TEKsystems

Our people make IT possible.

TEKsystems Education Services

presents

Intensive Intermediate Python

20750 Civic Center Drive
Suite 400, Oakland Commons II
Southfield, MI 48076
800.294.9360

COURSE CODE IN1468 / 3.25.2022

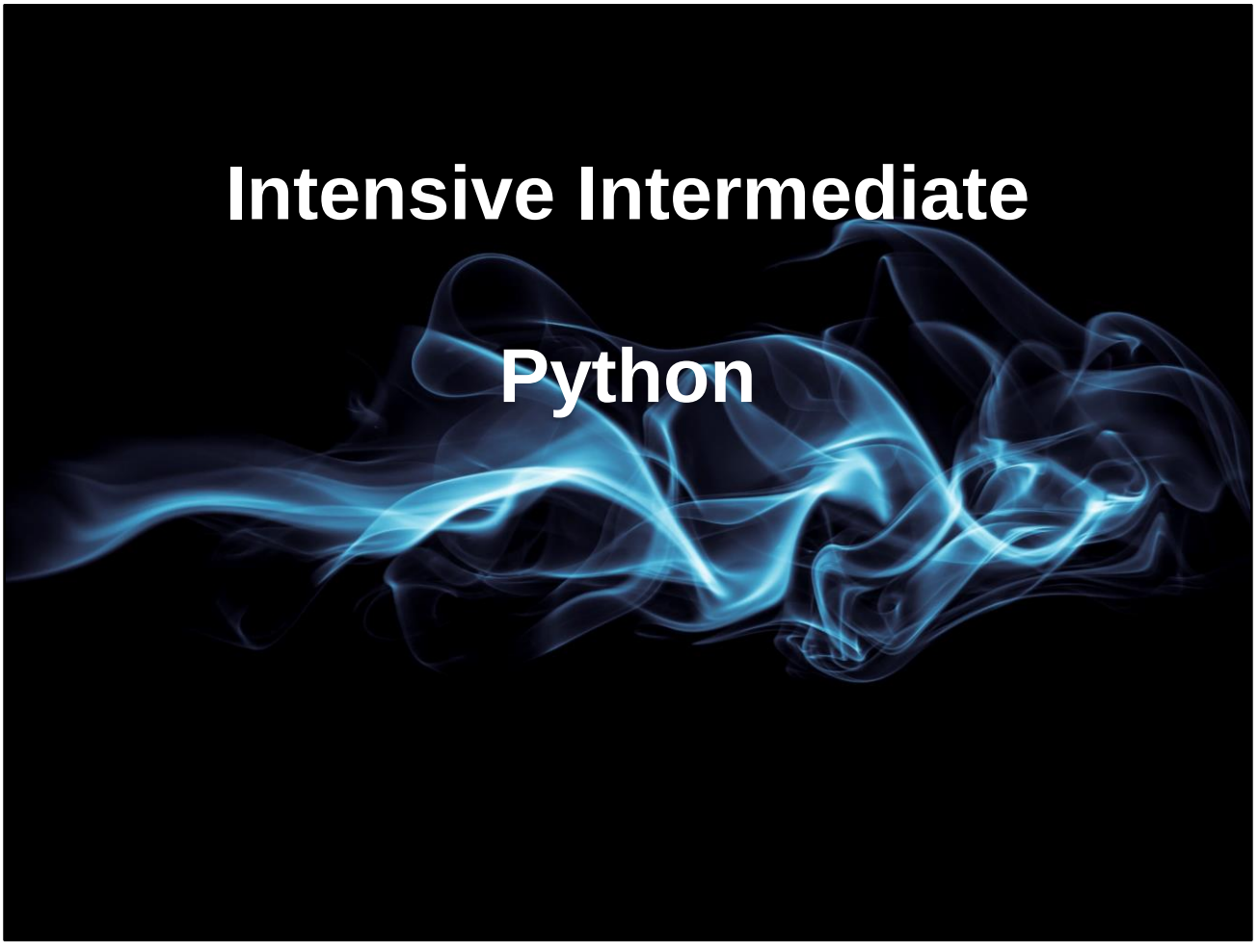
This course covers Intensive Intermediate Python

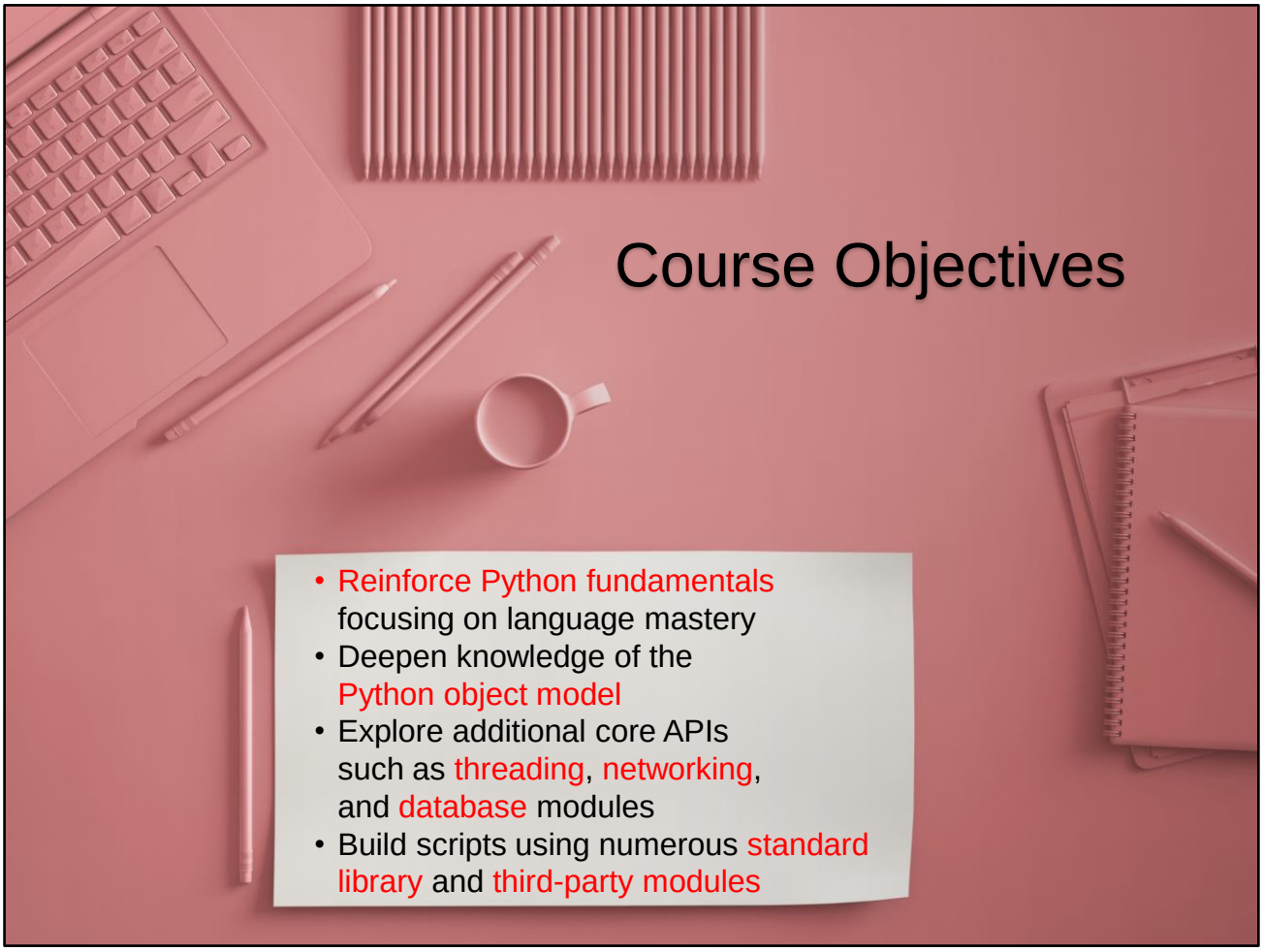
This version updated: 3/25/2022.

Notes

Chapters at a Glance

Notes





Course Agenda

Day 1

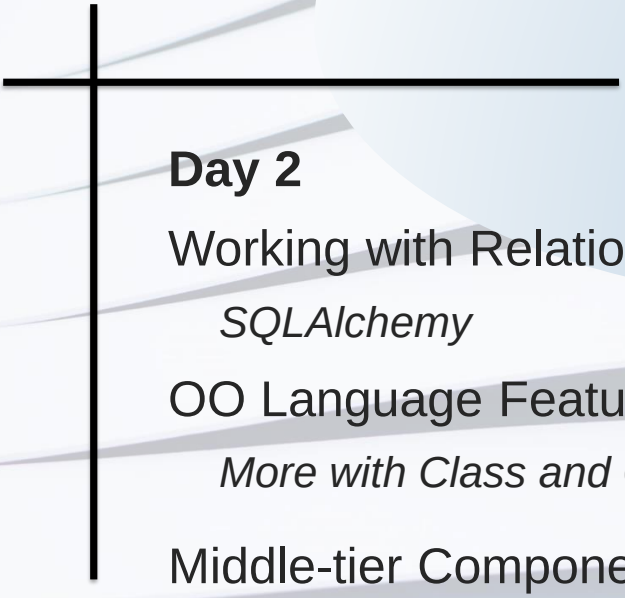
Virtual Environments

Language Fundamentals Review

Modularization

Documentation and Style

Course Agenda



Day 2

Working with Relational Databases

SQLAlchemy

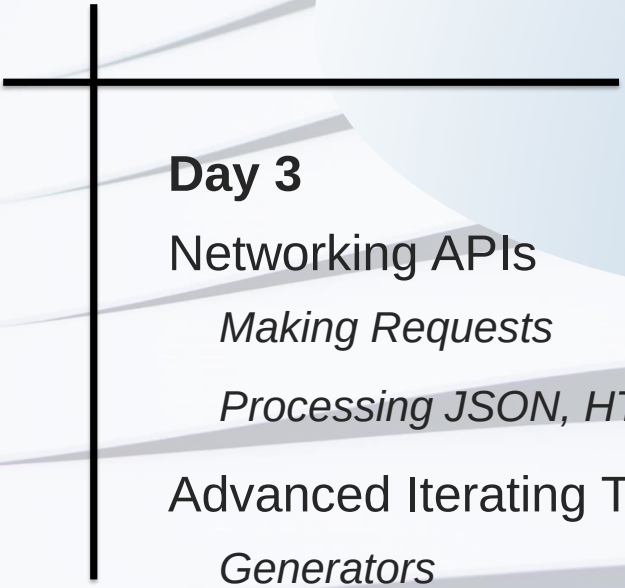
OO Language Features

More with Class and Objects

Middle-tier Components (Flask)

Flask-SQLAlchemy

Course Agenda



Day 3

Networking APIs

Making Requests

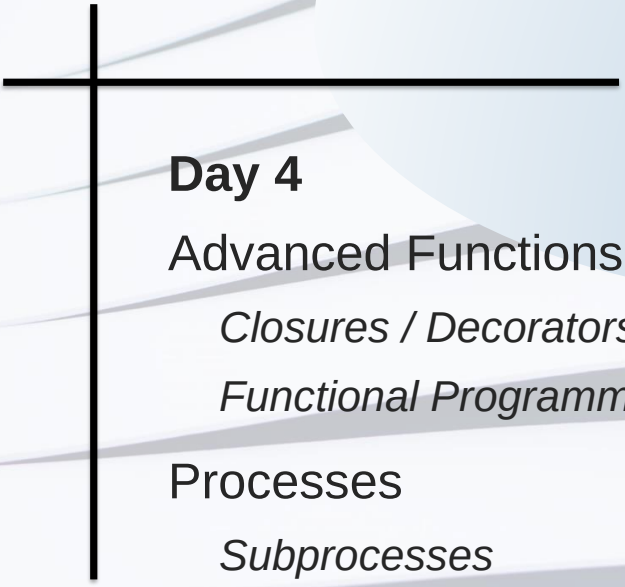
Processing JSON, HTML

Advanced Iterating Techniques

Generators

collections, itertools

Course Agenda



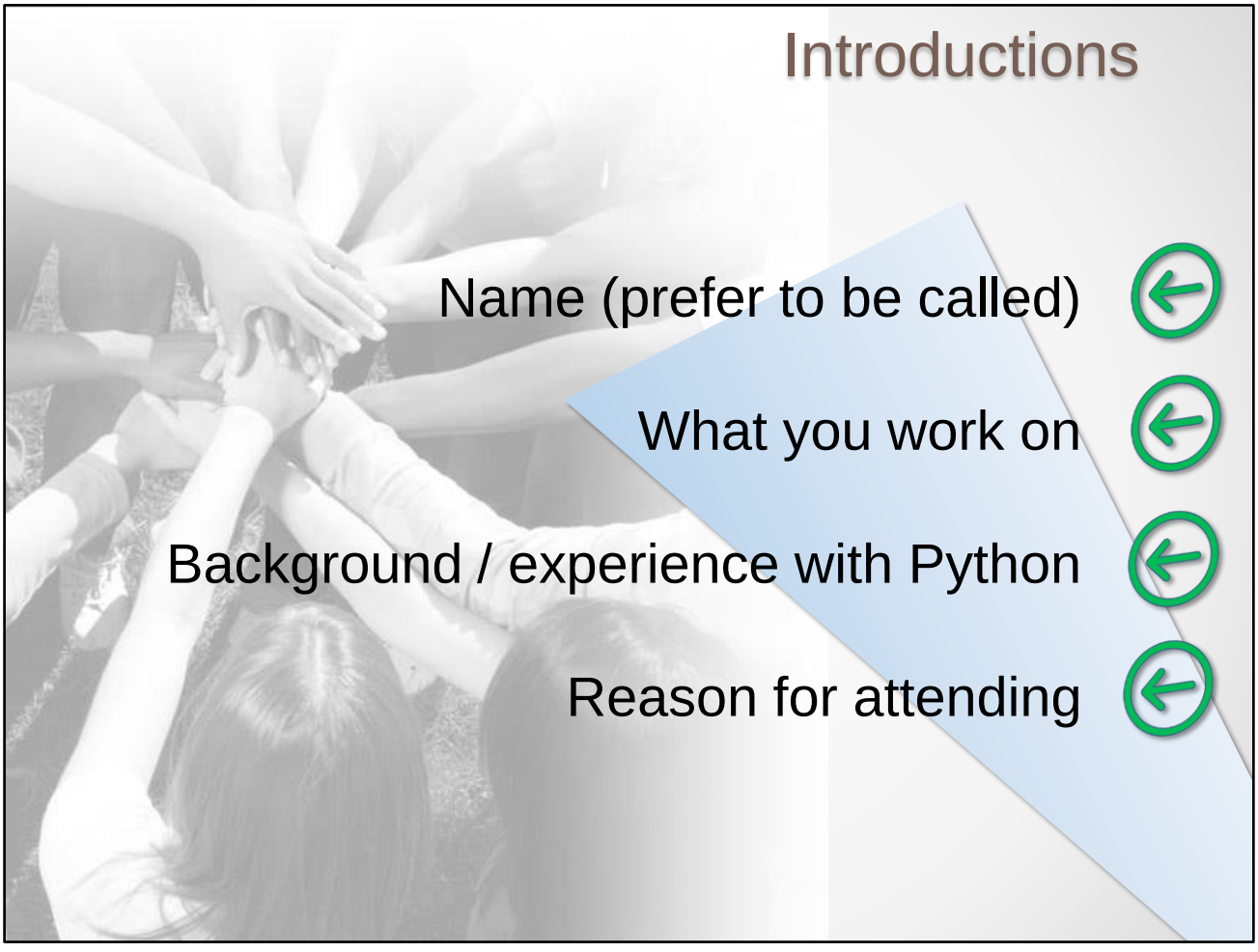
Day 4

Advanced Functions

- *Closures / Decorators*
- *Functional Programming Methods*

Processes

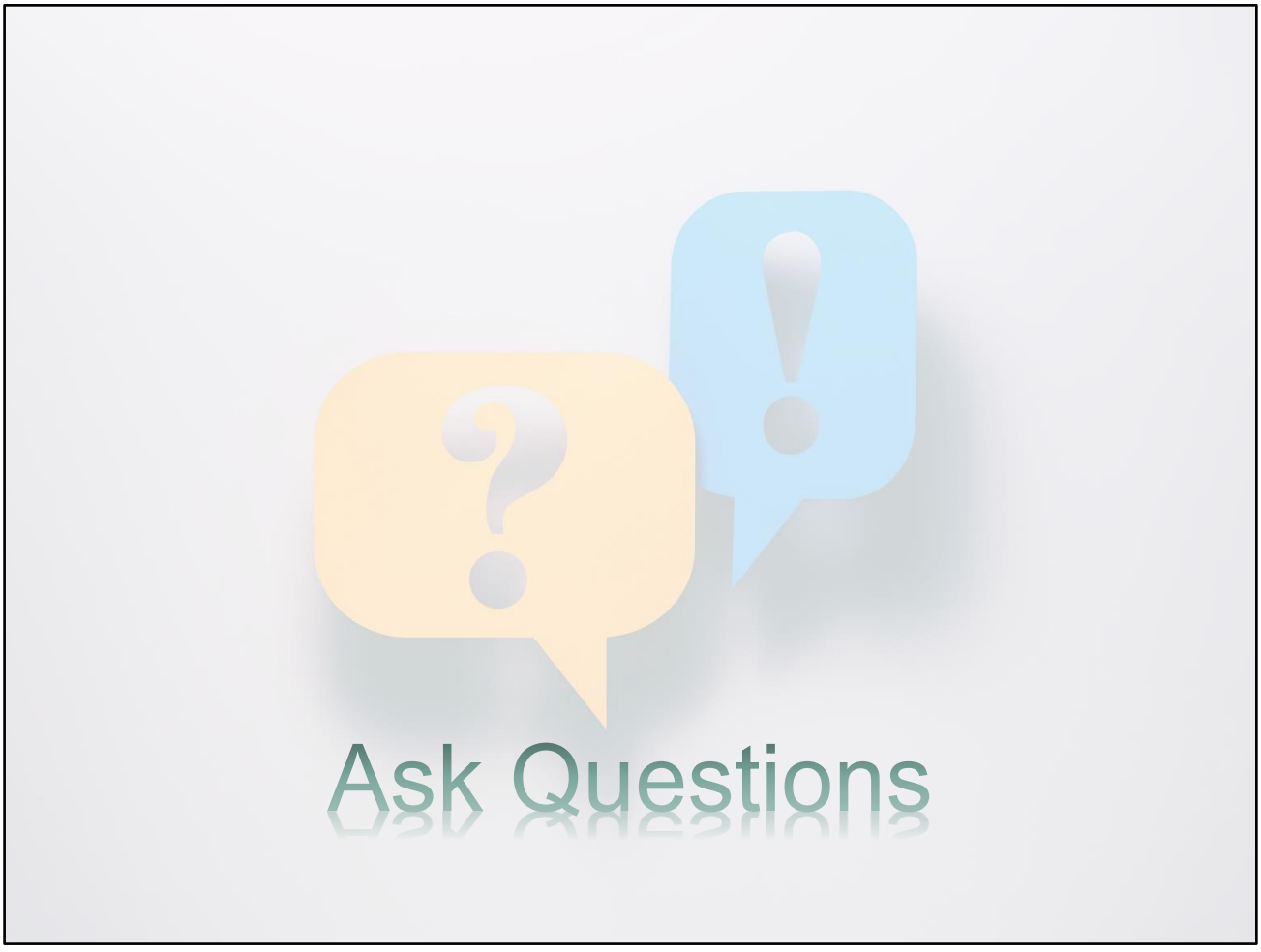
- *Subprocesses*
- *Threads / Multi-processing*

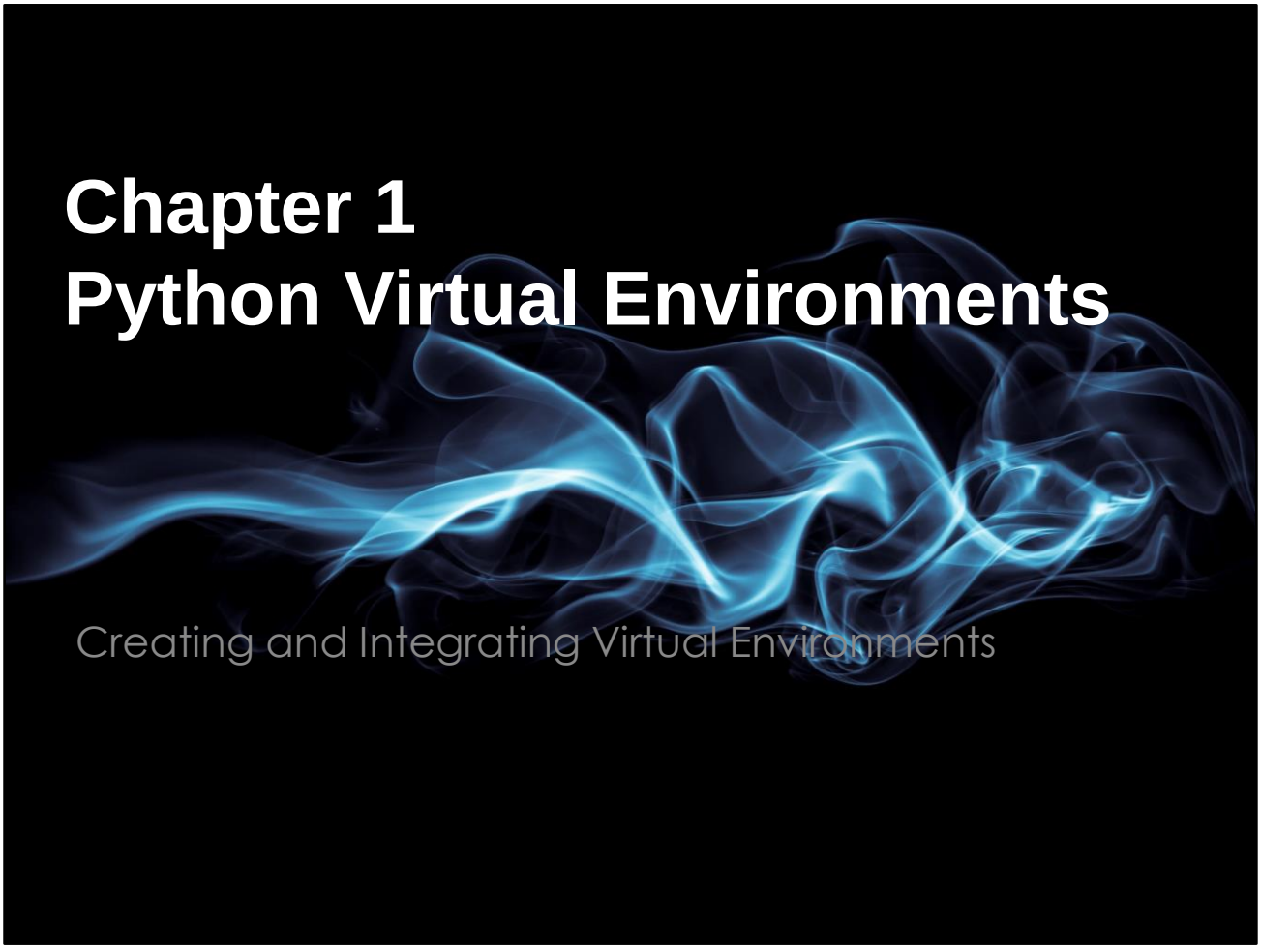


Typical Daily Schedule*

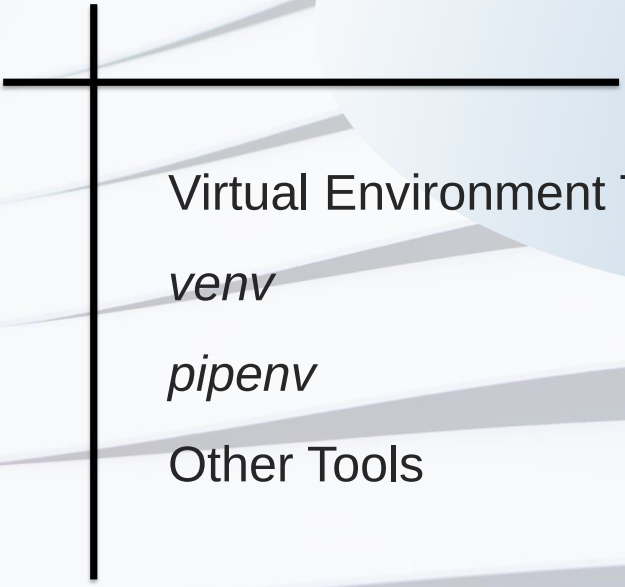
9:00	Start Day
10:10	Morning Break 1
11:20	Morning Break 2
12:30 – 1:30	Lunch
2:40	Afternoon Break 1
3:50	Afternoon Break 2
5:00	End of Day

** Your schedule may vary, timing is approximate*





Overview



Virtual Environment Tools

venv

pipenv

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 an early, popular tool commonly used for both Python 2 and 3 *(even today!)*
    - It needed to be *pip* installed
  - 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

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 is now largely defunct. *venv* is similar to virtualenv. It is packaged within Python 3.6+ and therefore doesn't require any additional pip installs to use it.

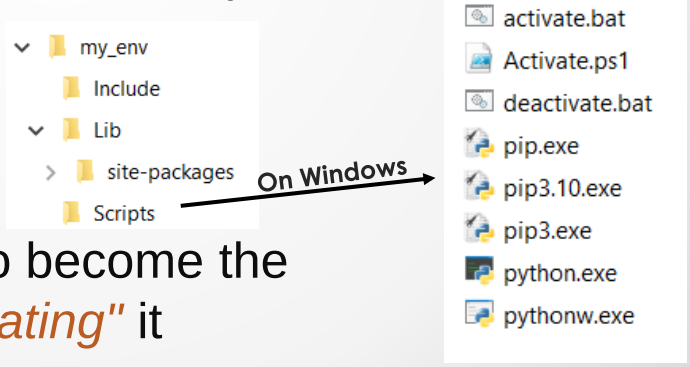
venv

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

Specify python, python3, python3.10, etc., (as needed) when running venv

```
python -m venv relative_or_absolute_path
```

- Using this command within the directory you specify (*my_env* above) will create a new virtual environment within that directory



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

The syntax for virtualenv would be: **`virtualenv my_env.`**

Activation, Deactivation

What actually happens when you activate?

- To enable the virtual environment, run the *activate* command
  - On Windows: my_env\Scripts\activate
  - On OS X: source ./my_env/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 my_env* directory
  - Optionally, use rm -rf my_env on OS X

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

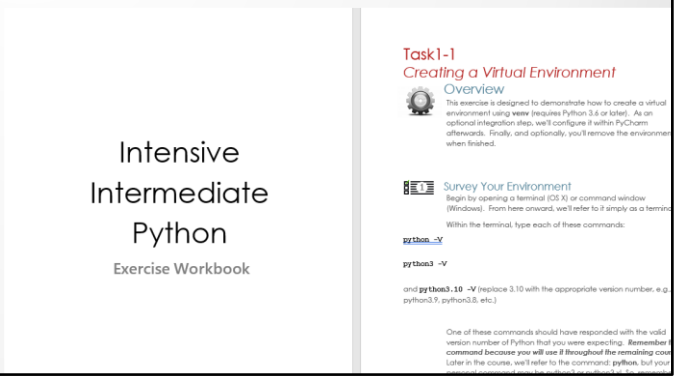
Activation will only take place in the terminal (command) window you are currently using.

