) to manage third-party packages
 - **virtualenv** was one the first popular tools to be used for both Python 2 and 3 (*even today!*)
 - It needs to be *pip* installed to use
 - **pyvenv** was used in early Python 3 versions
 - It had several issues and never stepped out of virtualenv's shadow and is not used now
 - **venv** became the official virtualization tool starting in Python 3.6
 - It ships with all current versions of Python and is the primary utility used

virtualenv is one of the oldest tools mentioned here. It was first used in 2007 and quickly became a preferred tool. Even today, many will opt to use it as it still does the job and is actively maintained. pyvenv never took off and was not widely adopted. It should not be used. venv was based on virtualenv. It is packaged within Python 3.6+ and therefore doesn't require any additional pip installs to use it. The syntax for virtualenv would be: `virtualenv venv` (where the second word refers to the directory being created).



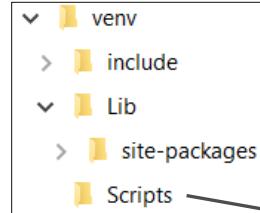
venv

The new directory gets created at the location where the command runs

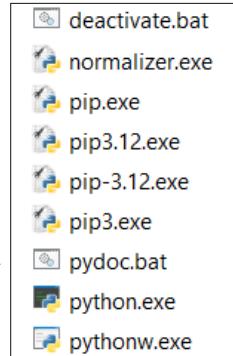
Specify `python`, `python3`, `python3.12`, etc., (as needed) when running `venv`

```
python -m venv relative_or_absolute_path
```

- Using this command creates a new virtual environment at the specified location



On Windows



- Once created, the environment needs to become the primary one by "*activating*" it

venv is the officially defined tool for creating Python virtual environments and the only one that ships with Python. While other tools exist, this is the most commonly used one.



Activation / Deactivation

- To enable the virtual environment, run the **activate** command
 - On Windows: `venv\Scripts\activate`
 - On OS X: `source ./venv/bin/activate`
- When finished, issue the **deactivate** command to end use of the virtual environment or simply close the terminal window
- To **remove** the virtual environment, first **deactivate**, then *delete the venv directory*
 - Optionally, use `rm -rf venv` on OS X

What actually happens when you activate?

Activation simply sets the path to the Python interpreter found in the new virtual environment directory (`venv/bin` or `venv\Scripts`) to the beginning of the OS PATH environment variable.

Activation only affects the terminal (command) window you are currently using.



freeze and requirements.txt

- Use the **freeze** command to identify the contents of currently installed packages

```
pip freeze > requirements.txt
```

requirements.txt can be generated based on the currently installed set of packages

```
Flask==3.0.0
Jinja2==3.1.2
SQLAlchemy==2.0.22
beautifulsoup4==4.12.2
colorama==0.4.6
prettytable==3.9.0
requests==2.31.0
wcwidth==0.2.9
```

- *pip* can be used to install packages from a **requirements.txt** file

```
pip install -r requirements.txt
```

pip freeze when used without -r requirements.txt will list the currently installed packages to the console.

Remember to invoke activate when installing packages using a requirements.txt file so they are directed into the correct virtual environment.



Your Turn! - Task 1-1

Create a Python Virtual Environment

- Work from the instructions provided in the workbook (*found in the back of the student manual*)
- You will perform the following
 - Create and activate a new virtual environment
 - *pip install* packages into it
 - Connect it to PyCharm (the IDE used for this course)

Locate the instructions for this exercise in the back of the student manual

Intensive
Intermediate
Python
Exercise Workbook

Task1-1
Creating a Virtual Environment

Overview
This exercise is designed to demonstrate how to create a virtual environment within PyCharm. This virtual environment becomes the Python used throughout the remainder of the course.

Survey Your System
Begin by opening a terminal (OS X) or command (Windows) window. From here onward, we'll refer to it simply as a terminal. Within the terminal, type each of these commands:

```
python -V
python3 -V
```

python3.12 -V (replace 3.12 with the appropriate version number, e.g., python3.11, python3.13, etc.)

One of these commands should have responded with a valid version number of Python that you were expecting. Remember this command because you will use it throughout the rest of the course. It's important to remember which command to use. On Mac OS X, your terminal command may be python3 or python3.x. So, remember it!

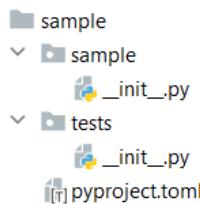


Poetry

- Poetry is a multi-purpose project manager for Python
 - It supports multiple subcommands for performing different actions

`poetry new sample`

Creates a new project with this structure



Symbols allowed:

"1.2.3"	exact version only
"^1.2.3"	Version up to leftmost non-zero value (1.2.3 >= x < 2.0.0)
"~1.2.3"	Changes up through least significant digit (1.2.3 >= x < 1.3.0)
Can also use >, <, <=, !=	
"1.2.*"	Latest version in the position indicated (1.2.0 >= x < 1.3.0)

`pip install poetry`

```
[tool.poetry]
name = "sample"
version = "0.1.0"
description = ""
authors = ["Your Name <you@example.com>"]
readme = "README.md"
```

Identify basic project info

```
[tool.poetry.dependencies]
python = "^3.12"
requests = "*"
prettytable = "*"
```

Add packages into the .toml file

```
[tool.poetry.dev-dependencies]
pytest = "^7.4.3"
```

```
[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Poetry was first released in 2018. It can support package dependency management, management of virtual environments, and the build and deploy process. Using `poetry new --src sample`, the sample package above would be placed into a /src directory (a common preferred format for some projects). The main configuration file, `pyproject.toml`, comes from PEPs 517 and 518. TOML stands for Tom's Obvious Minimal Language format.

`poetry init` is similar to `poetry new <project_name>` except it builds the project in the current directory. This is useful when an existing Python project is already present in the current directory. It will create the `.toml` file.

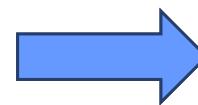


Poetry (continued)

- To create a virtual environment, issue `poetry env use path/to/python_exe` from within the project directory

`poetry env use` creates a virtual python environment in a pre-defined directory depending on the OS:
 Windows: C:\Users\<username>\AppData\Local\pypoetry\Cache
 OS X: ~/Library/Caches/pypoetry
 Linux: ~/.cache/pypoetry

- To install the dependencies defined in the `.toml` file, run `poetry install`



Creates a `poetry.lock` file containing exact versions installed

`poetry shell` - activates the new environment
`exit` - exits the shell
`poetry env remove <name of poetry_env>`
 - removes the virtual environment

poetry.lock
 ...
 [[package]]
 name = "prettytable"
 version = "3.9.0"
 ...
 [[package]]
 name = "requests"
 version = "2.31.0"

To test out using poetry, perform the following:

- In a command window, **activate** the virtualenv previously created.
- Run (within the `student_files` directory): **poetry new sample**
- Add `requests = "*"` and `prettytable = "*"` to the `pyproject.toml` file (by using PyCharm)
- `cd` into `sample` and run the command **poetry env use python**
- Run the command **poetry shell** followed by **pip install poetry**
- Run **poetry install**. Check the environment with **poetry env info**.
- Finally, type **exit** and then run **poetry env remove <env_name>** to remove it.



Other Tools

- **pyenv** is a Python *version* switching tool for Unix/Linux
 - A Windows version exists as well

`pip install pyenv`

`pip install pyenv-win`

```
$ pyenv install 3.12      # installs latest 3.12 version
$ pyenv versions          # list installed versions
$ pyenv global 3.10.5     # selects local version
$ pyenv local 3.11.3      # selects global version
```

- **pipenv** is another package management tool designed to emulate Ruby and Node.js environments

`pip install pipenv`

- Use **pipenv install** to create an environment at the location where you run the command
 - Run **pipenv shell** to activate
 - Deactivate (afterwards) by exiting the shell

pipenv installed modules are added to the `~/.virtualenvs` directory.

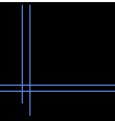
pipenv behaves somewhat like Ruby's gem or Node.js' npm package managers. It defines lock files that identify and manage specific versions of files that should be installed. It has lost popularity over the last few years due in part to inactive project maintenance



Chapter 1 Summary

- Python supports a number of packages and tools related to managing installed Python packages and environments
- For most requirements, **venv** works fine and is the official built-in tool for virtual environment creation
- **poetry** is a multi-purpose tool providing virtual environment creation, package management, and build/deploy capabilities
- Numerous other package installation and management tools exist
 - **pipenv**
 - **conda**
 - **wheel**

For more on the topic of package management tools: <https://packaging.python.org/guides/tool-recommendations/>



Chapter 2

Fundamentals Review



Basic Language Concepts (Some Review, Some New)



Chapter 2 - Overview

Key Data Structures

File Handling

Functions

Modules

Controlling Printed Output

- The print function has the following syntax

```
print(val1, val2, ..., sep=' ', end='\n', file=sys.stdout)
```

sep= String value inserted between val1 and val2, etc.

end= defines char(s) to add to end of a line (i.e., line-separator character)

file= defines where output will be sent

```
print('a', 'b', 'c', sep='_')
```

a_b_c

```
for item in ['a', 'b', 'c']:  
    print(item, end=' ')
```

a b c

Use **sep=** to define the string between val1 and val2. Use **end=** to define what is added to the output at the end of the printed result (the default is '\n').

F-Strings and Formatted Output

- A way to format output values is to use the string **format()** method

```
{ fieldname | index [:format] }
```

- f-strings allow previously defined variables to be formatted

```
employee = ['Bill', 'Smith', 37]
results = 'Name: {0} {1}, age: {2}'.format(*employee)
print(results)

def show_name():
    return ' '.join(employee[:2])

print(f'Name: {show_name()}, age: {employee[2]:0.2f}')
```

Name: Bill Smith, age: 37

Slice notation

Unpacking

Formatting specification

Name: Bill Smith, age: 37.00

Python 3.12 allows nested "same-type" quotes in f-strings

ch02_basics/01_formatting_values.py

f-strings are faster than using `format()`, `%`, or `+` in string operations. f-strings, have become a preferred syntax.

The format specification takes the following form:

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

More on the `format()` method: <https://docs.python.org/3/library/string.html>.



Working with Sequences

- Sequences are ordered collections of objects
 - Types include: **str, list, tuple**

```
sa_countries = [
    ('Brazil', 216_000_000), ('Columbia', 52_000_000),
    ('Argentina', 45_700_000), ('Peru', 34_300_000),
    ('Venezuela', 28_800_000), ('Chile', 19_600_000),
    ('Ecuador', 18_200_000), ('Bolivia', 12_400_000),
    ('Paraguay', 6_800_000), ('Uruguay', 3_400_000),
    ('Guyana', 813_000), ('Suriname', 623_000),
    ('French Guiana', 312_000), ('Falkland Islands', 3_800)
]
```

Numbers can contain underscores to ease readability

- Random access and slicing

```
sa_countries[2]
sa_countries[-2:]
```

('Argentina', 43100000)

[('Suriname', 56000), ('French Guiana', 262000)]

- Membership

```
'brazil'.capitalize() in [country for country, pop in sa_countries]
```

True

ch02_basics/02_lists.py

Sequences exhibit common behaviors such as the ability to be randomly accessed, support slicing, concatenation and replication through multiplication of a scalar value (the last two are not shown here).

The last example searches for 'Brazil' within the data structure. It does this by using a list comprehension. The list comprehension will generate a list containing the names of each country.



Declaring namedtuples, NamedTuples, Dataclasses

- Python provides several options for holding data records

```
# classic namedtuple
Country = namedtuple('Country', 'name population')

# typed NamedTuple
Country = NamedTuple('Country', [('name', str), ('population', int)])

# alternatively...
class Country(NamedTuple):
    name: str
    population: int = 0

# dataclass
@dataclass
class Country:
    name: str
    population: int = 0
```

	namedtuple (classic)	NamedTupl e (typed)	dataclass
Advantages	Dot notation to access attributes, Python 2 support	Supports type defs on fields	Supports types on fields, fields can be modified
Disadvantages	Fields are immutable, types on fields not supported	Fields are immutable	Fields are not indexable

ch02_basics/03_namedtuples_and_dataclasses.py

Typed NamedTuples provide a way to define a collection that still behaves like a tuple but provides dot notation to access its attributes. typing.NamedTuple is an improvement over the classic namedtuple type. This new version only differs in the fact that it supports types on its fields while the classic namedtuple doesn't support types.

Unlike namedtuples or NamedTuples (Py 3.6+), dataclasses (Py 3.7+) are mutable and NOT indexable. They perform well, perhaps even better than namedtuples and NamedTuples which make them ideal alternatives to the classic namedtuple.

Using namedtuples, NamedTuples, Dataclasses

- Each of the definitions on the previous slide would work the same

```
sa_countries = [  
    Country('Brazil', 216_000_000), Country('Columbia', 52_000_000),  
    Country('Argentina', 45_700_000), Country('Peru', 34_300_000),  
    Country('Venezuela', 28_800_000), Country('Chile', 19_600_000),  
    Country('Ecuador', 18_200_000), Country('Bolivia', 12_400_000),  
    Country('Paraguay', 6_800_000), Country('Uruguay', 3_400_000),  
    Country('Guyana', 813_000), Country('Suriname', 623_000),  
    Country('French Guiana', 312_000), Country('Falkland Islands', 3_800)  
]  
  
sa_countries.sort(key=lambda country: country.name)  
  
for ctry in sa_countries:  
    print(f'{ctry.name:<20}{ctry.population:>15,}')
```



Working with Files

- Use `open(file, mode, buffering, encoding, ...)` to work with files

```
import sys

f = None
lines = []
try:
    f = open('access_.log', encoding='utf-8')
    lines = f.readlines()
except IOError as err:
    print(err, file=sys.stderr)
finally:
    if f:
        f.close()
```

If the file is not found (let's say), open() fails raising an `IOError`

`file=` routes error messages to the `sys` module's `stderr`

Calling `f.close()` without checking the validity of `f` first could result in `None.close()` which would raise an `AttributeError`

ch02_basics/04_file_reading.py

In the example, an `IOError` can occur for failing to open the specified file. Therefore, `f` will only be conditionally valid (in the case where the file opens successfully). So, it is necessary to check the variable `f` for a value before attempting to close. The `finally` block is called whether the `open()` operation was successful or not. This makes it an appropriate location to close the file (if opened).



Initialization and Cleanup Using the *with* Control

- When working with resources, often initialization or cleanup operations are needed
 - Examples:* Closing files, database connections, network connections (sockets), locks on threads, and more...
- with* is a control structure that can simplify initialization and cleanup

do some initialization

do some work

do some cleanup

```
with contextmanager as obj:  
    do_work
```

A **context manager** is a special object that defines how to initialize at the beginning and cleanup afterwards

The *with* control has a few hidden features that make it very versatile. It requires the use of a special object called a **context manager**. The context manager is usually a class that implements two methods (discussed shortly).

Using the *with* Control

- The object returned from `open()` is a context manager
 - It can be used within the `with` control

```

import os
import sys

path = '../resources'
filename = 'access_.log'           Safely joins a path and filename
filepath = os.path.join(path, filename)
lines = []

try:                                The file gets closed automatically
    with open(filepath, encoding='utf-8') as f:
        lines = f.readlines()
except IOError as err:
    print(f'Error {err}', file=sys.stderr)

print(f'{len(lines)} lines read.')   100000

```

ch02_basics/05_with_files.py

This version will result in the same output and proper file closing as the previous file reading example (whether there is an exception or not). This version uses the `with` control providing a cleaner solution overall.

In addition to the `with` control, `os.path.join()` illustrates a clean way to join paths and filenames.

How *with* Works

```

from ipaddress import ip_address

class OutputRedirector:
    def __init__(self, filename, data):
        self.filename = filename
        self.data = data

    def __enter__(self):
        self.file = open(self.filename, 'wt', encoding='utf-8')
        self.stdout_prev = sys.stdout
        sys.stdout = self.file

    def __exit__(self, typ, val, tb):
        sys.stdout = self.stdout_prev
        if not self.file.closed:
            self.file.close()

with OutputRedirector('filtered_results.txt', results):
    for entry in results:
        print(ip_address(entry[0]))

```

Opens a file to write to, saves the current stdout, points stdout to the open file

Restores the stdout and closes the file

Writes data to file using a simple print statement

ch02_basics/06_custom_with.py

The example above is designed to show the customization that the `with` control supports. In the example, `OutputRedirector` temporarily redirects any `print()` statements into the file until the `with` control ends. After the `with` control ends, `print()` output is sent to the typical output console (device) once again.



A Utility For Output Redirecting

- The Python standard library provides a tool for accomplishing the previous example:

```
from contextlib import redirect_stdout

with redirect_stdout(open('filtered_results2.txt', 'wt', encoding='utf-8')):
    for entry in results:
        print(ip_address(entry[0]))
```

Output is sent to the open file

ch02_basics/06_custom_with.py

The previous example can be accomplished using a built-in utility found in the `contextlib` module of the standard library. It accepts an open file handle and will temporarily redirect any output from a `print()` function to the open file. For brevity, exception handling has been omitted.

Pathlib

- `pathlib.Path` provides an object-oriented wrapper for a file or directory
 - Many `pathlib.Path` object methods replace `os` functions

<code>pathlib.Path()</code>	<code>os</code> Module	Comment
<code>p = Path('.') p / 'file.txt'</code>		Create Path object Joins a dir & fn
<code>p.is_file(), p.is_dir()</code>	<code>os.path.isfile(item), os.path.isdir(item)</code>	Is it a file or directory?
<code>p.exists()</code>	<code>os.path.exists(item)</code>	Does it exist?
<code>p.resolve()</code>	<code>os.path.abspath(item)</code>	Absolute path
<code>[f for f in p.iterdir()]</code>	<code>os.listdir(dir)</code>	List contents of dir
<code>p.name</code>	<code>os.path.basename(item)</code>	Filename only
<code>p.parent</code>	<code>os.path.dirname(item)</code>	Containing directory

The table above shows some (near) equivalents between the `Path()` object and methods within `os` and `os.path`. In the table, `fn`=filename, `dir`=directory, and `item`=path+filename.



Variations on Ways to Read a File

```
from pathlib import Path

path = Path('../resources')
filename = 'access_.log'
filepath = path / filename
```

Creates a `pathlib.Path()` object

`filepath` is a new `Path()` object with the filename appended

Using `open(Path_obj)`

```
with open(filepath, encoding='utf-8') as f:
    lines = f.readlines()
```

`Path()` objects may be used within `open()`

Using `Path().open()`

```
with filepath.open(encoding='utf-8') as f:
    lines = f.readlines()
```

`Path` objects come with an `open()` method

Using `Path().read_text()`

```
lines = filepath.read_text(encoding='utf-8').split('\n')
```

`Path().read_text()` closes
the file automatically

ch02_basics/07_with_pathlib.py

For brevity, exception handling is not shown.

The `read_text()` method of the `Path()` object will automatically close the file. This approach will result in one extra (blank) line at the end of the file. It can be removed with `Path().read_text().split('\n')[:-1]`. `read_text()` is suitable for smaller to medium sized files but not for larger datafiles as it will attempt to read the file into a single string within memory.



Your Turn! - Task 2-1

Part 1

Reading From Files (part 1 of 3)

- cities15000.txt contains *population* and *elevation* info related to cities around the world

```
3041563 Andorra la Vella    Andorra la Vella    ALV,Ando-la-Vyey,Andora,Andora la Vela,Andora la Velja,Andora lja Vehl
290594 Umm al Qaywayn   Umm al Qaywayn   Oumm al Qaiwain,Oumm al Qaiwain,Um al Kawain,Um al Quweim,Umm al Qaiwain,Umm a
291074 Ras al-Khaimah   Ras al-Khaimah   Julfa,Khaimah,RKT,Ra's al Khaymah,Ra's al-Chaima,Ras al Khaima,Ras al Khaimah,
291696 Khawr Fakkān     Khawr Fakkan     Fakkan,Fakkān,Khawr Fakkan,Khawr Fakkān,Khawr al Fakkan,Khawr al Fakkān,Khor F
292223 Dubai            Dubai           DXB,Dabei,Dibai,Dibay,Doubayi,Dubae,Dubai,Dubai emiraat,Dubaija,Dubaj,Dubajo,Dubajus,Dub
```

- Your task is to [read elevation and population](#) info from this file
 - Note:** This file is a **tab-separated value** file not a csv!
- Work from [task2_1_starter.py](#)
 - Use a *with* control and **Path** object
 - Read records into a **City** typed **NamedTuple**

Read data into the provided
cities sequence



Sorting, Iterating Sequences

- To customize sorts, use `sort()` or `sorted()` with a `key=` parameter

```
sa_countries = [
    ('Brazil', 216_000_000),
    ('Peru', 34_300_000),
    ('Columbia', 52_000_000),
    ('Argentina', 45_700_000)
]
```

```
def sort_by_population(country):
    return country[1]
```

Returns the population field

```
def sort_by_name(country):
    return country[0]
```

Returns the country name

```
sa_countries.sort(key=sort_by_population)
print(sa_countries)
```

```
[('Peru', 34300000),
 ('Argentina', 45700000),
 ('Columbia', 52000000),
 ('Brazil', 216000000)]
```

```
new_ctys = sorted(sa_countries,
                  key=sort_by_name,
                  reverse=True)

print(new_ctys)
```

```
[('Peru', 34300000),
 ('Columbia', 52000000),
 ('Brazil', 216000000),
 ('Argentina', 45700000)]
```

ch02_basics/08_sorting.py

A function can specify specifically how to sort when the default is not sufficient. In a list of tuples (as the example provides), the default behavior is to sort on the first field (column) of the tuple. Therefore, the `sort_by_name()` function isn't necessary here.

Lambdas

- Lambdas are *inline-functions*
 - For quick, short, throw away uses such as sorting, closures, functional programming
- The following lambda sorts a list of tuples by the second field

```
sa_countries.sort(key=lambda country: country[1],  
                   reverse=True)  
  
print(sa_countries)
```

```
[('Brazil', 216000000),  
 ('Columbia', 52000000),  
 ('Argentina', 45700000),  
 ('Peru', 34300000)]
```

ch02_basics/08_sorting.py

Lambda limitations:

<ret_val> must be equivalent to what a legal return value would be. No assignments (e.g., x = 10) in the lambda.

No if or for loops allowed.

This example sorts a list of tuples using a lambda. It sorts records in descending order of their population (reverse=True) by using the key= parameter. The lambda receives one tuple at a time, upon which it returns country[1] (the population field).

 Your Turn! - Task 2-2

Part 2

Sorting and Max (*part 2 of 3*)

- Use sorting to determine
 - *the most populous city* in the world
 - the highest city (elevation)
- Continue working from your previous solution (task2_1_starter.py) or work from task2_2_starter.py
- *After getting it working, can you do it using the max() function?*
 - Which way is better?

```
max_obj = max(iterable, key=sort_by_func)
```

A second keyword parameter for the max() function is called default. It returns the value specified if the iterable is empty.

List Comprehensions

- List comprehensions provide a convenient way to create a list from another iterable

```
newlist = [ expression for var in iterable test_condition ]
```

```
larger = [country for country, pop in sa_countries
          if pop >= 20_000_000]
print(larger)
```

['Brazil', 'Columbia', 'Argentina', 'Peru', 'Venezuela']

```
from pathlib import Path
city_names = []
filepath = Path('../resources/cities15000.txt')
if filepath.exists():
    city_names = [line.split('\t')[1]
                  for line in filepath.open(encoding='utf-8')]
print(city_names)
```

['les Escaldes', 'Andorra la Vella', 'Umm al Qaywayn', ...]

ch02_basics/09_listcomps.py

The first example uses a list comprehension to create a new list containing countries with populations over 20 million.

The second example reads only the city names from cities15000.txt into a list.

Functions

- Functions must be defined before being called and all parameters without defaults must be provided in the call

```
from pathlib import Path

def check_files(path='..', filename=''):
    p = Path(path)
    fullpath = p / filename
    return fullpath.exists()

print(check_files('..//ch02_basics', '02_lists.py'))
print(check_files())
print(check_files(filename='02_lists.py',
                  path='..//ch02_basics'))
```

Provide *default arguments* to add flexibility for users when calling your functions

The function returns *True* or *False* if a specified file exists

True
True
True

Keyword arguments can also be used for most functions

ch02_basics/10_functions.py

Functions must be defined (or imported) before they can be called. Parameters passed must be provided if a default value isn't specified in the function definition. In this example, both parameters have default arguments, so it's possible to call this function with no arguments at all.

The `fullpath` object (on the left of the equation) will also become a `Path` object.



Multiple Positional Arguments

- To define a function in which the number of parameters passed are variable, use the *

The Python `all(iterable)` function returns `True` only if all items evaluate to True

```
def check_all_files(path='..', *files):
    p = Path(path)
    results = all([(p / x).exists() for x in files])
    return results

print(check_all_files('../ch02_basics',
                     '02_lists.py', '04_file_reading.py'))
print(check_all_files('..ch02_basics', '02_lists.py',
                     '04_file_reading.py', 'not_there.py'))
```

True

False

ch02_basics/10_functions.py

In the example above, the function returns True if all filenames provided exist.

As a convenience feature for those who call your functions, you might write the function to perform flexibly. Python provides a way that allows passing as many arguments as needed using the * notation. For this to work properly, the *files parameter MUST be declared after all positional arguments. It cannot appear before any other positional arguments and only ONE multiple positional argument can be supplied.

Multiple Keyword Arguments

```
def check_all_files_extra(path='..', *files, **kwargs):
    p = Path(path)
    results = [(p / x).exists() for x in files]
    if kwargs.get('individual'):
        return results
    return all(results)

print(check_all_files_extra('', '02_lists.py', '04_file_reading.py')) True

print(check_all_files_extra('', '02_lists.py', '04_file_reading.py',
                           'not_there.py', individual=True)) [True, True, False]

print(check_all_files_extra('', '02_lists.py', '04_file_reading.py',
                           'not_there.py', individual=False)) False
```

ch02_basics/10_functions.py

When numerous additional parameters may be passed into a function, you can optionally capture these as keyword arguments that are placed into the `**kwargs` dictionary short for "keyword arguments." This is a common name to use (though not required) in Python.

In the example, a single keyword argument is supplied. It is not a declared parameter and therefore needs to be viewed within the `kwargs` dictionary.



Your Turn! - Task 2-3

Part 3

Add a Search Capability (part 3 of 3)

- Create a function that performs a *search capability*
 - It should accept a city name (or partial name) and *return a list of valid matches and populations* for each city

Enter the (partial) name of the city: <i>Los Angeles</i>			
name	population	elevation	country
East Los Angeles	126,496	63	US
Los Angeles	3,971,883	96	US

- *Additionally:* Turn the code that calculates the largest and highest cities into functions also
- Continue from your previous solution (`task2_1_starter.py` or `task2_2_starter.py`) or work from `task2_3_starter.py`

Adding Flexibility: Using `getattr()`

- Our solution only works with `cities15000.txt`; it's not very versatile
 - Can we make it work with most *any* data file?
 - `attribute = getattr(obj, 'field_name')` can make the solution work for any field

```
def largest():
    return max(cities, key=lambda city: city.population)
```

These functions are very similar.
Can we make it more reusable?

```
def highest():
    return max(cities, key=lambda city: city.elevation)
```

How do we make an *attribute* a
value that could be passed in?

```
def most(field_name):
    try:
        return max(_items, key=lambda record: getattr(record, field_name))
    except ValueError:
        return None

most('population')
```

The object

Its attribute
(as a string)

ch02_basics/11_getattr.py

In our previous tasks, we ended up creating a couple of functions that weren't very reusable. They only work for the `cities15000.txt` file and only for the population and elevation fields. To make it more versatile, we could remove the hardcoded field names and pass in the field name we are interested in working with. But this causes a problem. The attribute is now provided as a string and needs to be used as an attribute of the object. How can we allow our object to work with the field name when it's provided as a string parameter to the function?



Modules

- Modules act as both *namespaces* and *code containers*
 - Modules should contain **functions**, **variables** and **classes**
 - Import the module to access items within it
- Modules can be imported several ways
- Organize imports into three categories

Standard library modules

```
import pathlib
p1 = pathlib.Path('foo.py').name

import pathlib as pl
p2 = pl.Path('foo.py').parent

from pathlib import Path
p3 = Path('foo.py').cwd()

from pathlib import Path as P
p4 = P('foo.py').suffixes
```

Third-party modules

```
import linecache
import sys
from os import getcwd, listdir

import requests
from pandas import read_csv

import mymodule
from mymodule import foo
```

Custom modules

There are several ways to import modules. There are no performance differences between each approach as they all require reading the module entirely. The choice of import technique is largely a matter of preference at the time. To determine the top-level items within a module, issue the `dir()` command after importing the module.

When organizing imports, create three sections as shown above. Place "import" statements first, then place "from ... import" statements next. Within each set of import and "from ... import" statement, place them in alphabetic order.

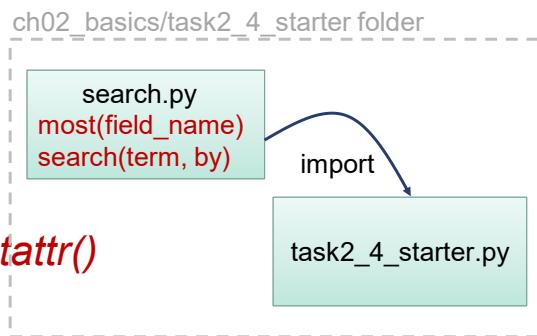


Your Turn! - Task 2-4

Part 4

Functions and Modules

- Refactor and modularize *Task 2-3* to make it more reusable
- Move `search()`, `largest()`, and `highest()` functions from the provided `task2_4_starter.py` into `search.py`
- Move the code that reads the data from the file into `search.py`
- Combine `largest()` and `highest()` to use `getattr()` making a single function called `most()`
- Modify `search()` to search on a given `field_name` (`by=` parameter)
- Test the search module using `task2_4_test.py`



[Open `search.py` for further notes](#)



Ternary Operator

- Python has a shorthand notation for an if-else structure called the ternary operator

```
result = true_value if expression else false_value
```

```
def convert(data):
    value, typ = data if isinstance(data, tuple) else (data, str)
    try:
        return typ(value)
    except Exception as err:
        raise TypeError('Error converting.') from err

test_values = [('16.6', float), 33.7, ('hello', int)]
for value in test_values:
    try:
        result = convert(value)
        print(result, type(result))
    except Exception as err:
        print(err.__cause__)
```

ch02_basics/12_ternary_raise_from.py

The ternary operator provides a concise syntax for evaluating if-else conditions. It can be convenient in certain situations. Be careful not to overuse this control structure. Overly complex solutions can be difficult to read. A fun but hacky way to achieve the ternary operator effect is as follows:

```
result = (false_value, true_value)[bool(expression)]
```

In the above example, convert() is called repeatedly. If a type is provided, an attempt to type cast it is made.

Using raise...from

- Chained exceptions can retain the root cause of an exception by using the *from* clause when re-raising exceptions

```
def convert(data):
    value, typ = data if isinstance(data, tuple) else (data, str)
    try:
        return typ(value)
    except Exception as err:
        raise TypeError('Error converting.') from err

test_values = [('16.6', float), 33.7, ('hello', int)]
for value in test_values:
    try:
        result = convert(value)
        print(result, type(result))
    except Exception as err:
        print(err.__cause__)
```

invalid literal for int() with base 10: 'hello'

Came from the int('hello') ValueError exception that gets raised originally

Exception handling is the preferred approach over the use of numerous if-checks when structuring code. Exceptions can result from other exceptions. When re-raising errors, the original exception information can be retained using the *from* clause, which sets the *__cause__* attribute of the exception.



Your Turn! - Task 2-5

Part 5

A Generic Function to Read Data

- Replace the current code that reads data from cities15000.txt
- Create a `read_data()` function that accepts the name (or Path object) of a file and returns a list of NamedTuple objects *for any file*
- Do this by defining a dict which identifies the desired fields to read

```
read_data('filename',
          data_fields={'colname1': (1, str), 'colname2': (2, int), ...},
          sep=',')
```

Diagram annotations:

- Filename or Path**: Points to the string 'filename' in the function call.
- dict**: Points to the variable 'data_fields' in the function call.
- File separator character**: Points to the string ',' in the function call.
- attribute name within the NamedTuple**: Points to the key 'colname1' in the dictionary 'data_fields'.
- column number in the file, and desired data type**: Points to the tuple value (1, str) in the dictionary 'data_fields'.



Type Hints (Annotations)

- Type hints inform users of required types for functions and variables
 - Useful within smart IDEs or type checkers such as **Mypy**, **Pyre**, or style checkers such as **pylint**, **flake8**, and **black** or for documentation

```
from dataclasses import dataclass
from typing import Tuple, Any, Union, Optional

Fullname = tuple[str, str] ← Fullname is a type alias

def display_info(name: Fullname, nickname: Any, age: int, spouse: str,
                 children: list,
                 parents: Tuple, ← As of Py3.9, use the native types
                 children_ages: list[Union[str, int]], ← (lower case) instead of the typing
                 parent_ages: list[str | int], ← module versions (upper case)
                 other_family: Optional = None) -> dict:
    return display_info.__annotations__

print(display_info(('John', 'Smith'), 'Johnny', 40, 'Sally', ['Tim', 'Sam'],
                  parents=('Martha', 'Frank'), children_ages=['22', '10'],
                  parent_ages=[65, '70']))
```

ch02_basics/13_annotations.py

The example shows how various annotations can be used. Many items from the typing module are no longer used because the native types can be used for annotations as of Python version 3.9. Union was replaced by the vertical bar (pipe) to indicate a choice. Optional implies it can be present or None but doesn't automatically set a value to None, so a default would still have to be specified. Any is not extremely valuable as it implies any value can be provided, but it does inform the user of this condition.

Your Turn! - Task 2-6

Part 6

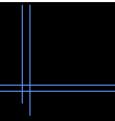
Adding Type Hints

- Revisit `search.py` from Task 2-5 adding type hints to functions by indicating types for parameters and return values
- There are no specific instructions for this task. Work from `search.py` in either your `task2_5_starter` folder or from the provided `task2_6_starter` folder



Chapter 2 Summary

- While Python is recognized as being a very readable and easy to learn language, it is evident that Python is deeper and more complex the more we look under the hood
- Python continues to evolve including new features such as type hints, ordered dictionaries, Path objects, and much more
 - Exercise care when using these features as they are not typically backward compatible with earlier versions



Chapter 3

Advanced Iterating



Additional Features and Modules to Aid with Iterating



Chapter 3 - Overview

Iterating in Python

Generators

Collections and Itertools

Argument Parsing



Creating Iterator Objects

- Python allows for creating custom iterators by implementing the `__iter__` and `__next__` magic methods

```
class MyIterable:
    def __init__(self, min=0, max=5):
        self.count = min
        self.min = min
        self.max = max

    def __next__(self):
        val = self.count
        self.count += 1
        if self.count > self.max:
            raise StopIteration
        return val

    def __iter__(self):
        self.count = self.min
        return self
```

Called repeatedly for each pass through the for-loop

Called once before the for loop begins

```
for i in MyIterable():
    print(i, end=' ')
```

0 1 2 3 4

ch03_iterating/01_iterating.py

The requirements when creating a class-based iterator are the implementation of `__next__()` and `__iter__()`.

`__iter__()` should return the object with the `__next__` method, which is generally the `self` (or current) object.

Special Iterators: Generators

- Special functions that return iterator objects
 - `yield` within the function causes the function to become a generator
 - Invoking a generator does not begin executing it
 - Generators resume where they left off

`__next__()` is considered private, however, it's okay to call `next(gen)`

When invoked, this function will return a generator object

```
print(gen.__next__())
print(gen.__next__())
print(gen.__next__())
print(gen.__next__())
```

<class 'generator'>

```
def a_generator(min_val: int, max_val: int):
    current_val = min_val
    while current_val <= max_val:
        yield current_val
        current_val += 1

gen = a_generator(3, 5)
print(type(gen))

for val in gen:
    print(val)
```

3
4
5

What will this bottom statement do?

ch03_iterating/02_generator.py

Generators can retrieve resources "on-demand." They work like the iterator created on the previous slide. Generators have a `__next__()` magic method. How does the generator pause and resume? Internally when a function is called it creates what is called a stack frame. The stack frame is suspended (placed on "pause") and resumed when it picks up again from the `yield` statement. When the generator completes, the function has run to conclusion and the stack frame is discarded.

The return value type hint for a generator is (from the typing module): `Generator[yield_type, send_type, return_type]`.



Retrieving One Word At a Time

- Consider the following example
 - Suppose we want to create a function that reads from a file and allows us to iterate over the results *printing one word at a time*

```
def read_words(filename: str = 'data.txt'):
    total_words = []
    try:
        with open(filename, encoding='utf8') as f:
            for line in f:
                line_words = line.strip().split()
                total_words.extend(line_words)
    except IOError as err:
        print(err, file=sys.stderr)

    return total_words

for word in read_words():
    print(word)
```

What happens if
data.txt is a 30Gb file?

ch03_iterating/03_read_words.py

This example is straight forward presenting nothing we haven't learned already. It opens a file, reads all the words in it, line-by-line, and stores the results in a list. It returns the list and then uses it in a for loop to individually present the words.



One Word At a Time (Generator Version)

- **yield** makes this version a generator

```
from typing import Generator
def read_words_generator(filename= 'data.txt'):
    try:
        with open(filename, encoding='utf8') as f:
            for line in f:
                line_words = line.strip().split()
                for one_word in line_words:
                    yield one_word
    except IOError as e:
        print(e)

for word in read_words_generator():
    print(word)
```

yield returns *one_word* and pauses execution

The generator can be passed into a for-loop. It runs to the *yield* each time and returns *one_word*

ch03_iterating/03_read_words.py

The generator-based version works more like what the requirements called for. It actually returns a single word at a time instead of returning a list as in the previous example. Also, this file can be as large as we want as only one line at a time is needed within memory.

Annotations and Generators

Type hint syntax for a generator:
`Generator[yield_type, send_type, return_type]`

```
from typing import Generator
def read_words_generator(filename: str = 'data.txt'):
    -> Generator[str, None, None]:
        try:
            with open(filename, encoding='utf8') as f:
                for line in f:
                    line_words = line.strip().split()
                    for one_word in line_words:
                        yield one_word
        except IOError as e:
            print(e)

for word in read_words_generator():
    print(word)
```

ch03_iterating/03_read_words.py

Annotations for the generator work the same as a function except for the return type. The return type annotation for a generator indicates three values: the type that is yielded, the type that is sent into the generator (this is a special condition known as a coroutine), and the type that is returned at the end of the generator.

Dictionary Comprehensions

- Similar to list comprehensions, **dictionary comprehensions** create dictionaries

```
new_dict = {key:value for item in iterable if conditional}
```

```
data = {'January': 31, 'February': 28, 'March': 31, 'April': 30,
        'May': 31, 'June': 30, 'July': 31, 'August': 31,
        'September': 30, 'October': 31, 'November': 30,
        'December': 31
    }

day31_months = { k:data[k] for k in data if data[k] > 30 }
print(day31_months)
```

```
{'August': 31, 'July': 31, 'December': 31,
 'January': 31, 'May': 31, 'March': 31, 'October': 31}
```

ch03_iterating/04_comprehensions.py

A dictionary comprehension is like a list comprehension except that the resulting structure is a dictionary. It follows a similar syntax to the list comprehension (except it uses curly braces).



Set Comprehensions

- Set comprehensions create sets from iterables

```
new_set = { item for item in iterable if conditional }
```

```
days_set = {days for days in data.values()}

print(days_set)
```

```
{28, 30, 31}
```

ch03_iterating/04_comprehensions.py



Generator Expressions

- Generator expressions are like list comprehensions
 - List comprehensions generate the entire data structure *into memory immediately*
 - Generator expressions will generate values that are used on-demand (only as needed)

```
gen_exp = ( item for item in iterable if conditional )
```

```
# list comprehension
lc = [k for k in range(55) if k % 5 == 0]
# generator expression
ge = (k for k in range(55) if k % 5 == 0)

for val1, val2 in zip(ge, lc):
    print(val1, val2)
```

```
0 0
5 5
10 10
...
50 50
```

Both of these yield equal results, so what is the difference?

ch03_iterating/04_comprehensions.py

The most useful in terms of performance and memory is also the least well-known of these controls. The generator expression yields values back as needed. Therefore, the entire data doesn't need to be resident in memory all at once.

The list comprehension version creates everything in memory up front and then begins working on it. So, at the start of the for-loop, the lc variable resides entirely in memory already while ge hasn't begun doing anything yet!



Tools within *collections*

- **defaultdict()** - a dict that doesn't raise a KeyError when an invalid key is supplied

```
d = defaultdict(function)
```

- If an invalid key is supplied, it invokes the provided function and inserts the return value into the dictionary for that key

```
from collections import defaultdict  
  
d1 = defaultdict(str)  
d1['greet1'] = 'hello'  
print(d1['greet1'])  
print(d1['greet2'])
```

Works as expected

Key not present, invokes the str() constructor as a default

Normally d1['greet2'] will be a KeyError; but here, the default value from str() which is "" (empty string) will be used

ch03_iterating/05_defaultdict.py

In the example above, an invalid key is supplied (greet2). However, a KeyError isn't raised. Instead, the key is added, and an empty string is provided as the value.



The Counter Class

- The Counter class is a *dict* type that can assist in tracking occurrences

```
from collections import Counter

items = [1, 2, 3, 4, 5, 4, 4, 3, 4, 5, 2, 0, 7, 4, 5, 6]

top_most = Counter(items).most_common(1)
print(top_most)                                [(4, 5)]

top_two_most = Counter(items).most_common(2)
print(top_two_most)                            [(4, 5), (5, 3)]

top_three_most = Counter(items).most_common(3)
print(top_three_most)                         [(4, 5), (5, 3), (2, 2)]
```

Each returned "most" is a tuple of two values:
most, num_occurrences

Defines how many "mosts" to return.
In this case, the top three are returned

ch03_iterating/06_counter.py

Counter is very easy to use. The trick is understanding what gets returned. `most_common(int)` accepts an integer that identifies how many top items to return. `most_common(10)`, for example, returns the top 10 items. Each returned item is actually a tuple. So, `most_common(10)` returns 10 tuples. Within each tuple will be two values. The first value is the item that occurred, and the second value is how many times the value occurred.



itertools Iterators

- **itertools** provides numerous types of iterators
 - Iterators behave like generators, use them iteratively

accumulate()	-	accumulates values from an iterable
chain()	-	chains multiple iterables together returning another iterator
count(start, step)	-	increment values
cycle()	-	cycles elements, repeating when exhausted
repeat(obj, n)	-	repeats an object n times
product()	-	cartesian product
starmap()	-	similar to map, arguments can be packaged up
filterfalse()	-	iterate items only if condition is true
takewhile()	-	iterate as long as a condition is true, including items
dropwhile()	-	iterate as long as a condition is true, excluding items
zip_longest(iters, fillvalue)	-	iterates parallel items stopping after the longest
compress()	-	provide individual filters on each item, drop false filter values

There are additional iterators in the `itertools` module.

Generators are iterators. Iterators are merely objects that implement `__iter__` and `__next__` (like our example at the start of the chapter). This was an iterator. Generators are forms of iterators that are created with a `yield` statement within a function.

Examples From *itertools*

```
from itertools import chain, islice, count

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]
chained_list = chain(list1, list2, list3)
print(list(chained_list))
```

chain() returns an iterator, not a new list

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

```
with Path('../resources/cities15000_info.txt')
        .open(encoding='utf-8') as f:
    for line in islice(f, 7, None):
        print(line.strip())
            start stop
```

islice(iter, start, stop) starts reading the file after the header

```
iterable = count(start=10, step=10)
while (val := next(iterable)) <= 50:
    print(val)
```

10
20
30
40
50

Walrus operator (Py3.8+)

count(start, step) iterates indefinitely until told to stop (useful when an upper bound is not known)

ch03_iterating/07_itertools.py

The first example iterates sequentially over multiple iterables without having to merge them.

The second example uses islice() to return lines from a file starting at line seven. It reads the first seven lines but throws them out (skips them) and returns lines starting at line 7. None represents a stop value (None = go until finished).

The last example iterates until it is told to stop. There is no upper bound. The walrus operator performs an assignment in the middle of an on-going expression.



Your Turn! - Task 3-1

Iterating Techniques

- Revisit `cities15000.txt` from earlier
- Use `collections.Counter`
 - Determine the country with the most cities from this file
 - Read column 9 (index value 8) which represents the `country_code`
 - See `task3_1_starter.py` for more info
 - Determine the top ten countries with the most cities listed in this file?

Index 8 (column 9)
Count these for each country

`cities15000.txt`

Frankston South AU
Sunshine West AU
Altona Meadows AU
Hurstville AU
West Pennant AU
Oranjestad AW
Mariehamn AX
Xankandi AZ

Work from the `task3_1_starter.py` file within the `ch03_iterating` directory

Your Turn! - Task 3-2

Generators and Iterators

- Revisit `task3_1_starter.py`
- Refactor the previous exercise
 - Modify the code used to read from the file
 - Convert it to a **generator**
 - Do this by placing the code in a function and **yielding** back a single country code each time
 - **Advanced:** After getting the generator version working, attempt it using a **generator expression**

If this were a very large file, what advantage would this version have over Task 3-1?

argparse

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-g', '--greet',
                    help='A howdy do.', required=True)
args = parser.parse_args()

print(args.greet)
```

Argparse is a standard library module that makes it easy to work with command-line arguments

Run with no arguments

```
>python 08_argparse1.py
usage: 08_argparse1.py [-h] -g GREET
08_argparse1.py: error: the following arguments are required: -g/--greet
```

Run with -h argument

```
>python 08_argparse1.py -h
usage: 08_argparse1.py [-h] -g GREET

options:
  -h, --help            show this help message and exit
  -g GREET, --greet GREET
                        A howdy do.
```

Run with -g argument

```
>python 08_argparse1.py -g Hello
Hello
```

Run with --greet arg

```
>python 08_argparse1.py --greet Hello
Hello
```

ch03_iterating/08_argparse1.py

This module will parse a list of arguments (usually sys.argv command-line arguments when parser.parse_args() is passed no arguments. There are quite a number of options associated with using this module. These slides attempt only to provide the basic options. Please consult the documentation for additional configuration options.



Argparse Options

```
def get_args():
    parser = argparse.ArgumentParser(description='Finds occurrences within datafile.')
    parser.add_argument('-n', '--filename', required=True, help='The file to search')
    parser.add_argument('-c', '--columns', nargs='*', required=True,
                        help='The columns to use, ex: name=(1,str) pop=(14,int)')
    parser.add_argument('-s', '--sep', default=',', help='File separator character')
    parser.add_argument('-q', '--query', help='Query term to search for')
    parser.add_argument('-f', '--field', help='Column to search within')
    args = parser.parse_args()
    return args

args = get_args()
search.read_data(args.filename, args.columns, args.sep)
search.search(args.query, args.field)
```

```
c:\>python 09_argparse2.py -n ../resources/cities15000.txt -c name:(1,str) population:(14,int) -q "new york" -f name -s
name          population
West New York      53366
East New York     173198
New York City     8804190
```

ch03_iterating/09_argparse2.py

The required=True option forces the user to provide a value. help= allows a description of that field. nargs='*' allows the field to contain more than one value. default= parameter allows for specifying a default value.

Other options exist as well. For example, the action='store_true' parameter would allow a command-line switch to act as a Boolean value.



Your Turn! - Task 3-3

Argument Parsing

- Re-visit the country counting exercise (either Task 3-1 or Task 3-2)
 - Use the `argparse` module
 - Display the top countries based on a command-line option, as in:

```
python task3_3_starter.py -c 5
```

or

```
python task3_3_starter.py --count 5
```

if no count argument is supplied let the default be 3

Chapter 3 Summary (1 of 3)

- Let's review the techniques we've learned about iterating

```
for item in enumerate([5, 8, 9], 10):
    print(item[0], item[1])
```

```
10 5
11 8
12 9
```

```
for item in enumerate([5, 8, 9], 10):
    print('{}, {}'.format(*item))
```

```
(10, 5)
(11, 8)
(12, 9)
```

```
for (count, value) in enumerate([5, 8, 9], 10):
    print(count, value)
```

```
10 5
11 8
12 9
```

```
for count, value in enumerate([5, 8, 9], 10):
    print(count, value)
```

```
10 5
11 8
12 9
```

ch03_iterating/11_iterating_summary.py

Chapter 3 Summary (2 of 3)

```
for item in reversed([5, 8, 9]):  
    print(item)  
  
for item in enumerate(reversed([5, 8, 9])):  
    print(item)  
  
array1 = ['a', 'b', 'c']  
array2 = [30, 40, 50]  
for item1, item2 in zip(array1, array2):  
    print(item1, item2)  
  
from itertools import chain  
for item in chain(array1, array2):  
    print(item, end=' ')
```

9
8
5

(0, 9)
(1, 8)
(2, 5)

a 30
b 40
c 50

a b c 30 40 50

ch03_iterating/11_iterating_summary.py

Chapter 3 Summary (3 of 3)

```
def my_generator():
    values = [1, 2, 3]
    for i in values:
        yield i
```

You can check for membership:

```
if 1 in my_generator():
    print('it\'s there!')
```

4 Ways to Run a Generator:

- 1 `for i in my_generator():
 print(i)`
- 2 `results = list(my_generator())
print(results)`
- 3 `print(*my_generator())`
- 4 `g = my_generator()
print(g.__next__())
print(next(g))
print(next(g))`

Doesn't behave like a normal function call

You cannot:

```
print(my_generator())
print(my_generator()[0])
print(len(my_generator()))
```

len() and indexing are not supported

ch03_iterating/12_generator_summary.py



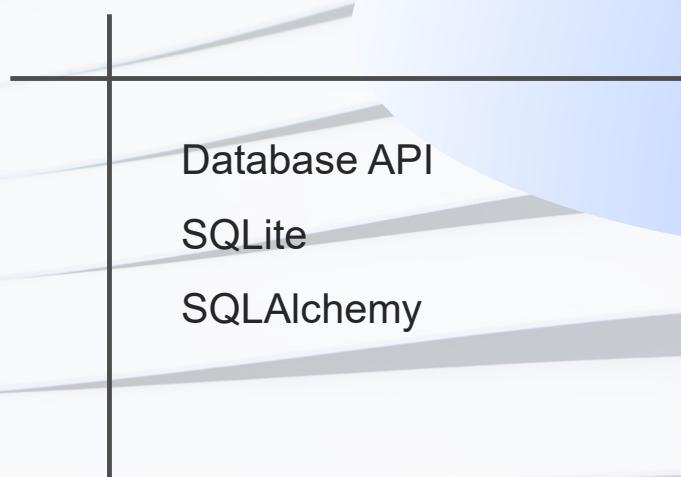
Chapter 4

Working with Relational Databases

Python Database API



Chapter 4 - Overview





Python Database API 2.0

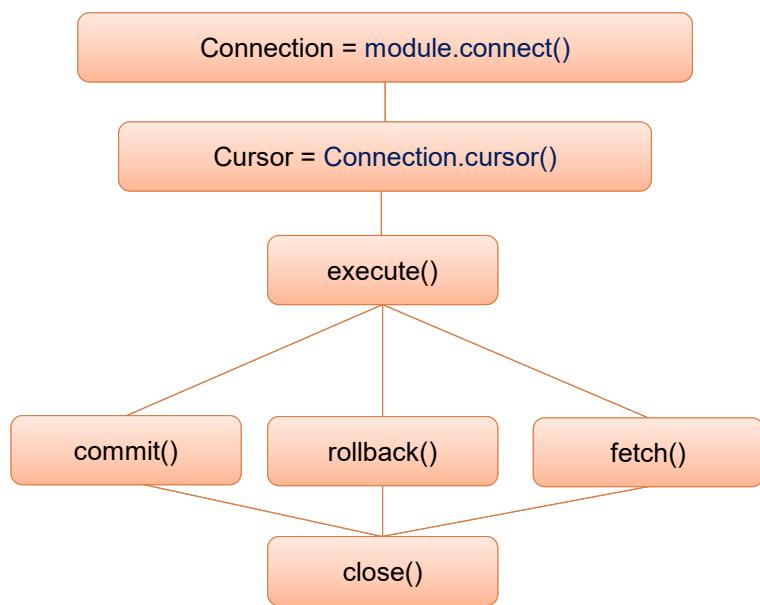
- The **Python DB API 2.0** (PEP 249) defines a common interface for modules to connect to and work with relational databases
- The interface defines:
 - Connection Objects and Transactions
 - Cursors Object Operations
 - Input, Output Data Types
 - Error Handling
 - Two-Phase Commits

Except for SQLite3, no standard library module exists for working with relational databases. Vendors or third-parties provide drivers for their databases

The Python Database API 2.0 is Python's version of Java's JDBC. It defines the interface for working with a driver that communicates with a specific database.

API Overview

- The following is a birds-eye view of a typical database interaction





Working with Various Databases

- Different databases require different modules to be installed

Database	Module	Install
MySQL	Connector Python Driver (Oracle official)	pip install mysql-connector-python
MS SQL Server	PyODBC Driver	pip install pyodbc
PostgreSQL	Psychopg2 Driver	pip install psycopg2
IBM DB2	IBM_DB Driver	pip install ibm_db
Oracle	cx_Oracle Driver python-oracledb Driver	pip install cx_Oracle (classic) pip install oracledb (newer)

- Modules will follow the Python DB standard

```
import cx_Oracle

conn = cx_Oracle.connect(user='scott', password='tiger',
                         dsn='182.100.212.18:1521/server')
cursor = conn.cursor()
```

Connecting to a relational database involves the steps mentioned in the birds-eye view on the previous slide. Usually, only the parameters to the connect() method will differ between databases.



Connections

- Each DB module features a `connect()` method that returns a **Connection** object
 - Parameters supplied will be database-specific

```
import mysql.connector

conn = mysql.connector.connect(
    host='localhost',
    user='c_eastwood',
    password='few$More'
)
cursor = conn.cursor()
```

To work with a database,
the driver module must be installed:

`pip install mysql-connector-python`

- Connection object methods:
 - `cursor()`- Obtains a cursor object
 - `commit()/rollback()`
 - `close()` - closes connection

While additional methods
may be defined, the
methods shown here are
defined in the spec

For MySQL, the main driver used is called the MySQL connector python driver. This is the official Oracle supported driver for MySQL.



Cursor Methods

- **Cursors** are used for fetch and execution operations
 - Once execute() is called, the cursor will be at the start of the result set
 - Cursors methods and attributes include

```
description
rowcount
execute(sql, [params])
executemany(sql, [seq_of_params])
callproc(proc_name, params)
fetchone()
fetchmany(size=cursor.arraysize)
fetchall()
close()
```

Atomicity - all parts of a transaction succeed or fail together

Consistency - the database will always be consistent

Isolation - no transactions can interfere with another

Durability - once the transaction is committed, it won't be lost



Working with SQLite

- **SQLite** is an in-process relational database embedded into Python
 - To use it, simply `import sqlite3`
 - Like other RDBMS, it follows the Python DB API
- Connect using `sqlite3.connect()`

```
import sqlite3

conn = sqlite3.connect('course_data.db')
cursor = conn.cursor()
cursor.execute('select fullname, city, state from schools \
                where state="CO"')
results = cursor.fetchmany(size=3)
print(results)
```

```
[('Fort Lewis College', 'Durango', 'CO'), ('Lamar Community College', 'Lamar',
'CO'), ('Colorado College', 'Colorado Springs', 'CO')]
```

ch04_database/01_quick_fetch.py

For more on SQLite, visit <https://www.sqlite.org>.



Cursor execute() Methods

- **execute(sql, params)** – executes the specified SQL

```
data = ('Bob', 100.0, 0.05, 'C')
cursor.execute(
    "INSERT INTO accounts(name,balance,rate,acct_type) VALUES \
    (?, ?, ?, ?)", data)
```

- **executemany(sql, [params])** – execute SQL repeatedly against all sequence params

```
data = [(100, 'John Smith', 5500.0, 0.025, 'C'),
        (101, 'Sally Jones', 6710.11, 0.025, 'C'),
        (102, 'Fred Green', 2201.73, 0.035, 'S'),
        (103, 'Ollie Engle', 187.30, 0.025, 'S'),
        (104, 'Gomer Pyle', 12723.10, 0.015, 'C')]

cursor.executemany("INSERT INTO accounts \
(id, name, balance, rate, acct_type) VALUES (?,?,?,?,?)", data)
```

ch04_database/02_execute.py

For the executemany() method, PEP249 leaves it up to the database vendors as to whether it will perform individual calls to execute() or whether it will perform a batch operation instead.

Fetching Data

- `fetchone()`, `fetchmany` and the `cursor` itself retrieve records after an `execute()` query

```

import sqlite3
import sys

state = 'CO'
try:
    conn = sqlite3.connect('course_data.db')
    cursor = conn.cursor()
    cursor.execute('SELECT fullname, city, state FROM schools \
                   WHERE state=?', (state,))

    print(cursor.fetchone())
    print(cursor.fetchmany(size=3))

    for sch in cursor:
        print(f'Name: {sch[0]}, ({sch[1]}), {sch[2]})')

except sqlite3.Error as err:
    print(f'Error: {err}', file=sys.stderr)

```

This query retrieves 11 records

First tuple returned

List of three tuples returned

Remaining 7 records returned Access columns by position

ch04_database/03_fetching_rows.py

This solution obtains records from the database. Once the cursor is executed, rows can be retrieved using fetch methods. `fetchone()` returns the first record in the record set. `fetchmany(size=3)` returns the next three records as a list of tuples (since we are re-using the same cursor it continues where we left off). Finally, the for-loop over the cursor returns the remaining seven records.