Your Turn! Task 1-1

Create a Virtual Python Environment

- Work from the instructions 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* a package into it (if your network allows it)
  - Deactivate it afterwards
  - Remove it (optionally)

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



Freezing and *requirements.txt*

- Use the *freeze* command to list the contents of a python's currently installed packages

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

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

```
certifi==2021.10.8
distlib==0.3.4
filelock==3.4.2
pipenv==2022.1.8
platformdirs==2.5.0
prettytable==3.2.0
six==1.16.0
virtualenv==20.13.1
virtualenv-clone==0.5.7
wcwidth==0.2.5
```

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

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

pip freeze when used without requirements.txt will list the currently installed packages to the console. When installing via a requirements.txt file into a new virtual environment, don't forget to *activate* first.

Other Tools

- pyenv is a Python *version* switching tool and only runs on Unix/Linux

```
$ pyenv install 3.7.3    # installs new python version
$ pyenv versions         # list installed versions
$ pyenv local 3.6.5      # sets local python version
```

- pipenv is another package management tool designed to emulate Ruby and Node.js environments
  - It must be installed in order to use it `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
    - Run pipenv --rm to remove the environment

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.

More on *pipenv*

- After creating a virtual env using *pipenv install*, a Pipfile and Pipfile.lock will exist

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]
flask = "*"

[packages]
requests = "*"
pandas = "*"

[requires]
python_version = "3.10"
```

- To save currently installed package versions to the Pipfile.lock file to reproduce later, run: `pipenv lock`

- If Pipenv.lock is present when running *pipenv install*, it will use the lock file to install packages

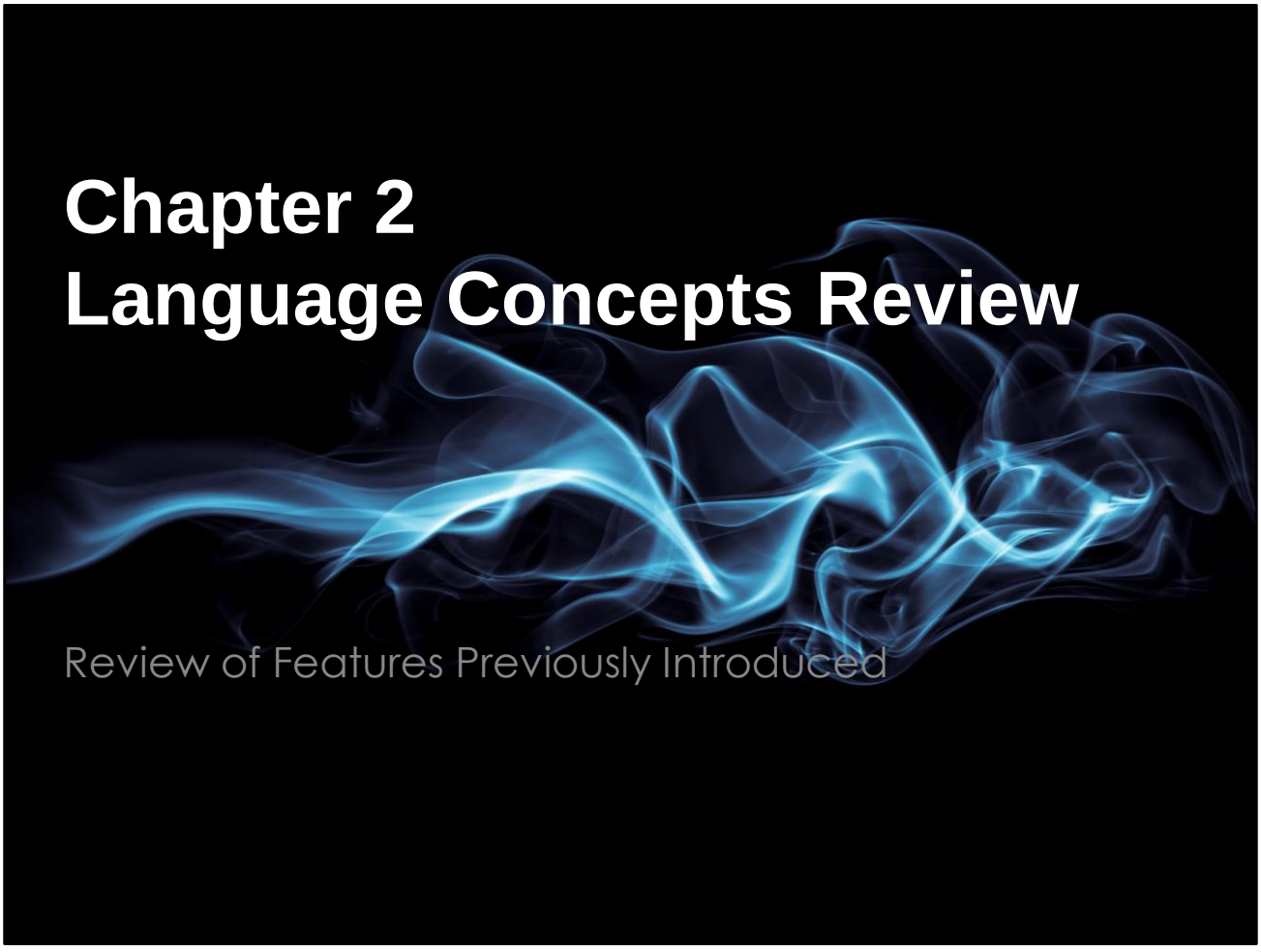
Pipfile and Pipfile.lock are analogous to Ruby's Gemfile and Gemfile.lock files. Pipfile is used to specify which dependencies should be installed in a project.

Pipfile.lock tells pipenv exactly which version of each dependency needs to be installed. Placing the lock file within source control will ensure everyone sees this file and can have proper dependencies installed. This provides consistency across all machines.

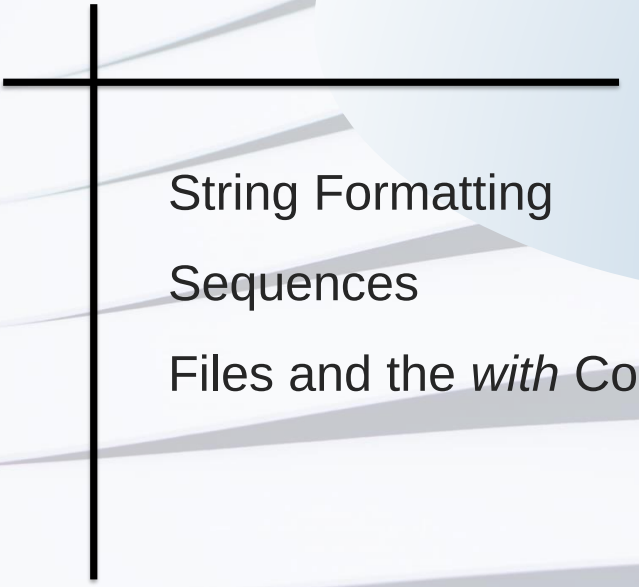
Summary

- Python supports (unfortunately) a widening number of packages and tools related to managing Python packages and environments
- For most basic needs, venv, will be satisfactory
- pipenv can provide better specific version management over venv due to the use of Pipfile and the lock file
- Other package installation and management tools that are beyond our scope include:
  - Poetry
  - Wheel
  - Conda
  - Spack

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



Overview



String Formatting

Sequences

Files and the *with* Control

Formatting 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').

String Immutability

- Python 3 strings are immutable sequences of Unicode characters

```
my_str = 'Python is fun'
my_str = 'Python is very fun'
```

← Creates a new string object

- The builtin id( ) function can illustrate this:

```
>>> my_str = 'Python is fun'
>>> hex(id(my_str))
'0x20661822f70'

>>> my_str = 'Python is very fun'
>>> hex(id(my_str))
'0x2066150c1c0'
```

Different address

In Python 3, strings are Unicode characters. Some characters must be escaped to use them. Escape characters include:

\\	backslash
\'	single quote (as shown in the slide)
\"	double quote
\b	backspace character (ascii)
\n	linefeed
\r	carriage return
\t	tab
\uXXXX	16-bit Unicode character
\uXXXXXXXX	32-bit Unicode character
\N{name}	Unicode named character

Random Access and Slicing

- Sequences support random access (square bracket) and slicing
  - Since strings are sequences, they support it also

```
import linecache
print(linecache.getline('../resources/simpsons.csv', 129))
```

127,Moe Szyslak,Hank Azaria,Owner of Moe's Tavern, …, 1989-12-17

```
print(line[4])
```
M

startval (included)

endval (excluded)

Slicing Syntax

```
new_str = string[start:end:step]
```

```
print(line[4:15])
```
Moe Szyslak

```
print(line[:3])
```
127

```
print(line[-10:])
```
1989-12-17

ch02_basics/01_strings.py

In the top example, the linecache module is used to read a specific line (line 129 in this case). Note: this technique is okay for smaller files but shouldn't be used for larger sized files. For larger files, read the file iteratively and use enumerate() as shown in the associated source file.

Two things to note about slicing:

1. Negative values supplied will wrap around from the end.
2. Slices return new objects in memory

Helpful String Methods

- strip()

Returns a new string with whitespace removed from each end

```
line = linecache.getline('resources/simpsons.csv', 129).strip()
```

```
127,Moe Szyslak,Hank Azaria,Owner of Moe's Tavern.,…,1989-12-17
```

- find()

```
line.find('Moe')
line.find('Homer')
```

```
4
-1
```

Finds the index of the first occurrence of the substring

- split()

Creates a list of strings based on a specified separator string

```
print(line.split(',')[2])
```

```
Hank Azaria
```

ch02_basics/01_strings.py

These are commonly used, basic string methods.

String format()

- A common way to format data is to use the format() method of the string class

{ fieldname | index [!spec] [:format] }

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

Bill Smith 37

```
employee = ['Bill', 'Smith', 37]
fmt_results = '{0:-^20} {1:-<30} {2:>8}'.format(*employee)
print(fmt_results)
```

--------Bill-------- Smith---------- 37

20 30 8

ch02_basics/02_string_format.py

[!spec] refers to either:
[!s] which executes the str() function on the value first before inserting into the string.
[!r] which executes repr(), and
[!a] which executes ascii()

The format specification is somewhat complex. It takes the following form:
[[fill] align] [sign] [#] [0] [width] [,] [.precision] [type]

fill is a fill character.
align is <, >, =, or ^ to indicate left align, right-align, padding between the sign and number, or centered.
sign is +, -, or (space).
width is the minimum field width.
precision is the number of digits displayed after the decimal point.
type can be s, b, c, d, o, x, n for the type of data. s, for string, is the default.

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

F-Strings *(Formatted Strings)*

- Starting with Python 3.6, formatted string literals may be used
  - This syntax won't work in earlier Python versions

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


def show_name():
    return first + ' ' + last


print(f'{first} {last} {age:0.2f} {show_name()}')
```

Bob Smith 37.00 Bob Smith

Specify an 'f' at the beginning of the literal

ch02_basics/02_string_format.py

A letter 'f' must precede the f-string. On a performance note, f-strings are faster than using format(), %, or + in string operations.

f-strings are preferred for Python 3.6+ or later.

Accessing Sequences

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

```
sa_countries = [
    ('Brazil', 204_000_000), ('Columbia', 48_500_000),
    ('Argentina', 43_100_000), ('Peru', 31_100_000),
    ('Venezuela', 30_600_000), ('Chile', 18_000_000),
    ('Equador', 16_300_000), ('Bolivia', 10_500_000),
    ('Paraguay', 7_000_000), ('Uruguay', 3_300_000),
    ('Guyana', 747_000), ('Suriname', 560_000),
    ('French Guiana', 262_000)]
```

(Python 3.6+) Numbers can have underscores

  - Random access, 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/03_lists.py

Sequences exhibit common behaviors such as the ability to be randomly accessed, perform slicing, get concatenated with the same type, or be replicated through multiplication of a scalar value (the last two are not shown here).

The last example is looking 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.

Lists and Tuples

```
sa_countries = [
    ('Brazil', 204_000_000), ('Columbia', 48_500_000),
    ('Argentina', 43_100_000), ('Peru', 31_100_000)]

sa_countries.append(('Falkland Islands', 3000))
sa_countries.insert(1, ('Chile', 18_000_000))
print(sa_countries)
```

```
[
    ('Brazil', 204000000), ('Chile', 18000000),
    ('Columbia', 48500000), ('Argentina', 43100000),
    ('Peru', 31100000), ('Falkland Islands', 3000)
]
```

- Generally, use lists, but when all things are equal, choose:
  - Lists when contents are all similar (homogeneous)
  - Tuples when contents are different (heterogeneous)

As an example, databases commonly return a list of tuples (each record is a tuple)

ch02_basics/03_lists.py

Use *append(value)* to add items onto the end of the list.

To add a value into the middle of a list, use *insert(position, value)* keeping in mind that sequence positions begin at zero. Due to immutability, tuples do not have append(), insert(), or several other methods that attempt to modify the data.

In addition to append() and insert(), other list operations include index(item), sort(), extend(list2), reverse(), pop(), and remove(item).

The following demonstrates removing a items from a list using del:

```
a_list = [1, 2, 3, 4, 5]
del a_list[1:3]
print(a_list)                          # [1, 4, 5]
```

Typed Named Tuples

- Python 3.6 introduced *typed* named tuples

```
from typing import NamedTuple

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

# alternate way...
Country = NamedTuple('Country',
                     [('name', str), ('population', int)])

sa_countries = [
    Country('Brazil', 204000000), ...
    Country('Falkland Islands', 3000)
]

sa_countries.sort(key=lambda country: country.name)


for record in sa_countries:
    print(f'{record.name:<20}{record[1]:>15}')
```

Use inheritance notation to declare the typed NamedTuple

Specify class attributes along with types and optional default values

Both attribute and index notation are still available

ch02_basics/04b_typed_namedtuples.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.

These types are also enforced only in the IDE, not at runtime! Nevertheless, this small distinction does give the typing.NamedTuple an advantage over the classic one.

Output from Above:

Argentina	43100000
Bolivia	10500000
Brazil	204000000
Chile	18000000
Columbia	48500000
Equador	16300000
Falkland Islands	3000
French Guiana	262000
Guyana	747000
Paraguay	7000000
Peru	31100000
Suriname	560000
Uruguay	3300000
Venezuela	30600000

Dataclasses *(Python 3.7+)*

- Dataclasses are similar to typed named tuples except they allow attributes to be modified

```
from dataclasses import dataclass

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

sa_countries = [
    Country('Brazil', 204000000),
    ...
    Country('Falkland Islands', 3000)
]

sa_countries.sort(key=lambda country: country.name)

for record in sa_countries:
    print(f'{record.name:<20}{record.population:>15}')
```

Implement the dataclass by importing the decorator and placing it above the class

Dataclass in use

Index notation is ***NOT*** allowed here

ch02_basics/04c_dataclasses.py

Unlike namedtuples or NamedTuples, dataclasses are *mutable* and NOT indexable. They perform well, perhaps even better than namedtuples and NamedTuples which make them idealalternatives to the classic namedtuple.

In reality, a dataclass simply provides an __init__()
that doesn't have to be declared.
The format of that __init__() is:

```
def __init__(self, val1: type1, val2: type2, ...):
    self.val1 = val1
    self.val2 = val2
```

Working with Files

- Open files using the global, builtin function open()
  - Due to differences in platform defaults, an encoding should always be specified
- Exception handling can be used to problems that arise when working 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 status of f first here could result in *None.close() which is* an AttributeError

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 *with*

- Proper cleanup is essential when working with external resources
  - True for files, database connections, network connections (sockets), locks on threads, and many other situations...

```
do some initialization

do some work

do some cleanup
```

- *with* is a control structure that performs initialization/cleanup
  - Requires an object called a *context manager*

```
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 but also somewhat tricky to figure out. 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 *with*

- The file object is a context manager
  - This makes it useable within the *with* control

```
import os
import sys

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

try:
    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.')
```

Safely joins a path and filename

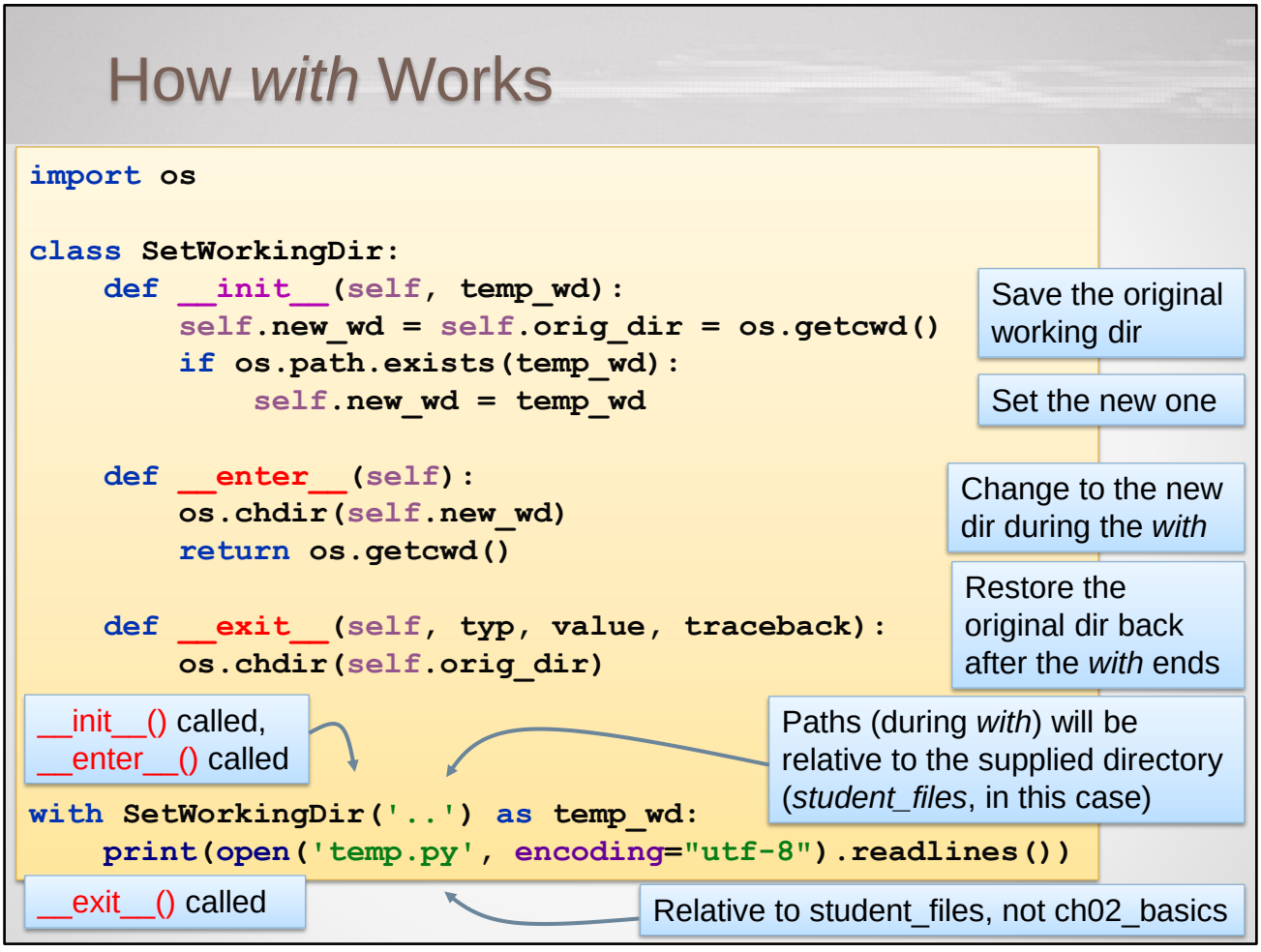
The file gets closed automatically

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.



ch02_basics/06_with_dissected.py

A context manager defines both an __enter__ and an __exit__ method. Though classes haven't been discussed thus far, the class concept is irrelevant to this discussion. When the *with* statement is encountered, the context manager's __enter__ method is invoked. The return value from __enter__ is passed to the optional **'as'** clause of the *with* control.

When the block within the *with* control is finished executing, the __exit__() method will be invoked. The __exit__() method will be called no matter what (whether an exception is raised or not).

This example sets the working directory to the student_files directory (the parent to where it is currently executed from) at the beginning of the *with* control. At the end of the *with* control, the working directory is set back to what it was before the *with* control began. During the *with* control, all paths for files referenced can be relative to this temporary (student_files) working directory.

Writing to Text Files

- Use Python 3's file= parameter to redirect output to a file:

```
data = [
    'Lorem ipsum dolor sit amet, consectetur ',
    ...
    'qui officia deserunt mollit anim id est laborum.',
]

try:
    with open('data.txt', 'w', encoding='utf-8') as f:
        for line in data:
            print(line, file=f)
except IOError as err:
    print(err, file=sys.stderr)
```

ch02_basics/07_writing.py

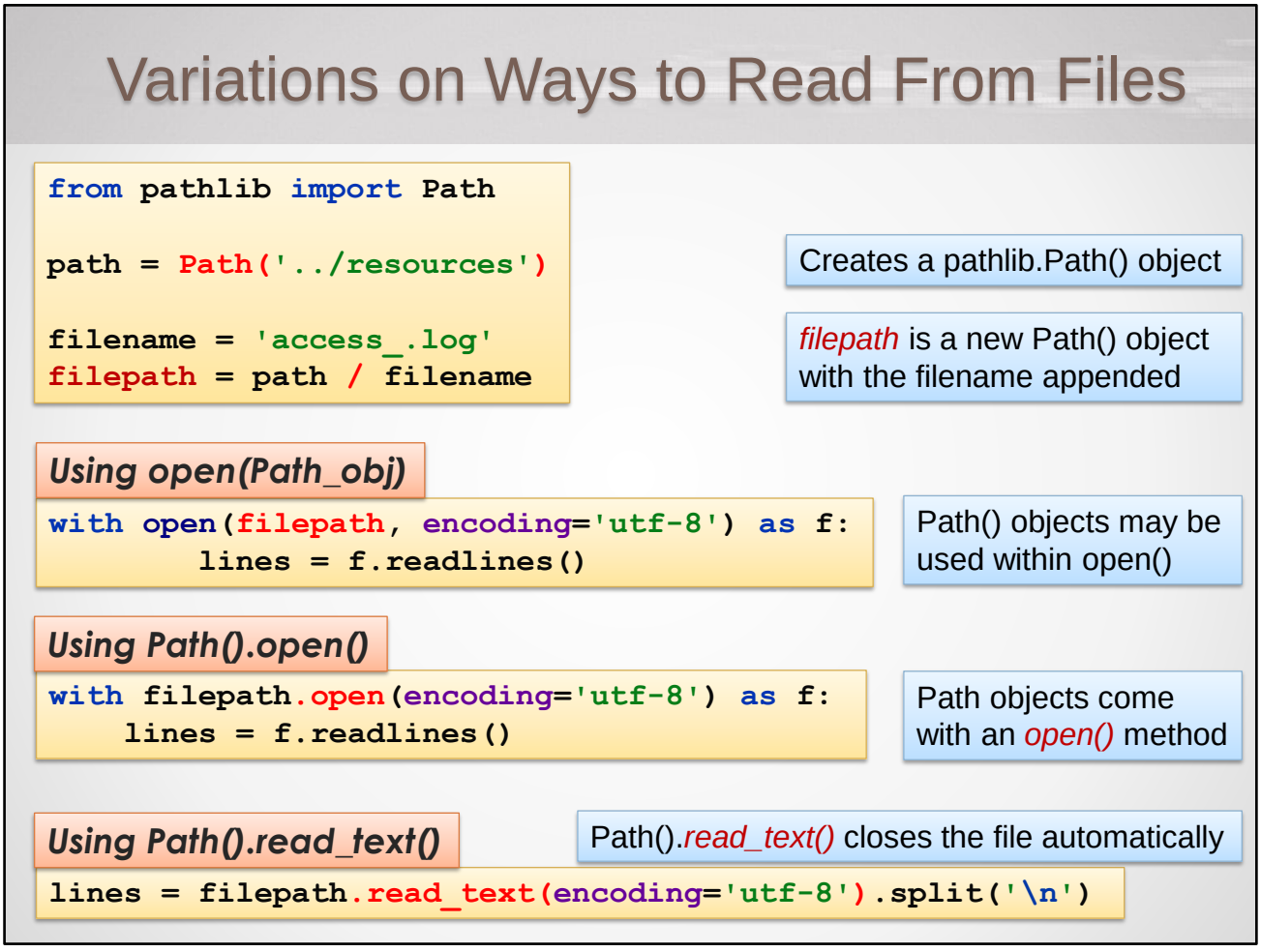
This example takes a list of strings and writes it to a file.

Using pathlib.Path

- Many pathlib.Path object methods replace os functions

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



ch02_basics/08_with_pathlib.py

For brevity, the exception handling was 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 Qaïwaïn,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,Dubay,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

cities15000.txt is data sourced from geonames.org. Files provided are raw data as found on geonames.org.

cities15000.txt contains city information for all cities with populations greater than 15000.

Note: the population field is the 15th one in the record (14th starting from 0), e.g., record[14] is the population field. The elevation field is the 17th field (16th starting from 0).

Working with Sequences

```
sa_countries = [
    ('Brazil', 204000000),
    ('Columbia', 48500000),
    ('Argentina', 43100000),
    ('Peru', 31100000)
]
```

- len()

```
print(len(sa_countries),
      len(sa_countries[0]),
      len(sa_countries[0][0]))
```

```
4, 2, 6
```

- max()

```
print(max([1973, 2001, 2015, 2013, 1994])
print(max(sa_countries, key=lambda country: country[1])[0])
```

```
2015
Brazil
```

  - A similar function exists for min()

key= can define how to calculuate max by population (key= is discussed shortly)

These functions are all global. They do not have to be imported to be used.

min() and max() also support a key function that can be supplied as an argument to define what biggest or smallest means in sequences where custom behavior is needed.

Consider the following example that uses a dictionary:

weights = {**'Paul'**: 195, **'Mary'**: 165, **'Tom'**: 182}
print(min(weights.keys(), key=**lambda** key: weights[key]))

Note the key as an argument to min().
The result displays 'Mary'.

Additional Iterating Techniques

```
from pathlib import Path
filepath = Path('../resources/sa_countries.csv')
sa_countries = []

with filepath.open(encoding='utf-8') as f:
    for line in f:
        sa_countries.append(line.strip().split(','))
print(sa_countries)
```

```
[['Brazil', 'Columbia', 'Argentina', 'Peru', 'Venezuela',
'Chile', 'Equador', 'Bolivia', 'Paraguay', 'Uruguay',
'Guyana', 'Suriname', 'French Guiana', 'Falkland Islands'],
['204000000', '48500000', '43100000', '31100000',
'30600000', '18000000', '16300000', '10500000',
'7000000', '3300000', '747000', '560000', '262000', '3000']]
```

```
for value in enumerate(reversed(sa_countries[0])):
    print(value)

for country, pop in zip(*sa_countries):
    print(f'{country:<20}{pop:>15}')
```

```
(0, 'Falkland Islands')
(1, 'French Guiana')
(2, 'Suriname')
(3, 'Guyana')
…
(10, 'Peru')
(11, 'Argentina')
(12, 'Columbia')
(13, 'Brazil')
```

```
Brazil               204000000
Columbia              48500000
Argentina             43100000
…
```

ch02_basics/09_iterating.py

These handy and globally available functions accept a sequence and return values in a different way.

enumerate() and zip() take an iterable while reversed() takes a sequence.
What's the difference?
Sequences are all iterable, but not all iterables are sequences. A dictionary is iterable but not a sequence. zip() will end when the SHORTEST iterable completes.

Sorting

```
sa_countries = [
    ('Brazil', 204000000),
    ('Columbia', 48500000),
    ('Argentina', 43100000),
    ('Peru', 31100000)
]
```

- Use sort() for simple, in-place sorting:

For lists only

```
sa_countries.sort()
print(sa_countries)
```

```
[ ('Argentina', 43100000),
 ('Brazil', 204000000),
 ('Columbia', 48500000),
 ('Peru', 31100000) ]
```

- Use sorted() to return a new sequence

Works for any iterable

```
new_countries = sorted(sa_countries)
print(new_countries)
```

```
[ ('Argentina', 43100000),
 ('Brazil', 204000000),
 ('Columbia', 48500000),
 ('Peru', 31100000) ]
```

ch02_basics/10_sorting.py

To sort in descending order, use *reverse=True*. See the example in the file.

When determining whether to use sort() or sorted() consider your needs. If a new list is desired with the original remaining intact, then use sorted(). However, if the added overhead of creating a new list doesn't make sense when simply modifying the one in memory will suffice then use sort(). Performance-wise, both are about the same.

Sorting with a Key

- Use key= to sort in a specialized way
  - Write a function to define how to sort
  - One item at a time from the collection is passed into the function, it should return a value to sort by

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

Returns the population field

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

[('Peru', 31100000), ('Argentina', 43100000), ('Columbia', 48500000), ('Brazil', 204000000)]

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

print(new_ctys)
```

[('Brazil', 204000000), ('Columbia', 48500000), ('Argentina', 43100000), ('Peru', 31100000)]

ch02_basics/10_sorting.py

Often the default sort method is not what is needed. To overrule the default sort used, supply a function that defines how to sort. This function can be provided using the *key=* parameter within sort() or sorted(). The value provided for *key=* should either be the name of a function (without the parentheses) or a lambda (mentioned next). The *key=* function should return a value defining how the record should be sorted. The original elements in the collection are not transformed, they are only sorted.