Optimizing `cursor.execute()`

- The cursor is an iterator to the result set
 - No records are actively obtained until a `fetchXXX()` method is invoked, or
 - The cursor is iterated over

```
conn = None
page_size = 10
try:
    conn = sqlite3.connect('course_data.db')
    cursor = conn.cursor()
    cursor.execute('SELECT fullname, city, state FROM schools')
    records = cursor.fetchmany(size=page_size)
    while records:
        print(f'{len(records)} records processed.')
        records = cursor.fetchmany(size=page_size)
finally:
    if conn:
        conn.close()
```

Use `fetchmany()` to process `page_size` records at a time

ch04_database/03_fetching_rows.py

Fetching multiple rows at a time can be faster than working with individual records. In the example above, 10 records at a time are fetched and processed before another set of 10 are retrieved.



Closing Connections

- Connections should be closed when done
 - A *finally* block can accomplish this

```
conn = None
page_size = 10
try:
    conn = sqlite3.connect('course_data.db')
    cursor = conn.cursor()
    cursor.execute('SELECT fullname, city, state FROM schools')
    records = cursor.fetchmany(size=page_size)
    while records:
        print(f'{len(records)} records processed.')
        records = cursor.fetchmany(size=page_size)
finally:
    if conn:
        conn.close()
```

ch04_database/03_fetching_rows.py

To ensure a connection is closed, we can place the close() call in a finally block. This way, whether there is an exception or not, the connection will be closed.

Using the *with* Control with the Database

- You might expect the *with* control to close database connections automatically
 - However, *with* behaves differently for different databases
 - The *with* behavior is not defined in the Python DB API specification

```
import mysql.connector
with mysql.connector.connect(host='localhost',
                             user='c_eastwood',password='few$More') as conn:
    cursor = conn.cursor()
    ...
    
```

The mysql.connector driver uses the *with* control to **close** the connection

Module	How <i>with</i> behaves	
cx_Oracle	Closes the connection	
ibm_db	Not supported	
pyodbc	Commit/Rollback	<i>With</i> behaves differently depending on the database
psycopg2	Commit/Rollback	

Different database modules will behave differently when using the *with* control. MySQL will close the connection at the end of the *with*.

When using the *sqlite3* module, *with* is used to automatically commit or rollback the transaction depending on whether an error occurs internally or not. Worth noting however, the connection is not closed by the *with* control.



Using *with* in SQLite

- In SQLite, *with* will perform commit and rollbacks

```
data = (110, 'Amber Willis', 3200.0, 0.025, 'C')

try:
    with sqlite3.connect('course_data.db') as conn:
        cursor = conn.cursor()
        cursor.execute('INSERT INTO accounts(id, name, balance, \
                        rate, acct_type) VALUES (?,?,?,?,?)', data)
        print('Record inserted into accounts')
except sqlite3.Error as err:
    print('Record not inserted into accounts', file=sys.stderr)
    print(f'Error: {err}', file=sys.stderr)
finally:
    if conn:
        conn.close()
```

The *with* will *commit()* if no errors occur at the end, otherwise it will *rollback()* if an error occurs

The connection is not automatically closed by the *with* control using `sqlite3`

ch04_database/04_inserting_with.py

Different database modules will behave differently when using the *with* control. MySQL will close the connection at the end of the *with*.

When using the `sqlite3` module, *with* automatically commits or rolls back the transaction depending on whether an error occurs internally or not. Worth noting however, the connection is not closed by the *with* control.



closing() to Close Connections

- The `contextlib` module provides utilities for using the `with` control

`closing()` adapts any object that has a `close()` method to work in a `with` control

```
from contextlib import closing

data = (110, 'Amber Willis', 3200.0, 0.025, 'C')

with closing(sqlite3.connect('course_data.db')) as conn:
    try:
        cursor = conn.cursor()
        cursor.execute('INSERT INTO accounts(id, name, balance, \
                        rate, acct_type) VALUES (?,?,?,?,?)', data)
        conn.commit()
    except sqlite3.Error as err:
        conn.rollback()
        print(f'Error: {err}', file=sys.stderr)
```

Wrapping `closing()` around the connection will automatically invoke its `close()` method *but now it won't automatically commit/rollback*

ch04_database/05_with_closing.py

In this version, the connection object will be closed at the end of the `with`. `closing()` is a utility that can work with any object as long as it has a `close()` method.

Accessing Data Using Column Names

```
from contextlib import closing
import sqlite3

state = 'CO'

with closing(sqlite3.connect('course_data.db')) as conn:
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()

    cursor.execute('SELECT fullname, city, state FROM schools \
                    WHERE state=?', (state,))
    for sch in cursor:
        print(f'{sch["fullname"]} ({sch["city"]}), {sch["state"]}')
```

sqlite3.Row is a class that allows accessing fields by column name

ch04_database/06_fetching_rows_column_names.py

This version uses a Row class instead of a tuple for the returned records. It allows fields to be accessed using column names instead of positions. This is not standard in the Python DB API so use it at your own discretion.



Your Turn! - Task 4-1

- Query the `schools` table within `course_data.db`
 - A function called `get_location(school_name)` queries the database returning the `name`, `city`, and `state` of schools that match or partially match the `school_name` provided

Sample Output:

```
School name (or partial name): loy
Matches for loy:
Loyola University Chicago (Chicago, IL))
Loyola Marymount University (Los Angeles, CA))
Loyola College in Maryland (Baltimore, MD))
Loyola University New Orleans (New Orleans, LA))
```

- A starter file, called `task4_1_starter.py` in the `ch04_database` folder has been provided for you
- Complete the provided `get_location()` function*
- Retrieve results into a [list of data classes](#)

Note: The database should already exist, however, if it doesn't, run `08_create_schools.py` to repair it.



ORM with SQLAlchemy

- Object-relational mapping is the act of converting Python objects to database relations
 - Several Python tools support ORM capabilities
 - The most popular tool in Python is called `sqlalchemy`
- Steps to working with SQLAlchemy
 - 1) *Initialize* SQLAlchemy providing connection info
 - 2) Define how *models (classes) map* to the database
 - 3) Perform database work within a *session*

```
pip install sqlalchemy
```

ORM tools are popular for mapping between Python objects and databases. SQLAlchemy can do this with little effort typically without the use of any manually created SQL.



SQLAlchemy Basics

- Step 1. Initialize SQLAlchemy using
`create_engine(dburl)`

```
from sqlalchemy import create_engine, Column
from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy.types import String

db = create_engine('sqlite:///course_data.db', echo=True)
```

Prepares to connect to the db. No actual connection occurs at this point

Other databases will have slightly different connection URLs

```
create_engine('postgresql://user:pswd@localhost:1542/schools')
```

ch04_database/07_schools_sqlalchemy.py

The `create_engine()` call doesn't actually connect to the database. It does, however, prepare a pool of connections and establish the dialect to be used.

Different databases will feature different connection URLs passed to `create_engine()`. For more on the connection strings of supported databases, visit: <https://docs.sqlalchemy.org/en/20/core/engines.html>



SQLAlchemy Class Mappings

- Step 2. Define table-to-class mappings

```

from sqlalchemy import create_engine, Column
from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy.types import String

Base = declarative_base()

class School(Base):
    __tablename__ = 'schools'
    school_id = Column(String(30), primary_key=True)
    fullname = Column(String(50))
    city = Column(String(50))
    state = Column(String(15))
    country = Column(String(50))

db = create_engine('sqlite:///course_data.db', echo=True)

```

Dynamically creates the base class for your classes to inherit from

Table name in database

This class defines what attributes of the School class map to what columns of the schools table

ch04_database/07_schools_sqlalchemy.py

SQLAlchemy allows for sophisticated mappings between your classes and the database relations. You can map one-to-one, many-to-many, one-to-many, and many-to-one relationships within your classes.

Use Column() definitions to map to attributes within your class. Other column types may be specified, such as Integer, Boolean, and Numeric.

SQLAlchemy Sessions

- Step 3. Use a **Session** to perform work
 - A Session marks transactional boundaries
 - sessionmaker** uses the engine to configure the Session

```
Session = sessionmaker(bind=db)

with Session() as session:
    firstSchool = session.query(School).first()
    print(firstSchool.fullname, firstSchool.country)
        Abilene Christian University USA
    firstSchool.country = 'U.S.'
    session.commit()

with Session() as session:
    school = session.query(School).first()
    print(school.fullname, school.country)
        Abilene Christian University U.S.
```

Begin a transaction

Obtain the *first* school from the schools table

Modify the *country* attribute

Commit the transaction.
Performs a SQL UPDATE

Queries the database again and
verifies the changes made

ch04_database/07_schools_sqlalchemy.py

The Session is created dynamically by the `sessionmaker()`. There are dozens of Session methods and attributes including:

- `add(obj)` - adds a new object to the session
- `delete(obj)` - deletes an object managed in the session
- `query()` - the main method for performing queries against the database
- `commit()/rollback()` - save or discard the changes made
- `get(class, id)` - retrieve an object by primary key

A Complete SQLAlchemy Interaction

```
from sqlalchemy import create_engine, Column
from sqlalchemy.orm import sessionmaker, declarative_base
from sqlalchemy.types import String

Base = declarative_base()

class School(Base):
    __tablename__ = 'schools'
    school_id = Column(String(30), primary_key=True)
    fullname = Column(String(50))
    city = Column(String(50))
    state = Column(String(15))
    country = Column(String(50))

db = create_engine('sqlite:///course_data.db', echo=True)
Session = sessionmaker(bind=db)

with Session.begin() as session:
    firstSchool = session.query(School).first()
    print(firstSchool.fullname, firstSchool.country)
    firstSchool.country = 'U.S.'
```

ch04_database/07_schools_sqlalchemy.py

A complete picture of the SQLAlchemy schools table database interaction is shown here.



Chapter 4 Summary

- Python doesn't have a single module when working with a database
 - A different module is required for each database
- The [Python DB API](#) defines basic operations to perform on the database
 - Vendors are free to provide additional operations to enhance interactions with their database
- [SQLAlchemy](#) is a popular tool for mapping relations to objects within Python



Chapter 5

More with Classes and Objects

Supported OOP Features within Python



Chapter 5 - Overview

Classes Review
Static Methods
Inheritance
Abstract Classes

Review of Classes (1 of 2)

```

from typing import NamedTuple

City = NamedTuple('City', [('name', str), ('population', int),
                           ('elevation', int), ('country', str)])


class CitySearch:
    def __init__(self, filepath: str):
        self.cities = []
        self.filepath = filepath
        self._read_data()

    def _read_data(self):
        with open(self.filepath, encoding='utf-8') as f:
            for line in f:
                fields = line.strip().split('\t')
                name = fields[1]
                country = fields[8]
                population = int(fields[14])
                elevation = int(fields[16])
                city = City(name, population, elevation, country)
                self.cities.append(city)

```

self must always be defined as the first argument in the constructor

ch05_oo/01_review.py

This example illustrates a Python class. It contains several methods, including `__init__()` and `_read_data()`. The class is used to "instantiate" an object. The instantiation occurs on the next slide (part 2 of 2). This part of the class reads the data from the file when the `__init__()` invokes the `_read_data()` method.

The constructor must always define a variable called `self`. `self` represents the current object being constructed.



Review of Classes (2 of 2)

```
... continued from previous slide...

def largest(self) -> City:
    return max(self.cities, key=lambda city: city.population)

def highest(self) -> City:
    return max(self.cities, key=lambda city: city.elevation)

def search(self, name: str) -> list[City]:
    results = []
    for item in self.cities:
        if name.lower() in item.name.lower():
            results.append(item)
    return results

c = CitySearch('../resources/cities15000.txt')
print(c.search('los angeles'))
```

[City(name='East Los Angeles', population=126496, elevation=63, country='US'),
 City(name='Los Angeles', population=3971883, elevation=96, country='US')]

ch05_oo/01_review.py

In this example, object 'c' is being created. In the `__init__()`, the 'c' object is what `self` is referring to. While `self` can be named anything else, in Python, we never do this. We only call it `self`.

Note that in other languages you might be tempted to use the `new` operator. But Python doesn't define a `new` operator. So, this statement:

`c = new City()` is `c = City()` in Python



Instances and Self

```
class CitySearch:  
    def __init__(self, filepath: str):  
        self.cities = []  
        self.filepath = filepath  
        self._read_data()  
  
    def _read_data(self):  
        ...  
  
    def largest(self) -> City:  
        return max(self.cities, key=lambda city: city.population)  
  
    def highest(self) -> City:  
        return max(self.cities, key=lambda city: city.elevation)  
  
    def search(self, name: str) -> list[City]:  
        ...  
  
c = CitySearch('../resources/cities15000.txt')  
print(c.highest().name)
```

self, should always
be the first parameter

Instances may invoke the methods defined in the
class, but shouldn't specify self in the call

ch05_oo/01_review.py

Methods added to a class should always specify the self argument first. When calling a method, you DO NOT specify self.

When a method is invoked, such as c.highest(), behind the scenes, Python will translate the call into CitySearch.highest(c).



Instances are Backed by Dictionaries

- Object instances are usually backed by dictionaries

```
class City:
    def __init__(self, name, country='', population=0, elevation=-1):
        self.name = name
        self.country = country
        self.population = population
        self.elevation = elevation

    def __str__(self):
        return self.name

data = linecache.getline('../resources/cities15000.txt',
                        22237).strip().split('\t')
c = City(data[1], data[8], data[14], data[15])

print(c.__dict__['name'])
print(vars(c)['name'])
```

New York City
New York City

ch05_oo/02_as_dicts.py

Most objects are backed by dictionaries under the hood. This means that there is a dictionary underlying that instance's implementation. Certain tools, such as the json module's dumps() method will only work properly if that object has a dictionary backing it.

The __dict__ property allows object instances to behave like dictionaries when properties are referenced as strings through the dictionary.

Defining Properties

```

class City:
    def __init__(self, name, country='', population=0, elevation=-1):
        self.name = name
        self.country = country
        self.population = population
        self.elevation = elevation

    @property
    def population(self):
        return self._population

    @population.setter
    def population(self, pop):
        self._population = 0
        if pop > 0:
            self._population = pop

    @property
    def elevation(self):
        return self._elevation

    @elevation.setter
    def elevation(self, elev):
        self._elevation = 0
        if -400 < elev < 30000:
            self._elevation = elev

```

→ Setters invoked

Use `@property` to define the "getter"

`@prop.setter` is the syntax to define a setter

The "private" property is set in the setter

Properties represent Python's version of getters and setters

ch05_oo/03_properties.py

Notice that the setter methods are invoked within the `__init__()`. So, if a `City` object is created and an invalid elevation or population is passed into it, the validation within the setter will be invoked. Properties provide an ability perform setter, getter, and deleter interactions.

Using Properties

- Use the properties as if they were typical attributes
 - Performing assignments will execute the setter methods

```
c = City('New York City', 'USA', 8175133)
c.elevation = 10
print(c.population)

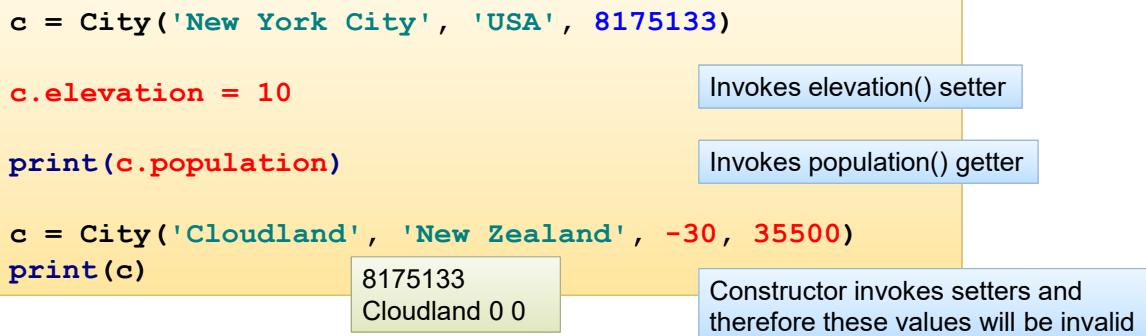
c = City('Cloudland', 'New Zealand', -30, 35500)
print(c)
```

8175133
Cloudland 0 0

Invokes elevation() setter

Invokes population() getter

Constructor invokes setters and therefore these values will be invalid



The property defined above is invoked when an assignment (e.g., `c.elevation = 10`) takes place. The assignment invokes the setter property defined in the class.

Class Attributes

- Variables created at the class level are called **class attributes**
 - They should be modified via the class

```
class RaceCar:  
    MAX_SPEED = 245  
  
    def __init__(self, speed):  
        self.current_speed = speed  
  
    def speed_up(self, amt):  
        self.current_speed += amt  
        if self.current_speed > self.MAX_SPEED:  
            self.current_speed = RaceCar.MAX_SPEED  
  
car1 = RaceCar(200)  
car2 = RaceCar(230)  
car1.speed_up(30)  
car2.speed_up(50)  
print(car1.current_speed, car2.current_speed,  
      car1.MAX_SPEED, car2.MAX_SPEED)
```

Class attributes can be accessed by both the class AND the instance

Do not modify MAX_SPEED through the instance, as in self.MAX_SPEED = 255

230 245 245 245

ch05_oo/04_attributes.py

As the note in the lower right on the slide indicates, you should be sure not to modify the class attribute through the instance. This will cause the instance add an acceleration attribute to itself while the class-level attribute will still exist separately, unchanged.

Static Methods

- Python defines static methods using the `@staticmethod` decorator

```
import urllib.request

class Page:
    @staticmethod
    def page_load(url):
        with urllib.request.urlopen(url) as f:
            results = f.read()

        return results

webpage = 'https://www.google.com'
print(Page.page_load(webpage))
```

What does this `@staticmethod` decorator do?

It allows the instance to call the method without having to declare a `self` parameter:

```
p = Page()
p.page_load(url)
```

Do not specify `self` when creating a static method. Call is accomplished directly via `Class.methodname()` or `instance.methodname()`

ch05_oo/05_static.py

Decorators are discussed in a later chapter. In order to use a static method (via an instance), we use the `@staticmethod` decorator.

Note: `@classmethod` also exists within Python, however, this is less commonly encountered.

Inheritance: Classic `__init__()` Call

```

class Contact:
    def __init__(self, name='', address='', phones=None):
        self.name = name
        self.address = address
        self.phones = phones

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

class BusinessContact(Contact):
    def __init__(self, name='', address='', phones=None,
                 email='', company='', position=''):
        Contact.__init__(self, name, address, phones)
        self.email = email
        self.company = company
        self.position = position

bc = BusinessContact('John Smith', '123 Main St.',
                     {'work': '(970) 322-9088', 'home': '(970) 455-2390'})
print(bc)

```

Call to object's base class constructor is required if initialization of base class attributes is needed

Calls the parent constructor

John Smith 123 Main St. {'work': '(970) 322-9088', 'home': '(970) 455-2390'}

ch05_oo/06_inheritance_classic.py

To ensure that instance attributes are properly set, your derived (child) class should call the base class constructor. There are two ways to do this. The classic approach, shown here, calls the parent constructors directly. This approach has the disadvantage of referencing the name of the parent class within the subclass.

Inheritance: Using *super()* - Preferred

```

class Contact:
    def __init__(self, name='', address='', phones=None):
        self.name = name
        self.address = address
        self.phones = phones

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

class BusinessContact(Contact):
    def __init__(self, name='', address='', phones=None,
                 email='', company='', position=''):
        super().__init__(name, address, phones)    super() invokes the parent __init__()
        self.email = email
        self.company = company
        self.position = position

bc = BusinessContact('John Smith', '123 Main St.',
                     {'work': '(970) 322-9088', 'home': '(970) 455-2390'})
print(bc)

```

John Smith 123 Main St. {'work': '(970) 322-9088', 'home': '(970) 455-2390'}

ch05_oo/07_inheritance_super.py

super() requires no arguments in Python 3, nor does the *self* object need to be passed when using it.



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
 - The child class should implement the abstract methods of the base class

What do I put in the abstract method?

- 'pass'
- raise NotImplementedError()
- Code - but it's still declared as abstract

```
import abc

class AbstractBase(abc.ABC):

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

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

ch05_oo/11_abstract_classes.py

Use the ABC class of the abc module and the @abc.abstractmethod decorator to define an abstract class. Methods marked as abstract must be implemented within child classes or the child class will not be instantiable.

There are options as to what to place into the body of the abstract method. All the options mentioned above are used. While the third option does not make the method concrete, the child class can still call the parent's abstract method if desired.

Using Abstract Classes

```
class TextMessage(abc.ABC):
    def __init__(self, msg):
        self.msg = msg
    @abc.abstractmethod
    def get_text(self):
        pass
```

Subclasses must define this method

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

    def get_text(self):
        return self.msg
```

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

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

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

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

```
def send_message(url: str, msg: Message) -> None:
    requests.post(url, data={'text': msg.get_text()})

value = 'The meeting today is at 3pm.'

send_message('https://www.reddit.com/r/test/submit',
             PlaintextMessage(value))

send_message('https://www.reddit.com/r/test/submit',
             Message(value))
```

TypeError: Can't instantiate abstract class Message without an implementation...

ch05_oo/11_abstract_classes.py

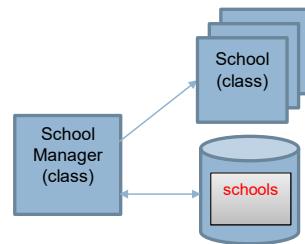
Note: the call into Reddit is for illustration only and actually returns a 405. In this example, the Message class has been made abstract by inheriting from abc.ABC and having at least one method marked as @abc.abstractmethod (or else it won't be abstract). In other words, if the Message class didn't mark the get_text() method as abstract, it would be able to be instantiated even if it still inherits from abc.ABC.

How can you prevent instantiation of a class without having any abstract methods? Mark the __init__() as abstract.



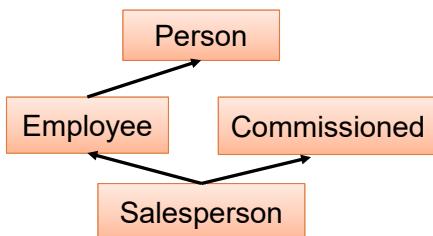
Your Turn! - Task 5-1

- Work from [ch05_oo/task5_1_starter.py](#) to create a *School* class containing `school_id`, `fullname`, `city`, and `state`, and `country` attributes
- Create a second class, *SchoolManager*, containing `__init__()`, and `find()` methods
 - The **SchoolManager find()** should search the `fullname`, `city`, `state`, or `country` columns
 - Test the SchoolManager as follows:



```
if __name__ == '__main__':
    print(SchoolManager('course_data.db').find('Loyola',
                                                column='fullname'))
    print(SchoolManager('course_data.db').find('ID',
                                                column='state', sort_by='fullname'))
```

Multiple Inheritance: Explicit `__init__()` Calls (1 of 2)



Python classes support having multiple parents

```

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
  
```

```

class Employee(Person):
    def __init__(self, name, age,
                 salary, dept):
        Person.__init__(self, name, age)
        self.salary = salary
        self.dept = dept
  
```

```

class Commissioned:
    def __init__(self, rate):
        self.comm_rate = rate

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

ch05_oo/08_multiple_inherit_explicit.py

To illustrate multiple inheritance in Python, consider the following relationship. Salesperson inherits from Employee and Commissioned. In order to properly call all the base class constructors directly, you can use the explicit approach (shown on this slide), or you may use the approach using super(). The explicit technique is preferred over the use of super() for multiple inheritance.

Multiple Inheritance: Explicit `__init__()` Calls (2 of 2)

```
class Salesperson(Employee, Commissioned):
    def __init__(self, name, age, salary, region,
                 dept='Sales', rate=0.02):
        Employee.__init__(self, name, age, salary, dept)
        Commissioned.__init__(self, rate)
        self.region = region

    def __str__(self):
        emp_str = Employee.__str__(self)
        comm_str = Commissioned.__str__(self)
        return f'{emp_str}, {comm_str} - Reg: {self.region}'
```

s = Salesperson('William', 37, 99275.00, 'Northeast')
print(s)

Specify both base classes

Use the classic (explicit) technique
to call both constructors

William (37) Sal: 99275.0 Dept: Sales, Rate: 0.02 - Reg: Northeast

ch05_oo/08_multiple_inherit_explicit.py

To illustrate multiple inheritance in Python, consider the following relationship. Salesperson inherits from Employee and Commissioned. In order to properly call all the base class constructors directly, you can use the explicit approach (shown on this slide), or you may use the approach using super(). The explicit technique is preferred over the use of super() for multiple inheritance.

Multiple Inheritance: Using *super()*

```
class Employee(Person):
    def __init__(self, name, age, salary, dept, **kwargs):
        super().__init__(name, age, **kwargs)
        self.salary = salary
        self.dept = dept
```

`**kwargs` captures the extra parameters sent to the other constructor

```
class Commissioned:
    def __init__(self, rate, **kwargs):
        super().__init__()
        self.comm_rate = rate
```

```
class Salesperson(Employee, Commissioned):
    def __init__(self, name, age, salary, region,
                 dept='Sales', rate=0.02):
        super().__init__(name=name, age=age, salary=salary,
                         dept=dept, rate=rate)
        self.region = region
```

```
s = Salesperson('William', 37, 99275.00, 'Northeast')
```

Only one `super()` call can exist, so both `__init__()` calls to the parents must be made at once

ch05_oo/09_multiple_inherit_super.py

For brevity, the complete class definitions are not shown.

We only get one call using `super()` in the `Salesperson __init__()`. Therefore, we send all arguments to both parents at once. The arguments that belong to the other parent will be placed into `kwargs` and not used. Therefore, `kwargs` must be provided in both parent `__init__()` declarations. This is not practical as it may not be possible to modify the parent classes. Therefore, use the explicit approach, previously discussed, for multiple inheritance.

Other Magic Methods

- Python classes support many magic methods

<code>__str__(self)</code>	<code>__ge__(self, other)</code>	<code>__iter__(self)</code>
<code>__abs__(self)</code>	<code>__get__(self, instance, owner)</code>	<code>__itruediv__(self, other)</code>
<code>__add__(self, other)</code>	<code>__getattr__(self, item)</code>	<code>__ixor__(self, other)</code>
<code>__and__(self, other)</code>	<code>__getattribute__(self, item)</code>	<code>__le__(self, other)</code>
<code>__bool__(self)</code>	<code>__getitem__(self, item)</code>	<code>__len__(self)</code>
<code>__bytes__(self)</code>	<code>__gt__(self, other)</code>	<code>__long__(self)</code>
<code>__call__(self, *args, **kwargs)</code>	<code>__hash__(self)</code>	<code>__lshift__(self, other)</code>
<code>__cmp__(self, other)</code>	<code>__hex__(self)</code>	<code>__lt__(self, other)</code>
<code>__coerce__(self, other)</code>	<code>__iadd__(self, other)</code>	<code>__mod__(self, other)</code>
<code>__complex__(self)</code>	<code>__iand__(self, other)</code>	<code>__mul__(self, other)</code>
<code>__contains__(self, item)</code>	<code>__idiv__(self, other)</code>	<code>__ne__(self, other)</code>
<code>__del__(self)</code>	<code>__ifloordiv__(self, other)</code>	<code>__neg__(self)</code>
<code>__delattr__(self, item)</code>	<code>__llshift__(self, other)</code>	<code>__new__(cls, *args, **kwargs)</code>
<code>__delete__(self, instance)</code>	<code>__imod__(self, other)</code>	<code>__oct__(self)</code>
<code>__delitem__(self, key)</code>	<code>__imul__(self, other)</code>	<code>__or__(self, other)</code>
<code>__delslice__(self, i, j)</code>	<code>__index__(self)</code>	<code>__pos__(self)</code>
<code>__divmod__(self, other)</code>	<code>__init__(self)</code>	<code>__pow__(self, power, module)</code>
<code>__enter__(self)</code>	<code>__int__(self)</code>	<code>__rand__(self, other)</code>
<code>__eq__(self, other)</code>	<code>__invert__(self)</code>	<code>__rdiv__(self, other)</code>
<code>__exit__(self, exc_type, exc_val, exc_tb)</code>	<code>__ipow__(self, other)</code>	<code>__rdivmod__(self, other)</code>
<code>__floordiv__(self, other)</code>	<code>__irshift__(self, other)</code>	<code>__reduce__(self)</code>
<code>__format__(self, format_spec)</code>	<code>__isub__(self, other)</code>	<code>__reduce_ex__(self, *a, **kw)</code>

Magic methods are so-named because they appear to get called "magically." In reality, they are called under very specific conditions. Most magic methods are never implemented.

Implementing Magic Methods

```
class Contact:  
    def __init__(self, name='', address='', phone=''):   
        self.name = name  
        self.address = address  
        self.phone = phone  
  
    def __eq__(self, second):  
        if type(self) is not type(second):  
            return False  
        elif self.name == second.name and  
                self.address == second.address:  
            return True  
        else:  
            return False  
  
c1 = Contact('John Smith', '123 Main St.', '(970) 322-9088')  
c2 = Contact('John Smith', '123 Main St.', '(970) 421-8032')  
c3 = Contact('John Smith', '321 Main St.', '(970) 421-8032')  
  
print(c1 == c2)      True  
print(c2 == c3)      False
```

Magic methods reference:
<https://rszalski.github.io/magicmethods/>

ch05_oo/10_magic_methods_again.py

Magic methods are each implemented differently. This example shows how to implement the `__eq__` method which allows instances to be directly compared to each other. Other magic methods will serve very different purposes.



Chapter 5 Summary

- Classes in Python have unique behaviors that make them different from classes found in other languages
 - Classes behave as a kind of **namespace** (code holders)
 - **self** is not implicit, but must be declared within methods
 - There are **no public or private** keywords
 - **Properties** can be defined to provide a version of setter/getter capability
 - Classes **support multiple inheritance**
 - Dozens of **magic methods** provide tailored behavior to classes and objects



Chapter 6

Python API Development



Clients and Servers: Working with Flask

A large, abstract graphic of blue smoke or a flame, swirling across the upper half of the slide, serves as a decorative background for the chapter title.



Chapter 6 - Overview

WSGI
Introducing Flask
RESTful API Development
SQLAlchemy / Marshmallow



Web Server Gateway Interface (WSGI)

- WSGI is a specification that defines the interaction between web server and applications
 - Provides a means for vendors to create servers using a consistent API
- WSGI compliant tools implement a "server" and "application" interface
 - To get a feel for what a WSGI server is, you can use the `wsgiref` module from the standard library...



WSGI (Example)

```
from wsgiref.simple_server import make_server
hostname = 'localhost'
port = 8051

def application(environ, start_response):
    response_body = '<html><body><h1>Request received.</h1></body></html>'
    status = '200 OK'
    headers = [
        ('Content-Type', 'text/html'),
        ('Content-Length', str(len(response_body)))
    ]
    start_response(status, headers)
    return [response_body.encode()]

httpd = make_server(hostname, port, application)
print(f'Server running on http://{hostname}:{port}...')

httpd.handle_request()           # serves one request
# httpd.serve_forever()         # serves continuously
```

ch06_api/01_wsgi.py

This example illustrates the structure (format) of the WSGI specification. Typically, normal users will not encounter this API. Instead, server vendors, or those desiring to create a framework for building and using web applications, will incorporate this spec.



Top Python Web/API Frameworks

- There are numerous web and API frameworks in Python
- Most popular among these include

Framework	Advantages	Disadvantages
Flask	Ease-of-use, 1000s of plugins, widely known/ used, short learning curve <i>(Pinterest, Twilio, Netflix)</i>	Needs to be deployed into a performance-worthy container (e.g., Gunicorn, waitress, etc.)
Django	All-in-one solution, Easy to use Angular/React, <i>(Instagram, many others)</i>	High learning curve, fewer plugins
Fast API	Fast, Built for serving high-performance APIs, ease-of-use, support for Pydantic	Newer (2018), few plugins, best for straightforward implementations

For more frameworks:
<http://wiki.python.org/moin/WebFrameworks>

The list of companies and sites using both Flask and Django are numerous.

<https://www.djangoproject.com/>
<https://careerkarma.com/blog/companies-that-use-flask/>



Flask Introduction

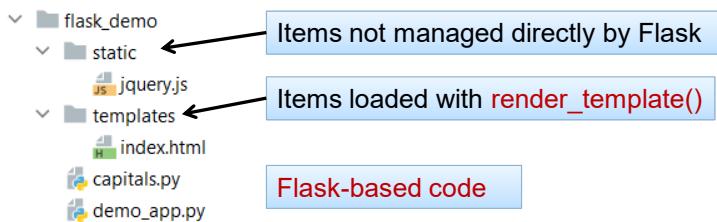
- A lightweight, extensible, WSGI web application framework (BSD license)
- A flask application should contain the following structure



Flask

<https://flask.palletsprojects.com/en/3.0.x/>

pip install flask



Quick overview:
<https://flask.palletsprojects.com/en/3.0.x/quickstart/>

Flask typically installs within seconds using pip. Once installed, you can begin running the Flask server and building web applications with it.

Routes

- `@app.route()` decorator defines which functions are called for which URLs
- The decorator identifies which URL mapping calls which function

`demo_app.py`

```
from flask import Flask, render_template, jsonify, Response  
from capitals import capitals
```

```
app = Flask(__name__)
```

Flask calls your function when a GET request matching `http://localhost:8051/` is received

```
@app.route('/')
```

```
def main_page():
```

```
    return render_template('index.html')
```

```
app.run(host='localhost', port=8051)
```

`render_template()` returns the contents of this file as a string of HTML

ch06_api/flask_demo/demo_app.py



Routes with Path Parameters

- Add additional methods and routes as more requests are needed

```
@app.route('/states/<name>', methods=['GET'])
def get_capital(name):
    try:

        resp = jsonify(state=name, capital=capitals[name])

    except KeyError:

        resp = jsonify(state=name, capital='not found')

    return resp
```

<name> allows for any value to be submitted
(e.g., http://localhost:8051/states/**Colorado**)

name is injected (by Flask) from the variable
in the URL as an argument to your function

Converts keywords into
JSON-based properties

Sends a JSON-based response back to the client

ch06_api/flask_demo/demo_app.py



Running the App

- Start the Flask server
- Browse to: <http://localhost:8051/>

State Name

State Name

Nebraska's capital is Lincoln.

```
* Serving Flask app 'demo_app'  
* Debug mode: off  
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.  
* Running on http://localhost:8051  
Press CTRL+C to quit
```

Production solutions using Flask should be deployed into environments such as [Gunicorn](#) or [Waitress](#), etc.

For more, see:
<https://flask.palletsprojects.com/en/3.0.x/deploying/gunicorn/>
<https://flask.palletsprojects.com/en/3.0.x/deploying/waitress/>

ch06_api/flask_demo/demo_app.py

Flask is not designed for large-scale applications without a little help. Fortunately, there are good options that implement multiple processes, they improve response times, are lightweight, and somewhat easy to deploy. While others exist, Gunicorn and Waitress are two popular options for deploying Flask and other WSGI servers. They are capable of supporting thousands of requests per second.

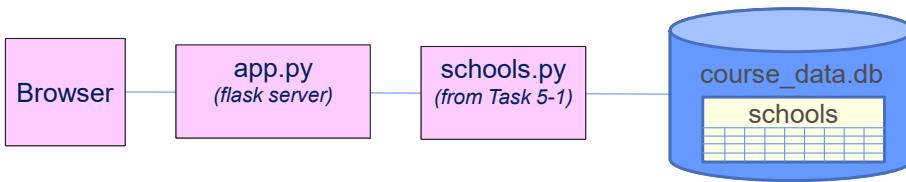


Your Turn! - Task 6-1

Working with Flask

- Complete the Flask-based application
 - Locate the [ch06_api/starter/task6_1/app.py](#) folder
- [schools.py](#) contains our completed version of [task5_1_starter.py](#)

Refer to this file for additional instructions



localhost:8051

School: poly

Results for poly:

California Polytechnic State University, Pomona (Pomona, CA)
California Polytechnic State University, San Luis Obispo (San Luis Obispo, CA)
Virginia Polytechnic Institute and State University (Blacksburg, VA)

Work from [ch06_api/starter/task6_1](#) folder in the student files



The Python *requests* Module

- A popular third-party tool, called **requests**, is ideally suited for constructing web application network requests

`pip install requests`

```
import requests

resp = requests.get('http://someURL')
resp = requests.post('http://someURL', data={'key': 'value'})
resp = requests.put('http://someURL', data={'key': 'value'})
resp = requests.delete('http://someURL')
resp = requests.head('http://someURL')
resp = requests.options('http://someURL')
```

Useful req object attributes:

`resp.text`
`resp.json()`
`resp.url`
`resp.status_code`
`resp.headers`
`resp.request.headers`
`resp.request.body`
`resp.cookies`
`resp.content`

The `requests` module is the premiere module in Python for making HTTP/S requests. It supports most kinds of authentication. While other options exist within Python for making HTTP requests (`pycurl`, `httplib`, `urllib`, `httplib2`, and `aiohttp`).

Using requests

```
feed = 'https://api.stackexchange.com/2.2/search?intitle=python&site=stackoverflow'
data_dict = requests.get(feed).json()
for idx, question in enumerate(data_dict.get('items', []), 1):
    print(idx, question.get('title'))
```

- 1 Updating class attributes in python
 2 Why do I get this TypeError using the randint function in Python?
 3 Why is python returning no result to my file search?
 ...

```
feed = 'https://api.stackexchange.com/2.2/search'
query_str = {
    'intitle': 'python',
    'site': 'stackoverflow'
}
data_dict = requests.get(feed, params=query_str).json()
print(f'URL used: {resp.url}')
```

Builds the query string

URL used: <https://api.stackexchange.com/2.2/search?intitle=python&site=stackoverflow>

ch06_api/02_requests.py

The example shows how requests can make an HTTP GET request. It retrieves some JSON-formatted data and converts it to a Python dictionary using the .json() method.

request.get() supports an optional params argument that can be a dict, list, or string to embed into the query string part of a URL.



Basic Authentication with *requests*

- Requests supports most types of authentication schemes including custom schemes
- To perform **basic authentication**, use the `auth=` parameter

```
import requests
from requests.auth import HTTPBasicAuth

page = 'https://jigsaw.w3.org/HTTP/Basic/'

auth = HTTPBasicAuth('guest', 'guest')
page_text = requests.get(page, auth=auth).text

print(page_text)
```

ch06_api/02_requests.py

Use the `HTTPBasicAuth` class in the `requests.auth` submodule to handle a basic authentication challenge. Requests will take care of the rest for you.



Manually Parsing JSON

- The **json** module is a way of parsing JSON data when not using the requests module

```
dict = json.loads(str)
json_str = json.dumps(dict)
```

```
import json

data = r'''
{
    "name": "burns",
    "results": [{"actor": "Harry Shearer",
                 "episode_debut": "\"Simpsons Roasting\"",
                 "name": "Charles Montgomery Burns",
                 "original_air_date": "1989-",
                 "role": "Owner of Nuclear Power Plant."}
    ]
}
'''


result = json.loads(data)
print(result.get('results')[0].get('name'))
```

Charles Montgomery Burns

ch06_api/03_parsing_json.py

The example takes the JSON data and passes it into the loads() method. The returned result is a dictionary. To extract a piece of information like 'actor' or 'name', you will need to access and drill down into the resulting dictionary.



Handling Non-serializable Fields (1 of 2)

- Some JSON fields can't be converted
 - These require special handling

```
@dataclasses.dataclass
class Employee:
    first: str
    last: str
    salary: float
    hiredate: datetime.date

    hired = datetime.date(1997, 10, 20)
    empl = Employee('Thomas', 'Hanks', 22_000_000, hired)

    json_str = json.dumps(vars(empl))
```

Error: Object of type `date` is not JSON serializable

```
json_str = json.dumps(vars(empl), default=str)
print(json_str)
```

Works now!

Quick, easy, but
not flexible

```
{"first": "Thomas", "last": "Hanks", "salary": 22000000, "hiredate": "1997-10-20"}
```

ch06_api/03_parsing_json.py

This example encodes the Employee object into a JSON string. But the date field can't be converted (serialized). An easy fix to this problem is to use `default=str` which converts the date to a string format first. But this doesn't give a lot of flexibility in formatting the date.

Handling Non-serializable Fields (2 of 2)

- Another way to handle fields that can't be converted to JSON is by using an encoder class and implementing the default() method

```

hired = datetime.date(1997, 10, 20)
empl = Employee('Thomas', 'Hanks', 22_000_000, hired)

class EmployeeEncoder(json.JSONEncoder):
    def default(self, obj):
        try:

            result = vars(obj)

        except (AttributeError, TypeError):
            if isinstance(obj, datetime.date):
                result = obj.strftime('%Y-%b-%d')
            else:
                result = '(unknown type)'

        return result

print(json.dumps(vars(empl), cls=EmployeeEncoder))

```

This fails on the date field with message:
Error: Object of type date is not JSON serializable

Manually defines how to
 serialize the date field

{"first": "Thomas", "last":
 "Hanks", "salary": 22000000,
 "hiredate": "1997-Oct-20"}

ch06_api/03_parsing_json.py

Alternatively, an encoder can be used to convert the hiredate field. The encoder should inherit from JSONEncoder and override the default() method. The default() method is only called on the hiredate field and it will fail in the vars(obj) conversion. When the hiredate fails, the except block will convert the (date) obj and return a formatted result.

 Your Turn! - Task 6-2

A Client to the Flask Server

- Create a client using the Python *requests* module that will make a request to the Flask Server created in Task 6-1

```
Enter a school name [Loyola]:  
Using: http://localhost:8051/api/schools/Loyola  
  
...  
  
Results for: Loyola  
+-----+  
| School |  
+-----+  
| Loyola College in Maryland (Baltimore, MD) |  
| Loyola Marymount University (Los Angeles, CA) |  
| Loyola University Chicago (Chicago, IL) |  
| Loyola University New Orleans (New Orleans, LA) |  
+-----+
```

Work from client.py provided in the
ch06_api/starter/task6_2 folder in the student files



The Flask *request* Object

- Flask provides a ***request*** object that collects information about the request sent by the client

Don't confuse the client-side ***requests*** module with Flask's ***request*** parameter

- Some of the ***request*** object attributes include

- **remote_addr** - (string) address of the client
- **path** - (string) part of the URL after the port but before the query string
- **form** - (dict) submitted form (body) parameters
- **args** - (dict) submitted query string parameters
- **headers** - (dict) client's request headers

Client

```
requests.get(url,
              params={'a_param': 'value'})
```

Handles query
string parameters

Flask Server

```
@app.route('/api/resource/<object>', methods=['GET'])
def handler(object: str) -> Response:
    a_param = request.args.get('a_param')
```

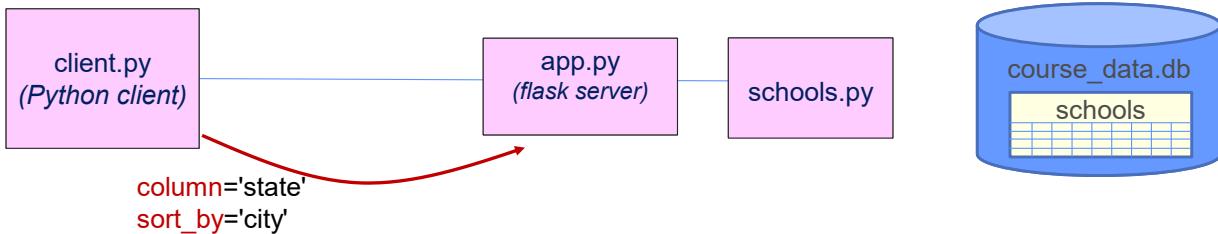
Information about the client, including the address, the requested URL, the body parameters, and headers are collected by Flask and provided to you within any route by accessing the `request` object. Shown above is a way to access query string parameters submitted by the client to the Flask app. The request app provides a dict that can be accessed as shown.



Your Turn! - Task 6-3

Submitting Query String Parameters

- Add the ability to send *column* and *sort_by* values from the client to the server when performing a search



Example URL sent by client:

```
http://localhost:8051/api/schools/ID?column='state'&sort_by='city'
```

Work from the provided **ch06_api/starter/task6_3/client.py** file first and then **app.py**

RESTful Services

- REST stands for *Representational State Transfer*



- *Resources* are the fundamental components in a RESTful solution
 - Resources are usually derived from the application's domain
 - Examples include:
 - *Invoices*
 - *Patients*
 - *Customers*
 - *Documents*



REST is a style of communication between two devices. REST involves sending objects to and from the server (the "transfer" part of the acronym). It often uses JSON (though it is not required) as the data interchange format. JSON objects "represent" data maintained on the server.



RESTful Design Principles

- The API is designed around the resource
 - Identify which operations will be allowed on that resource

GET `https://hostname/api/invoices/2952`

Reads a specific invoice

PUT `https://hostname/api/invoices/2952`

Updates a specific invoice

PATCH `https://hostname/api/invoices/2952`

Partially Updates a specific invoice

DELETE `https://hostname/api/invoices/2952`

Deletes a specific invoice

GET `https://hostname/api/invoices`

Retrieves all invoices

POST `https://hostname/api/invoices`

Creates a new invoice

Think of a RESTful request as being like a sentence. The HTTP method is the "verb" of the sentence while the resource is the noun. "GET (verb, HTTP method) the invoice 2952 (noun, resource)."

As previously mentioned, we tend to utilize the plural form of our resource (e.g., invoices) in the URL.

The PATCH method is used when you only wish to update one or two (or so) fields of an object. In doing so, you may PATCH by only sending the required fields that need updating. Generally, a PUT will "resend" the entire object, performing an overwrite of the entire object.



Flask Plugins: Using SQLAlchemy

- SQLAlchemy can integrate a database with Flask
- To use SQLAlchemy with Flask:
 1. Configure the Flask-SQLAlchemy plugin
 2. Define models to interact with the DB
 3. Work models in a session

```
from flask import Flask, jsonify, redirect
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

db_location = Path(__file__).parents[1] / 'resources/course_data.db'
print(f'Using database at location: {str(db_location)}')
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///+' + str(db_location)
app.config['SQLALCHEMY_ECHO'] = True

db = SQLAlchemy(app)
```

This is the location of our database

Displays generated SQL statements

For more on configuring other databases, visit:
<https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/config/>

ch06_api/04_flask_sqlalchemy.py

As is done with most plugins, the app object is passed into the SQLAlchemy object. When the Flask-SQLAlchemy plugin starts up, it will look within the Flask app's config object for this value. For other databases, other properties may be required, such as properties for a username and password to authenticate with the database.

The Flask-SQLAlchemy plugin is not always appropriate. Some apps may be better suited using the regular SQLAlchemy framework. In this way, the user can define the scopes of their sessions and reuse model objects even if not using Flask.



Defining a SQLAlchemy Model

```

class Airport(db.Model): ← The model inherits from db.Model
    __tablename__ = 'airports'
    airport_id = db.Column('airportid', db.String(10), primary_key=True)
    name = db.Column(db.String(100)) ← These fields identify the column name, type, and other info
    city = db.Column(db.String(100))
    country = db.Column(db.String(15))
    abbr = db.Column('IATA_FAAC', db.String(50))
    timezone = db.Column('timezone', db.String(50))

    def __init__(self, airport_id: str, name: str, city: str,
                 country: str, abbr: str, timezone: str):
        self.airport_id = airport_id
        self.name = name
        ...
    def _get_local_time(self): ← This function returns time in GMT (UTC) plus a timezone offset
        local_time = datetime.now(timezone.utc) + timedelta(hours=float(self.timezone))
        return local_time.strftime('%I:%M%p')
    def __str__(self):
        return f'Local time at {self.name} ({self.city}, {self.country}) is
               {self._get_local_time()}.'

```

ch06_api/04_flask_sqlalchemy.py

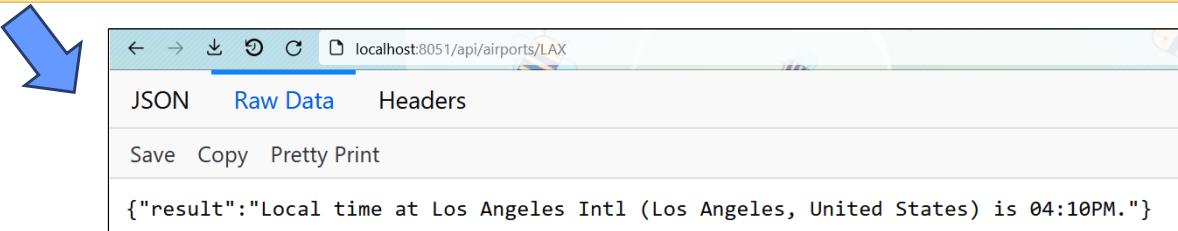
A model class should inherit from db.Model (db was created from the Flask-SQLAlchemy plugin object). Then (at the class level!) add field definitions to the class that will map to the database table columns. Optional `__init__()` and `__str__()` methods are provided for convenience for instantiation and debugging or logging purposes.

Using the SQLAlchemy Models

- Working with models involves using an object-oriented style syntax

```
@app.route('/api/airports/<search_val>', methods=['GET'])
def get_airport(search_val: str):
    try:
        result =
            db.session.execute(db.select(Airport).filter_by(abbr=search_val)).scalar_one()
        status = 200
    except Exception as err:
        result = err.args[0]
        status = 400

    return jsonify(result=str(result)), status
```



ch06_api/04_flask_sqlalchemy.py

Models can be used to perform queries to the database. To perform an insert, use `db.session.add(obj)`. To perform a delete, use `db.session.delete(obj)`. To modify a model object, make changes to the object and then perform a `db.session.commit()`.

The version shown above follows the newer-style ORM query syntax.

For more on using the SQLAlchemy query API, visit <https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/queries/>.

Flask Plugins: Incorporating Marshmallow

- What is Flask-marshmallow?
 - Flask-marshmallow is a helpful tool for converting objects from Python to JSON or JSON to Python
 - Ideal for returning a Flask database object as JSON from an API



Marshmallow homepage: <https://marshmallow.readthedocs.io/en/stable/>
Flask-Marshmallow (plugin) home: <https://flask-marshmallow.readthedocs.io/en/latest/>

- To use Marshmallow:
 1. **Configure** the plugin within Flask
 2. **Define and instantiate a "schema"** (the fields to be serialized)
 3. **Use the schema** to return from API methods (as needed)

Marshmallow is a standalone tool within Python. Marshmallow by itself is useful in converting Python objects to dictionaries. While numerous tools can do this, some of them have problems if the objects do not have underlying dictionaries. In other cases, the utilities for doing this are framework-specific (like Flask's `jsonify()`).

Flask-Marshmallow simply takes the easy-to-use marshmallow framework and brings it into Flask (as has been done with other plugins like SQLAlchemy).

Here, we make use of it to automatically convert SQLAlchemy objects into JSON directly.

Flask Plugins: Using Marshmallow

```

from flask_marshmallow import Marshmallow
app = Flask(__name__)
ma = Marshmallow(app)           ← This is how the plugin gets coupled to Flask

@app.route('/api/airports/<search_val>', methods=['GET'])
def get_airport(search_val: str):
    try:
        result = db.session.execute(db.select(Airport).filter_by(abbr=search_val))
        .scalar_one()
        return jsonify(result=airport_schema.dump(result), localtime=str(result)), 200
    except Exception as err:
        return jsonify(result=err.args[0]), 404
    ← The schema is applied, converting
    ← the result to a dictionary

class AirportSchema(ma.Schema):
    class Meta:
        fields = ('airport_id', 'name', 'city', 'country', 'abbr', 'timezone')

airport_schema = AirportSchema()
app.run(host='localhost', port=8051)           ← A schema is instantiated and used to define which
                                                ← attributes will be marshalled (converted to JSON)

```

ch06_api/05_flask_sqlalchemy_marshmallow.py

Configuring Flask-marshmallow is only two lines: an import and passing the Flask app object into the Marshmallow main object. Next, define a class that identifies which fields will be exposed (serialized). Instantiate these so that the instances can be used in your routes. The schema has a `dump()` method which can convert objects to dictionaries or a `jsonify()` method if you wish to return a JSON response of the object.

Accessing the Airport API

The screenshot shows a browser window with the URL `http://localhost:8051/api/airports/LAX` in the address bar. The page title is "Airports". Below the address bar, there are tabs for "JSON", "Raw Data", and "Headers", with "JSON" being the active tab. Underneath are buttons for "Save", "Copy", and "Pretty Print". The main content area displays a JSON object:

```
{  
    "localtime": "Local time at Los Angeles Intl (Los Angeles, United States) is 05:40PM.",  
    "result": {  
        "abrv": "LAX",  
        "airport_id": "3484",  
        "city": "Los Angeles",  
        "country": "United States",  
        "name": "Los Angeles Intl",  
        "timezone": "-8"  
    }  
}
```

A callout box with a black arrow points from the text "The returned Airport (SQLAlchemy) object is automatically converted to JSON by marshmallow" to the "result" key in the JSON response.

For this example, a browser can be used to view the GET response from the Flask app. The airport object queried for is returned according to the Marshmallow schema definition.



Your Turn! - Task 6-4

Integrating SQLAlchemy and Marshmallow with Flask

- Incorporate both SQLAlchemy and Marshmallow into the Flask school app
- The client will continue to make requests and receive a list of schools in return
- Objects are retrieved from the DB using SQLAlchemy
- The list of schools will be converted via Marshmallow plugin

Work from and follow additional instructions from the `ch06_api/starter/task6_4/app.py` file first and then run the provided `client.py`



Chapter 6 Summary

- Flask is a WSGI compliant web-based framework designed to easily create interactive client-to-Python applications
- Routes can be defined in the Flask app to support RESTful API development
- Flask incorporates many popular Python Frameworks via a pluggable architecture



Chapter 7

Intro to Data Analysis



NumPy, Matplotlib, Pandas Concepts



Chapter 7 - Overview

Working with Jupyter

Using NumPy

Matplotlib

Introducing Pandas



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

The screenshot shows the Google Colab interface. On the left, there's a sidebar with navigation links like 'Getting started', 'Data science', 'Machine learning', 'More Resources', and 'Machine Learning Examples'. The main area has a code cell containing:

```
[1]: import sys
import pandas as pd
sys.version, pd.__version__
('3.6.9 |Anaconda, Inc.| Jul 17 2020, 12:50:27 | \n[GCC 8.4.0]', '1.0.5')
[2]: print('You can create notebooks online in Google Colab.')
You can create notebooks online in Google Colab.
```

Below the code cell is a 'What is Colaboratory?' sidebar with a list of benefits:

- Zero configuration required
- Free access to GPUs
- Easy sharing

At the bottom, it says: "Whether you're a student, a data scientist or an AI researcher, Colab can make your work easier. Watch [Introduction to Colab](#) to learn more, or just get started below!"

<https://www.kaggle.com/kernels>

The screenshot shows the Kaggle Notebooks interface. It features a sidebar with 'Compete', 'Data', 'Notebooks' (which is selected), 'Discuss', 'Courses', 'Jobs', and 'More'. The main area lists several public notebooks:

- 169: Lyft: Understanding the data + baseline model (3w ago with multiple data sources)
- 150: Palmarony Fibrosis Dicom Preprocessing (1w ago by DataPalmarony Palmarony) (Beginner, exploratory data analysis, data cleaning, data visualization)
- 35: Heart Disease and Some scikit-learn Magic (3w ago by DataScienceDataScience) (Intermediate, Beginner, exploratory data analysis, dataset)
- 6: Stock price prediction using LSTM (3w ago with no data sources)
- 136: FiveThirtyEight Comics Analysis (1w ago in FiveThirtyEight Comic Characters Dataset) (Beginner, exploratory data analysis, data visualization)

Notebooks

Explore and run machine learning code with Kaggle Notebooks! Find help in the [Documentation](#).

+ New Notebook

Public	Your Work	Shared With You	Favorites	Sort by	Hotness
Categories	Outputs	Languages	Tags	Search notebooks	Q
169	Lyft: Understanding the data + baseline model	Py	29		
150	Palmarony Fibrosis Dicom Preprocessing	Py	92		
35	Heart Disease and Some scikit-learn Magic	Py	18		
6	Stock price prediction using LSTM	Py	3		
136	FiveThirtyEight Comics Analysis	Py	59		

<https://colab.research.google.com>

IPython is a code execution engine. Jupyter Notebook provides a browser-based interface for the IPython kernel.

Typing `ipython` from a command line launches an IPython interactive shell. Other ways to start IPython include:

`jupyter qtconsole` (which starts an enhanced console in a new GUI window).

There are repositories and online notebook creation capabilities. Google Colab allows for creating notebooks online, while Kaggle provides a repository for publicly shared notebooks. Other noteworthy notebook galleries include <https://nbviewer.org/> and <https://github.com/jupyter/jupyter/wiki>.



Working with Jupyter Notebook

- To use Jupyter, it must be installed first

`pip install ipython`

`pip install jupyter`

If using the Anaconda distribution,
these will already be present

- To launch Jupyter, first **cd** to the *student_files* directory within a command prompt (or terminal) and run the command:

`jupyter notebook`

```
c:\temp\notebooks>jupyter notebook
[W 14:12:24.283 NotebookApp] Unrecognized JSON config file version, assuming version 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 named 'nbbrowserpdf'
[I 14:12:25.767 NotebookApp] Serving notebooks from local directory: c:\temp\notebooks
[I 14:12:25.768 NotebookApp] 0 active kernels
[I 14:12:25.768 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 14:12:25.768 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

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

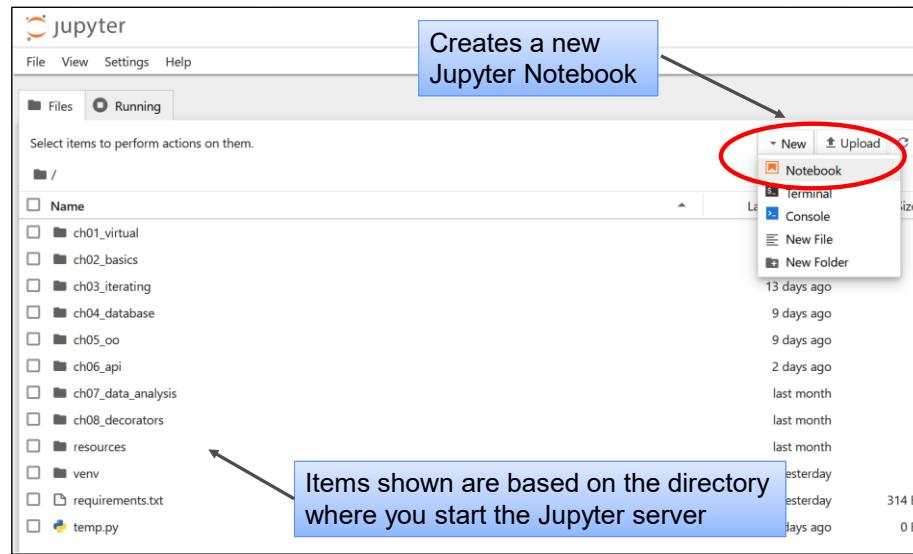
Note: In August 2023, Jupyter updated its interface so some interactions and menus may look different than yours if your version is older.