Lambdas

- Lambdas are *inline-functions*
  - For quick, short, throw away uses such as sorting, closures, functional programming

```
funcName = lambda <arguments> : <ret_val>
```

- The following uses a lambda to sort a list of tuples by the second field

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

print(sa_countries)
```

```
[('Brazil', 204000000),
('Columbia', 48500000),
('Argentina', 43100000),
('Peru', 31100000)]
```

ch02_basics/10_sorting.py

Lambda limitations:
- <ret_val> must be equivalent to what a *legal return value* would be. This means you may not have entire statements (e.g., x = 10) in the lambda.
- No variable assignments and 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] or the population field. This means that it will sort by the population.

List Comprehensions

- List comprehensions provide 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/11_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.

Your Turn! Task 2-2

Part 2

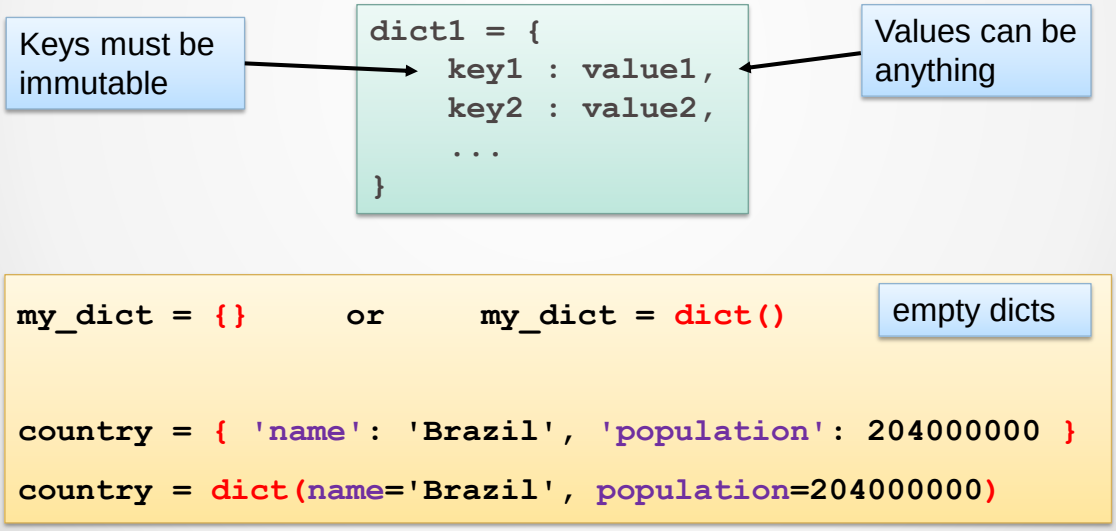
Sorting and Max *(part 2 of 3)*

- Determine
  - *the most populous city* in the world
  - the highest city (elevation) in this list
- 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?*

Creating Dictionaries

- Dictionaries are collections of name/value pairs
  - Ordered*, mutable, iterable
  - Supports len() and in comparisons

* dicts before Py 3.6 are unordered (see footnotes)




```
my_dict = {}      or      my_dict = dict()
```

empty dicts

```
country = { 'name': 'Brazil', 'population': 204000000 }
country = dict(name='Brazil', population=204000000)
```

Dictionaries are useful data structures as they provide a means to store any kind of data yet access that data via a key. Because dictionary keys must be unique, attempting to add an entry to a dictionary where the key already exists will replace the value with the new value.

Note on order within dicts: If you are writing tools that will run in mixed Python environments, it is best to treat dictionaries as unordered. Any Python 3.5 or earlier environment that runs the code will fail if order is assumed in a dictionary.

Accessing Dictionaries

```
country = { 'name': 'Brazil', 'population': 204000000 }
```

- Use get(key) to retrieve values

```
country.get('population')
country.get('capital')
country.get('capital', 'N/A')
```

```
204000000
None
N/A
```

- Direct access can generate a KeyError
  - Can be handled using exception handling

```
try:
    capital = country['capital']
except KeyError:
    capital = '(unknown)'
```

ch02_basics/13_dicts.py

Access a dictionary by using the get() method and by optionally supplying a second value as a fallback value if the key is not present.

Modifying and Iterating Dictionaries

```
country = { 'name': 'Brazil', 'population': 204000000 }
```

- Adding dictionary entries

```
country['capital'] = 'Brasilia'
```

- Iterating a dictionary returns the keys

```
for key in country:
    print(f'{key}: {country[key]}')
```

name: Brazil / capital: Brasilia / population: 204000000

- Iterating a dictionary's values

```
for value in country.values():
    print(value, end=' ')
else:
    print()
```

Brazil 204000000 Brasilia

ch02_basics/13_dicts.py

Accessing a dictionary in a *for* loop will return the keys. country.keys(), provides "a view" of the keys of the dictionary.

Iterating Dictionaries

- Accessing *both* key and value while iterating

```
for key, val in country.items():
    print(f'Key: {key}, Value: {val}', end=' ')
```

```
Key: name, Value: Brazil Key: population, Value: 204000000
Key: capital, Value: Brasilia
```

Other dict methods

```
dict.copy()
dict.clear()
dict.fromkeys([key_seq])
dict.pop(key, def)
dict.setdefault(key, def)
dict.update(d2)
```

ch02_basics/13_dicts.py

fromkeys() returns a dictionary based on the supplied sequence of keys

setdefault() returns the value for the key or if not found, will add the key and set the value to the default provided

update() merges d2 onto d1

Note about dictionary views:

dict.items(), dict.keys(), and dict.values() all return "views" that are iterable. A view is a read-only iterable, however if the dict the view refers to is changed, the view is changed also.

Functions (and *pathlib*)

- Functions must be defined before being called

```
from pathlib import Path


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

print(check_files('../ch02_basics', '01_strings.py'))
print(check_files())
print(check_files(filename='01_strings.py',
                  path='../ch02_basics'))
```

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

The function returns True or False on whether a file exists

```
True
True
True
```

ch02_basics/14_functions.py

Functions must be defined (or imported) before they can be called. Parameters passed into the function call must also match what is declared in the function definition.

The / operator is defined for the Path class. It merely creates a step in a path. The results object (on the left of the equation) is also a Path object.

Multiple Positional Arguments

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

The Python **all(*iterable*)** function returns *True* 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',
                      '01_strings.py', '03_lists.py'))

print(check_all_files('../ch02_basics', '01_strings.py',
                      '03_lists.py', '04_not_there.py'))
```

```
True

False
```

ch02_basics/14_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 may elect to write the function to be flexible. Python provides a way to pass as many arguments as needed using the * notation. For this to work properly, the *files argument MUST be supplied last. 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 'individual' in kwargs and kwargs['individual'] is True:
        return results
    return all(results)

print(check_all_files_extra('../ch02_basics', '01_strings.py',
                            '03_lists.py'))
```

True

```
print(check_all_files_extra('../ch02_basics', '01_strings.py',
                            '03_lists.py', '04_not_there.py',
                            individual=True))
```

[True, True, False]

```
print(check_all_files_extra('../ch02_basics', '01_strings.py',
                            '03_lists.py', '04_not_there.py',
                            individual=False))
```

False

ch02_basics/14_functions.py

When numerous or an unknown number of additional parameters may be passed into a function, you can optionally capture these as keyword arguments in a dictionary. Here, that dictionary has been called *kwargs*, 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)*

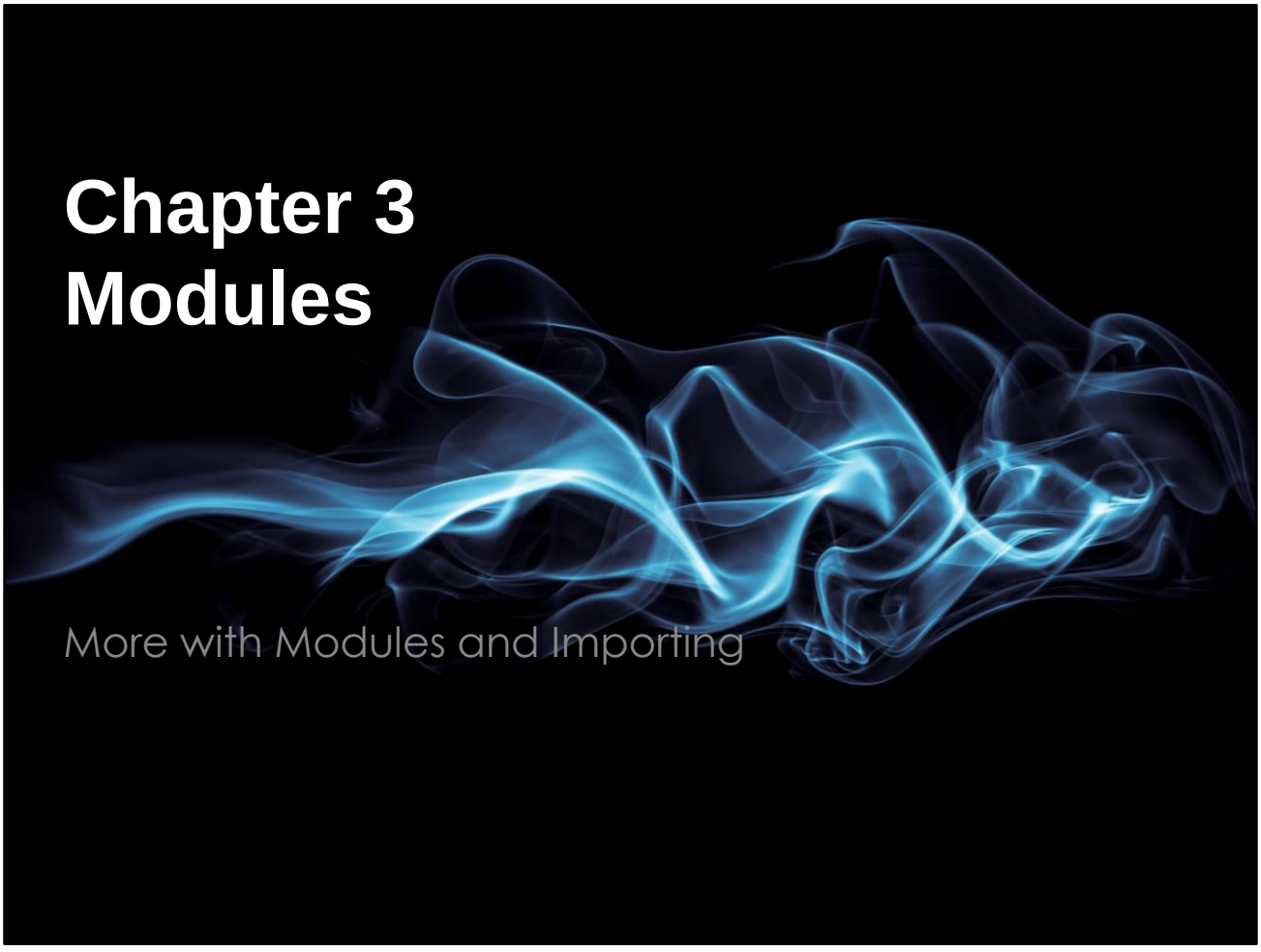
- Create a *search capability* that accepts a city name (or partial name) and *returns valid matches and populations* of each city

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

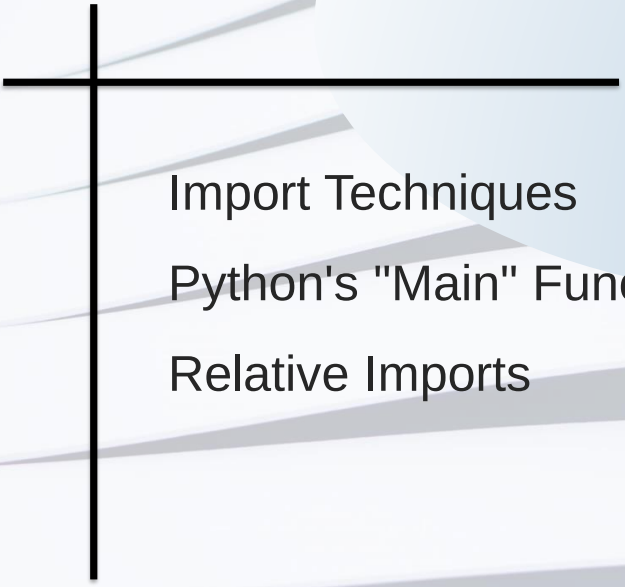
- Continue working from your previous part 2 solution (task2_1_starter.py or task2_2_starter.py) or you can work from task2_3_starter.py

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 function features (type hints), dictionary rules (ordered), Path objects, and much more
  - Exercise care when using these features as they are not typically backward compatible with earlier versions



Overview



Import Techniques

Python's "Main" Function

Relative Imports

Modules

- Modules are namespaces in Python
  - Each .py file represents a module
  - Functions, variables, classes are contained within modules
    - These attributes must be *imported* to use them
- A module's location can be determined after importing

```
import pathlib
print(pathlib.__file__)
```

```
<PYTHONHOME>/lib/pathlib.py
```

  - Most standard library modules are found in the PYTHONHOME/lib directory

What are these other files I see?
.pyc - compiled bytecode files when a module is first imported
.pyo - optimized bytecode files
.pyd - Windows only, compiled modules containing Python source code

Your source files are converted into bytecode (lower-level instructions) for use within the Python virtual machine.

.pyc files, once created, do not have to be created again and therefore improve loading and execution times on subsequent uses.

.pyo files are similar but are created with the -o flag is run python -O myfile.py. It's not common to run the -O flag.

.pyd files are encountered on Windows. They are somewhat like DLLs except they contain Python modules. They are compiled on Windows and placed into a directory in your sys.path. They are then imported like any other module. It is not common to create .pyd files.

Import Techniques

- There are several ways to import a module:

```
import pathlib
p1 = pathlib.Path('x/y/z.foo.py').name

import pathlib as pl
p2 = pl.Path('x/y/z.foo.py').parent

from pathlib import Path
p3 = Path().cwd() / Path('x/y/z.foo.py')
p4 = Path('x/y/z.foo.py').absolute()
p5 = Path('/x/y/z.foo.py').resolve()

from pathlib import Path as P
p6 = P('/x/y/z.foo.py').suffixes

print(p1, p2, p3, p4, p5, p6, sep='\n')
```

```
z.foo.py

x\y

C:\student_files\x\y\z.foo.py
C:\student_files\x\y\z.foo.py
C:\x\y\z.foo.py

['.foo', '.py']
```

- Use dir(*object*) to list an object's top-level items

```
dir(pathlib)
dir(p1)
```

ch03_modularization/01_importing.py

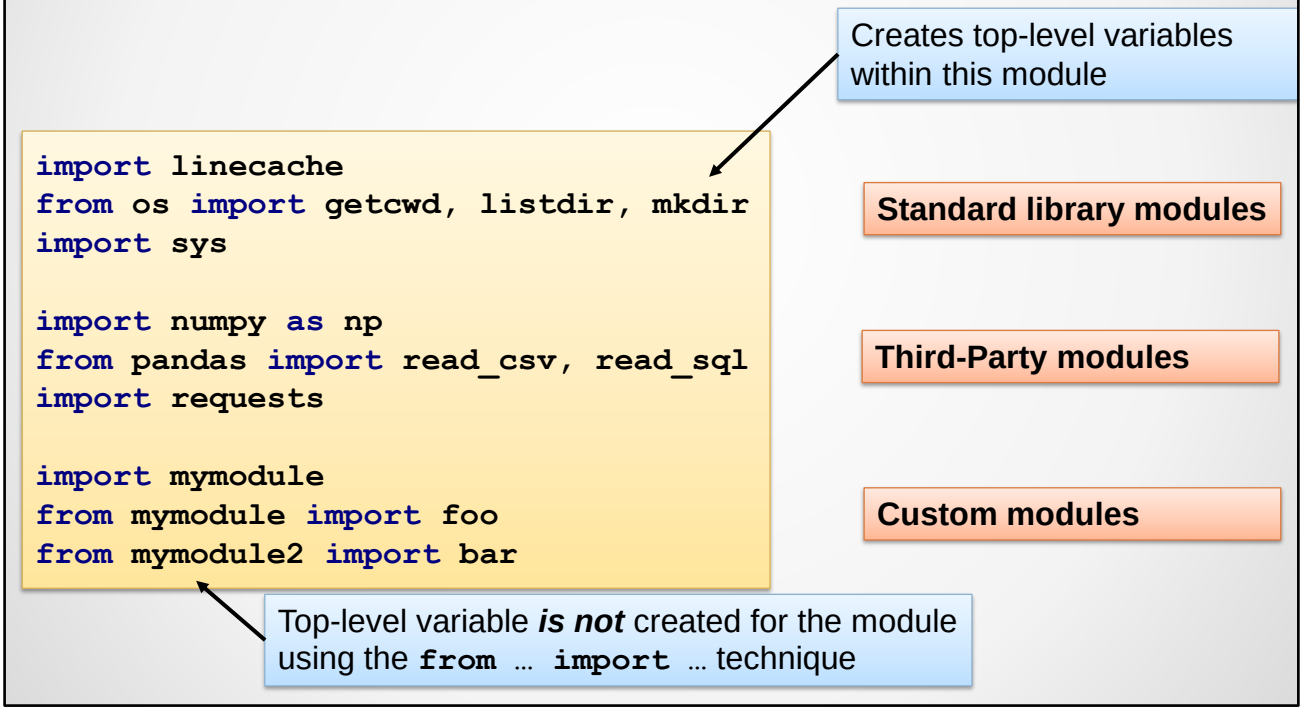
There are a number of different ways available to import modules. There are no performance differences between each approach as they all require reading the module entirely. The choice of import technique is largely a matter of preference at the time.

The values for p2, p3, p4, and p5 are shown with Windows-based results. For Unix/Linux expect forward-slashes to be used:
x/y
/student_files/x/y/z.foo.py
/student_files/x/y/z.foo.py
/x/y/z.foo.py

To determine the top-level items of a module, issue the dir( ) command after importing the module.

More on Importing

- Place module imports each on their own line:



The layout for importing is defined in PEP 8. It is worth noting that when organizing *import module* vs *from module import item*, PEP 8 does not define any rules (it only defines having the three sections as illustrated in the slide).

Some tools that auto organize imports will do it slightly differently by placing the f*rom module import item* type import at the bottom of each section. This is up to the user. Our convention will be to import and alphabetize each section by module name regardless of import mechanism used (as demonstrated in this slide).

Defining a Main

- The "name" of a module is determined by looking at the __name__ attribute
  - If a module is imported, __name__ is set to its fully qualified filename (without file extension)
  - If it is run directly, __name__ is set to __main__

driver.py

```
import ch03_modularization.main_example.two_ways as two
print(two.identify())
```

ch03_modularization.main_example.two_ways

two_ways.py

```
def identify():
    return __name__

if __name__ == '__main__':
    print(f'Running directly! Module __name__: {identify()}')
    print('This will not be seen when the module is imported')
```

Running directly. Module __name__: __main__
This will not be seen when the module is imported

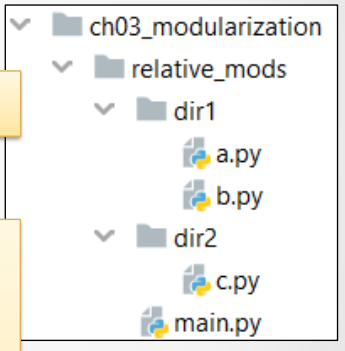
ch03_modularization/main_example/driver.py and two_ways.py

Test this out by running driver.py and then running two_ways.py. How does two_ways.py behave differently each time?

Relative Imports

- Generally, it is best not to rely heavily on relative imports
  - Relative imports *must use* from

The import keyword can not be used with relative path notation



main.py

```
import ch03_modularization.relative_mods.dir1.a
```

a.py

```
from .b import foo
print(foo)

from ..dir2 import c
print(c.foo)
from ..dir2.c import foo
print(foo)

from ...relative_mods.dir1.b import foo as foo2
print(foo2)
```

This is b.py
You are in b.py

This is c.py
You are in c.py

You are in c.py

You are in b.py

Use . or .. or ...

You can only go up as high as your main.py source file

ch03_modularization/relative_mods/main.py (also dir1/a.py, dir1/b.py, and dir2/c.py)

It is best to view and run these files directly to see the results on your own.

You can not run the a.py module directly or you will get an import error, as in:
ImportError: attempted relative import with no known parent package

This occurs because relative imports use the __name__ and __package__ to resolve the starting location. When a module is run directly, these values are __main__ and None respectively. If we run a.py directly, __name__ and __package__ will be __main__ and None, and therefore we won't have any info as to the starting package. So, when using relative imports, make sure you have a top-level module (like main.py) to run other modules that have relative references!

You can only use as many dots (e.g., ...) as levels up to your main file that you originally ran first.

Resolving Modules

- Python programs locate modules by looking at the sys.path variable

```
import sys
print(sys.path)
```

```
['C:\\temp\\python\\student_files',
'C:\\temp\\python39\\python39.zip',
'C:\\temp\\python39\\DLLs',
'C:\\temp\\python39\\lib', 'C:\\temp\\python39',
'C:\\temp\\python39\\lib\\site-packages',
'C:\\temp\\python39\\lib\\site-packages\\win32',
'C:\\temp\\python39\\lib\\site-packages\\win32\\lib',
'C:\\temp\\python39\\lib\\site-packages\\Pythonwin']
```

A sample sys.path on Windows

Python will automatically prepend the PYTHONPATH env variable to sys.path

Ways to get Python to recognize your own custom modules:

In Windows cmd shell, use: `set variable=value`
(e.g., `set PYTHONPATH=c:\module_location`).
Only lasts for that cmd session.

In Windows, select: Control Panel > System > Advanced System Settings > Advanced Tab > Environment Variables > User Variable > New... > PYTHONPATH, then add c:\module_location. This is a permanent fix.

In OS X terminal, `export variable=value`
(e.g., `export PYTHONPATH=~/module_location`).
Only lasts only for that terminal session.

In OS X, in a terminal run `sudo nano ~/.profile` (or .bashrc or .bash_profile) in your home directory. Add `export PYTHONPATH=~/module_location`
This will be a permanent fix.

In each case, substitute module_location for a valid location.

Installing Third-Party Modules (*pip*)

- Most third-party packages can be installed using pip (python package installation tool)
  - Packages are typically installed from PyPI (Python Package Index Repository)

pip list	Lists installed packages
pip freeze > requirements.txt	Output installed packaged to file: requirements.txt
pip install *package* pip3 install *package* pip3.10 install *package*	Examples: pip install pyyaml pip3 install pyyaml pip3.10 install pyyaml
python -m pip install *package*	Another way to run pip
pip uninstall *package*	

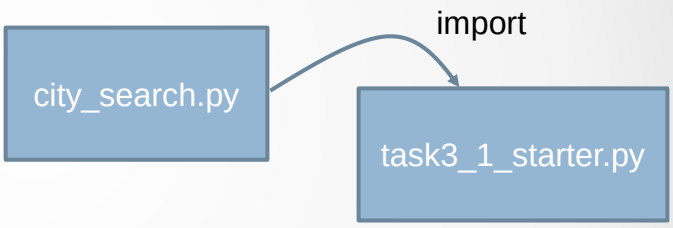
To use pip, ensure its location appears on your path:
<PYTHONHOME>/Scripts (Windows)
<PYTHONHOME>/bin (Unix/Linux)

You should always check the version of pip before executing it. Use *pip -V* to do this. Checking the version will reveal which pip tool you are about to execute. You may find that you are about to use the wrong pip utility. If it is the wrong one, you may need to try different combinations of the pip command until you find the desired one (pip3, pip3.9, pip.3.8, etc.).

Your Turn! Task 3-1

Creating Functions

- Refactor *task2_3_starter* using functions and modules
- Consider creating functions such as
  - read_data(filename)
  - search(cityname)
  - largest()
  - highest()



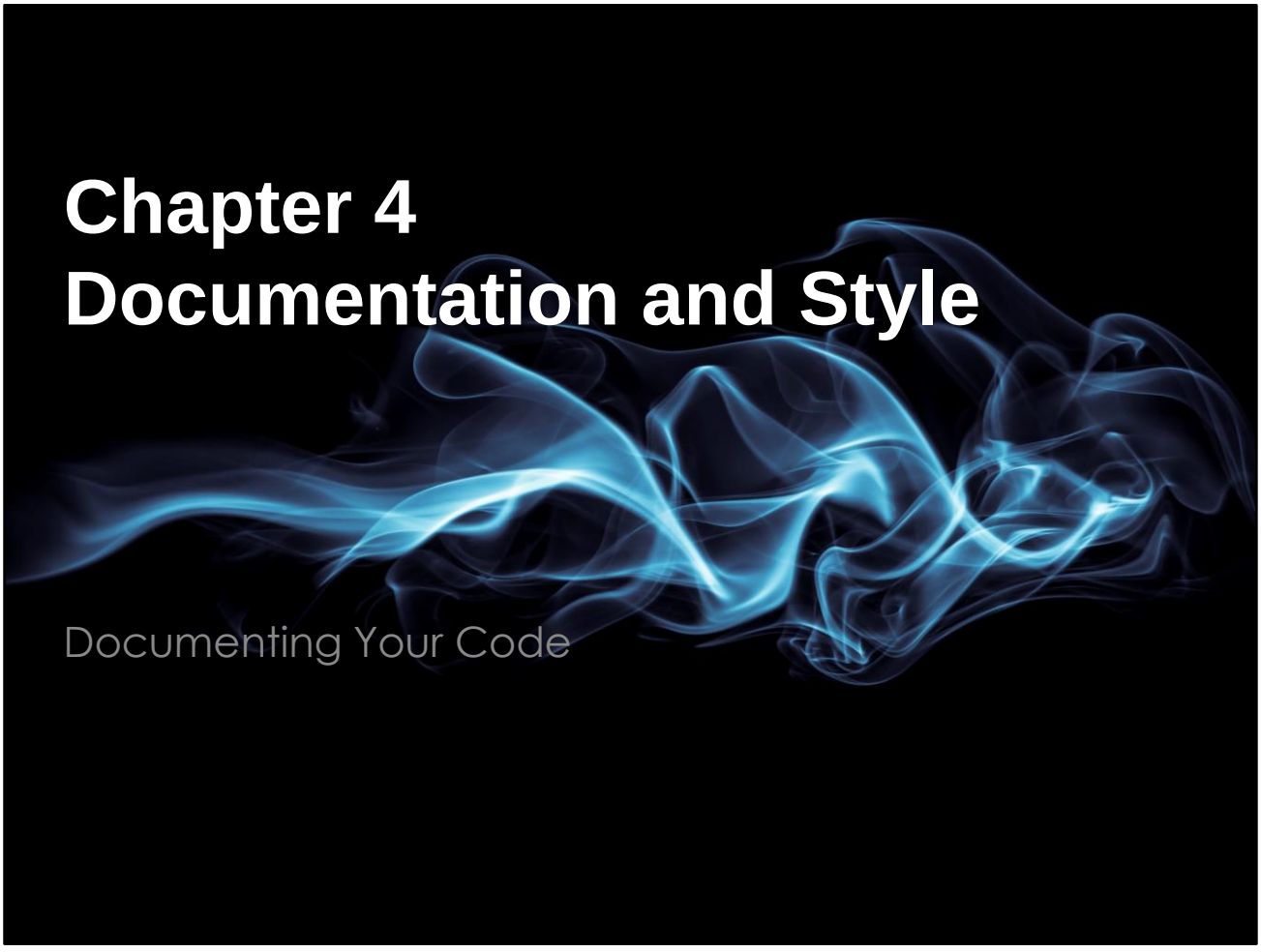
Use your completed task2_3_starter.py to help complete city_search.py and task3_1_starter.py

- Move functions into the city_search.py module
  - Modify the search() to return a list of *City* objects
  - (Optional) Use *PrettyTable* to display City search results (requires pip installing PrettyTable)

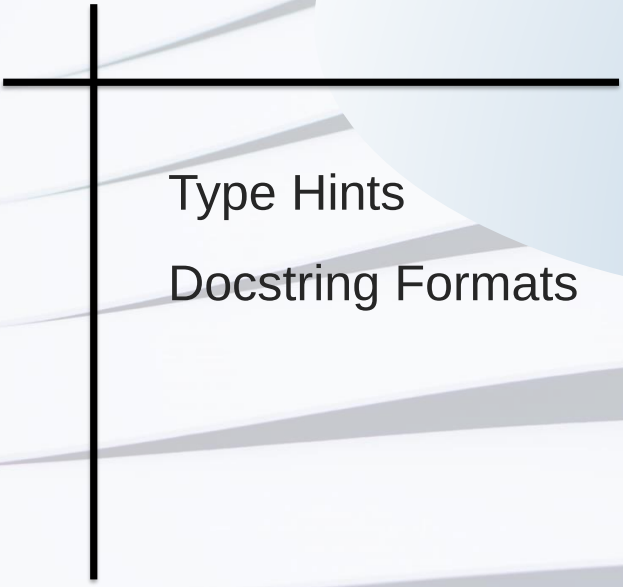
Requires being able to pip install PrettyTable

Summary

- Modules are Python's primary organizational unit
  - Create modules according to functionality (e.g., a module should do one thing well)
- Modules are imported according to the order identified in sys.path
  - If module names are not unique, the first one found in sys.path will be imported
- Relative imports only need to be used when you are distributing packaged software
  - And even then, use it sparingly



Overview



Type Hints

Docstring Formats

Type Hints (Annotations)

- Function type hints provide the ability to inform users of a desired type
  - Useful in IDEs or type checkers such as Pylint or for documentation