Additional startup options include:

--port <8888>
--no-browser (to avoid launching the browser when the server starts)
--version (displays the version number)
--paths (displays all Jupyter-related paths)

Creating a New Notebook

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



Create a new notebook by selecting the "New" dropdown box and then choosing "Python 3."

You should execute the `jupyter notebook` command from the directory you wish to start your Jupyter server. After starting Jupyter, browse to `localhost:8888` and the dashboard will display to contents of the directory where you launched Jupyter from.

To start Jupyter with a custom config file, invoke `jupyter notebook --generate-config` and then modify the settings found in `jupyter_notebook_config.py` found in the config directory (mentioned on the previous slide).

Notebook Interface

The screenshot shows the Jupyter Notebook interface. On the left, a sidebar indicates that notebooks consist of **markdown** and **code** cells. The main area displays a cell containing Markdown notation examples. The toolbar at the top includes buttons for file operations, cell type selection (Code, Markdown, Raw), and cell movement.

Notebooks consist of markdown and code cells

Select the type of cell desired

Move cells up/down

Insert cells above/below

Markdown Notation

```

#      H1      $inline_expr$  

##     H2      $$expr\_on\_own\_line$$  

###    H3  

####   H4  

#####  H5  

###### H6  

*Italic Text*
  
```

Code cells are the primary type of cell used. They contain your Python code.

Markdown cells allow for rendering formatted text. They support a special notational syntax shown above. One type of notation, called Latex, allows for depicting mathematical formulas. Refer to the Matplotlib section for more on Latex.

Code Assistance and Documentation

- Jupyter supports viewing code assistance

The screenshot shows two Jupyter notebook cells. The left cell has the code `[1]: import numpy as np`. The right cell has the code `[2]: np.`. A dropdown menu is open over the cursor, listing various methods and attributes starting with 'np.' such as abs, absolute, add, add_docstring, add_newdoc, add_newdoc_ufunc, all, allclose, ALLOW_THREADS, and alltrue. A callout bubble says "Shift-Tab (new version) to display documentation assistance". The right cell shows the full documentation for `np.array()`, including the docstring, parameters, and type hints.

Shift-Tab (new version) to display documentation assistance

```
[1]: import numpy as np
[2]: np.
```

abs
absolute
add
add_docstring
add_newdoc
add_newdoc_ufunc
all
allclose
ALLOW_THREADS
alltrue

```
[1]: import numpy as np
[2]: np.array()
Docstring:
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
      like=None)

Create an array.

Parameters
-----
object : array_like
    An array, any object exposing the array interface, an object whose
    ``__array__`` method returns an array, or any (nested) sequence.
    If object is a scalar, a 0-dimensional array containing object is
    returned.
dtype : data-type, optional
    The desired data-type for the array. If not given, NumPy will try to
    use
        a default ``dtype`` that can represent the values (by applying
        promotion
        rules when necessary.)
copy : bool, optional
```

Use Shift-Tab (older Jupyter version uses Shift-TAB-TAB) to examine the documentation for the attribute/method at the cursor's location. Make sure the library is imported and the cell for the import is executed first before using these shortcuts.



Your Turn! - Task 7-1

Using Jupyter

- Start your Jupyter Server within your `student_files` directory
(as discussed in the chapter)
 - Work from (open) the provided `task7_1_starter.ipynb` file
- 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 from the function 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

NumPy and the ndarray

- NumPy is a Python-based framework providing multi-dimensional array creation and manipulation tools
- NumPy's main object is the array (ndarray type)
 - Array dimensions in NumPy are called **axes**
 - The number of axes is its **rank**



Stands for n-dimensional array

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

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

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

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

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

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

ch07_data_analysis/01_numpy_array.py

NumPy arose from the combining of two older frameworks called Numeric and Numarray.

Note: The term axes (as used above) has nothing to do with coordinates in space. Here, it defines the number of subscripts needed to access a value within the array (e.g., arr[0][2][1]). This indicates an array with three axes (a 3-dimensional array).

Accessing the Array

- **ndarray types** (NumPy arrays) have several important attributes

```
<class 'numpy.ndarray'>
```

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

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

print(type(arr))

print(f'Shape: {arr3.shape}')
print(f'Size: {arr3.size}')
print(f'Axes: {arr3.ndim}')
print(f'Dtype: {arr3.dtype}')
```

- NumPy arrays are *homogenous*

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

Arrays are stored contiguously in memory

float64

ch07_data_analysis/01_numpy_array.py

NumPy arrays are stored contiguously in memory. By setting all data values to a fixed size, this enables pointer arithmetic to perform hundreds of thousands of operations as fast as possible. For performance, operations are performed outside of the Python virtual machine at the native code level.

Other Ways to Create 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

ch07_data_analysis/01_numpy_array.py

These are a few additional ways of initializing arrays in NumPy. zeros() fills the specified shaped array with all zero values. ones() does the same with one values. full() fills it with a specified value. eye() creates an identity matrix of that size.

linspace() generates a specific number of items as defined by total_items. total_items for linspace() should be an int. arange() generates a NumPy array of values separated by an amount indicated in step_by. The default step_by is 1.



Accessing Data within Arrays

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

Syntax:
array[rows, cols]

```
arr6 = 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

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

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

arr6[2:, 1:]

This notation is used by numerous Python data analysis frameworks such as Pandas

ch07_data_analysis/01_numpy_array.py

Selecting values from a NumPy array can be done using row, col notation. Negative values and slice notation is also supported.

Selecting Rows and Columns

- Select all rows or columns using the colon

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

`arr6[:, 1]`

Colon "selects all" rows

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

`arr6[2, :]`

Colon here "selects all" columns

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

Reshaped arrays must be the same overall size afterwards

`arr6.reshape((2, 6))`

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

ch07_data_analysis/01_numpy_array.py

Using reshaping, we can see an easier way of creating a multidimensional array. When reshaping, the total size of the array (total number of elements) cannot change. Pandas does not support higher-order arrays over two-dimensions.



Aggregation Functions

- NumPy provides functions for performing tasks on all selected elements within the array

```
arr7 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9],
                 [10, 11, 12]])
```

```
print(np.sum(arr7))          # 78
print(arr7.sum())            # 78

print(np.mean(arr7[:, 0]))    # 5.5
print(arr7[:, 0].mean())      # 5.5

print(np.prod(arr7[:2, :2]))  # 40
print(arr7[:2, :2].prod())    # 40

print(arr7.prod(axis=1))     # [ 6 120 504 1320]
```

Sums all elements

Average of column 0

Product of upper left 4 nums

Product of each row

ch07_data_analysis/02_aggregating.py

Numerous additional functions also perform aggregation operations, including:

`np.min()`, `np.max()` - find the min and max values within an array

`np.argmin()`, `np.argmax()` - find the index of the min and max values within an array

Selections Using Masks

- For selecting cells dynamically, this achieves a filtering (masking) effect:

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

Where a True occurs,
the item is selected

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

```
arr15 = np.arange(1, 7)
```

```
[1 2 3 4 5 6]
```

```
mask = arr15 <= 3
```

```
[True, True, True, False, False, False]
```

```
print(arr15[mask])
```

```
[1 2 3]
```

ch07_data_analysis/03_masking.py

Selecting columns using Boolean-valued arrays will select only the columns that are True. In the test above ($\text{arr15} \leq 3$), each element will be compared to the value 3. This results in a NumPy array of Booleans. The Boolean array (we called it mask) is plugged back into the original array to achieve the masking effect.



Masks on Rows and Columns

- Selecting columns based on a criterion:

```
arr16 = np.arange(11, 29, 2).reshape(3, 3)
mask = arr16[0] > 12 [False, True, True]
print(arr16[:, mask])
```

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

arr16
[[11 13 15]
[17 19 21]
[23 25 27]]

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

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

- Selecting rows based on a criterion:

```
arr16 = np.arange(11, 29, 2).reshape(3, 3)
mask = arr16[:, 2] > 15 [False, True, True]
print(arr16[:, mask])
```

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

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

arr16

ch07_data_analysis/03_masking.py

There are two examples shown here. Each presents how to select data within our arrays (doesn't matter if it's NumPy, Pandas, or numerous other Python data analysis frameworks). In the first example, columns are selected based on comparing the first row to greater than 12. If the conditional is true in that first row, the entire column is selected.

In the second example, rows are selected based on a comparison of the third column to 15. If the conditional is true in that third column, the entire row will be selected.

 Your Turn! - Task 7-2

Using NumPy

- Continue from Task 7-1, this time load the population data returned from the function into a NumPy array
 - Note: use `dtype=object` since values aren't homogeneous
 - Work from the provided `task7_2_starter.ipynb` file
(or your existing `task7_1_starter.ipynb`)
- 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

Introducing Matplotlib

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

Bar
Line
Scatter

Pie
Log
Bar

Line
Scatter
Polar

Streamplots
Ellipses
Table Demo
...and more...

- Common module import conventions

```
import matplotlib as mpl
```

We will refer to this module as **plt** from now on

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

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

The names shown on this slide represent a common way to import Matplotlib's main module and submodule. For reference, the PyPlot API docs can be found at <https://matplotlib.org/stable/>.

Plotting with Matplotlib

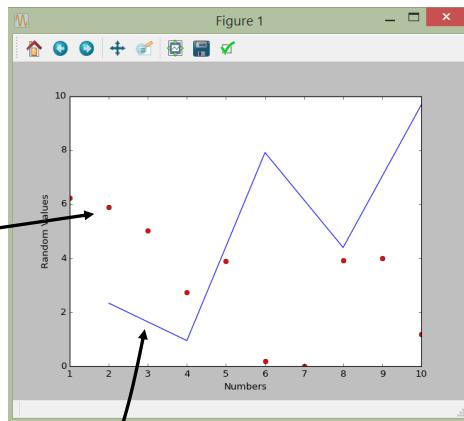
```
import matplotlib.pyplot as plt
import numpy as np

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

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

plt.show()
```

Use this (outside of Jupyter) to render output



ch07_data_analysis/04_simple_plot.py

In the example, two pairs of NumPy arrays are provided to the plt.plot() method. The first, x1, is a NumPy array from 1 to 10, while the second, y1, is a ndarray of 10 random values. They are rendered using red circles.

The second plot is overlaid onto the first. This one plots the x2 and y2 ndarrays using the default blue line. More on the line and marker styles is shown on the next slide. Labels have been added and show() is invoked to cause the back end to render the graph.

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

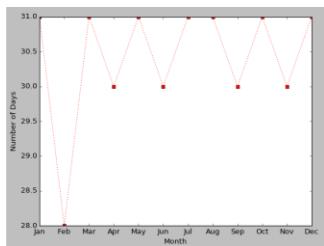
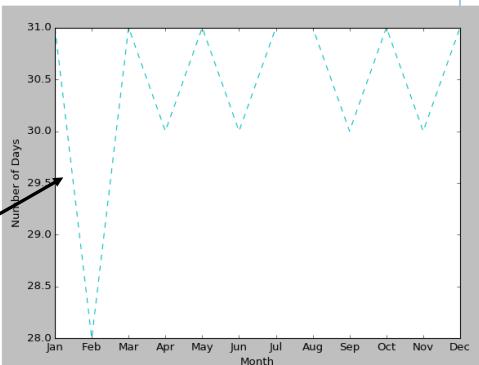
You can supply a list, ndarray, or other sequences for the x-axis and the y axes. In addition to the x and y axes, the style of marker/line can be provided. This argument is a string containing a color symbol (a letter from the color list on the left above) followed by a line style (an option from the right two lists). The default is 'b-', which refers to a solid blue line. Other examples are shown on the next slide.

Also, instead of supplying two individual arrays, a single, two-dimensional array may be supplied for the x and y axis.

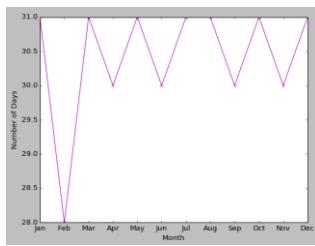
Changing Styles

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

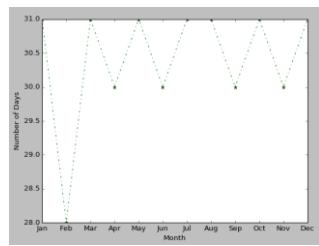
plt.xlabel('Month')
plt.ylabel('Number of Days')
plt.xticks(x, x_ticks)
plt.plot(x, y, 'c--')
plt.show()
```



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



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



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

ch07_data_analysis/05_styles.ipynb

The last day of the month is plotted against the month name. Because Matplotlib wants numerical values for the axes, `arrange()` is used for the x-axis, but `xticks()` provides the label names.

The top example uses '`c--`' for the style argument which displays a cyan colored dashed line while the bottom examples (from left to right) display: red square markers with a red dotted line, magenta solid line with x markers, and green dash-dot line with star markers.

Line Style Properties

- While style shortcuts can be used, there are other line styles that attributes:

```
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

Visit here for more marker styles: https://matplotlib.org/stable/api/markers_api.html

The drawstyle is the line type used to connect the data points.

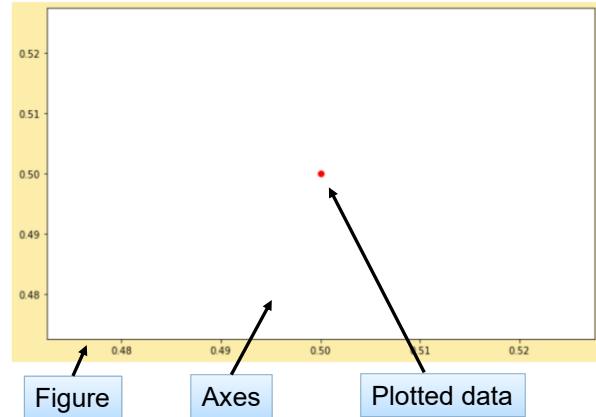
While the slide shows a call to plt.plot(), these values apply to any subplot/axes object as well.

Figures and Axes

- For more fine-grained 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



ch07_data_analysis/05_styles.ipynb

Figures represent the entire drawing canvas. An Axes object is a coordinate system placed onto the figure.

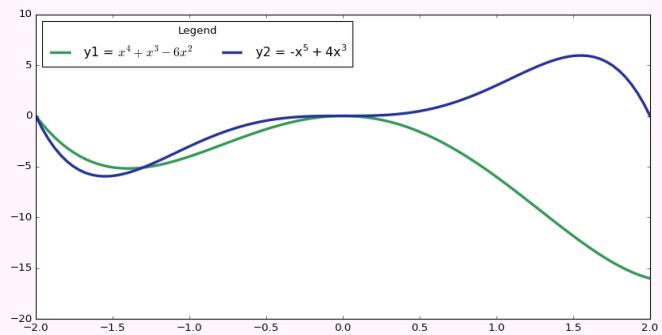
Legends

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

```
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='#ffff5ff')

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)
```



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

ch07_data_analysis/06_legends.ipynb

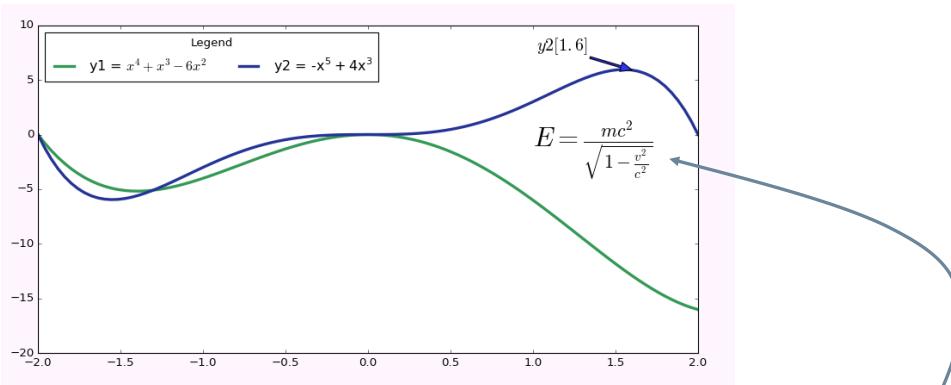
For more on Matplotlib's displaying of math symbols in text, refer to <http://matplotlib.org/users/mathtext.html>.

To generate superscripts, use the Matplotlib TeX syntax: $\$ \text{_____} \$$ as shown in the `legend()` statement above.

The `loc` specifies the location of the legend. Values include 'upper right', 'upper left', 'lower right', 'lower left'. The `ncol` argument defines how many columns to display for the legend. The default value is 1.

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)
```

ch07_data_analysis/06_legends.ipynb

The first annotation, ($E=mc2$), places the formula onto the plot specifying the `xycoords` value of 'figure fraction', which means it will be placed as a percentage of the figure. The `xy` argument places the text at 0.7 across the width and 0.6 up the height of the figure.

savefig()

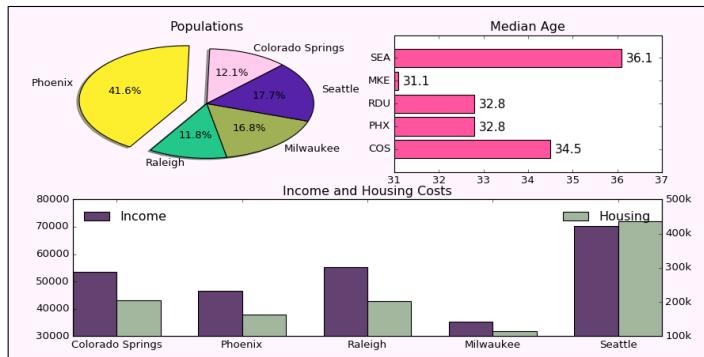
- Save figures using `figure.savefig()`:

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

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

`fname` is the name of the file the plot will be saved to. The format can be 'pdf', 'ps', 'svg', 'png', and 'eps'. The orientation may be 'portrait' or 'landscape'.

Other Chart Types (1 of 3)



```
figure = plt.figure(figsize=(12, 6), facecolor='#ffff55ff')

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]
```

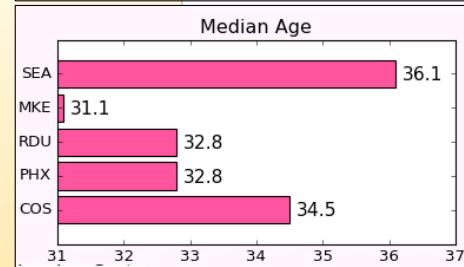
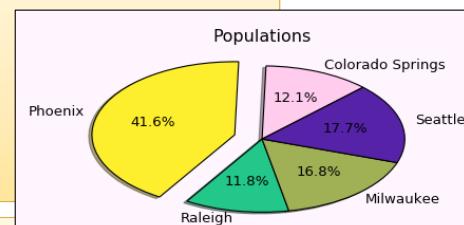
ch07_data_analysis/07_other_charts.py

The figure shown above is created using the data provided here. The subsequent two slides discuss the pie, horizontal bar, and grouped bar plots.

Other Chart Types (2 of 3)

```
sub1 = figure.add_subplot(221)
colors = ['#ffccce', '#fdef2e', '#23c78a', '#a0b055', '#5623a8']
sub1.pie(populations, labels=cities,
          colors=colors,
          explode=[0, 0.2, 0, 0, 0],
          autopct='%.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)
```

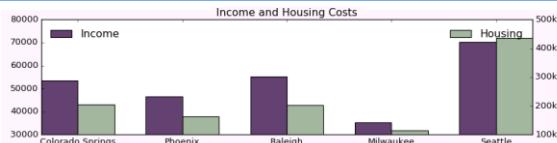


ch07_data_analysis/07_other_charts.py

The pie chart is created by invoking `pie()`. The `colors=` option provides the shading for each pie piece. The `explode=` option identifies which pieces to "break off" and by how much (fractionally). `autopct=` shows a numeric value within the pie piece.

On the bottom, the `barh()` method call creates a horizontal bar graph. Take note of the placement of annotations to identify the values of data points at the end of each bar. This was done by plotting the value of the `median_age` and then adding a small offset to move the text slightly farther to the right.

Other Chart Types (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_xlim(30000, 80000)
sub3.set_title('Income and Housing Costs')
sub3.legend(['Income'], loc='upper left', frameon=False)

sub3b = sub3.twinx()                                ← Creates axis on
sub3b.bar(x_ticks + bar_width, median_house,        other side of x
          width=bar_width, color='#a2b79e')
sub3b.set_xlim(100000, 500000)
sub3b.set_ylimits(np.arange(100000, 600000, 100000))
sub3b.set_yticklabels(['100k', '200k', '300k', '400k', '500k'])
sub3b.legend(['Housing'], loc='upper right', frameon=False)
```

ch07_data_analysis/07_other_charts.py

In this example, the main "new" concepts are the fact that there are two bars per data point (city) and that two axes are used on the left and right sides. Use the `twinx()` method to generate a new sub-axis that allows data to be plotted on the right side.

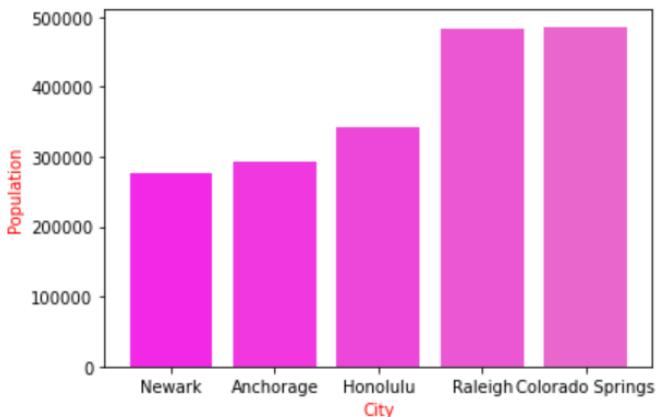
Note that there are two calls to `bar()` in this example to generate the dual bar graphs.



Your Turn! - Task 7-3

Using Matplotlib

- Continue working from Task 7-2
- Add an import for matplotlib.pyplot to your Notebook
- Create a bar plot of cities vs. population
 - Use `plt.bar()`
 - Use `x=` and `height=` for the parameters
 - Select the cities column and population column respectively





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

Common import syntax
We will refer to this module
as `pd` in further examples

```
import pandas as pd
```

Panel is deprecated as of version 0.20



Creating a 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	

```
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	

ch07_data_analysis/08_series.ipynb

A Pandas Series is based upon the NumPy array. NumPy does not have to be imported to create a Series, however.

A Series is a typical Python class; therefore, it follows conventional class usage rules. Keywords may be used when instantiating objects as shown in the top example. Similarly, a dictionary may be provided when creating a Series which uses the keys as the labels. If an index is provided along with the dictionary, the index will override the keys from the dictionary.



Accessing a Series

- Series allow access to values using methods similar to 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                                  # 619
ser4[['Chicago', 'Oklahoma City']]            # Chicago      619
                                                # Oklahoma City 1295

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

Mobile	218
San Diego	15
Chicago	619
New York City	13
Oklahoma City	1295
Name: City Elevations,	
dtype: int64	

ch07_data_analysis/09_accessing_series.ipynb

Accessing series appear to work very much like lists and dictionaries. They support access using get(), but they also support slice notation in a similar fashion to lists. Series can also use the axis labels as properties (as demonstrated with the ser4.Chicago example above).



Helpful Series Tools

- Several attributes and methods exist for learning about Series data

```
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]
```

ch07_data_analysis/09_accessing_series.ipynb

When accessing a Series (beyond what was mentioned on the previous slide), the index and values attributes as well as the describe() method can provide helpful information. Results of these are shown in the slide above.

Plotting Series

- Pandas' data structures support Matplotlib data plotting arguments

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

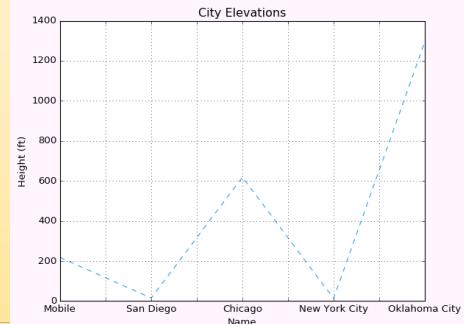
ser4 = pd.Series([218, 15, 619, 13, 1295],
                 index=['Mobile', 'San Diego', 'Chicago',
                         'New York City', 'Oklahoma City'],
                 name='City Elevations')

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



ch07_data_analysis/10_plotting.ipynb

Pandas supports plotting Series as part of the API. It uses the Matplotlib API, therefore all of the options discussed in the previous chapter are available as well. These options include kind ('bar', 'line', 'barh', 'hist', 'box', 'kde', 'density', 'area', 'pie'), ax, figsize, use_index, legend, grid, title, style, xticks, yticks, xlim, ylim, rot (rotation), label, fontsize, xerr, yerr, secondary_y, and more.

DataFrames

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

```
s = 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'])
```

The diagram illustrates the structure of a DataFrame. It shows a table with four rows labeled 'row0' through 'row3' and four columns labeled 'col0' through 'col3'. The cells contain numerical values from 0 to 15. Three callout boxes point to specific parts of the table: 'Index labels' points to the row indices ('row0'-'row3'), 'Column Labels' points to the column indices ('col0'-'col3'), and 'Values' points to the numerical entries in the cells.

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

DataFrames are the primary data structure of Pandas. They are essentially a Series of Series (similar to a dictionary of dictionaries). Each column can be a different data type and both rows and columns can be inserted and deleted.



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

ch07_data_analysis/11_dataframes.ipynb

The top half of the example shows a DataFrame created using two lists. The lists each form a row within the DataFrame. No index or column index is defined so positional numerical values (0, 1) and (0, 1, 2, 3, 4) are used automatically.

The bottom half of the example defines an index for the DataFrame. Notice how the index is used to define the rows.

Note: several displayed DataFrame results are actually taken from Jupyter output results.



Helpful DataFrame Tools

```

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

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

df4 = pd.DataFrame(data, index=index, columns=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

df4.to_numpy()

Returns a NumPy array containing just 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

ch07_data_analysis/12_dataframe_info.ipynb

There are lots of techniques for accessing and utilizing DataFrames.

If just the DataFrame values are desired, a NumPy array can be extracted from the DataFrame using the `values` attribute.

If statistical summary information is desired, use the `describe()` function which returns a new DataFrame.



More Helpful DataFrame Tools

df4.info()

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

ch07_data_analysis/12_dataframe_info.ipynb

The info() function can provide information about the structure of the DataFrame including column information and datatype information.

The T attribute can provide a transpose view of the DataFrame. It doesn't, however, modify the original DataFrame.



Accessing DataFrame Data Using 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

Placing a name in the brackets directly refers to a **column**

```
df5['col0']
```

row0	0
row1	4
row2	8
row3	12
Name:	col0, dtype: int32

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

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

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

	col2	col3
row2	10	11
row3	14	15

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

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

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

	col2	col3
row2	10	11
row3	14	15

ch07_data_analysis/13_indexing.ipynb

The two indexing properties shown here are similar but have some differences. iloc[index] only supports integer positional index values even if there are labels provided. The loc[index] is used with labels in the index.

Depending on what is selected, the results will either be a new Series, DataFrame, or individual value.

Sorting Using `sort_index()` or `sort_values()`

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()`

Reorders the index,
df4 is unchanged

	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

df6

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

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

Use `ascending=False` to sort from largest to smallest

Sorts the DataFrame on the
median_age column

ch07_data_analysis/14_sorting.ipynb

In the example in the middle of the slide, rows are reordered using the `sort_index()` method. Using `sort_values()`, as in the bottom example, will sort the DataFrame by the specified column label.

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 use the first line as a header

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

ch07_data_analysis/15_read_csv.py

`read_csv()` parameters are shown below:

```
pd.read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col,
            usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine,
            converters, true_values, false_values, skipinitialspace,
            skiprows, skipfooter, nrows, na_values, keep_default_na,
            na_filter, verbose, skip_blank_lines, parse_dates,
            infer_datetime_format, keep_date_col,
            date_parser, dayfirst, iterator, chunksize, compression,
            thousands, decimal, lineterminator, quotechar, quoting,
            escapechar, comment, encoding, dialect, tupleize_cols,
            error_bad_lines, warn_bad_lines, skip_footer, doublequote,
            delim_whitespace, as_recarray, compact_ints, use_unsigned,
            low_memory, buffer_lines, memory_map, float_precision)
```

Masking (Boolean Indexing) with DataFrames

- 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

ch07_data_analysis/16_boolean_indexing.py

The Python and operator has been written to take the left and right sides, convert them to Booleans first, and then evaluate the expression. This behavior can't be changed in Python. However, the & operator's behavior can be changed. The same is true for the or operation, use | instead.