```
from typing import Any, Union

def display_info(name: str, age: int, spouse: str,
      *children: Any, parents: list,
      **other_family: Union[str, int]) -> dict:
    return display_info.__annotations__

results = display_info('John', 40, 'Sally', 'Tim', 'Sam',
                parents=['Martha', 'Frank'], sister='Amy')
print(results)
```

Return value

children

other_family

```
{'name': <class 'str'>, 'age': <class 'int'>, 'spouse': <class 'str'>, 'children': typing.Any,
'parents': <class 'list'>, 'other_family': typing.Union[str, int], 'return': <class 'dict'>}
```

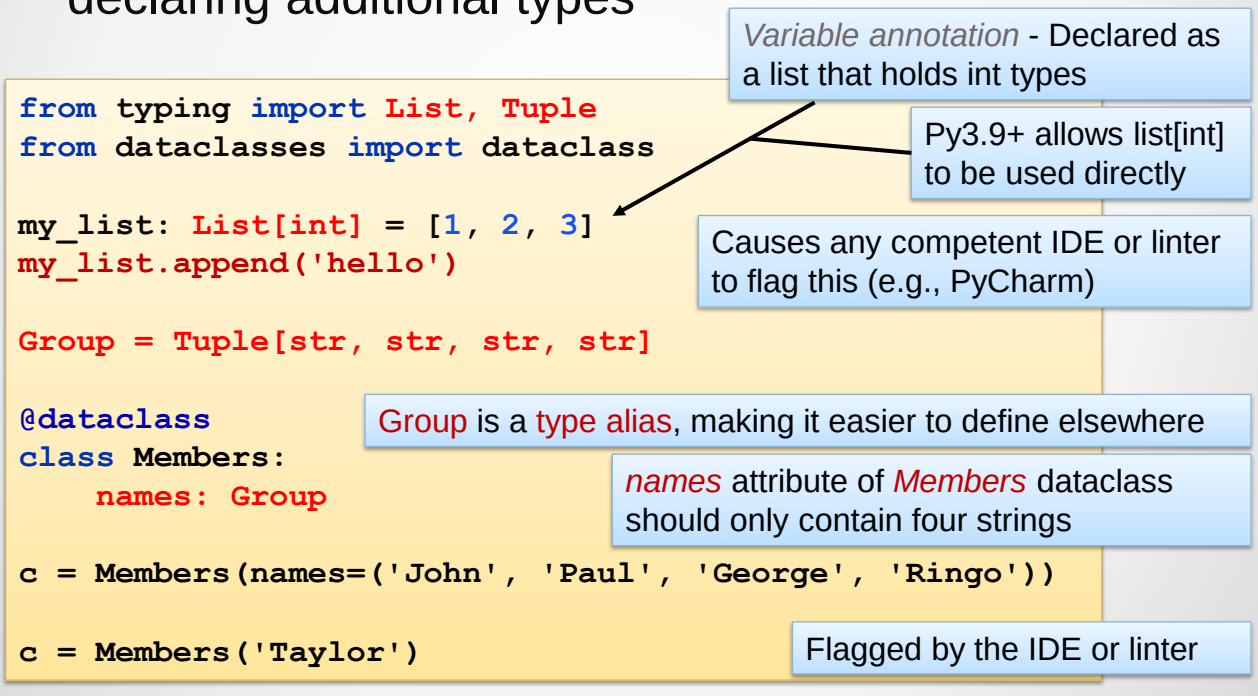
ch04_style/01_annotations.py

Annotations (type hints) are only allowed in Python 3 not Python 2. Use them liberally! The arrow -> dict statement identifies the return value.

Any and Union are defined in the typing module (see next slide). Union says that it can hold either int or string types (mostly for illustration here).

Newer Type Hints

- The typing module provides help when declaring additional types




```
from typing import List, Tuple
from dataclasses import dataclass

my_list: List[int] = [1, 2, 3]
my_list.append('hello')

Group = Tuple[str, str, str, str]

@dataclass
class Members:
    names: Group

c = Members(names=('John', 'Paul', 'George', 'Ringo'))

c = Members('Taylor')
```

ch04_style/01_annotations.py

In Python 3.6, variable annotations were introduced. These allow for variables and classes to be annotated (not just functions and function arguments).

The typing module contains numerous types to assist with type hint declarations. Additionally, the collections module has a submodule, called abc, that has abstract types, such as Sequence, Callable, Collection, Iterable, and more.

Starting with Python 3.9+, the container types themselves may specify generic types without the need to import the typing module (as in, list[int]).

Linting Tools

- To assist with code quality checking several tools can be installed:
  - flake8
  - pylint
  - pycodestyle (formerly pep8)
  - Several others

pip install flake8

```
>flake8
.\01_annotations.py:3:80: E501 line too long (80 > 79 characters)
.\01_annotations.py:26:1: E302 expected 2 blank lines, found 1
.\01_pep8.py:1:10: E401 multiple imports on one line
.\01_pep8.py:2:1: F811 redefinition of unused 'os' from line 1
.\01_pep8.py:2:1: F401 'os' imported but unused
.\01_pep8.py:3:1: F811 redefinition of unused 'sys' from line 1
.\01_pep8.py:3:1: F401 'sys' imported but unused
.\01_pep8.py:8:80: E501 line too long (83 > 79 characters)
```

Runs against a single file or a directory

Flake8 is generally a little more thorough than the others. It allows for ignoring rules (e.g., --ignore E401, E501) or examining only specific rules (e.g., --select F401).

Docstring Formats

- Docstrings are placed at the top of modules, within classes, and within functions
  - There are different docstring formats (see source file)
  - Most common: reStructuredText (reST) (PEP 287)
  - Other documentation formats include:
    - NumPy
    - Epytext
    - Google
    - Custom

```
C:/student_files/ch04_style/03_doc_styles.py
def reStructuredText_style(arg1: int,
                           arg2: object) -> list
```

This is the reST style. It supports HTML and LaTeX formats.

Note

Parameters

Params: arg1 – this is the first argument
arg2 – this is the second argument

Returns: the list of input arguments

```
def reStructuredText_style(arg1, arg2):
    """
    This is the reST style.  It supports HTML and LaTeX.
    .. note:: Parameters
    :param arg1: this is the first argument
    :type arg1: int
    :param arg2: this is the second argument
    :type arg2: typing.Any
    :returns: the list of input arguments
    :rtype: list
    """
    return [arg1, arg2]
```

ch04_style/03_doc_styles.py

Use formats, like restructuredText, to create documentation in other formats such as HTML.

Documentation Generation

- Numerous tools exist for generating documentation in Python
  - Sphinx — Most commonly used — **pip install sphinx**
  - Pdoc - one of the most versatile tools — **pip install pdoc3**
  - Pydoctor - Epytext (default) and reST support
  - Doxygen - support for many languages
  - Others

The *requests* (third-party module) documentation is generated using Sphinx

Inspect Module

- While not a common tool, Python's inspect module can examine underlying code fragments

```
from itertools import chain
import inspect

def my_func(a, b=1, c=2, *d, e, f=3, **g) -> list:
    frame = inspect.currentframe()
    args, varargs, varkwargs, locls = inspect.getargvalues(frame)


    print(f'args={args}')
    print(f'varargs={varargs}')
    print(f'varkwargs={varkwargs}')


    return [(i, locls[i])
                  for i in chain(args, varargs, varkwargs)]
print(f'all results={my_func(10, 20, e=40, x=50)}')
```

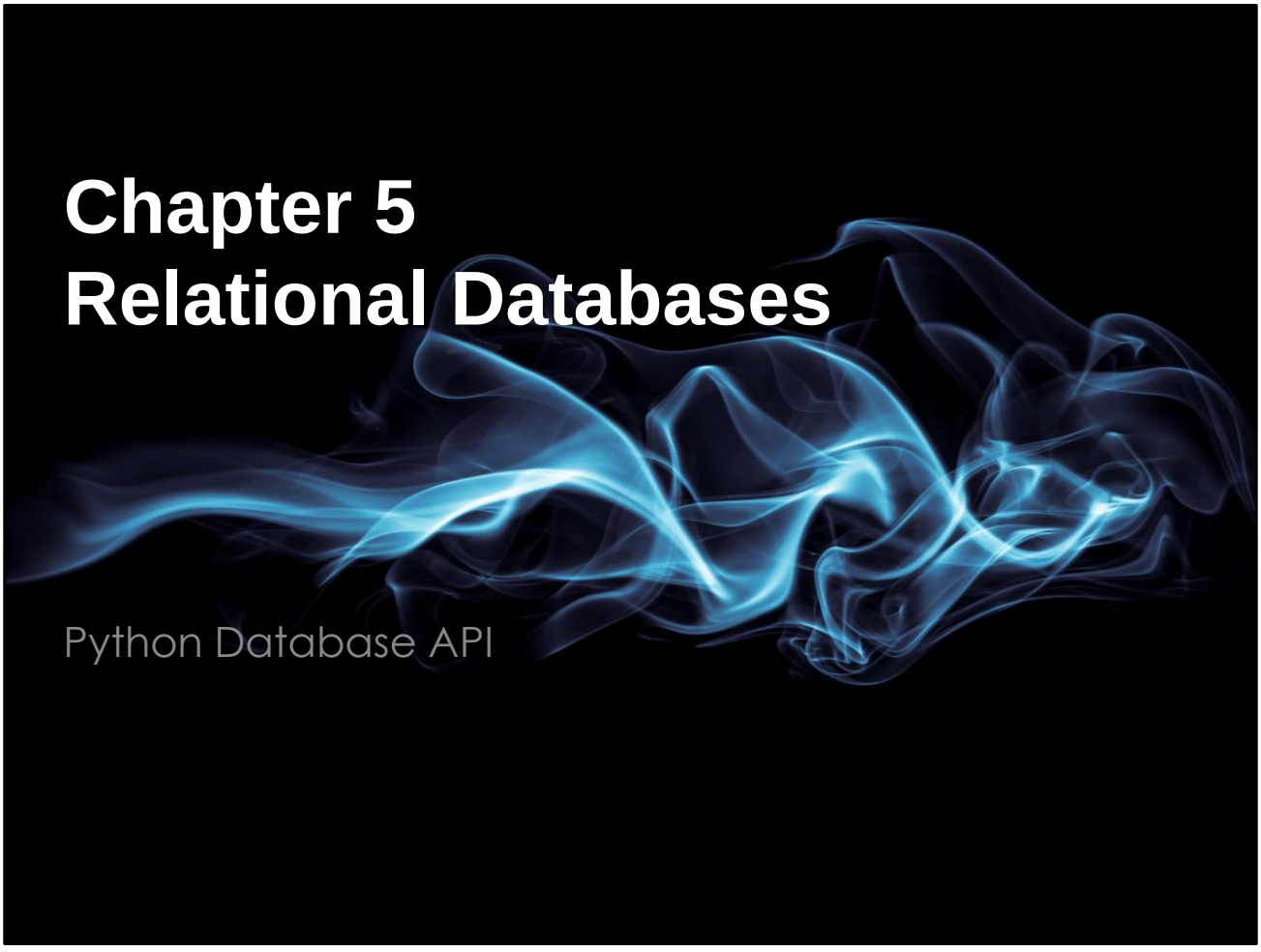
```
args=['a', 'b', 'c', 'e', 'f']
varargs=d
varkwargs=g
all results=[('a', 10), ('b', 20), ('c', 2),
('e', 40), ('f', 3), ('d', ()), ('g', {'x': 50})]
```

ch04_style/04_inspect.py

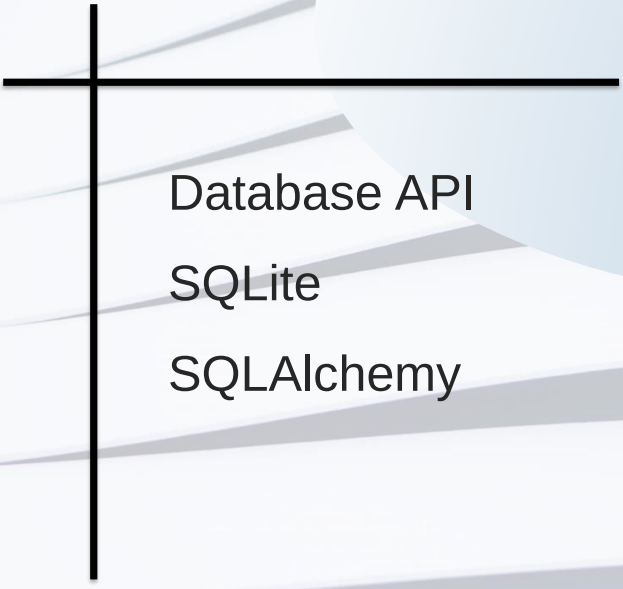
This example shows that we can inspect all kinds of information within a function. The inspect module allows us to view the current stack frame and then use it to obtain various pieces of information about the function. While this module is generally not necessary, it can be useful when creating tools that are "inspecting" code provided by the user. As such, a nice implementation to use here is a decorator. Decorators are discussed in a later chapter.

Summary

- Function and variable annotations have become an important part of the language and should be used frequently
- With a dynamically typed language, documentation is key for users to understand your code
  - Pick a style that works best for you
- Generate documentation using tools such as Sphinx



Overview



Database API

SQLite

SQLAlchemy

Python Database API 2.0

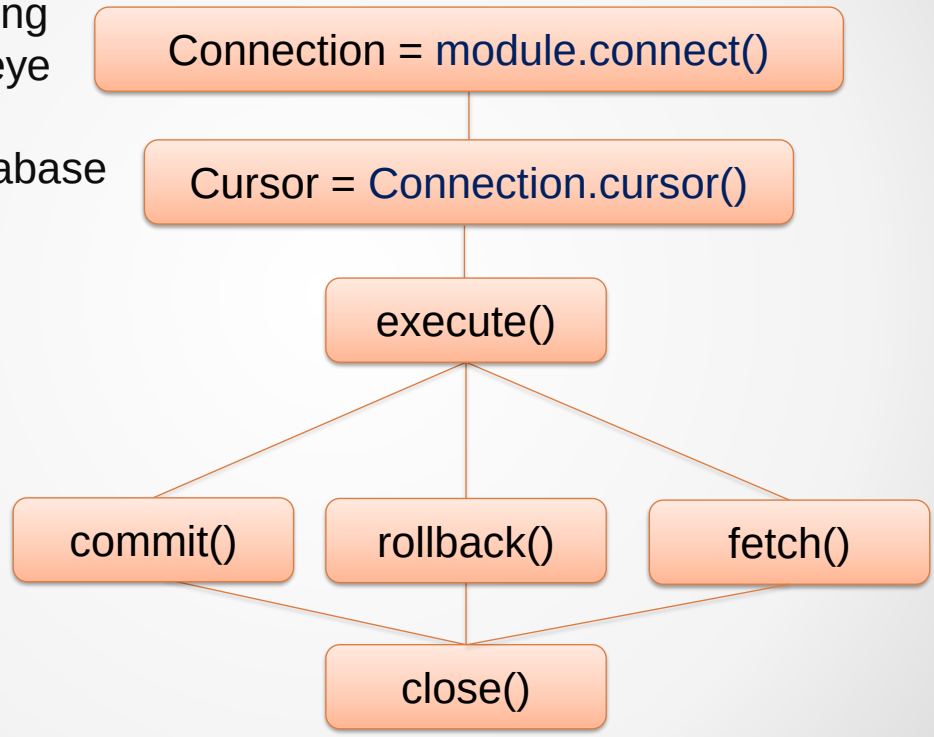
- The Python DB API 2.0 (PEP 249) defines a common interface for modules to connect to and work with relational databases
- The interface defines:
  - Connection Objects and Transactions
  - Cursors Object Operations
  - Input, Output Data Types
  - Error Handling
  - Two-Phase Commits

There is no standard library module (except SQLite3). 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	pip install cx_Oracle

- 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()
```

Keep in mind that each system is different, and you may need to activate a virtual environment and/or issue pip-specific commands, such as pip3, pip3.9, etc. Also, depending on your installation, you may need to run pip using sudo.

Notice how connecting to Oracle involves the steps mentioned in the birds-eye view on the previous slide. 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

Other vendor methods may exist, but these are defined by the spec

For MySQL, there are several prominent drivers. MySQLdb and MySQL Connector. MySQLdb lost favor since it didn't properly support Python 3 and gave rise to drop-in replacements such as mysqlclient and PyMySQL. However, the official driver respected by Oracle is the MySQL Connector driver.

Cursor Methods

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

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

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 that is 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('schools.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')]
```

ch05_database/01_quick_fetch.py

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 \
    (name, balance, rate, acct_type) VALUES (?,?,?,?,?)", data)
```

ch05_database/02_execute.py

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

ch05_database/03_fetching_rows.py

This solution reads back 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

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

```
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()
```

ch05_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()
```

ch05_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 *with*

- You might expect the *with* control to close database connections automatically
  - However, *with* behaves differently per database
  - It 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 with behaves:
cx_Oracle	Closes the connection
ibm_db	Not supported
pyodbc	Commit/Rollback
psycopg2	Commit/Rollback

With behaves differently depending on the database

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

ch05_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, it 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.

Closing Connections

- The contextlib module provides utilities for working with 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 the connection in closing() will automatically invoke its close() method *but now not automatically commit/rollback*

ch05_database/05_with_closing.py

In this version, the connection object will be closed for us 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

ch05_database/06_fetching_rows_column_names.py

This version uses a Row class instead of a tuple for the returned records. This allows fields to be accessed using column names instead of positions.

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

```
pip install 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*

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')
```

ch05_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().

SQLAlchemy Class Mapping

- 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

ch05_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. A one-to-many relationship could be modeled as follows:

```
class School(Base):
    __tablename__ = 'schools'
    school_id = Column(String(30), primary_key=True)
    students = relationship('Student')

class Student(Base):
    __tablename__ = 'students'
    student_id = Column(String, ForeignKey('schools.school_id'))
```

SQLAlchemy Sessions

- Step 3. Use a Session to perform work
  - Sessions mark 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

ch05_database/07_schools_sqlalchemy.py

The Session class is created dynamically by the sessionmaker() function based on the database engine object that is handed to it.

There are dozens of session methods and attributes including:

add(obj) - adds a new object into the session for SQLAlchemy to insert into the database and to track changes on

delete(obj) - deletes an object that has been previously retrieved from the database

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

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, autocommit=True)

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

ch05_database/07_schools_sqlalchemy.py

A complete picture can be seen of the SQLAlchemy schools table database interaction.

Your Turn! Task 5-1

- 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:

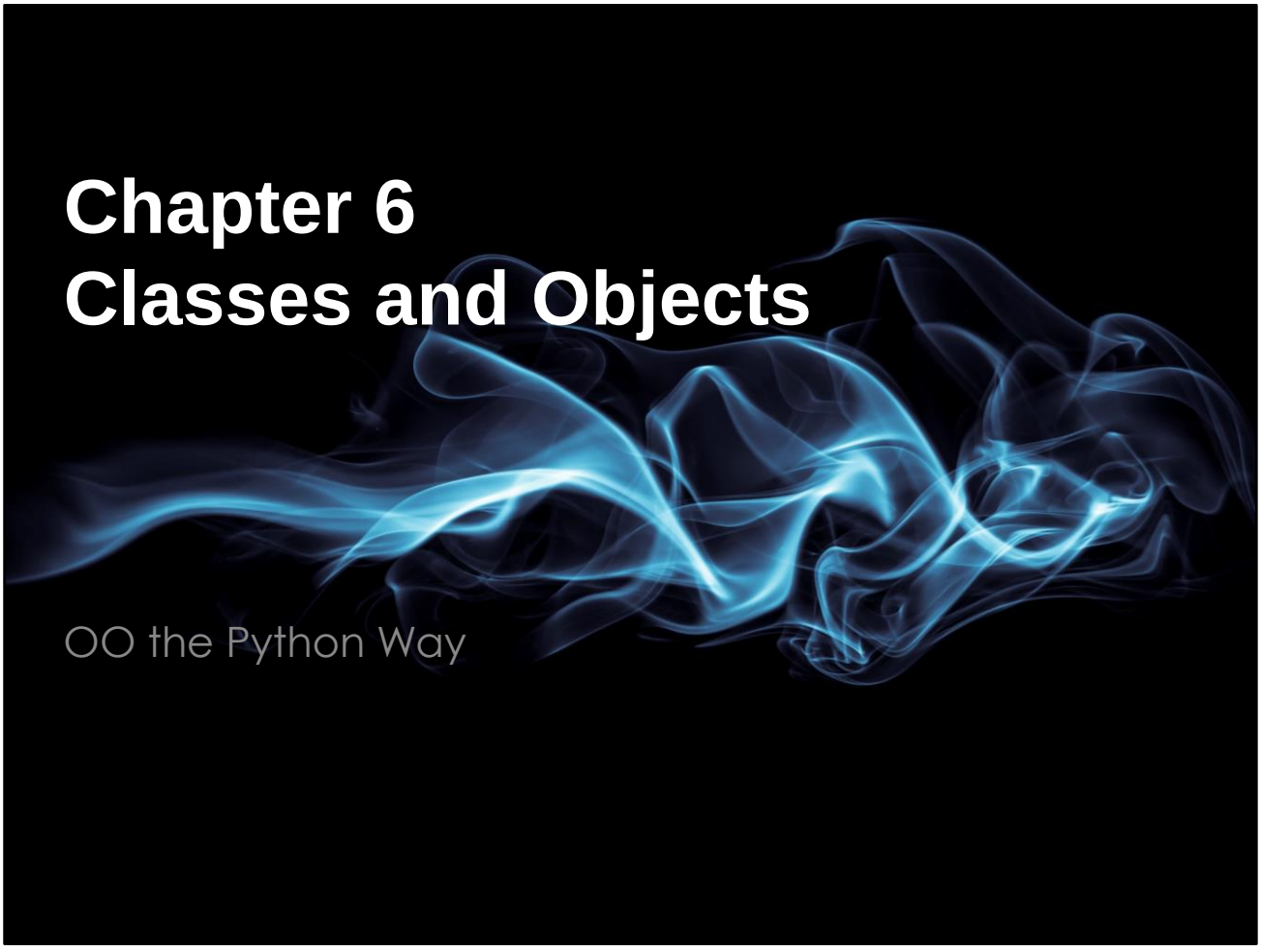
```
School name (or partial name): loy
Matches for loy:
Loyola University Chicago (Chicago, IL))
Loyola Marymount University (Los Angeles, CA))
Loyola College in Maryland (Baltimore, MD))
Loyola University New Orleans (New Orleans, LA))
```

  - A starter file, called task5_1_starter.py in the ch05_database folder has been provided for you
  - *Complete the provided get_location()* function
  - Retrieve results into a *list of data classes*

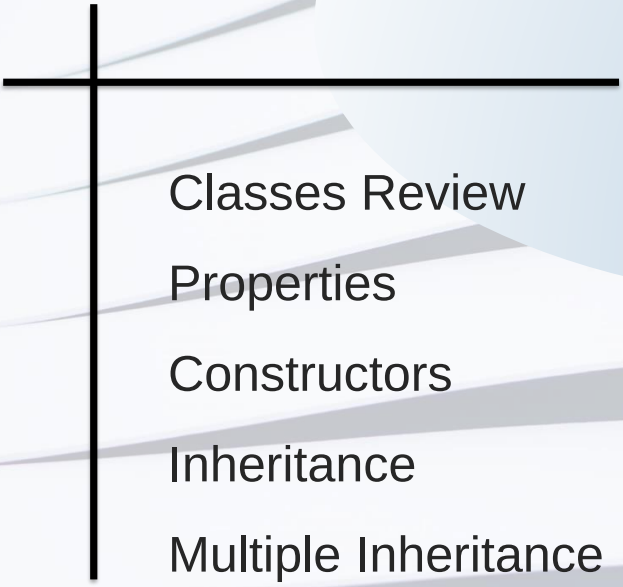
Note: The database should already exist, however, if it doesn't, run 08_create_schools_db.py to repair it.

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



Overview



- Classes Review
- Properties
- Constructors
- Inheritance
- Multiple Inheritance

Classes Review *(1 of 2)*

```
from typing import NamedTuple, List

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

ch06_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.

Classes Review *(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')]
```

ch06_oo/01_review.py

In this example, the object called 'c' is being created. In the constructor, the 'c' object is what self is referring to. While theoretically self can be renamed to something else, in Python, we ***never*** do this.

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 actually **c = City( )** in Python

Methods 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

ch06_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 re-organize the call into `CitySearch.highest(c)`.

Instances as 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
```

ch06_oo/02_as_dicts.py

Under typical circumstances, objects are backed by dictionaries. This means that under the hood, there is a dictionary as the basis for that instance's implementation. Certain tools, such as the json module's dumps() method will depend upon this concept.

The __dict__ property can allow 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 are 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

ch06_oo/03_properties.py

Notice that within the constructor the setter methods are actually invoked. This way if a City object is created and an invalid elevation or population is passed into it, it will still invoke the validation within the setter. Properties provide an ability perform setter, getter, and deleter interactions.

```
@property
def foo(self):
    return self._foo
```

```
@foo.setter
def foo(self, foo):
    self._foo = foo
```

```
@foo.deleter
def foo(self):
    del(self._foo)
```

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)
```

Invokes elevation() setter

Invokes population() getter

8175133
Cloudland 0 0

Constructor invokes setters and therefore these values will be invalid

Here, the property is used by performing an assignment (c.elevation = 10). The assignment invokes the property setter 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

ch06_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 a self passed, as in:

```
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()

ch06_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 Constructor 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'}
```

ch06_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)
        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)
```

super() invokes the parent constructor

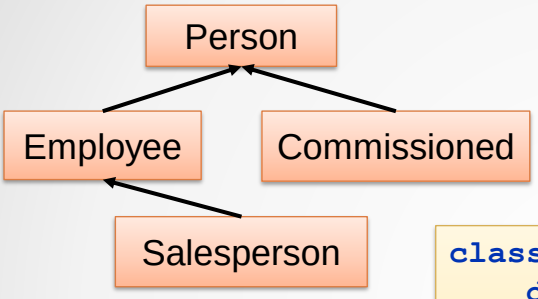
```
John Smith 123 Main St. {'work': '(970) 322-9088', 'home': '(970) 455-2390'}
```

ch06_oo/07_inheritance_super.py

By default, super() will automatically assume arguments of the current class and the self object.

In other words:
(Py3) super() = super(CurrentClass, self) <--super() will look to the immediate parent class to perform the call to the function that follows. self refers to the object that will be worked on. With single inheritance, the use of super() is preferred.

Multiple Inheritance (Explicit) *(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}'
```

ch06_oo/08_multiple_inherit_explicit.py

Multiple Inheritance (Explicit) *(2 of 2)*

Specify both base classes

```
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)
```

Use the classic (explicit) technique to call both constructors

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

ch06_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(). This approach 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
```

Only one super() call can exist, so both constructors must be called at once

```
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')
```

ch06_oo/09_multiple_inherit_super.py

For brevity, the complete class definitions are not shown.

We only get one call to super() in the Salesperson __init__. Yet we have to send both parents the needed arguments. 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 constructors for super() to be successful. Because this requires "touching" (going back and re-writing) both parent classes to support a **kwargs argument, it makes it impractical, "hackey," and in some cases not even possible. Therefore, use the explicit approach, previously discussed, for multiple inheritance.

Other Magic Methods

- Python classes support many magic methods

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

Many of these are rarely 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)
print(c2 == c3)
```

True
False

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

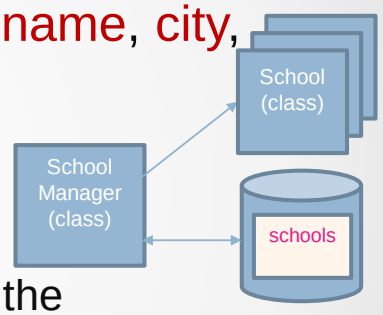
ch06_oo/10_magic_methods_again.py

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.

Your Turn! Task 6-1

Working with Classes

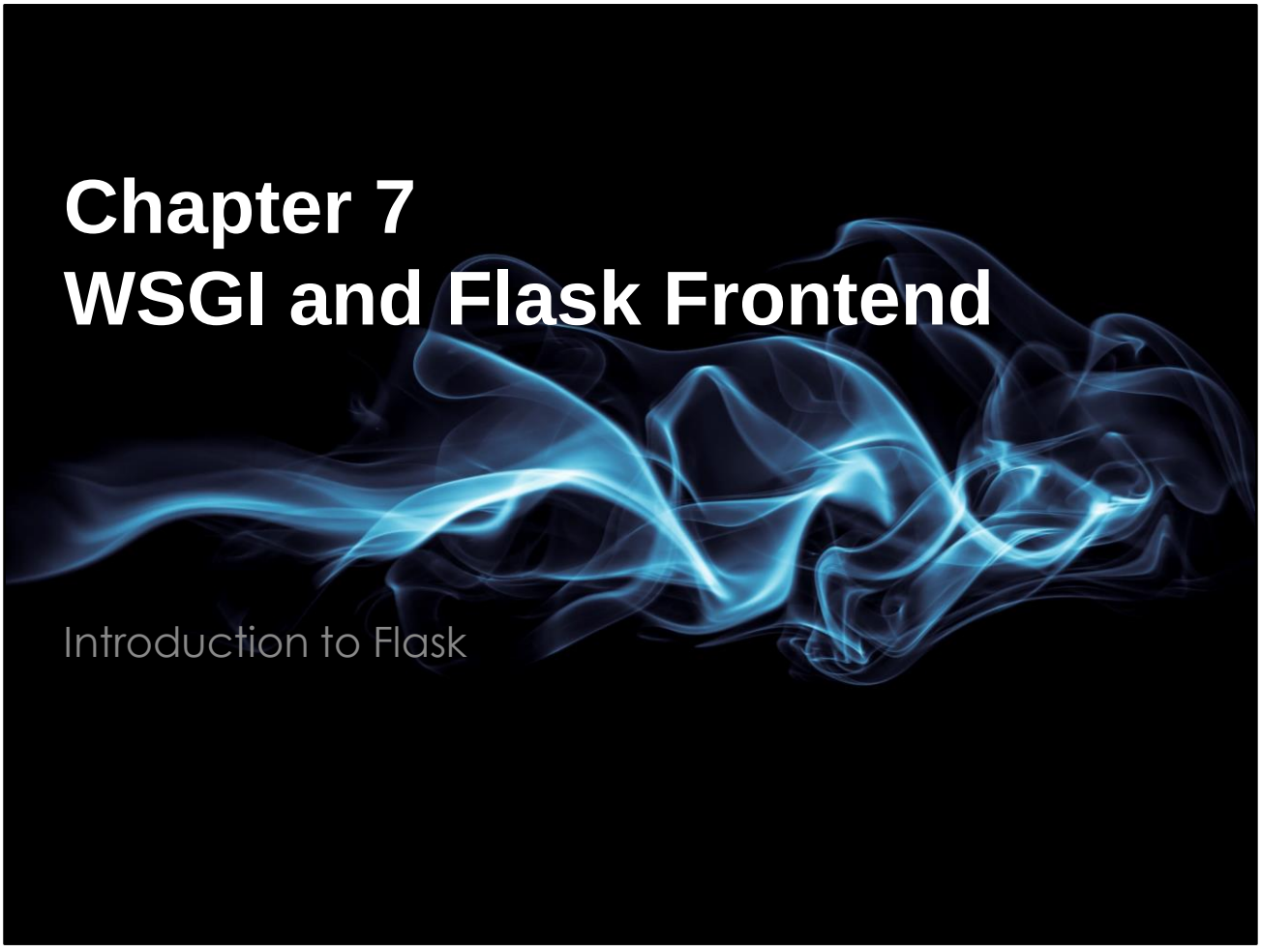
- Work from ch06_oo/task6_1_starter.py to create a *School* class containing school_id, fullname, city, and state, and country attributes
- Create a second class, *SchoolManager,* containing __init__(), and find() methods
  - The SchoolManager find() should search the fullname, city, state, or country columns
  - Test the SchoolManager as follows:



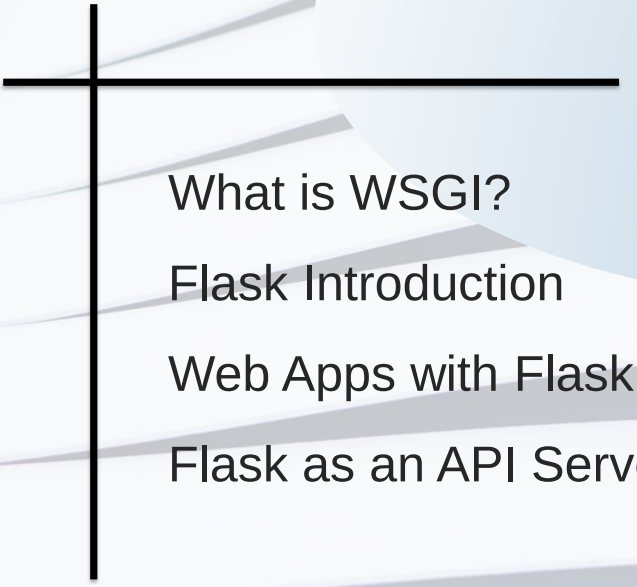

```
if __name__ == '__main__':
    print(SchoolManager('course_data.db').find('Loyola',
                                               column='fullname'))
    print(SchoolManager('course_data.db').find('ID',
                               column='state', sort_by='fullname'))
```

Summary

- Classes in Python have numerous differences from classes found in other languages:
  - They behave as a kind of namespace (code holders)
  - *self* is not implicit, but must be declared in methods
  - There are no public or private keywords
  - Properties can be defined to provide a form of setter/getter capability
  - Classes support multiple inheritance and super()
  - Dozens of magic methods tailor their behavior



Overview



What is WSGI?

Flask Introduction

Web Apps with Flask

Flask as an API Server

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 create your own WSGI server (not common), use the **`wsgiref`** module from the standard library…

WSGI (Sample)

```
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
```

ch07_frontend/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.

Python Web Frameworks

- A number of Python frameworks already exist that implement WSGI
  - Basic Frameworks
    - Bottle
    - Flask
    - CherryPy
    - FastAPI
  - Full-stack Frameworks
    - Django
    - web2py
    - TurboGears
    - Pylons
    - Zope
    - Wheezy

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

Framework Comparisons

- Comparing the two most popular frameworks

Flask

- Common as an API host
- Learning curve: shallow
- Ships bare bones, but pluggable
- Hundreds of plugins exist
- Easy to implement as an API host
- Can serve web apps, but less common for this

- Flask Implementations: Pinterest, Twilio, Netflix,

Django

- Common for web app hosting
- Learning curve: steep
- Fully featured out-of-the-box
- Plugins, less extensive than Flask
- Can be an API (Django Rest Framework) but not as common as Flask
- Common to implement Django frontend (e.g., serving Angular/React, etc.) while using Flask as an API

- Django Implementations: Instagram (on AWS), numerous others

The list of companies and sites using both are numerous.

https://www.djangosites.org/
https://careerkarma.com/blog/companies-that-use-flask/

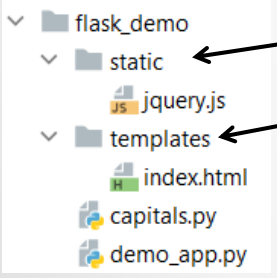
Flask Introduction

- A lightweight, extensible, WSGI web application framework (BSD license)

https://flask.palletsprojects.com

pip install flask

- A flask application should contain the following structure



Items not managed by Flask

Items loaded with render_template()

Flask-based code

Quick overview:
https://flask.palletsprojects.com/en/1.1.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
- Decorator defines URL to function mappings

demo_app.py

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

app = Flask(__name__)


@app.route('/')
def main_page():
    return  render_template('index.html')

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

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

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

ch07_frontend/flask_demo/demo_app.py

More with Routes

- Add additional methods and routes as more requests are needed

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

```
@app.route('/state/<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 Response(resp.data, status=200,
                    mimetype='application/json')
```

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

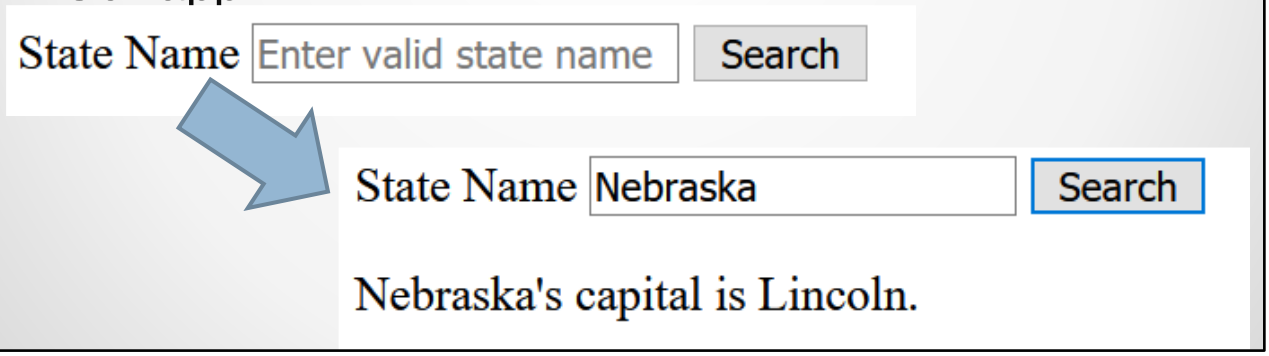
ch07_frontend/flask_demo/demo_app.py

More with Routes

```
@app.route('/state', methods=['GET'])
def list_states():
    resp = jsonify(names=[state for state in capitals], foo='bar')
    return Response(resp.data, status=200,
                    mimetype='application/json')
```

{"foo":"bar",
"names":["Alabama","Alaska","Arizona",...,"Wyoming"]}

- Our app:



ch07_frontend/flask_demo/demo_app.py

The Main Web Page

```
<div>
    <form id="stateSearch">
        <label for="name">State Name</label><input type="text" id="name">
        <input type="submit" value="Search">
    </form>
</div>
<div id="results"></div>
<script type="text/javascript">
    var url = '/state/';

    $('#stateSearch').submit(function(evt){
        evt.preventDefault();
        var name = $('#name').val();
        if (name.length > 0)
            $.ajax({url: url+name,
                dataType : 'json',
                success: function(results){
                    $('#results').append('<p>' + results.state +
                        '\'s capital is ' + results.capital +'.</p>') }
            });
    });
</script>
```

Simple browser (client) code communicates with the server using jQuery and Ajax

Partial listing of templates/index.html

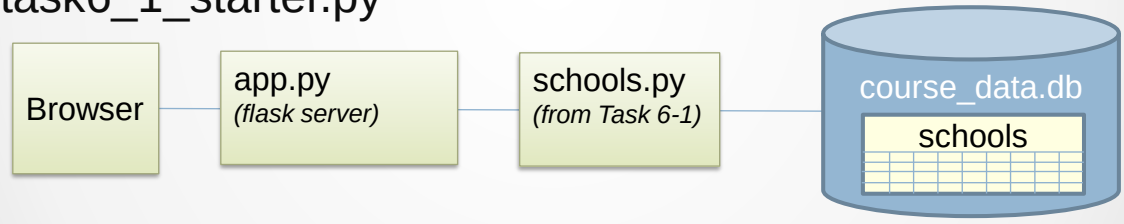
Your Turn! Task 7-1

Working with Flask

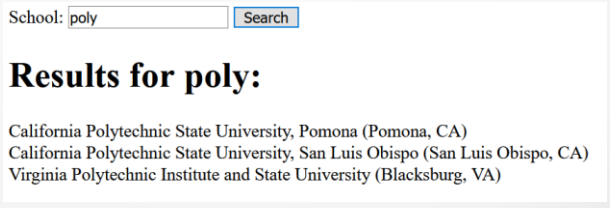
- Complete the Flask-based application:
  - Locate the ch07_frontend/starter/app.py folder

Refer to this file for additional instructions

- schools.py is now the completed version of task6_1_starter.py

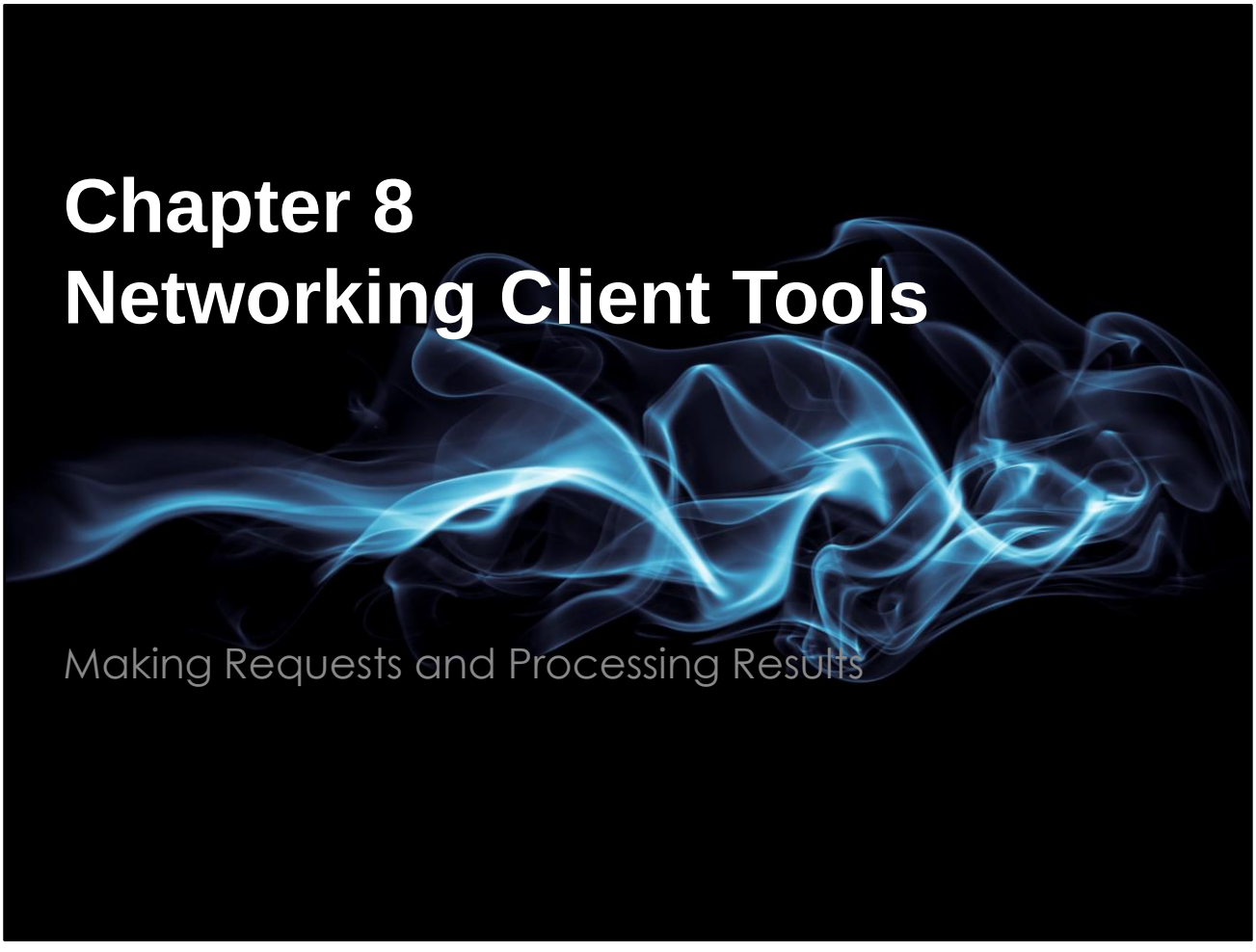


localhost:8051

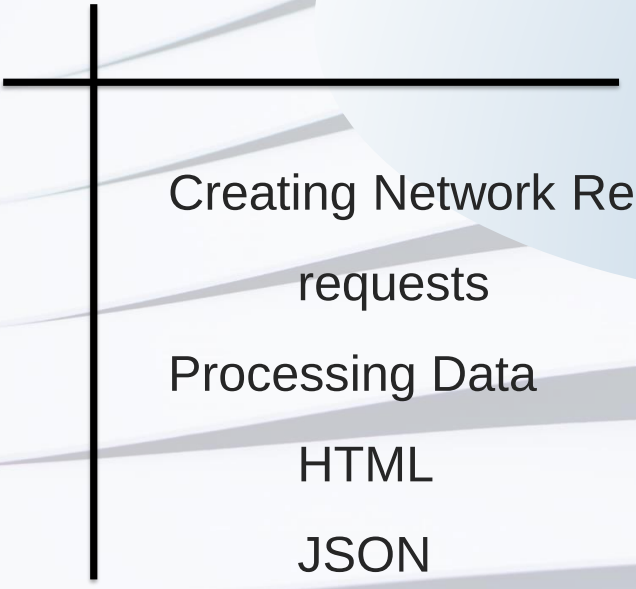


Summary

- Flask is a WSGI compliant web-based framework designed to easily create interactive client-to-Python applications
- It supports much more than we have discussed, including:
  - Web Forms (flask-wtf plugin)
  - Database ORM Integration (flask-sqlalchemy plugin)
  - Security and User Management (flask-login plugin)
  - Many other plugins



Overview



Creating Network Requests

requests

Processing Data

HTML

JSON

urllib

- The urllib is used to make network requests

urllib is broken across four modules
urllib.request - urlopen() and Request
urllib.response - read() and readline()
urllib.parse - URL building/parsing tools
urllib.error - Error handlers

```
from urllib.error import URLError
from urllib.parse import urlencode
from urllib.request import urlopen, Request

query_str = {'value': 'foo', 'column': 'fullname',
             'sort_by': 'state'}
url = 'http://localhost:8051/school?' + urlencode(query_str)

try:
    with urlopen(url) as f:
        results = f.read().decode()
        print(f.status, f.reason)
        print(results)
except (URLError, UnicodeDecodeError) as err:
    print(f'Error: {err}', file=sys.stderr)
```

http://localhost:8051/school?value=Loyola&column=fullname&sort_by=state

with closes the connection afterward

200
OK

{"column":"fullname","error_msg":"","results":["Foothill College (Los Altos Hills, CA)"],"sort_by":"state","value":"foo"}

ch08_network_apis/01_urlopen.py

Over the years, the urllib module has lost favor and is largely replaced by the requests module (upcoming slide).

Example using urllib.parse.urlparse():

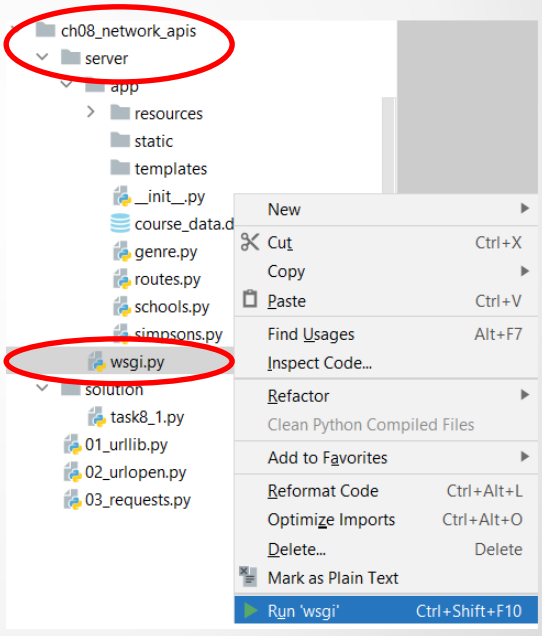
from urllib.parse **import** urlparse
result = urlparse(**'https://docs.python.org/3/library/index.html'**)
print(result)

Output:
ParseResult(scheme='https', netloc='docs.python.org',
path='/3/library/index.html', params='', query='', fragment='')

How to Run the Example

- Our Flask server from the previous chapter now supports additional requests
  - To run our server, from within PyCharm, right-click and Run the wsgi.py file within ch08_network_apis/server

You will need to approve when PyCharm prompts to allow the server to run



Working with the Server

- With the server running, the following URL are valid

```
http://localhost:8051/school?value=Loyola&column=fullname&sort_by=state
```

Sample output:


```
{"column":"fullname", "error_msg":"",
 "results":["Loyola Marymount University (Los Angeles, CA)",
            "Loyola University Chicago (Chicago, IL)",
            "Loyola University New Orleans (New Orleans, LA)",
           "Loyola College in Maryland (Baltimore, MD)"],
 "sort_by":"state", "value":"Loyola"
}
```


```
http://localhost:8051/simpsons?char_name=burns
```

Sample output:


```
{"name":"burns",
 "results":[{"actor":"Christopher Collins (early season 1), Harry Shearer",
             "episode_debut":"\"Simpsons Roasting on an Open Fire\"",
             "name":"Charles Montgomery Burns",
             "original_air_date":"1989-",
             "role":"Owner of the Springfield Nuclear Power Plant."}
]}
```


These URLs are available with the shown query strings (parameters) when the ch08_network_apis/server/wsgi.py server has been started.

The Python *requests* Module

- A popular Python 3rd party tool, called requests, is ideally suited for constructing web application network requests

pip install requests

  - Typical usage syntax

Useful req object attributes:

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

```
import requests

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

While requests must be installed, it is very much worth it and is one of the best modules in all of Python.

Using Requests

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

Google Speech Recognition Library in Python has Extremely Slow Times
How to call a python function that returns a different number of objects
Drop the value that matches the string from list in python
…
Asynchronous execution of Python subprocess.Popen with wait()
If statement combined with or statement to compare list elements in Python

- Alternate way (adding params later)

```
feed = 'https://api.stackexchange.com/2.2/search'
query_str = {'intitle': 'python', 'site': 'stackoverflow'}
data_dict = requests.get(feed, params=query_str).json()
print([question.get('title')
               for question in data_dict.get('items', [])])
```

ch08_network_apis/03_requests.py

The example shows how requests can make an HTTP get request, retrieve JSON-based data, and convert it to a Python dictionary using the .json() method.

request.get() supports an optional ***params*** argument that can be a dict, list, or string of query parameters to send in the request.

Requests and Basic Authentication

- Requests supports most types of authentication schemes including custom schemes
- It can make requests to perform Basic Authentication using the following syntax:

```
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)
```

ch08_network_apis/03_requests.py

Use the HTTPBasicAuth class in the requests.auth submodule to set up a Basic Authentication challenge. Requests will take care of the rest for you.