Groupby Operations

- Data can be grouped and then operated on in those groups

A red circle highlights the 'state' column header in the original DataFrame. A yellow box contains the code `bystate = contacts.groupby('state')` and `bystate.size()`. A pink arrow points from this box to a green box containing the resulting grouped DataFrame.

	name	address	state	zip	area_code	phone
0	Bob Green	4517 Elm St. Riverside	NJ	8075	301	356-8921
1	Violet Smith	220 E. Main Ave Philadelphia	PA	9119	202	421-9008
2	John Brown	231 Oak Blvd. Black Hills	SD	82101	719	303-1219
3	Ed Blumenthal	3012 Briarwood Ln. Denver	CO	80101	719	422-8091
4	Rosey Englund	1818 Mockingbird Ln. Aurora	CO	82101	719	286-1920
5	Tori Gray	2218 Masengild Ave.	NJ	8075	301	338-6571
6	Lisa Black	89 Prince Dr. Philadelphia	PA	9119	202	419-0650
7	Tom Redford	2323 Nicholas St. Newark	NJ	7101	862	227-8022
8	Sally White	3345 Spruce Cir. Harrisburg	PA	17105	717	429-1217
9	Goldy Simpson	4430 Mountainside Creek Rd Custer	SD	57730	605	689-3131
10	O. Range	1703 Treeline Dr. Denver	CO	80101	719	429-1356
11	Sil Verna	557 Pine Ave Aurora	CO	82101		
12	Pinky Tuscadero	601 Sapling Blvd	NJ	8501		
13	Hazel Sanford	27 Musket Dr. Pittsburg	PA	15201		

	name	address	state	zip	area_code	phone	email	company	position
3	Ed Blumenthal	3012 Briarwood Ln. Denver	CO	80101	719	422-8091	ep20002@gmail.com	Hanibow & Delite	Programmer
4	Rosey Englund	1818 Mockingbird Ln. Aurora	CO	82101	719	286-1920	ke7001@yahoo.com	Wadlow Inc.	Administrative Lead
10	O. Range	1703 Treeline Dr. Denver	CO	80101	719	429-1356	ffnine27@hotmail.com	n/a	Retired
11	Sil Verna	557 Pine Ave Aurora	CO	82101	719	286-1920	sil@yahoo.com	Music Enthusiasts	Salesperson

ch07_data_analysis/17_grouping.ipynb

Using the groupby() feature, data can quickly be organized into operational segments. The result (return value) of a groupby() call is a DataFrameGroupBy object, not a new DataFrame.

The DataFrameGroupBy object has a limited number of methods that can be used: size(), get_group(), all(), any(), filter(), count(), cumprod(), cumsum(), idxmax(), nth(), nunique(), prod(), std(), var(), sum() to name a few.



head() and tail()

- Use `df.head(n=5)` or `df.tail(n=5)` to display the first 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, ...
```

`head()` or `tail()` can provide a partial view of a data structure. By default, each returns either the first or last five rows in the DataFrame.

Your Turn! - Task 7-4

Using Pandas

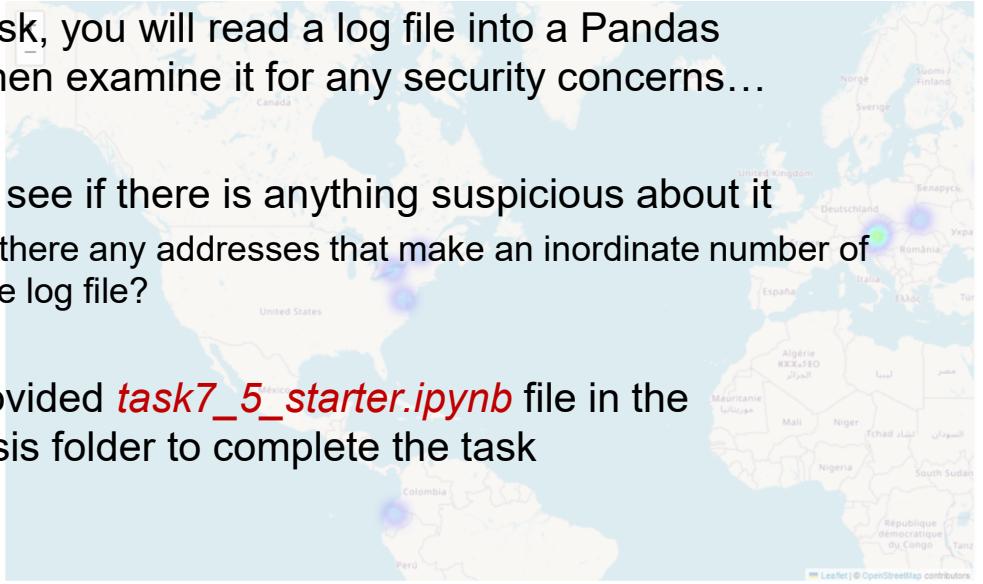
- In your existing notebook, import Pandas (in the import cell)
 - Load *population_data.csv* via Pandas' `read_csv()` method into a DataFrame
 - Use the file's headers as column names
1. Determine the shape
 2. Determine the average population using the `describe()` function
 3. Use masking to display only cities whose population is greater than 350,000



Your Turn! - Task 7-5 (*Optional, time-permitting*)

Analyzing a Log File

- In the following task, you will read a log file into a Pandas DataFrame and then examine it for any security concerns...
- We'll analyze it to see if there is anything suspicious about it
 - For example, are there any addresses that make an inordinate number of requests within the log file?
- Work from the provided **task7_5_starter.ipynb** file in the ch07_data_analysis folder to complete the task





Chapter 7 Summary

- NumPy provides a powerful array type, `ndarray`, that can be n-dimensions deep
 - Numerous operations manipulate, transform, and reshape data
 - The NumPy array is used as a foundational data structure in other frameworks
- Matplotlib is a foundational data visualization framework
- Pandas *Series* and *DataFrames* provide grouping, acquisition, and merging features beyond what NumPy supports
 - It makes data preprocessing a much easier task
 - These methods enable data cleaning capabilities that are largely unmatched within Python or many other languages for that matter



Chapter 8

Advanced Functions and Decorators



More Decorators Than You'll Know What To Do With



Chapter 8 - Overview

Closures

Decorators

Functional Programming Methods



Introducing Decorators

- Selling point: *what if we could modify someone's code without ever touching that original code?*
 - **Decorators** can do this (sort of)
- Decorators give their creators the ability to take control of someone else's code, modify it, and conform it to the way they would like it to behave
- *Decorators are a little tricky to wrap your head around,* so, we'll start by introducing some concepts related to functions...



Names of Functions are References

```
def some_func(val):
    print(val)

some_func('Calling the function.')

another_func = some_func
another_func('Calling the function.')
```

The name of a function is a reference to that function in memory

If we create another variable, it could be made to "point" to this function also

Calling the function.
Calling the function.

ch08_decorators/01_decorators.py

Functions are First-Class Objects

```
def increment(val, amount):
    return val + amount

def decrement(val, amount):
    return val - amount

def op(func, data, amt):
    result = func(data, amt)
    return result

print(op(increment, 5, 3))
print(op(decrement, 3, 2))
```

Functions can be passed
into other functions

8
1

ch08_decorators/02_decorators.py



Creating and Returning Functions

```
def display_width(width):
    def display(val):
        print(val[:width] + '...')
    return display

formatter = display_width(15)

data = 'This is a long string that will be truncated.'
formatter(data)
```

The display() function is created within another function

It also gets returned from the outer function

This is a long ...

ch08_decorators/03_decorators.py

Closures

```
from urllib.request import urlopen

def set_url(url):
    def load():
        return urlopen(url).read()
    return load

get_google = set_url('http://www.google.com')
results = get_google()
print(results)
```

load() is created and returned within set_url()

load() is called a **closure**

load() can "see" the variables of the outer function (like url)

ch08_decorators/04_decorators.py

Closures are functions defined within other functions. They have several interesting properties/benefits:

They provide a form of encapsulation (private-ness).

They can "see" the variables of the outer function.

They lead to the decorator pattern (discussed shortly).

Issues with Closures and *nonlocal*

- Closures can't easily modify variables of the outer functions

```
def outer_function():
    x = 3

    def my_closure():
        x += 1
        print(x)

    my_closure()

outer_function()
```

Error!

```
def outer_function():
    x = [3]

    def my_closure():
        x[0] += 1
        print(x[0])

    my_closure()

outer_function()
```

Works, but hacky!

```
def outer_function():
    x = 3

    def my_closure():
        nonlocal x
        x += 1
        print(x)

    my_closure()

outer_function()
```

Works!

ch08_decorators/05_decorators.py

The closure can access the outer function's variables; however, it cannot modify them if they are immutable. This is because it would be perceived as trying to "create" a new, local variable within the closure function.

The fix, while hacky, solves the dilemma of being unable to modify the variable of the outer function. It uses a "mutable" object whose contents can be modified without ever changing the reference to the original data structure.

nonlocal brings in the variable from the outer-scoped function, making it usable within our nested function. It provides the most elegant solution to our dilemma.

An Undecorated Function

```
def shortener(func):
    width = 15
    def wrapper(val):
        val = val[:width] + '...'
        func(val)
    return wrapper

def display(val):
    print(val)

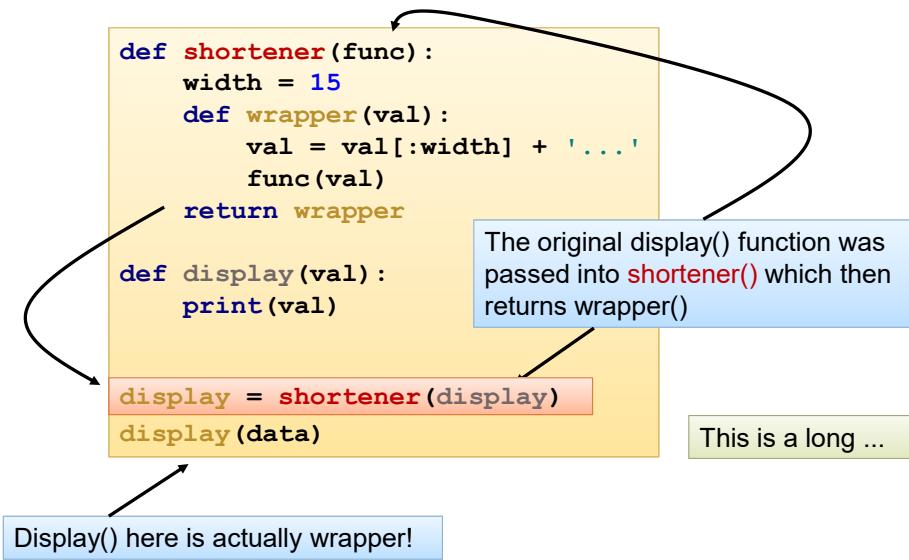
data = 'This is a long string that will be truncated.'
display(data)
```

This is a long string that will be truncated.

Despite the strange looking function at the top, there is nothing new happening in this example yet

ch08_decorators/06_decorators.py

Wrapping the Original Function



ch08_decorators/07_decorators.py

This example shows how when wrapper is returned from shortener, display receives the returned value and therefore "becomes" the wrapper function.

In effect, we have "swapped" out (replaced) the `display` function for `wrapper`.



The Decorator Notation

- The following notation is equivalent to the previous example

```
def shortener(func):
    width = 15
    def wrapper(val):
        val = val[:width] + '...'
        func(val)
    return wrapper

@shortener
def display(val):
    print(val)

data = 'This is a long string that will be truncated.'
display(data)
```

There it is! The decorator syntax in Python. This syntax is equivalent to:
`display = short_formatter(display)`

ch08_decorators/08_decorators.py

We finally arrive at the Python decorator notation.

The Python Decorator Pattern

```
def decorator(some_func):  
    def wrapper(name):  
        print('This is the wrapper.')  
        return some_func(name)  
    return wrapper
```

```
@decorator  
def display(name):  
    print(f'Hi, {name}')
```

With this notation, you never had access to the original function

```
display('Johnny')
```

This is actually calling `wrapper`

ch08_decorators/09_decorators.py

Improper Arguments

```
def shortener(func):
    width = 15
    def wrapper(val):
        val = val[:width] + '...'
        return func(val)
    return wrapper

@shortener
def display(val):
    print(val)

@shortener
def display_info(name, address):
    print(name, address)

data = 'This is a long string that will be truncated.'
display(data)
display_info('Johnny', '124 Main St.')
```

The decorator works fine here

Yikes!--these arguments do not match wrapper()
and will cause an error when this code runs

This is a long ...

TypeError: wrapper() takes 1 positional
argument but 2 were given

ch08_decorators/10_decorators.py

Decorators are usually meant to be applied to any function. This means the functions could have varying arguments. When we swap out the original function with the wrapper, the arguments may not match up properly.

Making Decorators More Flexible

```

def shortener(func):
    width = 15
    def wrapper(*args):
        arguments = []
        for arg in args:
            if isinstance(arg, str):
                arguments.append(arg[:width])
            else:
                arguments.append(arg)
        return func(*arguments)
    return wrapper

@shortener
def display(val):
    print(val)

@shortener
def display_info(name, address):
    print(name, address)

data = 'This is a long string that will be truncated.'
display(data)
display_info('Kiefer William Frederick Dempsey George Sutherland',
             '123 Chancellor Matheson Road')

```

Modifying wrapper() allows any positional arguments to be passed in

Any arguments that are strings are shortened, anything else is left as is

Our decorator now works for both of these functions

This is a long
Kiefer William 123 Chancellor

ch08_decorators/11_decorators.py

In this revised version, wrapper() has been refactored to support any number of positional arguments. This allows it to replace the display_info() function.

Even More Flexibility (1 of 2)

```
def shortener(func):
    width = 15
    def wrapper(*args, **kwargs):
        arguments = []
        for arg in args:
            if isinstance(arg, str):
                arguments.append(arg[:width])
            else:
                arguments.append(arg)

        key_args = {key: val[:width] for key, val in kwargs.items()
                    if isinstance(val, str)}

        return func(*arguments, **key_args)
    return wrapper

@shortener
def display(val):
    print(val)
```

wrapper() now accepts and works with any arguments

ch08_decorators/12_decorators.py

In this final version, our wrapper() now supports all positional and all keyword arguments. This makes it flexible enough to use with any function that we want to decorate.



Even More Flexibility (2 of 2)

```
@shortener
def display_info(name, address):
    print(name, address)

data = 'This is a long string that will be truncated.'
display(data)                                This is a long

display_info('Kiefer William Frederick Dempsey George Sutherland',
            address='123 Chancellor Matheson Road')
```

Keyword arguments are no problem now

Kiefer William 123 Chancellor

ch08_decorators/12_decorators.py

Five Examples of Decorators: #1

```
def trace(orig_func):
    def wrapper(*args, **kwargs):
        print(f'Calling: {orig_func.__name__}... ')
        return orig_func(*args, **kwargs)
    return wrapper

@trace
def func2(val):
    print(val)

@trace
def func1(val):
    func2(val)

func1(val='hello')
```

Calling: func1...
Calling: func2...
hello

ch08_decorators/13_decorators.py

This example will inform us when a function has been invoked. This version requires us to place the decorator above the function, but don't forget we can always use the alternate syntax:

```
orig = decorator(orig)
```

Five Examples of Decorators: #2

```
def count(func):
    call_count = 0

    def wrapper(*args, **kwargs):
        nonlocal call_count
        call_count += 1
        print(f'{func.__name__}, call #{call_count}')
        return func(*args, **kwargs)
    return wrapper

@count
def func1(greeting: str = 'hi, there'):
    print(greeting)

func1('hello')
func1(greeting='howdy')
func1('hey')
func1()
```

```
func1, call #1
hello
func1, call #2
howdy
func1, call #3
hey
func1, call #4
hi, there
```

ch08_decorators/14_decorators.py

This example tracks the number of times a function is called. Note the use of nonlocal in order to be able to modify the variable of the outer function (call_count).

Five Examples of Decorators: #3

```

from itertools import islice
from pathlib import Path
import time

def profile(orig_func):
    def wrapper(*args, **kwargs):
        start = time.time()
        ret = orig_func(*args, **kwargs)
        finish = time.time()
        print(f'{orig_func.__name__} took
              {(finish - start):.2f}sec')
        return ret
    return wrapper

@profile
def func1(filepath):
    with Path(filepath).open(encoding='utf-8') as f:
        for count, line in enumerate(islice(f, None, None, 2)):
            print(line.strip()[:-1])
    return count

print(f'{func1("../resources/access_.log")}\nlines read.')

```

This function merely opens the file and reads every other line from it, printing the lines out in reverse

func1 took 0.45sec
49999 lines read.

ch08_decorators/15_decorators.py

This decorator measures the start and end time when executing a function. func1() in this example will read every other line from the file (via islice), reverse the line, and print it. The decorator measures how long this will take.

The parameters for islice() are islice(iterator, start, stop, step).



Five Examples of Decorators: #4

```
class Transactional(object):
    def __init__(self, session):
        self.session = session

    def tx(self, func):
        def wrapper(*args, **kwargs):
            ret=None
            self.session.begin()
            try:
                ret = func(*args, **kwargs)
                self.session.commit()
            except Exception as err:
                print(err, file=sys.stderr)
                self.session.rollback()

            return ret
        return wrapper
```

Any method decorated with Transactional.tx will begin a transaction and then either commit() or rollback()

```
class Session:
    def begin(self):
        print('begin')

    def commit(self):
        print('commit')

    def rollback(self):
        print('rollback')
```

```
tx = Transactional(Session())
```

```
@tx.tx
def work():
    print('doing work')
work()
```

Replaces work with wrapper

ch08_decorators/16_decorators.py

The example demonstrates the use of a class that contains a method that holds the decorator function (tx). Within the decorator, a transaction is begun and then committed or rolled back depending on whether an exception occurs along the way. Notice the calling of func() before committing. func() refers to our original function (or work() in this case).

The begin/commit/rollback methods aren't real here, in fact, they come from the contrived Session class. This class simulates our transactional behavior.

Five Examples of Decorators: #5 (1 of 2)

```
def cache_call(orig_func):
    cache = {}

    def wrapper(*args):
        if args not in cache:
            ret = orig_func(*args)
            cache[args] = ret
            wrapper.misses += 1
        else:
            wrapper.hits += 1

        return cache[args]

    wrapper.hits = 0
    wrapper.misses = 0

    return wrapper
```

The example shows a decorator that behaves like a function-level cache

The `cache_call` decorator can be used on slow functions that are called frequently with the repeated parameter sets

These track how much the cache is used

ch08_decorators/17_decorators.py

When a function is called, the results are stored in a dictionary (the actual cache). Any time the function is called again, the cache can be checked to see if the arguments have already been used and if so, draw from the cache instead of calling the actual function. This type of solution can be useful anytime a function call is expensive (time consuming).

Five Examples of Decorators: #5 (2 of 2)

```
@cache_call
def func1(*args):
    return f'Calling {func1.__name__} with args: {args}'

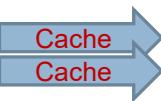
print(func1(1))
print(func1(1, 2, 3))
print(func1('hello'))
print(func1(1))
print(func1('hello'))
print(func1(1, 2))

print(f'Cache hits: {func1.hits}, cache misses: {func1.misses}')
```

Each of these calls is actually the wrapper

These will not call the original function,
instead they will draw from the cache

Calling func1 with args: (1,)
Calling func1 with args: (1, 2, 3)
Calling func1 with args: ('hello',)
Calling func1 with args: (1,)
Calling func1 with args: ('hello',)
Calling func1 with args: (1, 2)
Cache hits: 2, cache misses: 4



ch08_decorators/17_decorators.py

Our function, `func1()`, is called six times with two of those calls using repeated arguments. This causes the cache to be hit instead of the function getting called.

Using the *functools* Cache Decorator

```
from functools import lru_cache

@lru_cache
def func1(*args, **kwargs):
    return f'Calling {func1.__name__} with:{args} and {kwargs}.'

print(func1(1))
print(func1(1, 2, 3))
print(func1('hello'))
print(func1(1))
print(func1('hello'))
print(func1(1, 2))
print(func1(1, 2, 'hello'))
print(func1(1, val=2, item=3))
print(func1(1, val=2))
print(func1(1, val=2, item=3))
print(func1(1, item=3, val=2))
print(func1.cache_info())
```

The *functools* module provides cache decorator already called `@lru_cache()`

Calling func1 with: (1,) and {}.
 Calling func1 with: (1, 2, 3) and {}.
 Calling func1 with: ('hello',) and {}.
 Calling func1 with: (1,) and {}.
 Calling func1 with: ('hello',) and {}.
 Calling func1 with: (1, 2) and {}.
 Calling func1 with: (1, 2, 'hello') and {}.
 Calling func1 with: (1,) and {'val': 2, 'item': 3}.
 Calling func1 with: (1,) and {'val': 2}.
 Calling func1 with: (1,) and {'val': 2, 'item': 3}.
 Calling func1 with args: (1,) and {'item': 3, 'val': 2}.

`CacheInfo(hits=3, misses=8, maxsize=128, currsize=8)`

ch08_decorators/18_decorators.py

The cache decorator already exists in Python. It's called `lru_cache` and can be found in the *functools* module. It provides a `cache_info()` method for examining how much the cache is used. It is also parameterized so that it can identify how big the cache can be.



Can You Decorate a Decorator?

Yes!

```
def lowercaser(func):
    def wrapper(*args, **kwargs):
        arguments = []
        for arg in args:
            if isinstance(arg, str):
                arg = arg.lower()
            arguments.append(arg)

        kwargs = {key:val.lower() for key, val in kwargs.items()
                  if isinstance(val, str)}
        return func(*arguments, **kwargs)
    return wrapper

@lowercaser
@shortener
def display_info(name, address):
    print(name, address)

display_info('Kiefer William Frederick Dempsey George Sutherland',
            address='123 Chancellor Matheson Road')
```

It is possible to apply multiple decorators by "stacking" them

kiefer william 123 chancellor

ch08_decorators/19_decorators.py

It is legal for a decorator to "decorate" an already decorated function. This will first decorate the original function with dec2, then it replaces dec2's wrapper with dec1's wrapper function.



Can a Class Be a Decorator?

Yes!

Implement the `__call__` magic method, which serves as the "wrapper"

```
from numbers import Number

class check_numeric:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        retval = 'Invalid argument supplied. Must be numeric.'

        check_args = all(isinstance(arg, Number) for arg in args)
        check_kwargs = all(isinstance(val, Number)
                           for val in kwargs.values())
        if check_args and check_kwargs:
            retval = self.func(*args, **kwargs)

        return retval

@check_numeric
def sum(x, y):
    return f'Result: {x + y}'

print(sum(10, 3.5))
print(sum(0, 'Johnny'))
```

Here, our `check_numeric` **decorator** verifies if arguments to a function are numeric

Result: 13.5
Invalid argument supplied. Must be numeric.

ch08_decorators/20_decorators.py

The `__call__` method makes a class appear "invokable". When a class decorator is used, the `__init__` is still called to initialize the new object created, but the `__call__` method actually becomes the "wrapper" function. `@dec` replaces the original function with the `__call__` version.



Can You Decorate a Class?

Yes!

```

def dec(cls):
    orig_init = cls.__init__      Save the class' original constructor

    def __init__(self, *args, **kwargs):
        print('doing something before')
        orig_init(self, *args, **kwargs)
        print('doing something after')

    cls.__init__ = __init__        Replace the class's
                                constructor with a new one

    return cls                   Return the modified
                                class containing the
                                modified constructor

@dec
class Foo(object):
    def __init__(self):
        print('original init')

Foo()

```

A great example of this is the `@dataclass` decorator

do something first
original init
do something last

ch08_decorators/21_decorators.py

This example uses a function decorator to decorate a class. It does this by passing the class to be decorated into the function, saving the class's original constructor, and changing the constructor to a new function (that invokes the original internally).



Can Decorators Take Arguments? (1 of 2)

Yes!

It is possible to create decorators that accept arguments

```
class output_formatter:
    def __init__(self, mask=None, allcaps=False):
        self.mask = mask
        self.allcaps = allcaps

    def __call__(self, f):
        def wrapper(*args):
            if self.allcaps:
                args = tuple([arg.upper() for arg in args])
            if self.mask:
                args = (args[0], '*****') + args[2:]

            retval = f(*args)
            return retval
        return wrapper
```

Arguments are now passed into the decorator class which will show up in the class `__init__()`

The wrapper can "see" the `allcaps` and `mask` attributes via the `self` parameter

ch08_decorators/22_decorators.py

This example illustrates the use of a class decorator that accepts arguments. A mask and allcaps argument can be supplied when using `output_formatter()`. The solution technically uses a wrapper around a wrapper. The function called `wrapper` "wraps" the original function, while the function called `__call__` wraps that `wrapper` function. The `__call__` function provides instance variables that are visible to the `wrapper` function, while the `wrapper` function can replace and invoke the original function. Think of it as two layers of wrapping to achieve the effect.



Can Decorators Take Arguments? (2 of 2)

The arguments are available in the
__call__() method through self

```
@output_formatter(mask=True, allcaps=True)
def my_func(first, last):
    print('{fn} {ln}'.format(fn=first, ln=last))

@output_formatter(False, allcaps=False)
def my_func2(first, last):
    print('{fn} {ln}'.format(fn=first, ln=last))

my_func('Bill', 'Smith')
my_func2('Bill', 'Smith')
```

BILL ****
Bill Smith

ch08_decorators/22_decorators.py



The logging Module

- Use the logging module to log output to a file (or elsewhere)
 - Import `logging` for this purpose
 - It provides many basic logging capabilities
 - Formatting of output
 - Log levels
 - Multiple output handlers
 - Set your log filename logging level
(DEBUG, INFO, WARNING, ERROR, CRITICAL)

```
import logging
logging.basicConfig(level=logging.DEBUG,
                    filename='/temp/logfile.log')
logging.debug('Logging started.')
```

Module-level logger used here

ch08_decorators/23_logging.py



Additional Configuration

- To create a module-level logger

```
my_logger = logging.getLogger(__name__)
my_logger.setLevel(logging.DEBUG)

handler1 = logging.FileHandler('c:/temp/logfile.log')
formatter=logging.Formatter(fmt='%(asctime)s %(name)-12s
                                %(levelname)-8s %(message)s',
                            datefmt='%m-%d %H:%M:%S')
handler1.setFormatter(formatter)

my_logger.addHandler(handler1)
```

Logger named after
the module name

Use a *RotatingFileHandler* instead of a *FileHandler* to
configure how to swap out log files that grow large

ch08_decorators/23_logging.py



Loading Config from a File

- Loggers may load configuration from a file

```
[loggers]
keys=root

[handlers]
keys=consoleHandler,
fileHandler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=consoleHandler,
fileHandler

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stdout,)
```

```
import logging
import logging.config

logging.config.fileConfig('./logging.ini',
                        disable_existing_loggers=True)

logging.info('Loaded from ini.')
```

[handler_fileHandler]
class=FileHandler
level=DEBUG
formatter=formatter
args=('demo_config.log',)

[formatter_formatter]
format=%(asctime)s - %(name)s - \
%(levelname)s - %(message)s

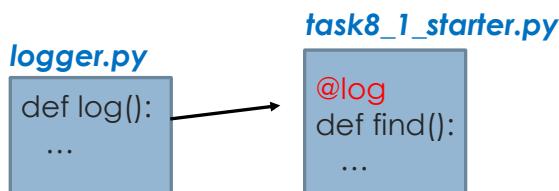
ch08_decorators/24_more_logging.py and logging.ini

In addition to an .ini configuration, python allows config using a dictionary, JSON data, or YAML data.



Your Turn! - Task 8-1

- Modify the database access app from Task 5-1
- Within the provided **logger.py** module, in the **ch8_decorators/starter** folder:
 - Configure a logger (basicConfig is sufficient)
 - Create a decorator in the same module called **log**
 - Import the **log** decorator into your main application file
 - Add the decorator above the **find()** method within your main application file (**task8_1_starter.py**)





Chapter 8 Summary

- Functions in Python are first-class objects
- They have special capabilities that include
 - Support for **functional programming** techniques
 - The ability to be **nested** in other functions
 - Being **passed into and returned** from functions
 - **Decorating** other functions
 - Becoming **iterable** by **yielding** (making it a generator)

Course Summary



What did we learn?

Virtual Environments	Pandas
Review of Fundamentals	Matplotlib
Python DB API 2.0	<i>collections</i>
SQLAlchemy	<i>itertools Iterators</i>
Instances as Dictionaries	<i>argparse</i>
Static Methods	Closures
Magic Methods	Decorators
super()	Functional Programming
Inheritance	Generators
Multiple Inheritance	Generator Expressions
Jupyter Notebooks	Dict & Set Comprehensions
NumPy	<i>logging</i>
API Fundamentals, Flask	WSGI
Marshmallow	

What's After This?

Intensive Advanced Python

Review of Intermediate Topics
NoSQL and Non-relational Databases
Advanced Context Managers
Automating Tools
PDF Creation
Email Creation / Sending
Machine Learning Fundamentals
Fast API
Django
Packaging and Distribution
Threads and Multiprocessing
Asynchronous I/O

Evaluations

- Please take the time to fill out an evaluation
- All evaluations are read and considered

Thank you for your feedback which is critical to this process.

Questions



Intensive Intermediate Python

Exercise Workbook

Task 1-1

Creating a Virtual Environment



Overview

This exercise is designed to demonstrate how to create a virtual environment using **venv**. We'll also configure the virtual environment within PyCharm. This virtual environment becomes the Python used throughout the remainder of the course.



Survey Your System

Begin by opening a terminal (OS X) or command (Windows) window. From here onward, we'll refer to it simply as a terminal.

Within the terminal, type each of these commands:

```
python -v
```

```
python3 -v
```

```
python3.12 -v
```

 (replace 3.12 with the appropriate version number, e.g., python3.11, python3.10, etc.)

One of these commands should have responded with a valid version number of Python that you were expecting. **Remember this command because you will use it throughout the rest of the course.** Later in the course, we'll refer to the command: **python**, but your personal command may be python3 or python3.x! So, remember it!

Check the location of the python interpreter using: **which** (use **where** on Windows). Type:

OS X: **which python** (again, this may be python3, python3.10, etc.)

Windows: **where python**

View the results. The first entry will be the Python interpreter that will be invoked by default.

Also view your path environment variable. Type the following:

OS X: **echo \$PATH**

Windows: **echo %PATH%**

Examine the entry on your PATH that identifies the location of the Python interpreter that you just listed.



Create the Virtual Environment

Locate your student files directory. In the terminal window, cd (change directory) to this location. This is where we'll create the virtual Python directory.