BeautifulSoup to Parse Results

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <title>Basic Authentication test page</title>
<!-- Changed by: Yves Lafon, 22-Feb-1999 -->
  </head>

<BODY BGCOLOR="white">
<P>
      <A HREF=".."><IMG SRC="/icons/jigsaw" ALT="Jigsaw" BORDER="0"
WIDTH="212"
              HEIGHT="49"></A><IMG SRC="/icons/jigpower.gif" WIDTH="94"
HEIGHT="52" ALT="Jigsaw Powered !"
            BORDER="0" ALIGN="Right">

<P>
<HR>
<P>Your browser made it!
  </body>
</html>
```

The following is the result following the Basic Auth request

```
soup = BeautifulSoup(page_text,
                     'html.parser')

print(soup.select('p')[2].text)
```

ch08_network_apis/03_requests.py

BeautifulSoup even handled the malformed HTML.

BeautifulSoup to Parse Results

- BeautifulSoup is a popular third-party Python add-on for parsing single HTML pages

```
from bs4 import BeautifulSoup

page = requests.get('http://www.cisco.com').text

soup = BeautifulSoup(page, 'html.parser')

print(soup.title)
print(soup.find_all('h2'))

p_tags = soup.find_all('p')
print(len(p_tags))
print(p_tags[2].text)

high_priority = soup.select('.pri-0')
if len(high_priority) > 0:
    print(high_priority[0].text)
```

find_all() gives back a collection of elements

Uses CSS selector syntax to extract data from a page

ch08_network_apis/03_requests.py

Here, the Cisco home page is loaded by requests, then it is parsed within the BeautifulSoup parser. Numerous things are parsed from the document, including:

- The page title
- All of the <h2> tags on the page
- The text for the 3rd <p> tag on the page
- The text of the first element encountered with the class name of .pri-0.

Using the JSON Module

- 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

ch08_network_apis/04_parsing_json.py

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

The very bottom line works as follows:

1) Obtain the list associated with the results "property" in the JSON data. If there is no "results" property, fallback to a dictionary with an empty list.
2) We then did a .get() on the returned dictionary to retrieve the "actor."

Advanced JSON Handling

- JSON and Python type comparisons

Python Type	JSON Type
int, float	Number
str	string
list, tuple	array
dict	object
True	true
False	false
None	null

- To save a Python object as JSON to a file

```
with open('json_file_save.json', 'wt') as f:
    json.dump(result, f)
```

dumps() converts to a string, while
dump() requires a file-like object to send the output to

ch08_network_apis/04_parsing_json.py

In an opposite way, the Python dict (result) on the previous slide is now converted back to JSON and saved to a file using dump()

Handling Non-serializable Fields

- 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!

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

ch08_network_apis/04_parsing_json.py

Here we are trying to encode the Python Employee object into a JSON format. But we are thwarted by the date field because it can't be converted (serialized). An easy fix to this problem is to use default=str which will convert the date to a string format first. But this doesn't give a lot of flexibility in formatting the date.

Custom Handling Non-serializable Fields

```
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

We manually define how to serialize it in the except block

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

ch08_network_apis/04_parsing_json.py

To customize the conversion on the hiredate, when the hiredate fails to serialize, the default() method of the EmployeeEncoder is invoked. Obj will represent the hiredate object. It will fail in the try block because it has no __dict__ attribute causing vars() to fail. The except block will convert the obj for us and return a formatted result.

Your Turn! Task 8-1

Requests and Processing Responses

- Create a data class to encapsulate Simpsons data
- Retrieve this data from the server (it must be running) using requests
- Use the following URL

You may insert this value any way you wish

```
http://localhost:8051/simpsons?char_name=Krusty
```

  - Insert the results into a dataclass and then write the object out to a file with the air date formatted as `%Y-%b-%d`

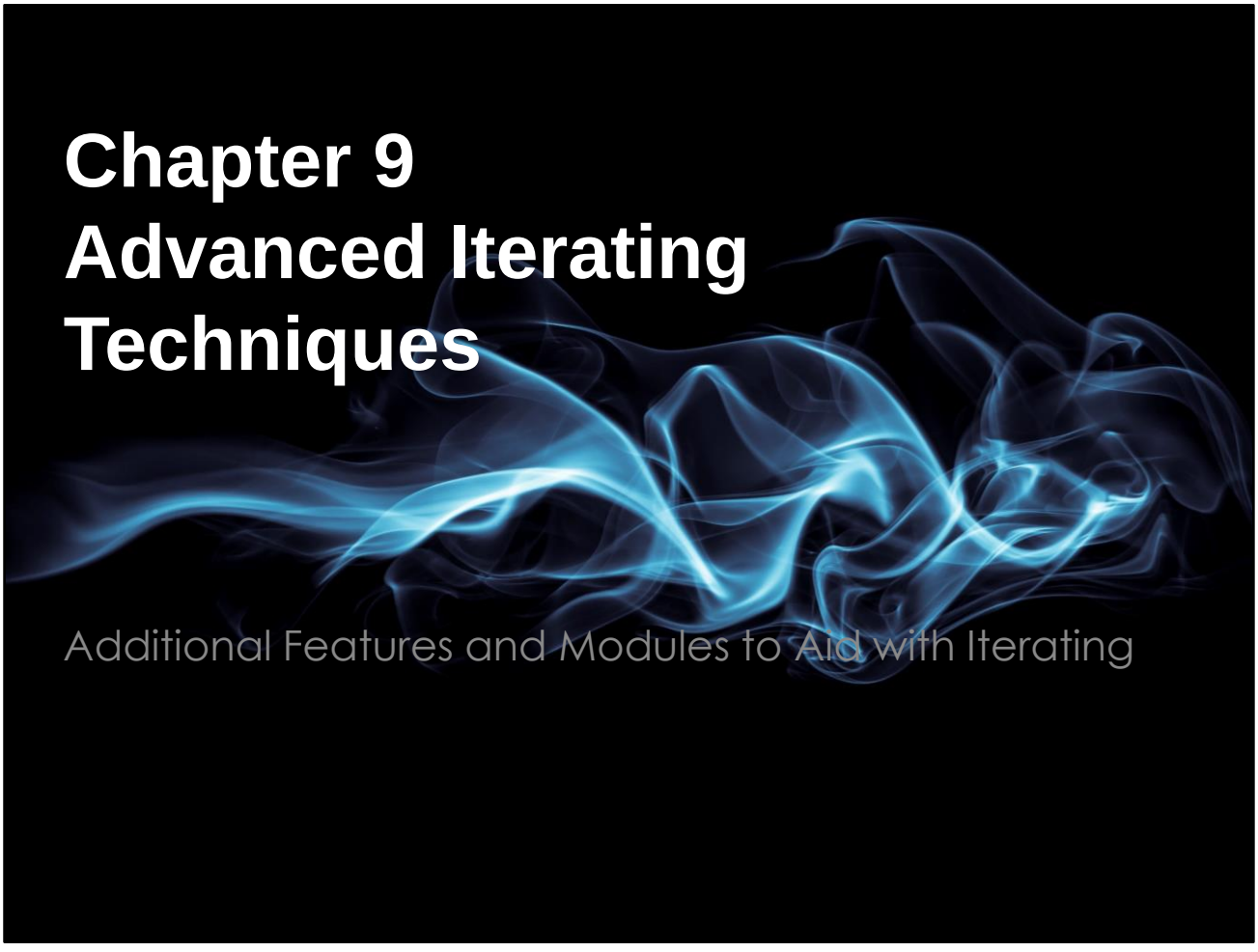
```
@dataclasses.dataclass
class Character:
    name: str
    actor: str
    role: str
    original_air_date: date
```

Follow the instructions within ch08_network_apis/task8_1_starter.py

Summary

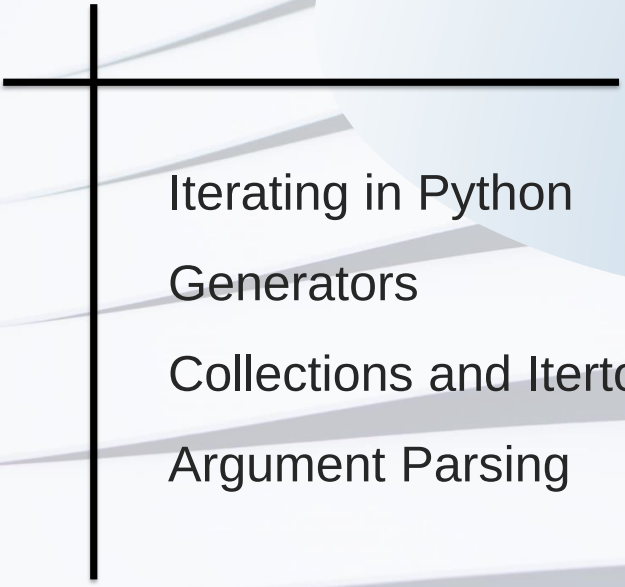
- Prefer requests over urllib whenever possible
- Use requests to parse JSON data whenever possible
- When parsing JSON data, be careful of fields that can't be JSON serialized
  - You will need often to deal with these manually

Chapter 9
Advanced Iterating Techniques



Additional Features and Modules to Aid with Iterating

Overview



Iterating in Python

Generators

Collections and Itertools

Argument Parsing

Creating Iterator Objects

- Python allows for creating custom iterators
  - It must meet two criteria:

```
class Something:
    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
```

```
s = Something()
for i in s:
    print(i, end=' ')
    if i == 3:
        break

for i in s:
    print(i, end=' ')


s1 = Something(2, 5)
s2 = Something(10, 13)
for i in s1:
    print(i)
    for j in s2:
        print(j)
    print()
```

```
0 1 2 3
```

```
0 1 2 3 4
```

```
2
10 11 12
3
10 11 12
4
10 11 12
```

ch09_iterating/01_iterating.py

The two 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
  - A yield in the function makes it return a generator
  - Invoking a generator *does not* begin executing it
  - Generators resume where they left off

When invoked, this function will return a generator object

```
def a_generator(min_val, max_val):
    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

<class 'generator'>

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

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

What will this bottom statement do?

ch09_iterating/02_generator.py

Generators can retrieve or return resources "on-demand" or "as-needed". They work in a similar fashion to the iterator created on the previous slide. Generators have a __next__() method.

How does the generator pause and resume? Internally when a function is called it creates what is called a stack frame. This stack frame is suspended (placed on "pause" if you will) 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.

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='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?

ch09_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)*

Presence of *yield* prevents this from being executed like a typical function

```
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

ch09_iterating/03_read_words.py

The generator-based version works more like 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.

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}
```

ch09_iterating/04_comprehensions.py

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}

ch09_iterating/04_comprehensions.py

Generator Expressions

- Generator expressions are like list comprehensions
  - List comprehensions generate the entire data structure *into memory*, however
  - 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?

ch09_iterating/04_comprehensions.py

By far, the most useful in terms of performance and memory consideration is also the least well-known of this group of structures. 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, on the other hand, creates everything in memory up front and then you begin working on it. So, in the example above, 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

Not valid, invokes the str() constructor as a default

Initially `d1['greet2']` will be a KeyError, so the default value from str() which is '' (empty string) will be used

ch09_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)

top_two_most = Counter(items).most_common(2)
print(top_two_most)

top_three_most = Counter(items).most_common(3)
print(top_three_most)
```

[(4, 5)]

[(4, 5), (5, 3)]

[(4, 5), (5, 3), (2, 2)]

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

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

ch09_iterating/06_counter.py

Counter is very easy to use. The tricky part is understanding what is 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))


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


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

chain() returns an iterator, not a new list

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

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

start

stop

Walrus operator (Py3.8+)

10
20
30
40
50

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

ch09_iterating/07_itertools.py

Here are 3 examples of various iterators found in the itertools module.

The first example returns an iterator that iterates (sequentially) over multiple iterables without creating a single structure first.

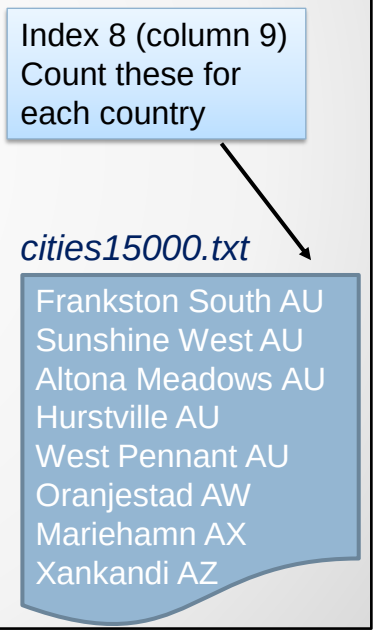
The second example uses the islice() iterator to begin returning lines from a file starting at line seven. It reads the first seven lines (it starts at zero) but throws them out (skips them) and begins providing lines from the file starting at line 7. The None value represents a stop value (None = go until file is finished).

The last example illustrates the count() iterator. It iterates until it is told to stop. There is no upper bound. This can be useful when you don't know how high you need to iterate up to. This example also uses a walrus operator, which performs an assignment in the middle of an on-going expression. Here, due to the assignment syntax, the parentheses are needed around the entire expression in the while.

Your Turn! Task 9-1

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 task9_1_starter.py for more info
  - Determine the 10 ten countries with the most cities over 15000 people?



Your Turn! Task 9-2

Generators and Iterators

- Revisit *task9_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
  - Test it out
  - *Advanced:* If you get it working, attempt your solution using a generator expression

If this were a very large file, what advantage would this version have over Task 9-1?

argparse

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

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

print(args.greet)
```

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 howdya do.
```

Run with -g argument

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

Run with --greet arg

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

ch09_iterating/08_argparse1.py

This module will parse a list of arguments (usually sys.argv command-line arguments when parser.parse_args() is left empty. 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 *(1 of 2)*

```
def get_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--value', nargs='*', required=True)
    parser.add_argument('-c', '--column', default='fullname')
    parser.add_argument('-s', '--sort_by', default='fullname')
    parser.add_argument('-e', '--show_error', action='store_true')

    return parser.parse_args()


args = get_args()
results, error_msg = find(args.value, args.column, args.sort_by)
if results:
    print(results)
elif args.show_error:
    print(error_msg, file=sys.stderr)
```

From our *schools* exercises

ch09_iterating/09_argparse2.py

From the above example:

'-v' or '--value' is required. No default value is supplied.

nargs="*" says to accept multiple values separated by a space. ?, +, or an integer may also be provided to indicate optional, 1 or more, or an exact number of arguments.

Though not shown, values can be typed, as in **type=int**.

action='store_true' means if the switch is present, save it as a boolean with the value True

Argparse Options *(2 of 2)*

Run with no arguments

```
>python 09_argparse2.py
usage: 09_argparse2.py [-h] -v [VALUE ...] [-c COLUMN] [-s SORT_BY] [-e]
09_argparse2.py: error: the following arguments are required: -v/--value
```

Run with -v loyola

```
>python 09_argparse2.py -v loyola
Database connection opened!
Querying...
Database connection closed!
[Loyola College in Maryland (Baltimore, MD), Loyola Marymount University (Los Angeles, CA), Loyola
 University Chicago (Chicago, IL), Loyola University New Orleans (New Orleans, LA)]
```

Run with -v nv -c state -e

```
>python 09_argparse2.py -v nv -c state -e
Database connection opened!
Querying...
Database connection closed!
[College of Southern Nevada (Henderson, NV), University of Nevada-Las Vegas (Las Vegas, NV), Unive
rsity of Nevada-Reno (Reno, NV)]
```

Run with -v co -c state -e -s city

```
>python 09_argparse2.py -v co -c state -e -s city
Database connection opened!
Querying...
Database connection closed!
[University of Colorado (Boulder, CO), Colorado College (Colorado Springs, CO), University of Denv
er (Denver, CO), Fort Lewis College (Durango, CO), Colorado State University (Fort Collins, CO), C
olorado School of Mines (Golden, CO), Mesa State College (Grand Junction, CO), University of North
ern Colorado (Greeley, CO), Lamar Community College (Lamar, CO), Colorado State University-Pueblo
(Pueblo, CO), Trinidad State Junior College (Trinidad, CO)]
```

Run with -v nv -c foo -e

```
>python 09_argparse2.py -v nv -c foo -e
Column not allowed.
```

ch09_iterating/09_argparse2.py

The script is run numerous times with varying options. Each time, argparse is used to parse the arguments and perform the task.

Your Turn! Task 9-3

Argument Parsing

- Re-visit the country counting exercise (either task9-1 or task9-2)
  - Use the argparse module
  - Display the top countries based on a command-line option, as in:

```
python task9_3_starter.py -c 5
```

or

```
python task9_3_starter.py --count 5
```

if no count argument is supplied let the default be 3

Summary *(1 of 3)*

- Let's review the techniques we've learned for 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
```

ch09_iterating/11_iterating_summary.py

Summary *(2 of 3)*

```
for item in reversed([5, 8, 9]):
    print(item)
```

```
9
8
5
```

```
for item in enumerate(reversed([5, 8, 9])):
    print(item)
```

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

```
array1 = ['a', 'b', 'c']
array2 = [30, 40, 50]
for item1, item2 in zip(array1, array2):
    print(item1, item2)
```

```
a 30
b 40
c 50
```

```
from itertools import chain
for item in chain(array1, array2):
    print(item, end=' ')
```

```
a b c 30 40 50
```

ch09_iterating/11_iterating_summary.py

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))
```

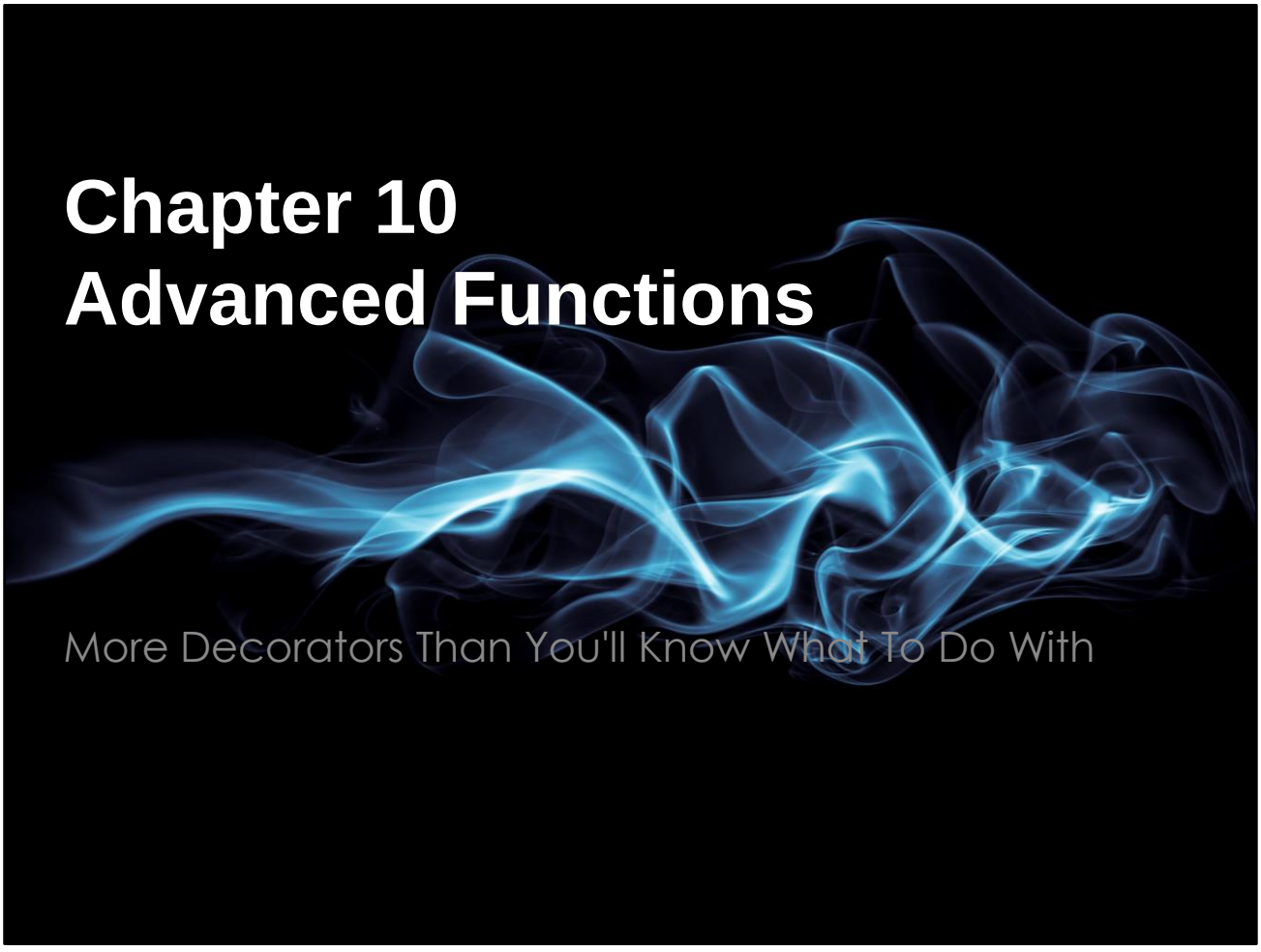
Doesn't behave like a normal function call

You cannot:

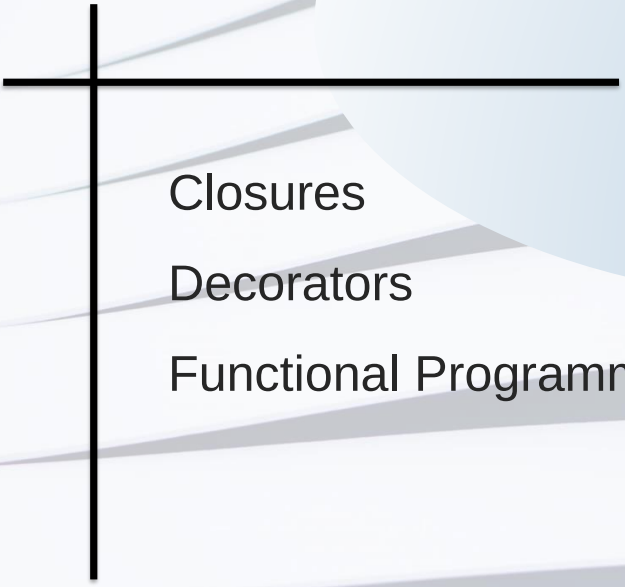
```
print(my_generator())
print(my_generator()[0])
print(len(my_generator()))
```

len() and indexing are not supported

ch09_iterating/12_generator_summary.py



Overview



Closures

Decorators

Functional Programming Methods

Introducing Python Decorators

- Selling point: *what if we could modify someone's code without ever touching that original code?*
  - Decorators can do this *(in a way)*
- Decorators give creators the ability to take control of someone else's code, modify it, and conform it to the way they would like
- *Decorators are a little tricky to wrap your head around*, and as such, we'll ease into them 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.
```

ch10_adv_functions/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
```

ch10_adv_functions/02_decorators.py

Creating and Returning Functions

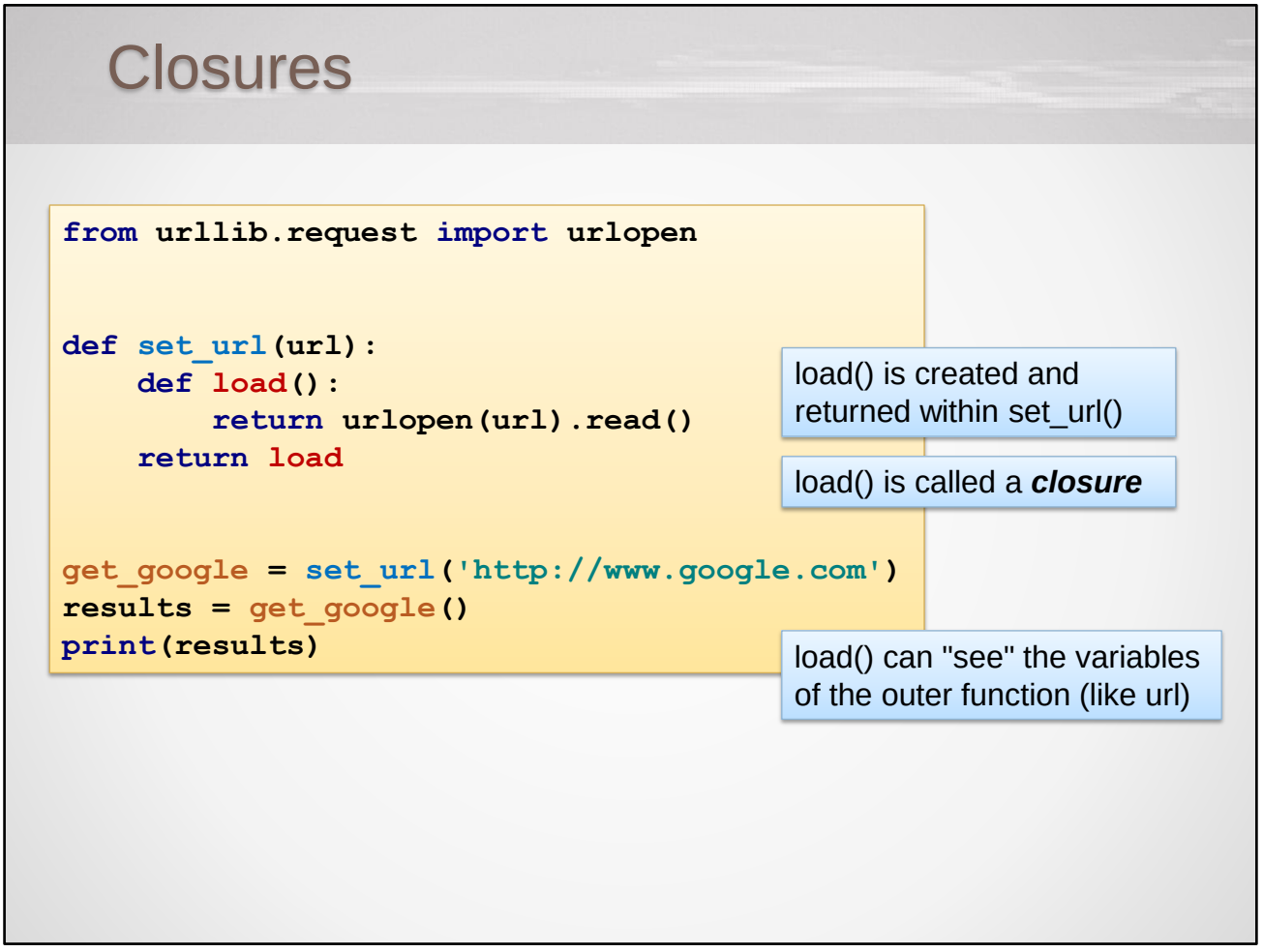
```
def display_width(width):
    def display(val):
        print(val[:width] + '...')
    return display


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 ...
```

ch10_adv_functions/03_decorators.py



ch10_adv_functions/04_decorators.py

Closures are functions defined within other functions. They have several interesting properties/benefits:

1. They provide a form of encapsulation (private-ness).
2. They can "see" the variables of the outer function.
3. Thy 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!

ch10_adv_functions/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, thus making it work the way we want. Yet it is not really ideal.

The Python keyword, *nonlocal*, provides this specific capability we need. It brings in the variable from the outer-scoped function, making it usable within our nested function.

An Undecorated Function

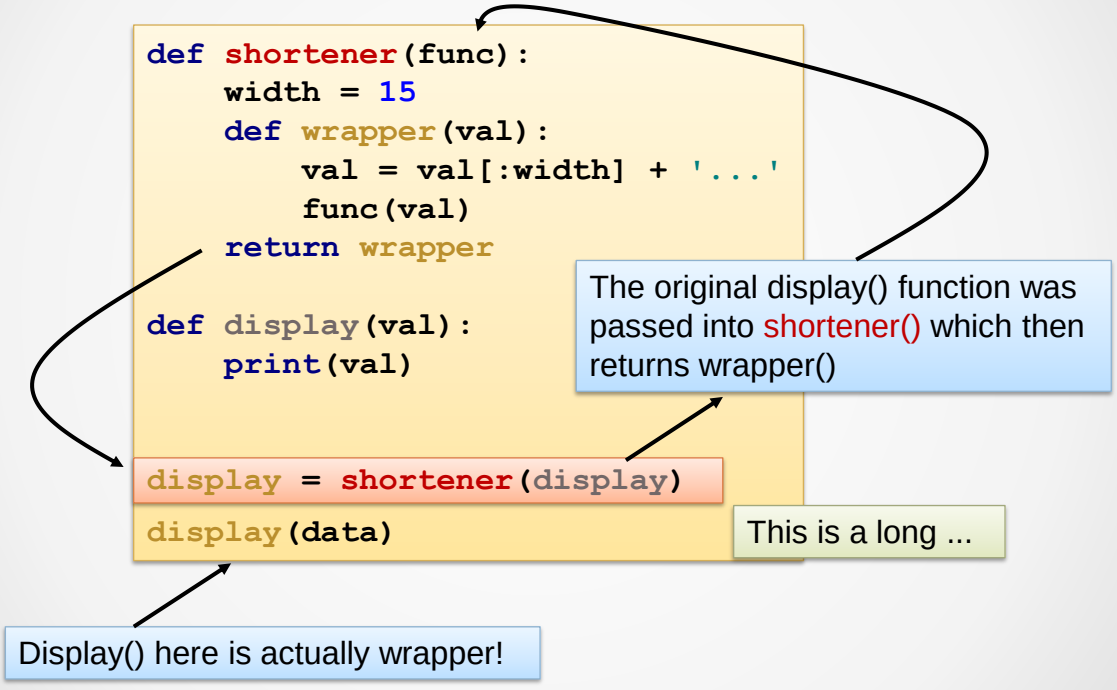
```
def shortener(func):
    width = 15
    def wrapper(val):
        val = val[:width] + '...'
        func(val)
    return wrapper


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

ch10_adv_functions/06_decorators.py

Wrapping the Original Function



ch10_adv_functions/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.

Applying the Decorator Syntax

```
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)

ch10_adv_functions/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}')


display('Johnny')
```

With this notation, you never had access to the original function

This is actually calling wrapper

ch10_adv_functions/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

ch10_adv_functions/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 Work Flexibly

```
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
```

ch10_adv_functions/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 Flexibly *(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

ch10_adv_functions/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 Flexibly *(2 of 2)*

```
@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',
             address='123 Chancellor Matheson Road')
```

This is a long

Keyword arguments are no problem now

Kiefer William 123 Chancellor

ch10_adv_functions/12_decorators.py

5 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
```

ch10_adv_functions/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)

5 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
```

ch10_adv_functions/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).

5 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")} lines read.')
```

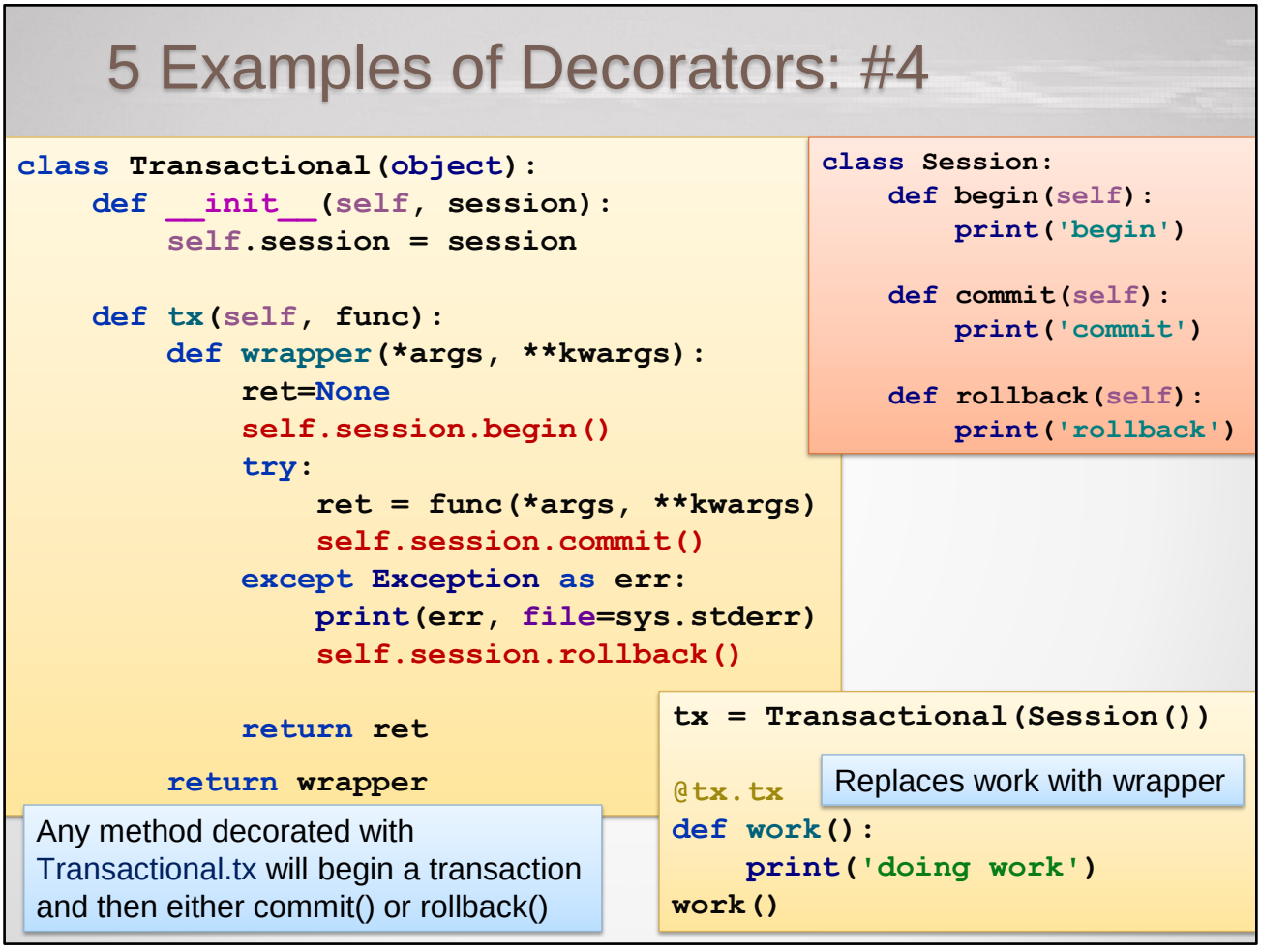
This function merely opens the file and reads every other line from it, printing the lines out in reverse

```
func1 took 0.45sec
49999 lines read.
```

ch10_adv_functions/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).



ch10_adv_functions/16_decorators.py

This example is slightly more complex than the previous ones, yet it achieves some unique and powerful features. First of all, it 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.

5 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

ch10_adv_functions/17_decorators.py

The example uses a decorator to cache results. 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 expense (time consuming).

Not shown above: To avoid adding confusion, the use of the **@wraps** decorator (shown in the source file) was not included here. @wraps comes from the functools module and causes the name of the original function to be retained. In other words, our output shows the name "func1" in the output (see next slide) instead of "wrapper" because of the use of the @wraps decorator placed above wrapper. To see its usage, view the source file.

5 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

Cache
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
```

ch10_adv_functions/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)
```

ch10_adv_functions/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?

```
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
```

ch10_adv_functions/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?

```
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

Implement the __call__ magic method, which serves as the "wrapper"

```
Result: 13.5
Invalid argument supplied.  Must be numeric.
```

ch10_adv_functions/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?

```
def dec(cls):
    orig_init = cls.__init__

    def __init__(self, *args, **kwargs):
        print('doing something before')
        orig_init(self, *args, **kwargs)
        print('doing something after')

    cls.__init__ = __init__

    return cls

@dec
class Foo(object):
    def __init__(self):
        print('original init')

Foo()
```

Save the class' original constructor

A great example of this is the @dataclass decorator

Replace the class's constructor with a new one

Return the modified class containing the modified constructor

```
do something first
original init
do something last
```

ch10_adv_functions/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)*

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

ch10_adv_functions/22_decorators.py

This example illustrates the use of a class decorator that accepts arguments. It allows for a mask and allcaps argument to make the arguments to the function uppercase if set to true and will mask the last name if set to True. 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
```

ch10_adv_functions/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, WARN, ERROR, CRITICAL)

```
import logging
logging.basicConfig(level=logging.DEBUG,
                    filename='/temp/logfile.log')
logging.debug('Logging started.')
```

Module-level logger used here

ch10_adv_functions/23_logging.py

Additional Configuration

- To create 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

ch10_adv_functions/23_logging.py

Loading Config From a File

- Loggers may load info from a config 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
```

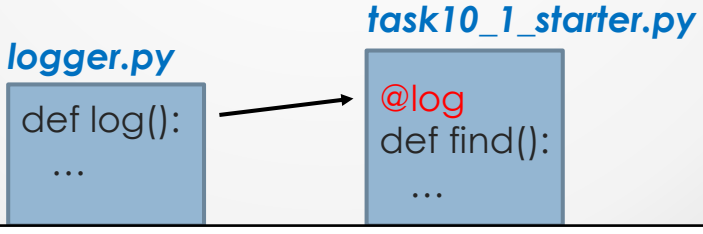
ch10_adv_functions/24_more_logging.py and logging.ini

In addition to an .ini configuration, python allows config using a dictionary, JSON data, or YAML data.

Your Turn! Task 10-1

Decorators

- Modify the database access app from Task 6-1
- Within the provided logger.py module, in the ch10_adv_functions/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 (task10_1_starter.py)



The solution assumes the student_files directory is on your PYTHONPATH. Import references are from that point.

(Optional Topic)

Functional Programming Methods

- Python is not traditionally considered a functional programming language
  - With some support for FP, coupled with its ease of readability, Python can still be useful with the functional programming style
  - Tools include:
    - First-class functions and lambdas
    - Map, reduce, filter, any, and all
    - Short-circuiting and/or operators

Helpful Functions

- map() - creates a map of values as a result of passing an iterable through a specified function
- reduce() - causes the results of each return value to be accumulated in a result variable
- filter() - returns a new sequence for each True value returned from a function
- any() - returns True when *at least one element* is True
- all() - returns True when *all elements* are True

Using *map()*

map() creates an iterator that executes a function for every item in an iterable

- Syntax

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

Returns a map object

- Examples

```
list(map(double, [1, 2, 3]))
```

[2, 4, 6]

```
from random import shuffle

def shuffler(normal_str):
    temp_list = list(normal_str)
    shuffle(temp_list)
    return ''.join(temp_list)

mapobj = map(shuffler, ['monday', 'tuesday',
               'wednesday', 'thursday', 'friday'])
print(list(mapobj))
```

['oymnad',
'teusyda',
'deedsynaw',
'ytusrdha',
'irfyda']

ch10_adv_functions/25_map_example.py

Note that the syntax for map() changed from Python 2 to Python 3. In Py2, map() returned a list. In Py3, it returns a map object that must be iterated over or converted to a list (as shown).

Using *reduce()*

reduce() repeatedly applies a function accepting two arguments returning a single result

- Syntax

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

Returns a single object

- Examples

```
reduce(lambda x,y: x+y, [1, 2, 3])
```

6

```
from functools import reduce
def sentence_maker(word1, word2):
    return word1 + ' ' + word2

results = reduce(sentence_maker, ['Four', 'score', 'and',
                                  'seven', 'years', 'ago'])
print(results)
```

Four score and seven years ago

ch10_adv_functions/26_reduce_example.py

Note: In Python 2, reduce() is a global, built-in function. In Python 3, it was moved to the functools module.

Using *filter()*

filter() creates an iterator from an iterable whose elements pass a specified test (boolean_function)

- Syntax

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

- Examples

```
filter(lambda x: x % 2 == 0, range(1, 15, 3))
```

[4, 10]

```
months = ['January', 'February', 'March', 'April',
          'May', 'June', 'July', 'August', 'September',
          'October', 'November', 'December']

def not_ber_months(month):
    return month.lower()[-3:] != 'ber'

filter_obj = filter(not_ber_months, months)
results = list(filter_obj)
```

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

ch10_adv_functions/27_filter_example.py

Using *any()* and *all()*

- Syntax

```
bool_result = any(iterable)
```

```
bool_result = all(iterable)
```

- Examples

```
any([0, '', ()])
```
False

```
all([1, ' ', {}])
```
False

```
months = ['January', 'February', 'March', 'April',
          'May', 'June', 'July', 'August', 'September',
          'October', 'November', 'December']

all(len(mo) < 10 for mo in months)
any(len(mo) > 8 for mo in months)
```
True
True

Generator expression!

Note: any() returns after any evaluation is True, all() returns after any evaluation is False (this is called short-circuiting).

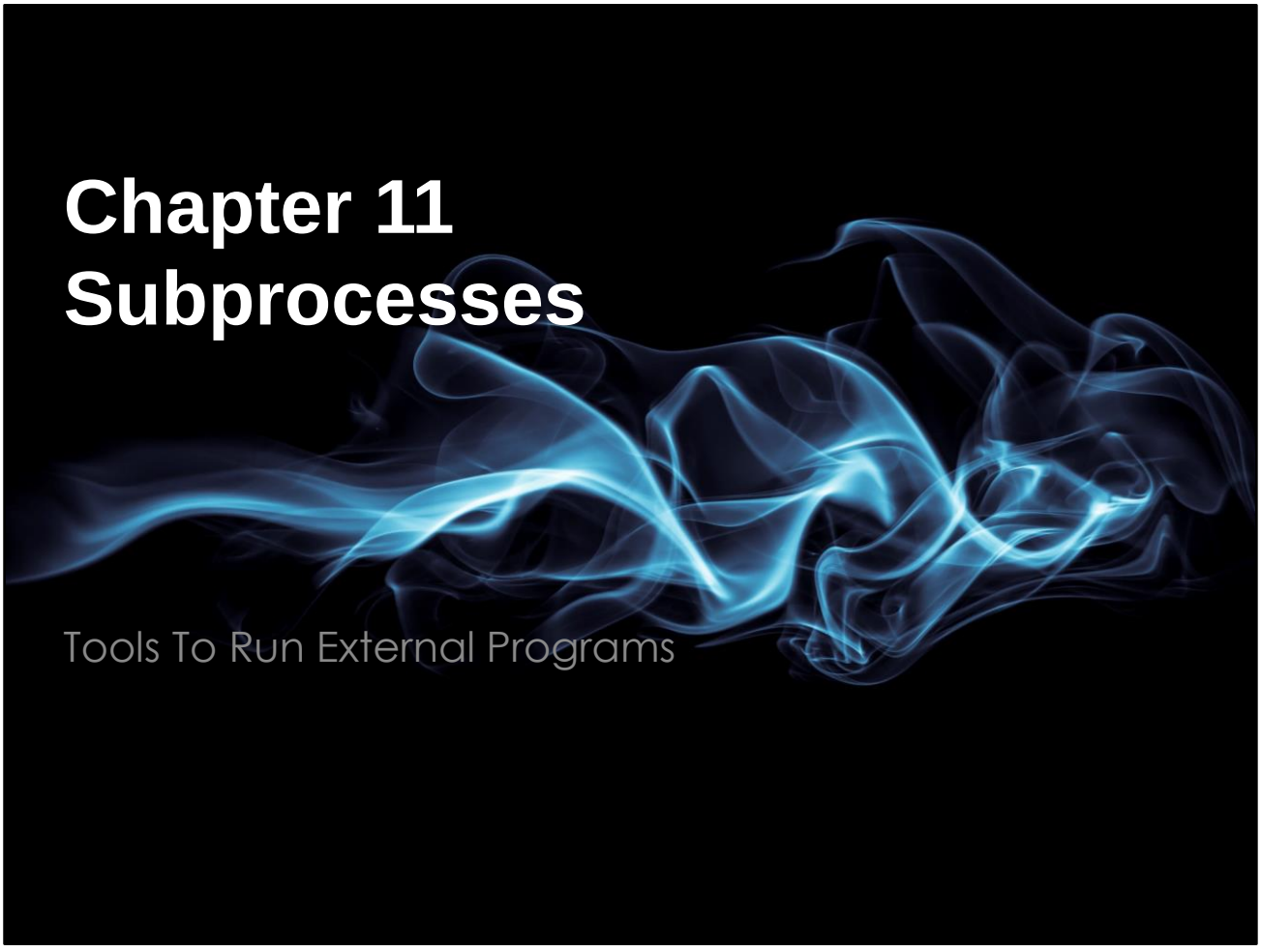
ch10_adv_functions/28_any_all.py

In the middle examples, because 0, empty string, and empty tuple all evaluate to False, the any() function returns False. Similarly, because 1, a string with a space evaluate to True but the empty dict evaluates to False, the all() function returns False.

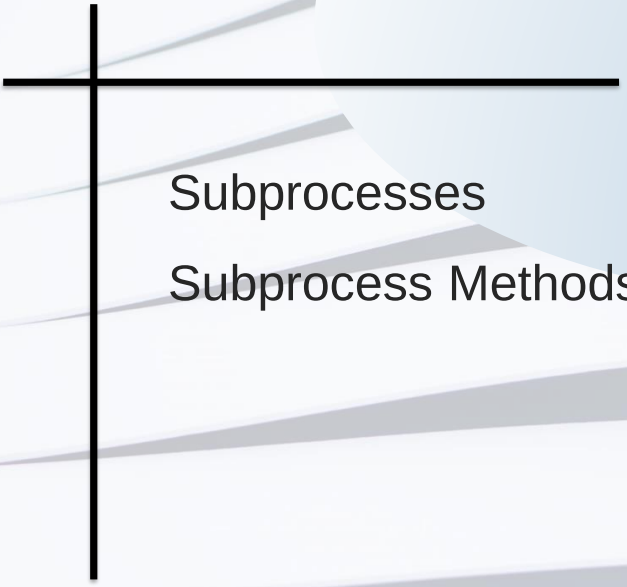
Note the case: `all([])` returns True (any empty iterables will return True).

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)



Overview



Subprocesses

Subprocess Methods

Processes

- *Processes* are programs running on your OS
- Some programs *can launch new processes*
  - These new processes are separate programs that run concurrently with the original program
  - These types of processes are often called **subprocesses**

Subprocesses

- Python has a subprocess module that allows a Python script to launch another program
- When launched from within a Python script, the script can perform some *limited communication* with that process and *get its return code*
- Two preferred techniques exist for launching subprocesses from within a Python script
  - run() - should be used in most situations
  - Popen() - use for more complicated situations

Note: older techniques, such as os.system(), os.popen(), os.spawn(), os.popen2(), subprocess.call(), subprocess.check_call(), subprocess.check_output() can all be replaced by either subprocess.run() or subprocess.Popen().

subprocess.run()

- subprocess.run() runs an OS command or script
  - By varying the arguments to run(), it can emulate the behavior of other methods

```
import subprocess
ret_code = subprocess.run(['echo', 'hello'])
```

run() is a *blocking operation* which means it doesn't continue until the spawned process finishes

  - On Windows, the above command fails
    - For OS-specific commands on Windows (e.g., dir, pathping, etc.), use the shell=True argument

```
import subprocess
ret_code = subprocess.run(['echo', 'hello'], shell=True)
```

ch11_threads/01_subprocess_run.py

In Python 3.5, subprocess.run() was added which effectively performs the same functionality as all of the older subprocess methods (except for Popen) by varying the arguments. The **shell=** argument appears in both run() and Popen(). The **shell=** argument is False by default which is generally the desired value. **shell=True** executes the arguments in an intermediate shell. Generally, this is not desired, but is *required for Windows OS commands*. The bottom line is, if executing a Windows-base OS command (e.g., mkdir, dir, echo, etc.) you will specify shell=True and in all other conditions, use the default (shell=False) value.

run() Variations *(1 of 4)*

- subprocess.run() with no additional arguments

sample.py

```
import random
import sys

print('This is sample.py')
ret_code = random.randint(0, 1)
 if ret_code:
    print('A random error occurred',
          file=sys.stderr)
sys.exit(ret_code)  # will be 0 or 1
```

This script can be run by itself.
50% of the time it will output:
This is sample.py

50% of the time it will output:
A random error occurred.
This is sample.py

02_run_no_args.py

```
import subprocess
result = subprocess.run(['python', 'sample.py'])
try:
    result.check_returncode()
except subprocess.CalledProcessError as err:
    print(f'Error running command: {err}',
          file=sys.stderr)
```

From within this script, we launch the one above

The return from run(), ***result***, is a ***CompletedProcess*** object which has *args*, *stdout*, *stderr*, *returncode* and *check_returncode()* attributes

ch11_threads/02_run_no_args.py

In sample.py above, when it is invoked, it will always print "This is sample.py" to the stdout. Randomly it will also print "A random error occurred" to the stderr.

subprocess.run() gives back a CompletedProcess object.

The check_returncode() method either:
- returns None (when the external program returned 0), or
- raises a CalledProcessError (if it received a non-zero return code from the external program)

If you don't want to raise an exception you can also just get the *returncode* attribute directly using *result.returncode*.

run() Variations *(2 of 4)*

- subprocess.run(stdout=PIPE, stderr=PIPE)

In order to receive any output from the external script, you must set **stdout=PIPE**

```
result = subprocess.run(args=['python', 'sample.py'],
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE)

print(f'The program said: {result.stdout.decode()}')
print(f'Error messages: {result.stderr.decode()}')
```

Typically, **stderr=PIPE** is also set in order to receive error messages

```
result = subprocess.run(args=['python', 'sample.py'],
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        text=True)

print(f'The program said: {result.stdout}')
print(f'Error messages: {result.stderr}', file=sys.stderr)
```

Use **text=True** to get returned results back as strings instead of bytes

ch11_threads/03_piping.py

Setting the **text=** or encoding= or universal_newlines= arguments will cause stdout and stderr to be returned as strings instead of bytes.

In Python 3, the universal_newlines argument to Popen, when set to True, will use the os.linesep character to represent newline characters and will open the PIPES as text streams instead of binary streams thus causing decoding to occur automatically.

Note: when running these in PyCharm, sometimes the output can appear out of order. This is due to how the output is buffered within PyCharm. If you wish to avoid this, you can choose to *Emulate terminal in output console* by selecting it in the Run Configuration dialog. This, however, will remove the colored output for the stderr.

run() Variations (3 of 4)

- subprocess.run(check=True)

If **check=True** then run() raises a CalledProcessError immediately without even setting the result variable Exception handling would be needed in this case

```
result = None
try:
    result = subprocess.run(args=['python', 'sample.py'],
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            text=True, check=True)
except subprocess.CalledProcessError as err:
    print(f'Error occurred on external process: {err}',
          file=sys.stderr)

if result:
    print(f'The program said: {result.stdout}')
```

ch11_threads/04_check_true.py

If check=True is set, then the run() method itself will raise an exception immediately when the return code from the external program is non-zero.

Note that if the command (python) is bad, an exception will be raised no matter what because the process was never created in the first place.

If an exception is raised, result will not be set to anything (it will remain a None value)

run() Variations *(4 of 4)*

- subprocess.run(timeout=n)

If **timeout=n** is used, the run() method will return back.

sample2.py always times out

```
result = None
try:
    result = subprocess.run(args=['python', 'sample2.py'],
                            stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE,
                            text=True,
                            check=True,
                            timeout=3)
except subprocess.CalledProcessError as err:
    print(f'Error occurred on external process: {err}',
          file=sys.stderr)
except subprocess.TimeoutExpired as err:
    print(f'Timeout on external process: {err}', file=sys.stderr)

if result:
    print(f'The program said: {result.stdout}')
```

ch11_threads/05_timeout.py

When a timeout occurs (which it will if sample2.py is used), a TimeoutExpired exception occurs. When subprocess.run() is used, the child (external) process will be killed automatically by Python. When Popen() is used, the child (external) process is not automatically killed.

Popen()

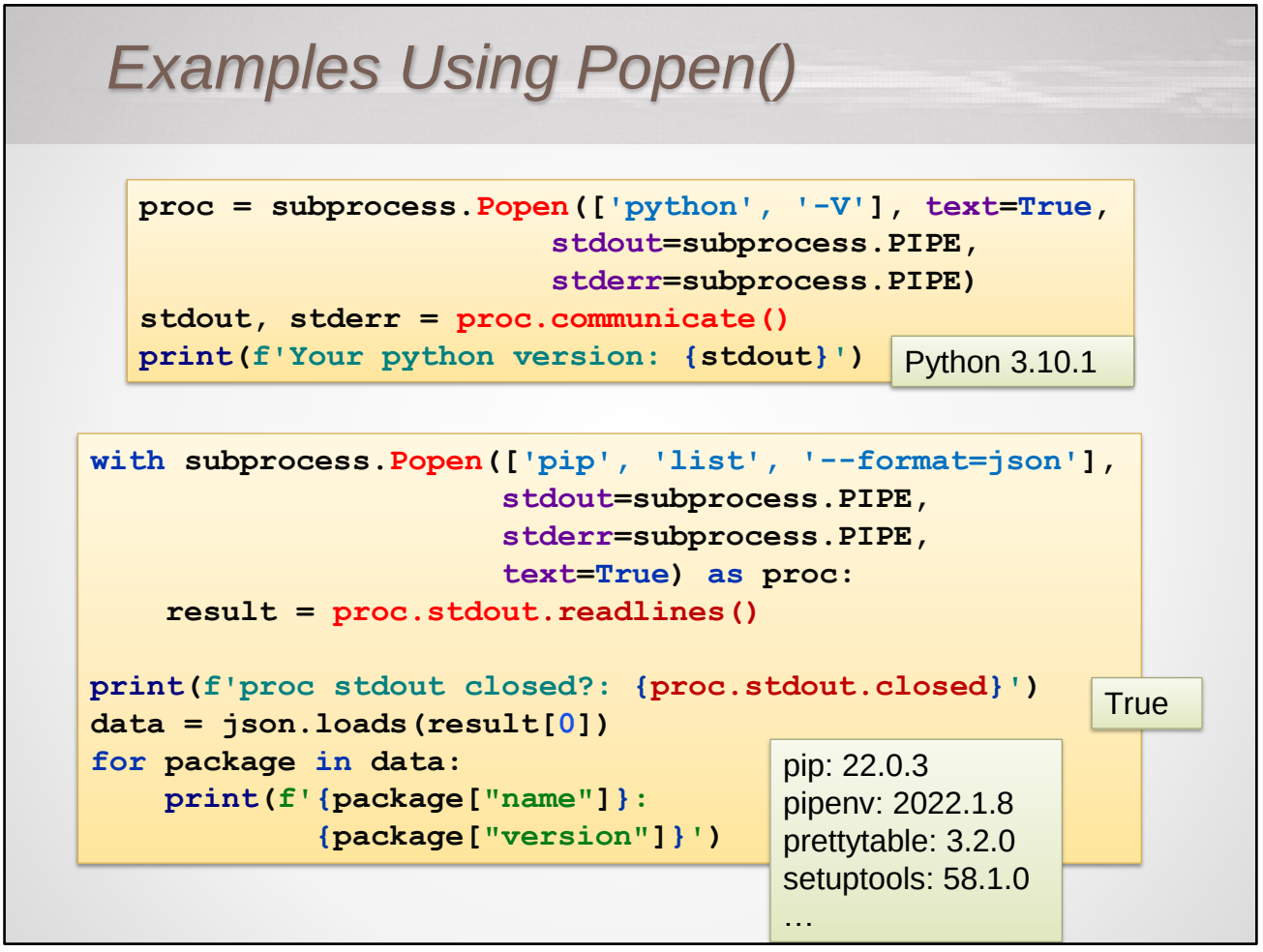
- The subprocess.Popen() is similar to run() but doesn't block when executed
  - The subprocess begins running immediately, however

```
proc = Popen([command, args], shell=False, stdout=PIPE, stdin=PIPE, stderr=PIPE)
(stdout, stderr) = proc.communicate(input_data)
```

communicate() blocks until the process finishes

  - Popen() can also be used in a *with* control
    - On exit, stdout and stderr are closed and the wait() method is called

Popen() can handle all of the same situations that run() can but doesn't block when executed. This means the external process executes immediately. Either call proc.communicate() or proc.wait() to cause the code to block until the external process finishes.



ch11_threads/06_popen.py

In the two examples above, the first simply runs the python command and checks the version. It uses communicate() to get the piped results back.

In the second example, the pip list command is executed (with results returned in a JSON format). We did this one in a with control to show that once the with control ends, the file descriptors for that process are closed. In this case, the stdout is closed as tested above. The remaining lines of code simply take the returned results from pip list and convert it to a dictionary. We then reformatted the dictionary using something called a dictionary comprehension (discussed previously).

Examples Using Popen()