```
cd <path_to_student_files>
```

Within the student files directory, type the command:

```
python -m venv venv
```

(where **venv** becomes the name of the directory that will be created within the student files)

Within a few seconds the virtual directory will be created.

Examine the new **venv** directory. Look inside the **bin** (or **Scripts**, on Windows) directory. Notice the reference to the interpreter. Look within the **lib** directory and then within the **site-packages** directory.

There's not much here yet. Type the following:

```
pip list
```

You should see a list of installed packages from your main Python interpreter (again, **use pip3 or pip3.12, etc.**, to mirror your Python command).

Type:

OS X: **which python**

Windows: **where python**

Notice that the same (main or original) Python interpreter is listed first. The virtual one doesn't appear in the list. Not yet, at least...



Activate the Environment, Test with *pip*

For your new Python environment to take effect, it must be activated. Do this by changing to the **venv/bin** (or **venv/Scripts** on Windows) directory. Then type:

OS X: **source ./activate.sh**

Windows: **.\activate** (or just **activate**)

Now, type:

OS X: **which python** (again, remember to type your specific Python version if needed, e.g., python3 or python3.12)

Windows: **where python**

Notice how the directory containing your new Python interpreter is listed first on your PATH environment variable now. This is what activation does!

Type:

pip list (as we did a moment earlier).

This should now list only *setuptools* and *pip*.

Type:

pip -v (revealing you are working within your virtual environment)!

Next, we'll install several modules used during this course. These are used later, so we need to install them. First, try installing a single item. Type:

pip install prettytable

Perform a **pip list** again. It should be there now, and you can see the installed module in your site-packages directory.

Now let's attempt to install all needed packages during this course. **cd** to the student files directory and type the following command:

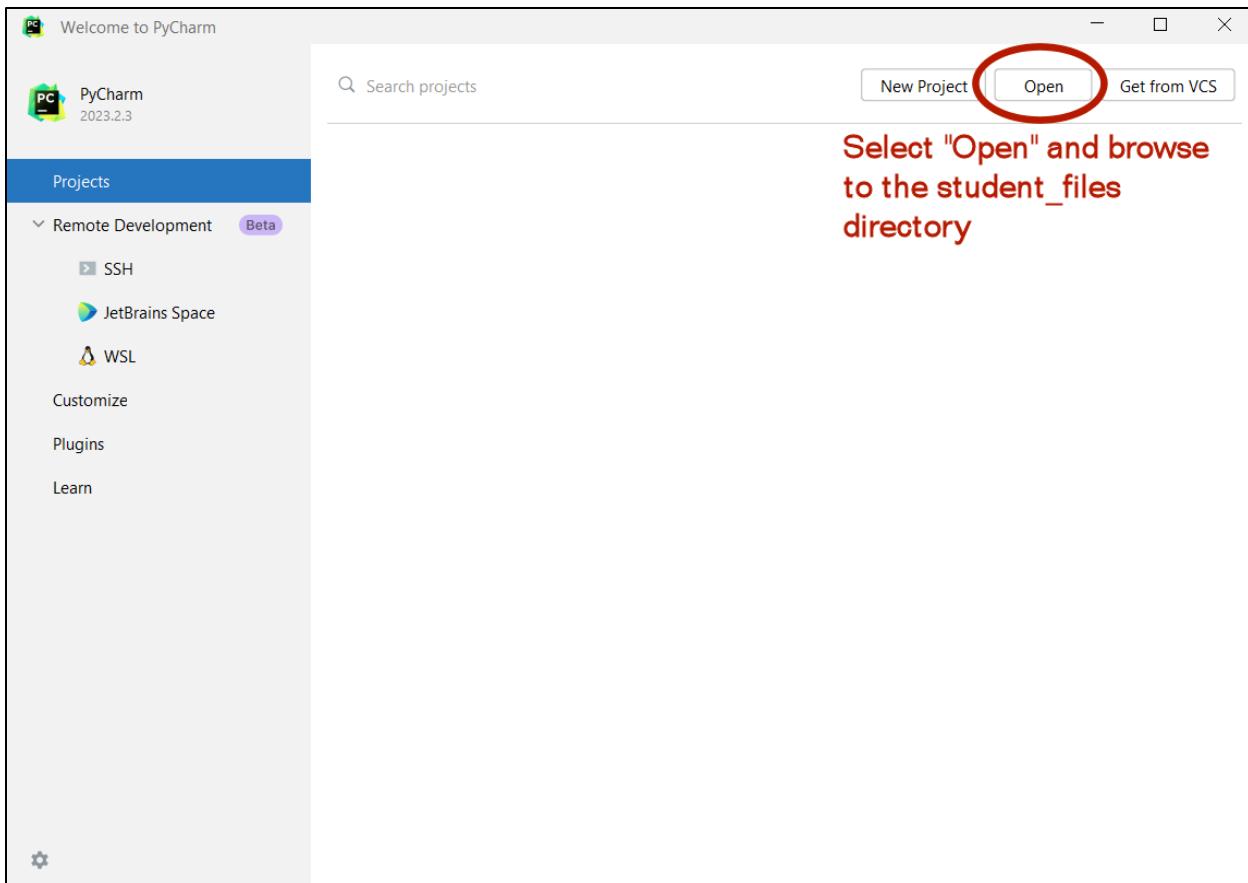
```
pip install -r requirements.txt
```

After a short moment, all packages should be installed. Don't worry about a warning that the pip module has a later version available.

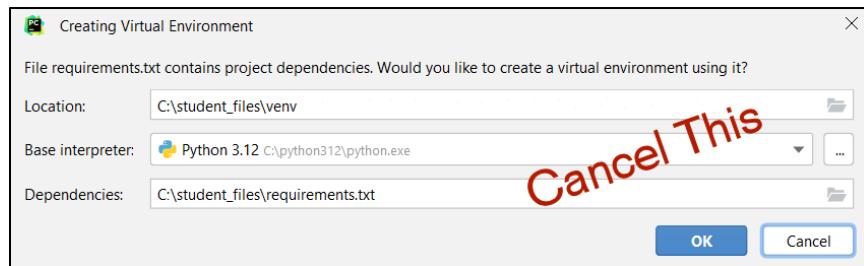


Connect the Environment to PyCharm

With the newly created virtual environment, we'll now connect it to PyCharm. **Open the PyCharm IDE**. At the initial "Welcome" dialog, PyCharm may prompt you to select a project directory. **Choose "Open"** and **browse to the student files** directory (see the following screenshot).



If prompted to create a new virtual environment (shown below)—cancel this as you've already done it in the previous steps.

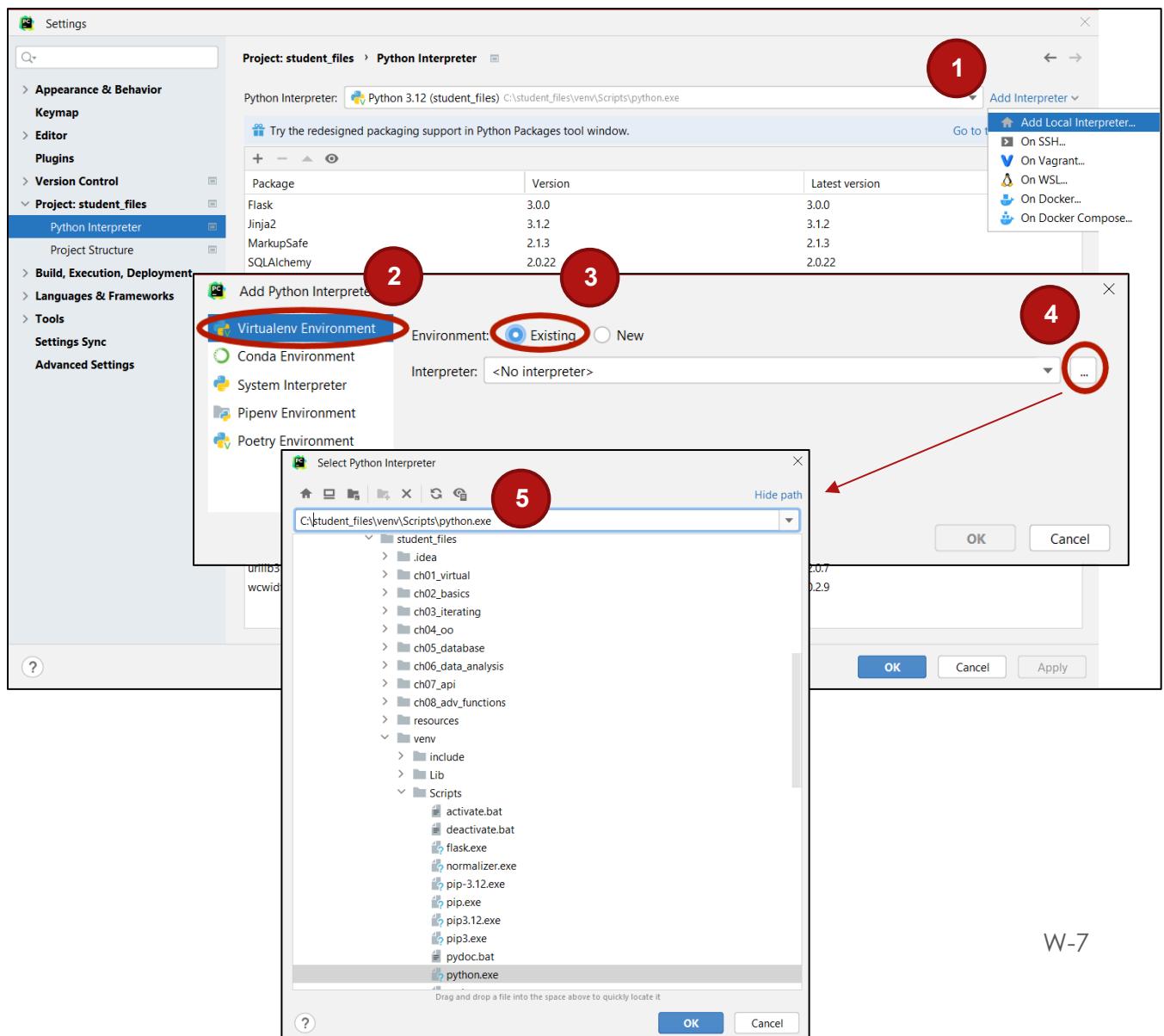


Once within the main project, select the settings menu as follows:

OS X: **PyCharm > Settings**
(some versions: PyCharm > Preferences)

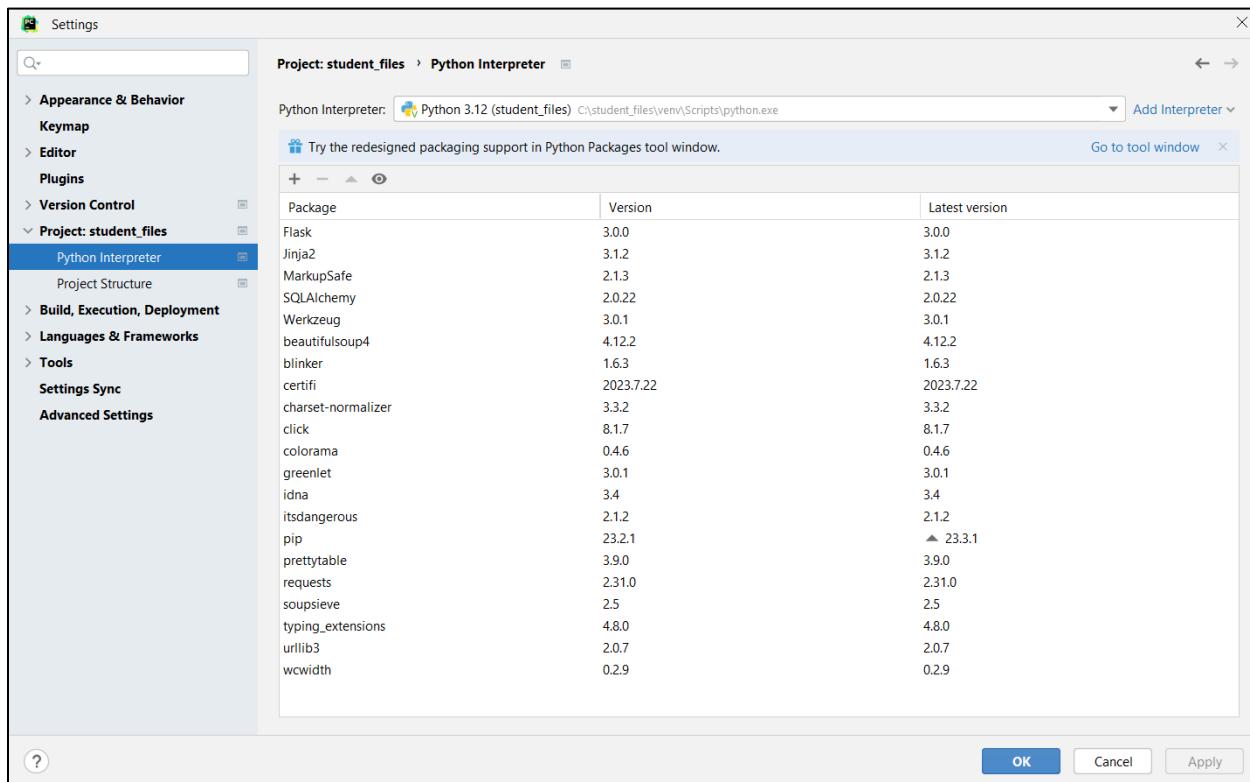
Windows: **File > Settings**

Within the settings, expand the **Project: student_files** item. Select **Project Interpreter**. On the right side, we'll have to point PyCharm to the newly created virtual environment. Do this as shown below:



- 1- Select the setting (cog wheel) symbol or "Add Interpreter" item
- 2- Leave selected "Virtualenv Environment" option
- 3- Select "Existing" Environment radio button
- 4- Select the "..." (Ellipsis) button (far to the right)
- 5- Browse to the new Interpreter location (python.exe on Windows, python or python3.x on OS X)

Click Ok twice to return to the packages screen. You should see prettytable and other packages listed now. Click Ok again.



Note: To **test out the prettytable** module, run the *01_prettytable.py* file in the ch01_virtual folder. Do this by right clicking the *01_prettytable.py* file and selecting **Run '01_prettytable'**.

That's it!

Task 2-1

Python Overview



Overview

This exercise is designed to review basic Python concepts and fill in gaps in introductory topics. In this exercise, you will read from the file **cities15000.txt**. In the follow-on tasks, you will determine the highest and most populous cities. Later, you will add a search capability such that when a user inputs a city name, the results will yield a list of cities that match the search. Finally, we'll look at making the solution more reusable.

Note: **cities15000.txt** is a tab separated value (TSV) file.



Create the City Named Tuple Definition

Work from **task2_1_starter.py** found in the **ch02_basics** directory. Examine the file **cities15000.txt** found in the resources folder. You will use a typed NamedTuple to store one record. Create a NamedTuple as follows:

```
header = [('name', str), ('population', int),
           ('elevation', int), ('country', str)]
City = NamedTuple('City', header)
```



Read from the File

Before reading from the file, use a Path object to wrap the working_dir variable. Change the provided line from

```
working_dir = '../resources'
```

to

```
working_dir = Path('../resources')
```

Then create a Path() object to the file itself. Change the following line from

```
city_data = 'cities15000.txt'
```

to

```
city_data = working_dir / 'cities15000.txt'
```

Now, use a **with** control to open the file. Read a record from the file. Don't forget to split it on a tab (not a comma). Read columns 1 (name), 8 (country), 14 (population), and 16 (elevation) from the split record. Values should be placed into a City NamedTuple and then appended into the cities list. The following will accomplish this:

```
header = [('name', str), ('population', int),
           ('elevation', int), ('country', str)]
City = NamedTuple('City', header)

with city_data.open(encoding='utf-8') as f:
    for line in f:
        fields = line.strip().split('\t')
        name = fields[1]
        country_code = fields[8]
        population = int(fields[14])
        elevation = int(fields[16])
```

```
cities.append(City(name, population, elevation,
                    country_code))
```

Finally, place exception handling around the `with` control to handle any file read errors.

```
try:

    # your with control from above goes here

except IOError as err:
    print(f'Error: {err}', file=sys.stderr)
    sys.exit(1)
```

Test it out and verify that no errors arise.

That's it for now. We'll revisit this exercise shortly.

Task 2-2 Finding Largest Elevation and Population

Python Overview



Overview

This is a continuation of the previous exercise (Task 2-1) in which you already read from a file and captured the elevation and population data of cities around the world.

This time, you should determine the highest city as well as the most populous city. You may continue from your previous solution (`task2_1_starter.py`) or you can optionally work from the `task2_2_starter.py` file which is a completed version of Task 2-1.



Get the Largest City

To get the largest city, you can use the `sort()` method or `sorted()` function. Both versions use `key=` to determine how to sort. The `population` attribute in each `City` NamedTuple can be used to sort the sequence. By sorting from largest to smallest, the largest population will be the 0th record.

```
header = [('name', str), ('population', int), ('elevation', int), ('country', str)]
City = NamedTuple('City', header)

try:
    with city_data.open(encoding='utf-8') as f:
        for line in f:
```

```

        fields = line.strip().split('\t')
        name = fields[1]
        country_code = fields[8]
        population = int(fields[14])
        elevation = int(fields[16])
        cities.append(City(name, population, elevation,
country_code))
    except IOError as err:
        print(f'Error: {err}', file=sys.stderr)
        sys.exit(1)

print(f'{len(cities)} cities read.')

largest = sorted(cities, reverse=True
                  key=lambda city: city.population,) [0]

```



Get the Highest City

Can you repeat the process from step 1 in a similar way to determine the highest city (city with the highest elevation)?

Use a lambda similar to the one used in step 1 except use the elevation attribute of the City NamedTuple instead.

Note: (Optional) Results are in meters, if you wish to view results in feet, multiply the value by 3.28.



Display the Results for Steps 1 and 2

Display the results from the previous steps.

```

print(f'Largest city: {largest.name}, {largest.country} with:
      {largest.population:,} people')
print(f'Highest city: {highest.name}, {highest.country} at:
      {highest.elevation} meters ({highest.elevation * 3.28}
      feet)')

```



(Optional) Solve It Using the max() Function

The **max()** function could be used to retrieve the results as well. Can you do it? Hint: Pass the cities list and use key= much the same way as steps 1 and 2. The result is a named tuple.

```
print(max(cities, key=lambda city: city.population,  
          default=None) .name)  
print(max(cities, key=lambda city: city.elevation,  
          default=None) .name)
```

Test out your solution. We'll return to this task again shortly!

Task 2-3 Add a Search Function Capability

Python Overview Overview



This is a continuation of the previous exercise (Task 2-2) in which you already determined the city with the highest elevation and largest population.

This time, you should create a capability for searching for cities by inputting the city name (or partial name). You will also convert the code that found the highest and largest cities into functions. You may continue from your previous solution (`task2_1_starter.py` or `task2_2_starter.py`) or you can optionally work from `task2_3_starter.py` now.



User-Defined City Population Search

Prompt the user to supply the name (or partial name) of a city.

```
search_term =  
    input('Enter the (partial) name of the city: ')
```



Create a Search Function Definition

Create a function that takes the value of the search term.

```
def search(term):  
    ...to be filled in below...
```

Lowercase the user's input and then use this to search the list of named tuples. Save the matched results for display.

```
def search(term):  
    results = [city for city in cities  
              if term.casefold() in city.name.casefold()]
```



Display the Search Results

Within the function, display the search results in a similar fashion to how it was shown on the slide. Here we used Python's expand operator with .format() for convenience since f-strings can't be used with this (*) operator.

```
def search(term):  
    results = [city for city in cities if term.casefold()  
              in city.name.casefold()]  
  
    if results:  
        print('{0:<35}{1:>15}{2:>15}{3:>10}'  
              .format(*[h[0] for h in header]))  
  
        for city in results:  
            print('{0:<35}{1:>15,}{2:>15,}{3:>10}'  
                  .format(*city))  
    else:  
        print('No cities found.')
```



Invoke the Search Function

Invoke the search function passing the search term into it.

```
def search(term):
    results = [city for city in cities if term.casefold()
               in city.name.casefold()]

    if results:
        print('{0:<35}{1:>15}{2:>15}{3:>10}'
              .format(*[h[0] for h in header]))

        for city in results:
            print('{0:<35}{1:>15,}{2:>15,}{3:>10}'
                  .format(*city))
    else:
        print('No cities found.')

search(search_term)
```



Convert Part 2 Solution into Functions

Refactor the solution from part 2 so that the code that calculates the largest population and highest elevation are functions. You may alternately use the max() for this step instead of sorted.

We've used the max() function for this step. Call each function and display the results.

```
# replace the part 2 code with the code below...

def largest():
    return max(cities, key=lambda city: city.population)

def highest():
    return max(cities, key=lambda city: city.elevation)

print(f'Largest city: {largest().name}.')
print(f'Highest city: {highest().name}.')
```

That's it--test it out!

Task 2-4

Functions and Modules



Overview

This exercise continues from Task 2-3. In this exercise, you will refactor the previous solution by refactoring functions and moving them into a module called **search.py**. The **task2_4_starter.py** file serves as the starting point. You will move functions from the **task2_4_starter.py** file. At the end of this task, no new functionality is introduced. **Task2_4_starter.py** currently runs and **task2_4_test.py** should be usable at the end of this exercise.



Move the Three Functions

Open **ch02_basics/task2_4_starter/search.py**. You will be moving items into this file.

Open **ch02_basics/task2_4_starter/task2_4_starter.py**. You will be moving items from this file.

Begin by moving the `largest()`, `highest()`, and `search()` functions into `search.py`. Don't move the `input()` function statement.

After the move, your `search.py` will look like the following:

```

import sys
from pathlib import Path
from typing import NamedTuple

working_dir = Path('..../resources')
city_data = working_dir / 'cities15000.txt'
cities = []

header = [('name', str), ('population', int),
          ('elevation', int), ('country', str)]
City = NamedTuple('City', header)

def largest():
    return max(cities, key=lambda city: city.population)

def highest():
    return max(cities, key=lambda city: city.elevation)

def search(term):
    results = [city for city in cities
               if term.casefold() in city.name.casefold()]

    if results:
        print('{0:<35}{1:>15}{2:>15}{3:>10}'
              .format(*[h[0] for h in header]))
        for city in results:
            print('{0:<35}{1:>15,}{2:>15,}{3:>10}'
                  .format(*city))
    else:
        print('No cities found.')

```



Move Code Related to Reading Data

Move the code related to reading the data from the file into this module. There is no need to move the associated print() function. Place it after the NamedTuple definition. You should grab all code beginning from the try block until the end of the except block.

Afterwards, your search.py will look partially like this:

```
import sys
from pathlib import Path
from typing import NamedTuple

working_dir = Path('..../resources')
city_data = working_dir / 'cities15000.txt'
cities = []

header = [('name', str), ('population', int),
           ('elevation', int), ('country', str)]
City = NamedTuple('City', header)

try:
    with city_data.open(encoding='utf-8') as f:
        for line in f:
            fields = line.strip().split('\t')
            name = fields[1]
            country_code = fields[8]
            population = int(fields[14])
            elevation = int(fields[16])
            cities.append(City(name, population,
                               elevation, country_code))
except IOError as err:
    print(f'Error: {err}', file=sys.stderr)

def largest():
    return max(cities, key=lambda city: city.population)
```



Rename Variable Names

Use PyCharm to refactor variable names. Change the following names:

cities	=>	_items
city	=>	item
City	=>	Item

To change these in PyCharm, select the variable name, right-click on it, and then choose Refactor...

Type the new name and hit Enter.

After refactoring, your solution will look like the following:

```
from typing import NamedTuple
from pathlib import Path
import sys

working_dir = Path('.../../resources')
city_data = working_dir / 'cities15000.txt'
_items = []

header = [('name', str), ('population', int),
          ('elevation', int), ('country', str)]
Item = NamedTuple('Item', header)
```

```

try:
    with city_data.open(encoding='utf-8') as f:
        for line in f:
            fields = line.strip().split('\t')
            name = fields[1]
            country_code = fields[8]
            population = int(fields[14])
            elevation = int(fields[16])
            _items.append(
                Item(name, population, elevation, country_code))
except IOError as err:
    print(f'Error: {err}', file=sys.stderr)

def largest():
    return max(_items, key=lambda item: item.population)

def highest():
    return max(_items, key=lambda item: item.elevation)

def search(term):
    results = [item for item in _items
               if term.casifold() in item.name.casifold()]

    if results:
        print('....'.format(*[h[0] for h in header]))
        for item in results:
            print('....'.format(*item))
    else:
        print('No cities found.')

```



Combine largest() and highest()

Create a function that can replace both largest() and highest() and supports any other field that can use max().

Replace both functions with:

```
def most(field_name):
    return max(_items,
               key=lambda item: getattr(item, field_name))
```



Modify the search() Function



Modify search() to accept a second parameter that indicates the field name to search on. Use getattr() (as in step 4) to make search() work properly.

```
def search(term, by):
    results = [item for item in _items
               if term.casifold() in getattr(item, by).casifold()]

    if results:
        print('...'.format(*[h[0] for h in header]))
        for item in results:
            print('...'.format(*item))
    else:
        print('No cities found.')
```



Test It Out!

Run the task2_4_test.py file which will import search and test out your solution.

Task 2-5

Creating the read_data() Function



Overview

This exercise continues from Task 2-4. In this exercise, you will refactor the previous solution by rewriting the code that reads the data from the cities15000.txt file. You will create a new function, called `read_data()` that generically reads most data files and places data into a list of NamedTuples. You will begin by modifying `search.py`.



Remove Unneeded Code

Open `ch02_basics/task2_5_starter/search.py`. We will remove code we no longer need for this task.

Remove the code related to reading data from the cities15000.txt file. Remove lines 96-109 from this file. Also, remove lines 91-92.

The following code should be removed:

```
working_dir = Path('..../resources')
city_data = working_dir / 'cities15000.txt'

header = [('name', str), ('population', int),
          ('elevation', int), ('country', str)]
Item = NamedTuple('Item', header)

try:
    with city_data.open(encoding='utf-8') as f:
        for line in f:
            fields = line.strip().split('\t')
```

```

        name = fields[1]
        country_code = fields[8]
        population = int(fields[14])
        elevation = int(fields[16])
        _items.append(Item(name, population,
                           elevation, country_code))
    except IOError as err:
        print(f'Error: {err}', file=sys.stderr)

```

Note: don't remove the following (line 94):

```
_items = []
```



Create the read_data() Function Definition

Below the `_items = []` statement, create the function definition as shown on the slide. Add the following function definition:

```
_items = []
```

```
def read_data(filename, data_fields, sep=','):
    ... (to be completed next steps)...
```

```
def most(field_name):
    return max(_items,
               key=lambda item: getattr(item, field_name))
```



Dynamically Create the NamedTuple

Within the function, create a NamedTuple definition based on the provided data_fields parameter:

```
_items = []

def read_data(filename, data_fields, sep=','):
    _items.clear()

    attributes = []
    for key, val in data_fields.items():
        typ = val[1] if isinstance(val, tuple) else str
        attributes.append((key, typ))
    Record = NamedTuple('Record', attributes)
```



Start Reading Data from the File

Within a 'with' control, read line-by-line from our data file. strip() and split() the fields based on the provided sep= value.

```
def read_data(filename, data_fields, sep=','):
    ...

    try:
        with open(Path(filename), encoding='utf-8') as f:
            for line in f:
                data_values = line.strip().split(sep)
                ... (to be completed next step) ...
    except IOError as err:
        print(f'Error reading from file. Err: {err}', file=sys.stderr)
        sys.exit(1)
```



Read Data into NamedTuples

Complete reading the data into Record() NamedTuples at the location marked "(to be completed next step)" above. For each of the data_fields supplied, get its column number (col) and type (typ). Perform a type cast. Add the value into a list of desired values. Use that list to create the Record NamedTuple. Add it to our_items.

```
try:  
    with open(Path(filename), encoding='utf-8') as f:  
        for line in f:  
            data_values = line.strip().split(sep)  
            desired_values = []  
            try:  
                for value in data_values.values():  
                    col_num, typ = (value[0], value[1])  
                    if isinstance(value, tuple) else (value, str)  
                    desired_values.append(  
                        typ(data_values[col_num]))  
                record = Record(*desired_values)  
                _items.append(record)  
            except (TypeError, ValueError):  
                pass  
    except IOError as err:  
        print(f'Error reading from file. Err: {err}',  
              file=sys.stderr)  
        sys.exit(1)
```

(note: some lines above, due to their length) could not be written without adding line breaks).



Clean Up the search() Function's Output

Replace the print statements in the search() function. These were fixed for the City NamedTuple. But now we have variable sized columns due to unknown data field sizes.

Replace the if ... else ... at the bottom of the search() function with the following:

```
if results:
    column_format = ''
    padding = 7
    for i in range(len(results[0]._fields)):
        max_width = max([len(str(record[i])) for record in results])
        column_format += f'{{{{i}}:<{max_width + padding}}} '
    print(column_format.format(*results[0]._fields))
    for item in results:
        print(column_format.format(*item))
```



Test It Out!

Run the task2_5_test.py file which will import the search module and test the functions in your solution.

Task 3-1

Using the collections Module & Counter



Overview

In this exercise, you will use the **collections.Counter** class to determine which country has the most cities (with a population over 15000). Work from the provided **task3_1_starter.py** file. Locate and open this file now.



Add the Country field to the cities List

Read records from the file adding the country column value for each city record to a list. You will need to create this list variable.

```
cities = []

with open(datafile, encoding='utf-8') as cities_file:
    for line in cities_file:
        cities.append(line.strip().split('\t')[8])
```



Calculate Country with Most 15k+ Cities

Use the collection module's Counter class to find the most common country in the list.

Display your results.

```
most_common = Counter(cities).most_common(10)
print(most_common)
```

That's it!

Task 3-2

Using Generators



Overview

This exercise refactors the previous exercise (Task 3-1). It will modify the previous solution by incorporating a generator function. The generator function will be used to read the country code.



Create and Utilize the Read File Generator

Work from either your completed task3_1_starter.py file or you may work from the provided task3_2_starter.py file.

Modify the previous code that read from the cities15000.txt. **Move it into a function.** Have this function **yield** the country_code. Try this on your own first, if you get stuck, look to the next page for a possible solution.

Once complete, invoke the generator by passing it into the Counter() constructor. Again, try this on your own before referring to the next page.

```
def country_generator(filename):
    with open(filename, encoding='utf-8') as cities_file:
        for line in cities_file:
            yield line.strip().split('\t')[8]
```

Invoke the generator and run it as follows:

```
most_common =
Counter(country_generator(datafile)).most_common(5)
print(most_common)
```



Test It Out

No other changes should be needed. You should be able to test out the solution now.

That's it!

Task 3-3

Using argparse



Overview

This exercise will incorporate the argparse module for use with your Task 3-2 solution. It adds the ability determine how many "mosts" to get as a command line argument. Your previous solution should be refactored to accept the following two command-line syntaxes now:

`python task3_3_starter.py -c 5`

or

`python task3_3_starter.py --count 5`

Note: you may work from your task3_2_starter.py solution if you wish or you may use the provided task3_3_starter.py file.



Define argparse Arguments

Import the argparse module. Create the solution to allow -c and --count command-line arguments. You should create a function called get_args() as follows:

```
def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', '--count',
                        default='3', type=int,
                        help='Num countries to retrieve')
    return parser.parse_args()
```



Invoke the `get_args()` Function

Invoke the previously defined function to retrieve values:

```
args = get_args()
```



Use the Arguments to Display Results:

Using the `args` object from step 2, use it to display <count> number of results. Do this as follows by modifying your final lines of code to use `args.count`:

```
most_common = Counter(country_generator(datafile))
               .most_common(get_args().count)

print(most_common)
```

That's it! Test out your solution.

Task 4-1

Python DB API 2.0 and Data Classes



Overview

This exercise works with a pre-defined database for access to the school data. It also uses data classes as an underlying data holder.



Create the School Data Class

Define a School dataclass as shown:

```
from dataclasses import dataclass

@dataclass
class School:
    name: str
    city: str
    state: str
```



Create a Function to Access the Database

Create a function to open a connection, query the database and return a list of dataclass records as follows:

```
def get_location(school_name: str) -> list[School]:
    results = []
    try:
        # work with database (next step)

    except sqlite3.Error as err:
        print(f'Error working with database: {err}',
              file=sys.stderr)
    return results
```

Within the try block, connect to the database, and allow rows to be accessed by their name. We'll use the `closing()` function of the `contextlib` module as discussed in the chapter notes.

```
with closing(sqlite3.connect(data_sourcefile)) as conn:  
    # this is completed as described below...
```

Above the function call, define our SQL string:

```
SELECT_SCHOOLS_SQL =  
    'SELECT fullname, city, state FROM schools WHERE fullname like ?'  
  
def get_location(school_name: str) -> list[School]:  
    results = []
```

Within the `with` control, obtain the cursor and perform the query:

```
def get_location(school_name: str) -> list[School]:  
    results = []  
  
    with closing(sqlite3.connect(data_sourcefile)) as conn:  
        cursor = conn.cursor()  
        cursor.execute(SELECT_SCHOOLS_SQL,  
                        (f'%{school_name}%',))
```

Iterate over the cursor, instantiate School data class object instances from the results, add the data class to the results:

```
with closing(sqlite3.connect(data_sourcefile)) as conn:
    cursor = conn.cursor()
    cursor.execute(SELECT_SCHOOLS_SQL,
                   (f'%{school_name}%',))
    for sch in cursor:
        results.append(School(*sch))
```

Prompt the user for a school name (or partial name) and query the results by calling get_location():

```
def get_location(school_name: str) -> list[School]:
    results = []
    with closing(sqlite3.connect(data_sourcefile)) as conn:
        cursor = conn.cursor()
        cursor.execute(SELECT_SCHOOLS_SQL,
                       (f'%{school_name}%',))
        for sch in cursor:
            results.append(School(*sch))

    return results

search_name = input('School name (or partial name): ')
results = get_location(search_name)

print(f'Matches for {search_name}:')
for school in results:
    print(f'{school.name} ({school.city}, {school.state}))')
```

Test your results.

As a reference, below is an example of our completed get_location() function:

```
SELECT_SCHOOLS_SQL =
    'SELECT fullname, city, state FROM schools WHERE fullname like ?'

def get_location(school_name: str) -> list[School]:
    results = []
    try:
        with closing(sqlite3.connect(data_sourcefile)) as conn:
            cursor = conn.cursor()
            cursor.execute(SELECT_SCHOOLS_SQL,
                           (f'%{school_name}%',))
            for sch in cursor:
                results.append(School(*sch))
    except sqlite3.Error as err:
        print(f'Error working with database: {err}', file=sys.stderr)

    return results
```

Task 5-1

Python Classes



Overview

This exercise creates two classes: a School and a SchoolManager class. Both classes are to be created within the task5_1_starter.py file. The SchoolManager class will define two methods: `__init__()` and `find()` and will be used to perform a search against an SQLite database. The database file is called `course_data.db` and is found within the `ch06_oo` chapter. The schools table can be found within this database.



Create the School Class

Within `ch05_oo/task5_1_starter.py` create a School class that contains a `school_id`, `fullname`, `city`, `state`, and `country` attributes. All fields are strings and may optionally be annotated as such.

```
class School:
    def __init__(self, school_id, fullname, city, state,
                 country):
        self.school_id = school_id
        self.fullname = fullname
        self.city = city
        self.state = state
        self.country = country

    def __str__(self):
        return f'{self.fullname} ({self.city}, {self.state})'

    __repr__ = __str__
```



Examine the SchoolManager Class

Examine the SchoolManager class that has been started for you already:

```
class SchoolManager:  
    SELECT_SCHOOLS_SQL = 'SELECT school_id, fullname, city,  
                          state, country FROM schools WHERE column like ?'  
  
    PERMITTED_SEARCH_COLUMNS =  
        ['fullname', 'city', 'state', 'country']
```

Notice the SQL strings are defined at the class level. Also, notice the WHERE clause contains a variable (column) that will need to be replaced by one of the values in the PERMITTED_SEARCH_COLUMNS list. This is to prevent SQL injection.



Build Upon the SchoolManager Class

Add the following methods to the SchoolManager class: `__init__()`, and `find()` as follows:

```
class SchoolManager:  
    ...  
    def __init__(self, db_filename):  
        pass  
  
    def find(self):  
        pass
```



Complete the `__init__()` Method

Remove the `pass` statement, pass in the database filename and save it.

```
class SchoolManager:  
    ...  
    def __init__(self, db_filename):  
        self.db_filename = db_filename  
  
    def find(self):  
        pass
```



Build Out the `find()` Method

Remove the `pass` statement within `find()`. Add (and return) a list representing the results that we will eventually get. Of course, at this point, the list will be empty.

Complete the `find()` method signature, passing the 3 required items as shown below. `value` represents the search term. `Column` represents the column we will search on. `sort_by` represents the column to sort on. Each are strings and may optionally be annotated as such.

```
class SchoolManager:  
    ...  
    def __init__(self, db_filename):  
        self.db_filename = db_filename  
  
    def find(self, value, column='fullname',  
            sort_by='fullname'):  
  
        results = []  
  
        # to be filled in next  
  
        return results
```



Complete the Database Interaction

Within `find()`, check the column name provided by the user to see if it is in the list of valid column names. Again, this is to prevent SQL inject attacks, only "approved" column names can be supplied.

```
def find(self, value, column='fullname',
        sort_by='fullname'):

    results = []

    if column in self.PERMITTED_SEARCH_COLUMNS:
        select_schools_sql =
self.SELECT_SCHOOLS_SQL.replace('column', column)

    return results
```

Then, connect to and query from the database. This part will be similar to what we encountered in `task4_1_starter.py`:

```
def find(self, value, column='fullname',
        sort_by='fullname'):

    results = []

    if column in self.PERMITTED_SEARCH_COLUMNS:
        select_schools_sql =
self.SELECT_SCHOOLS_SQL.replace('column', column)

        with closing(sqlite3.connect(self.db_filename))
                as conn:
            cursor = conn.cursor()
            params = (f'%{value}%',)

            cursor.execute(select_schools_sql, params)

            for record in cursor:
                results.append(School(*record))

    return results
```



Sort Results within the find() Method

Sort the results according to the `sort_by` parameter of the `find()` method.

```
def find(self, value, column='fullname',
        sort_by='fullname'):

    results = []

    if column in self.PERMITTED_SEARCH_COLUMNS:
        select_schools_sql =
            self.SELECT_SCHOOLS_SQL.replace('column', column)

        with closing(sqlite3.connect(self.db_filename))
            as conn:
                cursor = conn.cursor()
                params = (f'%{value}%',)

                cursor.execute(select_schools_sql, params)

                for record in cursor:
                    results.append(School(*record))

    if sort_by in self.PERMITTED_SEARCH_COLUMNS:
        results.sort(key=lambda s: vars(s).get(sort_by))

    return results
```



Test it Out!

Run the module. The `if`-block of code at the bottom of the module should execute successfully.

Task 6-1

Flask



Overview

This exercise will incorporate a web-based server using the Flask framework. It will return a list of schools given a search term sent from a browser. The search functionality in the browser incorporates the `search()` function from the previous exercise. Begin by locating the `ch06_api/starter/task6_1` folder.



Import the schools.py Module

Locate `app.py` within `ch06_api/starter/task6_1` and import the `SchoolManager` class from the `schools.py` module. Use the following:

```
from schools import SchoolManager
```

or

```
from ch06_api.starter.task6_1.schools import SchoolManager
```



Call the find() Method

At the location specified for step 2 within `app.py`, instantiate the `SchoolManager` and call the `find()` method as shown. Get the returned school objects and convert them into a list of school strings.

```
results = [str(school) for school in SchoolManager(database)
           .find(school_name)]
```



Return JSON-based Results

As part of the return statement of the get_schools() method, invoke the jsonify() method passing two keyword arguments into it: school_name and schools.

```
@app.route('/api/schools/<school_name>', methods=['GET'])
def get_schools(school_name):
    try:
        results = [str(school) for school in
                   SchoolManager(database).find(school_name)]
    except Exception as err:
        results = err.args

    return jsonify(schools=results, school_name=school_name)
```

To test out your solution, first run app.py and then browse to <http://localhost:8051> in your browser.

Task 6-2

A Client to the Flask Server: *requests*

Overview



This exercise requires making an HTTP GET request to our running Flask server (Task 6-1), retrieving JSON data and parsing it into a Python dictionary.



Make Sure the Server is Running

Ensure the Flask server from the previous exercise is still running. Do this by running the `app.py` file within the `ch06_api/starter/task6_1` folder. If you didn't complete Task 6-1 or your server won't run, you may use the `app.py` file provided in the `solutions` folder instead.



Finish the `make_request()` Method

Open `ch06_api/starter/task6_1/client.py`. The URL for making requests has been passed into the `make_request()` function. Provide it to a `requests.get()` call, as shown:

```
def make_request(url: str) -> dict:  
    resp = requests.get(url)
```

Within the function, display a few other attributes of the response: `url`, `headers`, `text`, and `status_code`. Return a dictionary of the response by using `resp.json()`. The following shows a possible way to write the `make_request()` function:

```
def make_request(url: str) -> dict:  
    resp = requests.get(url)  
  
    print('Using: ', resp.url)  
    print('\nThe Raw JSON response values:\n', resp.text)  
    print('\nThe HTTP status:', resp.status_code)  
    print('\nThe response headers:\n', resp.headers)  
  
    return resp.json()
```



Call the make_request() Function

Call the function above that you just wrote at the location marked for step 3.

```
school_name = click.prompt('Enter a school name',  
                           default='Loyola')  
url = f'http://localhost:8051/api/schools/{school_name}'  
  
results = make_request(url)
```



Obtain and Print the Results

Take the results returned from the make_request() function call and obtain the matching schools from within. Do this by using the dictionary get() method and supply the key of 'schools' which gives

back a list. Print this list any way desired. The solution below uses PrettyTable to print the list:

```
school_name = click.prompt('Enter a school name',
                           default='Loyola')
url = f'http://localhost:8051/api/schools/{school_name}'

results = make_request(url)

print(f'\nResults for: {results.get("school_name")}')
schools = results.get('schools', [])

if schools:
    pt = PrettyTable(['School'])
    pt.align['School'] = 'l'
    for school in schools:
        pt.add_row([school])
    print(pt)
```

That's it!



Test it Out!

To test it, run the client.py file. Verify correct matching values are returned.

Task 6-3

Supporting Query String Parameters



Overview

This exercise combines both the server (created in Task 6-1) and the client (created in the previous task, Task 6-2). This time we will add the ability to submit the column name to search on from our data and the column name to sort by. This will involve modifying the client to be able to send query string parameters and the server to receive them.

Note: If using PyCharm, shut down the server running from Task 6-1. Do this by pressing the red button on the left as shown below:

```
Run: app ×
C:\student_files\venv\Scripts\python.exe C:\student_files\ch06_api\starter\task6_1\app.py
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://localhost:8051
Press CTRL+C to quit
```



Modify the make_request() Function

Work from the `ch06_api/starter/task6_3/client.py` file. Enable the ability to send request parameters in the `requests.get()` call by passing in a dictionary to the `make_request()` function and then passing that into the `requests.get()` call.

```
def make_request(url: str, params: dict = None) -> dict:
    resp = requests.get(url, params=params)

    print('Using: ', resp.url)
    print('\nThe Raw JSON response values:\n', resp.text)
    print('\nThe HTTP status:', resp.status_code)
    print('\nThe response headers:\n', resp.headers)

    return resp.json()
```



Modify the Call to make_request()

Modify the call to the make_request() function. Pass in a params dictionary. Set keys called 'column' and 'sort_by' in this dictionary.

```
url = f'http://localhost:8051/api/schools/{search_val}'

search_column = 'state'
sort_column = 'city'

results = make_request(url,
params={'column': search_column, 'sort_by': sort_column})
```



Import the request object from Flask

Open the `app.py` file. Update your imports by bringing in the `request` object from Flask.

```
from flask import Flask, render_template, jsonify,
Response, request
```



Modify the Route on the Server to Accept Parameters

Obtain the parameters sent by the client using the Flask request object.

```
@app.route('/api/schools/<search_val>', methods=['GET'])
def get_schools(search_val: str) -> Response:

    search_column = request.args.get('column')
    sort_by = request.args.get('sort_by')

    try:
        results = [str(school) for school in
                   SchoolManager(database).find(
                       search_val,
                       column=search_column,
                       sort_by=sort_by)]
    except Exception as err:
        results = err.args
```

That's it!



Test it Out!

To test it, ensure you have started (or restarted) your app.py file (the server). Next, run the client.py file. Verify correct matching values are returned.

Task 6-4

Integrating SQLAlchemy & Marshmallow



Overview

In this exercise, you will use a slightly revised School Search Flask Application to integrate the SQLAlchemy and Marshmallow frameworks. Work from the provided **ch06_api/starter/task6_4** folder. You only need to modify **app.py** within this location.

Note: If using PyCharm, shut down the server running from the previous tasks (such as Task 6-1 or Task 6-3). Do this according to the following screen shot:

The screenshot shows a PyCharm terminal window titled "Run: app". The output shows the following text:
Run: app
C:\student_files\venv\Scripts\python.exe C:\student_files\ch06_api\starter\task6_1\app.py
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on <http://localhost:8051>
Press CTRL+C to quit



Integrate the Plugins with Flask

Work from the provided **ch06_api/starter/task6_4/app.py** file. Integrate the plugins by passing the app object into each of them as shown:

```

app.config['SQLALCHEMY_ECHO'] = True

db = SQLAlchemy(app)
ma = Marshmallow(app)

PERMITTED_SEARCH_COLUMNS =
    ['fullname', 'city', 'state', 'country']

```



Create a Model

Our model is a single School class. Uncomment the provided class in the code and then have it inherit from a parent class called `db.Model`.

```

class School(db.Model):
    __tablename__ = 'schools'
    school_id = db.Column(db.String(40), primary_key=True)
    fullname = db.Column(db.String(40))
    city = db.Column(db.String(40))
    state = db.Column(db.String(40))
    country = db.Column(db.String(40))

    def __init__(self, school_id: str, fullname: str,
                 city: str, state: str, country: str):
        self.school_id = school_id
        self.fullname = fullname
        self.city = city
        self.state = state
        self.country = country

    def __str__(self):
        return f'{self.fullname} ({self.city}, {self.state})'

    __repr__ = __str__

```



Create the Marshmallow Schema

The Schema is used to define how to convert Python objects to JSON format. Complete the schema as follows:

```
class SchoolSchema(ma.Schema):
    class Meta:
        fields = ('school_id', 'fullname', 'city', 'state', 'country')
```



Complete the SQLAlchemy Code within the Route

Within the route, add SQLAlchemy code to query the database. Use the following:

```
try:
    schools = db.session.execute(db.select(School)
                                .filter(getattr(School, column)
                                        .ilike(f'%{search_val}%'))
                                .order_by(getattr(School, sort_by)))
    .scalars()

    results = school_schema.dump(schools)
    status = 200
except Exception as err:
    results = err.args
    status = 400
```



Examine the Client, Run It

To test it, first examine the code within the `client.py` file. It is nearly the same as before with just a few variable names changed. Also, the output format is changed slightly to accommodate the fact that objects returned are not just simple strings any longer.

Run `client.py`.

That's it!

Task 7-1

Using Jupyter



Overview

This exercise is broken into three short parts numbered 7-1 through 7-3. Each part allows for continuing from the end of the previous part.



Create a Function to Load the Data File

For this exercise, open the provided notebook called **task7_1_starter.ipynb**. Do this from within the Jupyter dashboard. Look over the newly opened notebook.

The following function is only a suggestion and there are multiple ways to write it. In the cell marked “Step 1,” write the `read_data()` function to read data from the specified file as shown:

```
def read_data(filepath: str):
    data = []
    with Path(filepath).open(encoding='utf-8') as f:
        f.readline()
        for line in f:
            city, state, year, population = line.strip().split(',')
            data.append((city, state, int(year), int(population)))
    return data
```



Call the Function

In the cell marked “Step 2,” call the function you just created passing in the provided filename variable. Output its return value:

```
results = read_data(filename)  
results
```

That's it--run the notebook and verify the data is loaded!

Task 7-2

Using NumPy



Overview

While you can continue working from the notebook you created previously (in Task 7-1), you may also use the provided `task7_2_starter.ipynb` file. Answer the questions from the slide related to this task.



Create the NumPy Array

In the cell marked “Step 1,” create a NumPy array out of the data from the last task. Be sure to use `dtype=object` as a parameter when creating the array!

```
city_pop = np.array(results, dtype=object)  
city_pop
```



Shape, Size, Number of Dimensions

Find the shape, size, and number of dimensions of our data array created in the previous step. In the cell marked “Step 2,” within the notebook (if you are using `task7_2_starter.ipynb`), add:

```
f'Shape: {city_pop.shape}, size: {city_pop.size},  
num. dims: {city_pop.ndim}'
```

This exact output format is not critical and any output that displays the requested values will work.



Calculate the Average Population

In the cell marked “Step 3,” select the population column, and then perform `mean()` on it.

```
city_pop[:, 3].mean()
```



Determine How Many 2020 Records

In the cell for step 4, create a mask that checks that the third column is equal to the value of 2020. Pass that mask into the array. Get the first value of the shape property.

```
mask = city_pop[:, 2] == 2020  
city_pop[mask].shape[0]
```



Display Records with Populations > 350k

Finally, in the cell for step 5, create a mask that checks that the fourth column is greater than 350,000. Pass that mask into the array.

```
mask = city_pop[:, 3] > 350_000  
city_pop[mask]
```

That's it! Fix any typos and test it out by running the notebook!

Task 7-3

Plotting Data



Overview

In this exercise, you will use Matplotlib to render the population data against the city names as a bar plot. While you can continue working from the notebook you worked on previously (Task 7-1 or task 7-2), a notebook with instructions is provided for you. You may use the provided **task7_3_starter.ipynb** file.



Create the Bar Plot

Using Matplotlib, create the desired plot:

```
plt.bar(x=city_pop[:, 0],  
        height=city_pop[:, 3],  
        color=['#f327e6', '#f137e0', '#ed47da',  
               '#eb57d3', '#e967cc'])  
plt.xlabel('City', color='red', fontsize=16)  
plt.ylabel('Population', color='red', fontsize=16);
```

Note: the added references to color are not required, they just add some flare to the look and feel.

That's it for all three parts! Test out your whole solution.

Task 7-4

Using Pandas and DataFrames



Overview

In this exercise, you will repeat the loading of the **population_data.csv** accomplished in Tasks 7-1 through 7-3. This time, however, you will exclusively use Pandas and instead of reading the file data manually, you will use Pandas **read_csv()**.



Load the Data Using `read_csv()`

Work from the provided **task7_4_starter.ipynb** notebook (that you created in Task 7-1). Place the following statements into the cell marked “Step 1” within the notebook:

```
population_data = pd.read_csv(filepath)  
population_data.shape
```

Display the DataFrame in another cell.

```
population_data
```



Calculate the Population Average

In the “Step 2” cell, calculate the average city population using the `describe()` function. There are several ways to do this.

```
population_data.describe().loc['mean', 'population']
```



Filter the City by Population

In the Step 3 cell, display only cities whose population is greater than 350,000.

```
mask = population_data.population > 350_000  
population_data[mask]
```

That's it!

Task 7-5

Using Pandas: Working with a Log File



Overview

This exercise corresponds to the file in `ch07_data_analysis` called `task7_5_starter.ipynb`. With your Jupyter server already running, open this file within your browser and follow the instructions within the notebook. The steps listed below correspond to the steps within the notebook and provide quick answers to the cell prompts.

Note: Steps may have 1-3 cells associated with them. So, values to be inserted into each cell are separated by a dash ----- below.



Load the Data Using `read_csv()`

Perform the following in the 3 cells provided for step 1:

```
log = pd.read_csv(filepath, sep=r'\s+', low_memory=False,  
                  usecols=[0, 3, 5, 6, 7, 9],  
                  names=['addr', 'req_date', 'path',  
                         'status', 'size', 'browser'])
```

```
log.shape
```

```
log.head()
```

```
log.info()
```



Viewing Unique Status Values

Complete the following statements in the two cells provided for "Step 2."

```
log.status.unique()
```

```
log.status.value_counts()
```



Convert the Date Column Format

Complete the following statements in the three cells provided for "Step 3" to convert the *req_date* column to a datetime.

```
log.req_date = pd.to_datetime(log.req_date,  
                               format='[%d/%b/%Y:%H:%M:%S'])
```

```
log.head()
```

```
log.info()
```



Examine the Most Frequent IP Addresses

Complete the following statements in the two cells provided for “Step 4” to group and plot the addresses found in the log file.

```
top10_addrs = log.groupby('addr').size().nlargest(10)  
top10_addrs
```

```
top10_addrs[::-1].plot(kind='barh');
```



Create a New ‘Method’ Column

Complete the following statements in the two cells provided for “Step 5” to create a column related to the HTTP request method.

```
log['method'] = log.path.apply(lambda cell:  
cell.split()[0])
```

```
log.head()
```



Determine and Plot Top 10 Post Requests

Complete the following statements in the two cells provided for “Step 6” to create a column related to the most common IP addresses making the most POST HTTP requests.

```
mask = log.method == 'POST'  
  
top10_posts = log[mask].groupby('addr').size().nlargest(10)  
  
top10_posts
```

```
top10_posts[::-1].plot(kind='barh', color=['#d09a33']);
```



Obtain Geolocation Info on Addresses

Complete the following statements in the three cells provided for “Step 7” to determine information about the location of IP addresses from the file.

```
geoip = GeoIP(ip_addr_lookup_file)  
geoip.record_by_addr('149.56.83.40')
```

```
results = log.addr.apply(get_location)
```

```
locations.head()
```



Merge the Two DataFrames

Complete the following statements in the one cell provided for "Step 8" to merge the original (log) and locations DataFrames.

```
merged = pd.concat([log, locations], axis=1)  
merged.head()
```



Plot the Top 10 Cities on a 'barh' and World Map

Complete the following statements in the three cells provided for "Step 9" to plot the top cities on both a 'barh' chart and on a world map using the Folium framework.

```
top10_cities = merged.groupby('city').size().nlargest(10)
top10_cities[::-1].plot(kind='barh');
```

lat_long

```
map = Map(location=[40, -60], tiles="OpenStreetMap",
          zoom_start=3)
hm = plugins.HeatMap(lat_long, radius=15, blur=10)
hm.add_to(map)
map
```

That's it!

Task 8-1

Logging and Decorators



Overview

This exercise will incorporate the logging module as well as utilize decorators. In this task, you will create a **@log** decorator and then apply it to our earlier Task5-1 (equivalent) exercise. The exercise has been reproduced in this chapter folder, so you do not need to return to the earlier chapter folder to work on it.



Define the Logger

Open the `ch8_decorators/starter/logger.py` file. Define a `basicConfig()` logger:

```
logging.basicConfig(filename='./logfile.log',
                    level=logging.DEBUG)
```



Create a @log Decorator

In the same file, create a function that will serve as the decorator. Call it `log()`.

Within the decorator, add a function that will log whenever a function (that is decorated) is called. You can write a simple decorator:

```

def log(orig_func):
    def wrapper(*args, **kwargs):
        ret = orig_func(*args, **kwargs)
        logging.info(f'{orig_func.__name__}() called.')
    return wrapper

```

or you can process and log the arguments to the call (optional):

```

def log(orig_func):
    def wrapper(*args, **kwargs):
        ret_val = orig_func(*args, **kwargs)

        arguments = []
        for arg in args:
            if not isinstance(arg, str):
                arguments.append(type(arg).__name__)
            else:
                arguments.append(arg)

        for kw, val in kwargs.items():
            if not isinstance(val, str):
                arguments.append(type(val).__name__)
            else:
                arguments.append(f'{kw}={val}')

        logging.info(f'{orig_func.__name__}() called. \
                    Arguments: {" ".join(arguments)} . \
                    Return value: {ret_val}')
    return ret_val
    return wrapper

```



Import the Decorator into the Starter File

As part of our last steps, we'll apply the decorator we created. First, we need to import it. Open the

`ch08_decorators/starter/task8_1_starter.py` file. Near the top of the file, add the import as shown:

```
from logger import log
```



Decorate the find() Method within schools.py

Locate the `find()` method and apply the decorator.

```
@log
def find(self, value: str, column: str = 'fullname', sort_by:
str = 'fullname'):
    results = [
        ...
    ]
```

Test out your solution by running the `task8_1_starter.py` file directly.

Look for a log file that gets created, called `logfile.log` in the same directory. Open and view the log file.

That's it!