```
import subprocess
import sys

cmds = {'win32': 'netstat -an',
        'darwin': 'netstat -a | grep -i "listen"',
        'linux': 'netstat | grep LISTEN',
        'linux2': 'netstat | grep LISTEN'}
command = cmds.get(sys.platform)
print('Using: ', command)
with subprocess.Popen(command.split(),
                      stdout=subprocess.PIPE,
                      stderr=subprocess.PIPE,
                      text=True) as proc:
    result = proc.stdout.readlines()

print(''.join(result))
```

```
Using:  netstat -an
Active Connections
  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:808            0.0.0.0:0              LISTENING
...
```

ch11_threads/06_popen.py

In the final example, we ran a command that is different for each operating system. We used the sys module's platform attribute to do a little OS detection and then provided a command based on this.

Your Turn! Task 11-1

Subprocesses

- Finish the starter file to create a subprocess that uses the OS ping utility
  - Use the subprocess module to execute the operating system's ping command

```
import subprocess
import sys

cmds = {'win32': 'ping', 'darwin': 'ping -c 4',
        'linux': 'ping -c 4', 'linux2': 'ping -c 4'}
command = cmds.get(sys.platform)

address = 'www.google.com'

# Your code here:
# Split the command into a list
# Add the address to this list
# Run the subprocess and display results
```

Note: For Unix-based systems, use the syntax: ping -c 4 *address*

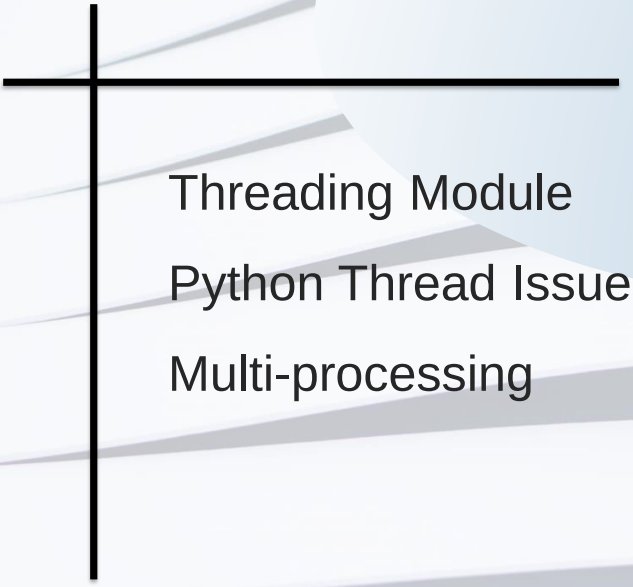
Summary

- Make use of the subprocess module to execute commands from the operating system and view results back in the Python script
- Use subprocess.run() for most needs
- Vary the options to tailor the behavior of the run() method

Chapter 12
Multi-threading
Multi-processing

Creating Multiple Paths of Execution

Overview



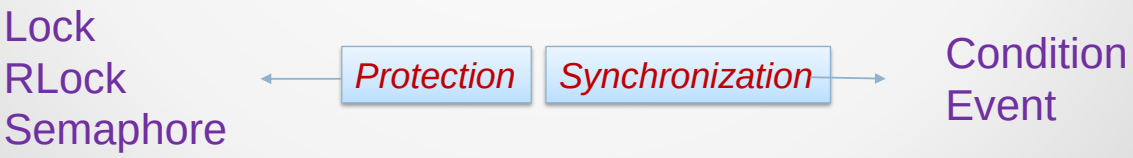
Threading Module

Python Thread Issues

Multi-processing

Thread Class Methods

- Threads provide multiple paths of execution within a program
  - This means multiple tasks can often execute faster than single threads*
- To use threads, the **Thread** class within the **threading** module is commonly used
- Other components of the threading module:



* More details on this later.

Events and Conditions are not discussed here. Events allow threads to communicate state information with other threads. The state info can be toggled between set and unset. Conditions allow two threads to synchronize properly by using wait() and notify() methods.

Threads

- The *threading* module provides several module-level functions

```
threading.active_count()    # of alive thread objects
threading.current_thread()  the currently executing thread object
threading.main_thread()     the main thread (program launch thread)
threading.enumerate()       list of active thread objects
```

- Some of the *Thread* class methods

```
Thread.run()          the method defining what a thread does
Thread.start()        begins the thread
Thread.is_alive()     is the thread still active? (isAlive() before Py3.8)
Thread.join()         main thread blocks until worker thread terminates
```

Creating and Running a Thread

```
from threading import Thread
import requests

def load_data(url):
    data = None
    try:
        r = requests.get(url)
        data = r.json()
    except requests.exceptions.RequestException as err:
        print('Error: {1}'.format(err))

    return data

class DataFetcher(Thread):
    def __init__(self, url, name=''):
        super().__init__(name=name)
        self.url = url

    def run(self):
        print('Results: {0}'.format(load_data(self.url)))

m = DataFetcher('https://inciweb.nwcg.gov/feeds/json/markers/', 'Worker')
m.start()
```

The work to perform

To create a thread, subclass *Thread*, implement *run()*, and call *start()*

Called when the thread starts

Creates (instantiates) and starts a new thread

ch12_threads/01_threading.py

Custom threads should inherit from the Thread class imported from the threading module.

The thread in this example is launched by calling the thread's start() method. The run() method is invoked indirectly (via the parent thread). The run() method invokes load_data() which obtains the remote data.

Creating Threads without Subclassing

For simpler situations, threads can be created without classes

```
from threading import Thread

import requests

url = 'https:/inciweb.nwcg.gov/feeds/json/markers/'

def load_data(url):
    data = None
    try:
        r = requests.get(url)
        data = r.json()
    except requests.exceptions.RequestException as err:
        print('Error: {1}'.format(err))

    return data

th = Thread(target=load_data, args=(url,))
th.start()
```

Use the ***target=*** attribute of the Thread class to point to your work function

ch12_threads/02_no_subclassing.py

Threads can be created without subclassing by using the target= attribute of the Thread class constructor.

Controlling Thread Resource Access

```
from threading import Thread, Lock
from time import sleep

message = ''
lock = Lock()

def set_message(msg):
    global message

    internal_message = message
    internal_message += msg
    sleep(0.3)
    message = internal_message

    print(f'Message: {internal_message}\n')

t1 = Thread(target=set_message, args=('First thread, ',))
t2 = Thread(target=set_message, args=('Second thread, ',))
t3 = Thread(target=set_message, args=('Third thread, ',))

t1.start();t2.start()t3.start()
t1.join();t2.join();t3.join()

print(f'End Message: {message}')
```

Three threads come in at once

At the sleep(), internal_message will be the value of the last thread to hit it

Last one out of sleep() sets the global

ch12_threads/04_locks.py

In this example, each thread comes through the function, sets the *internal_message* to "" + its name. The last thread to do this sets internal_message to its own name. After the sleep() statement, message = internal_message will be based on the last thread to hit the internal_message += msg statement.

Check this and you will see the global message will only be set to one thread's name.

Note: the start() and join() calls were placed on a single line here for space purposes.

Using a Lock

```
message = ''
lock = Lock()

def set_message(msg):
    global message
    lock.acquire()
    internal_message = message
    internal_message += msg
    sleep(0.3)
    message = internal_message

    print(f'Message: {internal_message}\n')
    lock.release()
t1 = Thread(target=set_message, args=('First thread ',))
t2 = Thread(target=set_message, args=('Second thread ',))
t3 = Thread(target=set_message, args=('Third thread ',))

t1.start();t2.start()t3.start()
t1.join();t2.join();t3.join()

print(f'End Message: {message}')
```

The lock will limit access to only one thread at a time after acquire()

Updates the global variable with no interference from the other threads now

ch12_threads/04_locks.py

When the lock is used, each thread must wait at the acquire(). One thread at a time is allowed passed acquire() and it gets the global message, assigns it to internal_message, appends its own name to internal_message, then sets the global one to the internal_message variable. No other threads will interfere now. After the release(), the next thread is allowed to repeat this process.

RLocks

- RLocks – (re-entrant locks) block only if held by *another* thread
  - Useful if a thread may (recursively or otherwise) re-acquire a lock that the thread has already acquired

```
lock = threading.Lock()
lock.acquire()
lock.acquire()              # this will block


lock = threading.RLock()
lock.acquire()
lock.acquire()              # this won't block
```

Semaphores allow a limited number of threads to access a resource. Useful for pools of resources (such as connections).

Using *with* on Locks

```
message = ''
lock = RLock()

def set_message(msg):
    global message

    with lock:
        internal_message = message
        internal_message += msg
        sleep(0.3)
        message = internal_message
        print(f'Message: {internal_message}\n')

t1 = Thread(target=set_message, args=('First thread ',))
t2 = Thread(target=set_message, args=('Second thread ',))
t3 = Thread(target=set_message, args=('Third thread ',))

t1.start();t2.start()t3.start()
t1.join();t2.join();t3.join()

print(f'End Message: {message}')
```

Lock is automatically acquired and released using 'with'

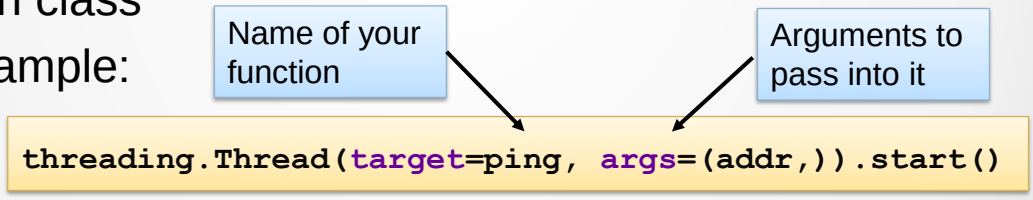
ch12_threads/05_with.py

This version implements a RLock although a regular Lock would work equally well here. The *with* control performs the acquire() and release() automatically yielding the same result as the previous version.

Your Turn! Task 12-1

Threads

- Revisit the *subprocess ping task*, this time making it a multi-threaded pinger
- Create a thread per address to ping
  - Use the Thread() class rather than creating your own class
  - Example:



- Work from task12_1_starter.py
  - Either use the hints in the source file or work from the back of the student manual

The Global Interpreter Lock (GIL)

- A mutual exclusion lock is held by the Python Interpreter (within a PVM)
  - Only the current thread holding the GIL may operate
  - The interpreter releases the lock every 100 instructions or when blocking operations occur
    - sys.setswitchinterval(*seconds*) can change this value
- Applications that feature more I/O operations may benefit more from multiple threads
- Apps that are purely computational or work intensive may lose performance due to the switching incurred by the GIL

Note: Built-in mutable objects such as lists and dictionaries are considered thread-safe because updates to these data structures are considered atomic. This means that if a list or dict is updated, the Python interpreter will not release the GIL until that update has finished.

Despite the GIL, Can Threads Still Work?

Test the example and determine for yourself

```
num_worker_threads = 20
tasks = ['http://python-requests.org','https://www.google.com',…]

req_queue = Queue()
results_queue = Queue()
for url in tasks:
    req_queue.put(url)

class WorkerThread(Thread):
    def run(self):
        while not req_queue.empty():
            url = req_queue.get()
            results_queue.put(get_data(url))
            req_queue.task_done()

for i in range(num_worker_threads):
    t = WorkerThread()
    t.start()
```

Queues implement required locking logic so they can be shared by many threads

URLs are placed onto a job queue (req_queue)

The thread grabs a URL from the queue, performs the get_data() task, then places the results onto the results queue

Create and start the threads, run gets called automatically

ch12_threads/06_queue.py

In this example, 16 tasks (each independent web requests) are placed on a queue to be executed by a pool of 20 threads.

The work that each task will perform can be found in the get_data() function. It retrieves a web page. For brevity, it is not shown here.

Multiprocessing

- To overcome the GIL's limitations, Python 2.6+ introduced the **`multiprocessing`** module
- The multiprocessing module creates full processes rather than worker threads to perform multi-processing operations
  - It does this using the Process() class
  - Processes create their own Python virtual machines and are not bound by single CPU core limitations as threads are

The *multiprocessing* Module

```
import multiprocessing

def square(val: int):
    print(f'Square result: {val * val}')

def cube(val: int):
    print(f'Cube result: {val * val * val}')

if __name__ == '__main__':
    p1 = multiprocessing.Process(target=square, args=(4,))
    p2 = multiprocessing.Process(target=cube, args=(10,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
    print('Tasks done!')
```

View your processes in your task manager

Each process instantiation is a separate OS-level process.

```
Square result: 16, process id: 19716
Cube result: 1000, process id: 9416
Tasks done!
```

ch12_threads/07_multi_processing.py

Multiprocessing and Queues *(1 of 2)*

```
from multiprocessing import Process, freeze_support, Queue
from queue import Empty

from bs4 import BeautifulSoup
import requests

num_processes = 20
tasks = ['http://python-requests.org', ...]

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

def create_tasks(req_queue, tasks, num_processes):
    for url in tasks:
        req_queue.put(url)
    for i in range(num_processes):
        req_queue.put('DONE')
```

We'll create 20 processes to handle 16 tasks

Queues up URLs into the req_queue

ch12_threads/08_multiprocessing_queue.py

This example is similar to the threaded example previously discussed. It uses a multiprocessing queue to queue up multiple tasks. It then spawns multiple processes, 20 in this case, to handle the tasks.

Multiprocessing and Queues *(2 of 2)*

```
def work(req_queue, results_queue):
    while True:
        val = req_queue.get(timeout=10)
        if val == 'DONE':
            break
        results_queue.put(get_data(val))
```

Grab from the queue

Perform the work

Place results on results_queue

```
def main():
    processes = []
    req_queue = Queue()
    results_queue = Queue()
    for i in range(num_processes):
        p = Process(target=work, args=(req_queue, results_queue))
        p.start()
        processes.append(p)

    create_tasks(req_queue, tasks, num_processes)

    for p in processes:
        p.join()
    print_results(results_queue)

if __name__ == '__main__':
```

Spawn 20 processes

work() is the task each process will perform

Create the task and results queues

ch12_threads/08_multiprocessing_queue.py

This is the continuation of our multiprocessing example. The freeze_support() (not shown but appears in the source code) method is for Windows-based machines due to their improper support for forking and initializing processes.

The main() method creates a tasks queue (req_queue) and a results queue. URLs are stored in req_queue and pulled out later from within the work() method. The instantiation of the req_queue is critical. It should be instantiated before creating the processes, but it should not be loaded up with tasks until after the processes are created. create_tasks() is responsible for adding tasks to the req_queue queue, which occurs after the creation of all of the processes.

Summary

- Use the threading module and the Thread class to create multi-threaded apps
  - Threads in Python work fine if those threads see blocking operations
- For CPU intensive tasks, incorporate the multiprocessing module into apps to spawn *child processes* instead of threads
  - Multiprocessing brings additional complexities, such as how to handle resources used by all processes

What Did We Learn?

- Virtual Environments
- Review of Fundamentals
- Documentation Techniques
- Python DB API 2.0
- SQLAlchemy
- Instances as Dictionaries
- Properties
- Static Methods
- Magic Methods
- *super()*
- Inheritance
- Multiple Inheritance
- Sockets
- *urllib*
- *requests* Module
- Subprocesses
- *run()* and *Popen()*
- Threads
- Locks, RLocks
- Global Interpreter Lock
- Queues
- Multi-processing
- *collections*
- *itertools* Iterators
- *argparse*
- Closures
- Decorators
- Functional Programming
- Generators
- Generator Expressions
- Dict & Set Comprehensions
- *logging*
- WSGI
- Flask

Intensive Advanced Python
What's in Part III?

Review of Intermediate Topics
Performance Monitoring, Profiling, Memory Profiling, Tracing
Advanced Context Managers
Tools to Assist in Automation
PDF Creation
Email Creation and Sending
Advanced Classes
Introduction to Numerical Analysis: NumPy, Matplotlib, Pandas
Pandas
Testing with unittest and unittest.mock
Monkey Patching
Network monitoring/sniffing, port scanners, ssh
Django
Packaging and Distributing
AsyncIO, gevent, Twisted *(optional)*
Python GUIs *(optional)*

Evaluations

- Please take the time to fill out an evaluation
- All evaluations are read and considered

Intensive Intermediate Python

Exercise Workbook

Task1-1
Creating a Virtual Environment

Overview

This exercise is designed to demonstrate how to create a virtual environment using **venv** (requires Python 3.6 or later). As an optional integration step, we'll configure it within PyCharm afterwards. Finally, and optionally, you'll remove the environment when finished.

Survey Your Environment

Begin by opening a terminal (OS X) or command window (Windows). From here onward, we'll refer to it simply as a terminal.

Within the terminal, type each of these commands:

```
python -V
```

```
python3 -V
```

and **`python3.10 -V`** (replace 3.10 with the appropriate version number, e.g., python3.9, python3.8, etc.)

One of these commands should have responded with the valid version number of Python that you were expecting. ***Remember this command because you will use it throughout the remaining 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 listing will be the Python interpreter that is 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

Create a temporary directory on your system. This directory can be placed anywhere and called anything. For example, use: c:\temp on Windows (or something similar).

cd to your *temp* directory

Type the command: **`python -m venv my_env`**

This should take a few seconds and creates the virtual directory.

Examine the new *my_env* directory. Look inside the ***bin*** (or ***Scripts****, on Windows*) directory. Notice the interpreter. Look within the Lib directory and then within the site-packages directory.

There's not much here yet.

Type: **`pip list`**

You should see a list of installed packages from your main Python interpreter (again, ***use pip3 or pip3.10, etc.***, to match your Python command).

Again, (as a reminder) type:

OS X: **`which python`**

Windows: **`where python`**

Notice that the same (original) Python interpreter is listed first. The virtual one doesn't appear in the list. Not yet, at least...

Activate the Environment, Test *pip*

In order for your new Python environment to take effect, it must be activated. Do this by changing to the **`my_env/bin`** (or **`my_env/Scripts`** on Windows) directory. Then type:

OS X: **`source ./activate.sh`**

Windows: **`.\activate`** (or just activate)

Now, type:

OS X: **`which python`** (again, remember, type your specific Python version if needed, e.g., python3 or python3.10)

Windows: **`where python`**

Notice how the directory containing your new Python interpreter is listed first on your PATH environment variable. This is what activation does!

Type: **`pip list`** (as we did a moment earlier). This should now list only *setuptools* and *pip*. pip -V reveals that you are working within your virtual environment now!

(Optional, if your network doesn't block this.) Let's install a utility into this new environment. The utility is called **prettytable**. It is a simple Python module that creates ASCII tables.
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.

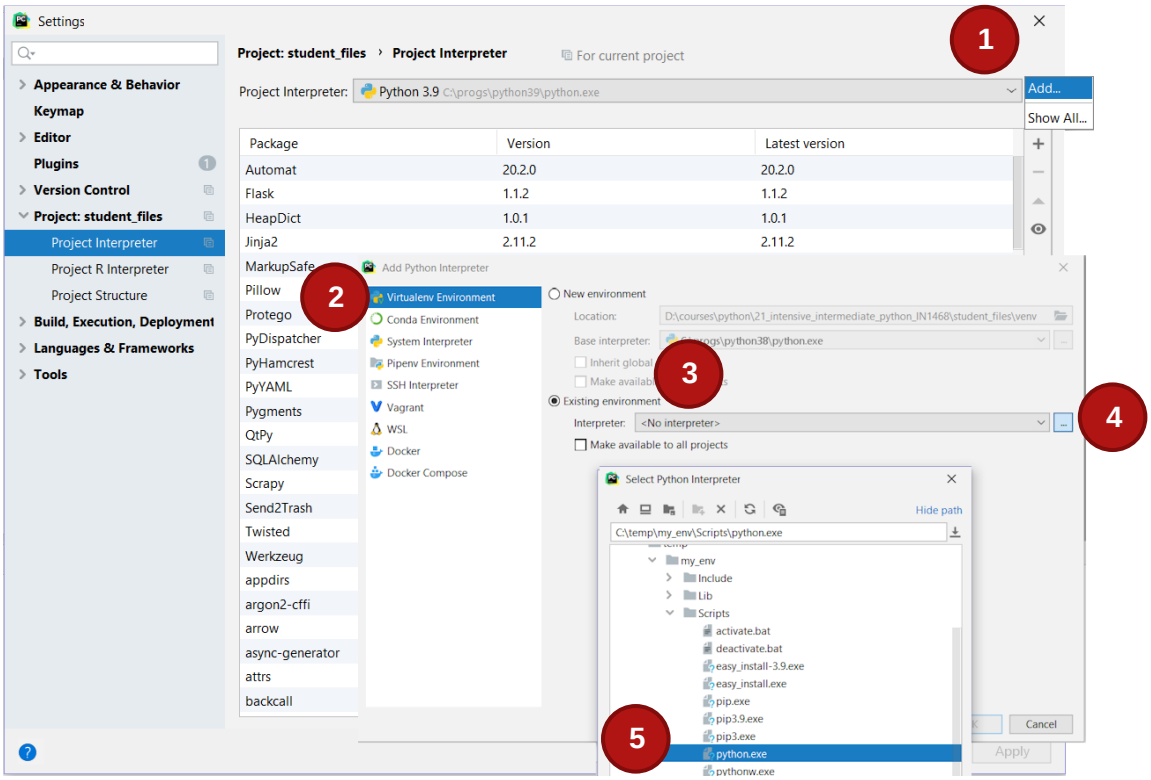
Connect the Environment within PyCharm

With the newly created virtual environment, we'll now connect it to PyCharm. First, select:

OS X: **`PyCharm > Preferences`**

Windows: **`File > Settings`**

Then, 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
2- Leave selected "Virtualenv Environment" option
3- Select "Existing Environment" button
4- Select the ... 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 listed here now.

Note: If you wish to test out the prettytable module, you can run the *ch01_virtual/01_prettytable.py* example in the student files. It should run.

Undo, Deactivate, and Remove (Optional)

You do not have to do this step if you wish to keep the virtual environment.

To demonstrate removing everything now, first, let's remove prettytable. From within your open terminal, type:

`pip uninstall prettytable`

(You don't really have to uninstall each package; this is just meant to show you *how* to uninstall individual packages)

Now, within PyCharm, restore the original interpreter that was selected when you first visited the Python Interpreter drop-down box. Click Ok to set it.

Next, back in the terminal window opened earlier, type: **`deactivate`**.

Note: deactivate is not a shell script nor a file in the *Scripts* or *bin* directories. Deactivate gets created by activate as an internally available shell command. So, don't look for deactivate.sh or deactivate.bat.

Type **`cd c:\temp`** (or whatever your temp directory name is). This should be one level above your virtual environment directory.

Now type:

OS X: **`echo $PATH`**

Windows: **`echo %PATH%`**

Notice how your virtual environment is no longer seen on your PATH. You may delete the *my_env* directory if you wish.

That's it!

Task2-1
Python Overview

Overview

This exercise is designed to review basic Python concepts and fill in gaps in introductory topics. In this exercise, you should read from the file cities15000.txt. You should then determine the highest city as well as the most populous city. Finally, you should 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.

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. All of 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)
```

Test it out and verify that no errors arise.

That's it for now. We'll revisit this exercise shortly.

Task2-2 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() or sorted() function. The sorted() function uses a key to determine how to sort. The population attribute in each City named tuple can be used to sort key 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)

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 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, multiple 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).name)
print(max(cities, key=lambda city: city.elevation).name)
```

Test out your solution so far. We'll return to this task again shortly!

Task2-3 Implementing a Search Feature

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 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. Lowercase the user's input and then use this to search the list of named tuples. Save the results of the matches for display.

```
search = input('Enter the (partial) name of the city: ')
results = [city for city in cities
                  if search.casefold() in city.name.casefold()]
```

Display the results. Here we used Python's expand operator with .format() for convenience since f-strings can't be used with this operator (*).

```
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.')
```

Test the solution.

Task3-1
Functions and Modules

Overview

This exercise continues from Task 2-3. In this exercise, you will refactor the previous solution by creating functions and a module to place those functions into. The task3_1_starter.py file serves as the driver and will import this new module invoking the functions from it. No new functionality is introduced. The only changes will be structural.

Create the read_data() Function

Open **ch03_modularization/city_search.py**. Some design decisions will need to be made such as where to define the list of cities and the named tuple. We will define our list of cities as **_cities** inside the module. We'll define our typed NamedTuple inside of the function. Most of the rest of the code transfers over from task2_3_starter.py as written previously. Here is our read_data() function:

```
from typing import NamedTuple

_cities = []


def read_data(fullname):
    City = NamedTuple('City', [('name', str),
                               ('population', int),
                               ('elevation', int),
                               ('country', str)])

    with open(fullname, 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)
        _cities.append(city)
```

Define Your largest() Function

Beneath read_data(), create a new function. There are several ways to write this function, keep the body of the function the same as how it was accomplished in Task 2-3. Simply move this code over and into the largest() function definition. Do this as follows (your code may vary slightly):

```
def largest():
    return sorted(_cities, key=lambda city: city.population,
reverse=True)[0]
```

Repeat for the highest() Function

Can you repeat step 2 except this time with the code to determine the highest city? Write a highest() function and move your code into it. Here's an example (yours may vary slightly):

```
def highest():
    return sorted(_cities, key=lambda city: city.elevation,
reverse=True)[0]
```

Complete the search() Function

Create a function within city_search.py called search() that accepts a name.

Move your code related to the search functionality from Task2-3 into this function. It should look a little bit like the following:

```
def search(name):
    results = []
    for item in _cities:
        if name.lower() in item.name.lower():
            results.append(item)
    return results
```

Check for Errors

Now is a good time to look over this module and see if you have any syntax errors. The next step is to complete the driver file. Open **task3_1_starter.py**.

Add the city_search Import

Within task3_1_starter.py add the import needed to import city_search.py.

```
import ch03_modularization.city_search as cs
```

or

```
import city_search as cs

working_dir = '../../resources'
data_file = 'cities15000.txt'
```

Call the read_data() Function

Call the read_data() function passing the fullpath to the file.

```
fullname = Path(working_dir) / data_file

cs.read_data(fullname)
```

Call the largest() and highest() Functions

Call the largest() and highest() functions. Retrieve and display the values. Keep in mind the format of the return value may be a City namedtuple or you may have returned a string or int.

```
print(cs.largest().name)
print(cs.highest().name)
```

Call the search() Function

Call The search() function passing the desired city search term. Obtain your results. The search phrase is up to you.

```
results = cs.search('new')
```

Display the Results

Display the results from step 9 in any fashion you deem appropriate. Here's one way which is similar to how we did it in Task 2-3:

```
if not results:
    print('No cities found.')
else:
    for city in results:
        print('{0:<35}{1:>15,}{2:>15,}{3:>10}'.format(*city))
```

That's it!--Test it all out!

Task5-1
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 data class 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 data class records as follows:

```
def get_location(school_name):
    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):
    results = []
```

Within the *with* control, obtain the cursor and perform the query:

```
def get_location(school_name):
    results = []

    with closing(sqlite3.connect(data_sourcefile)) as conn:
        cursor = conn.cursor()
        cursor.execute(SELECT_SCHOOLS_SQL,
                                    ('%' + 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,
                                    ('%' + 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):
    results = []
    with closing(sqlite3.connect(data_sourcefile)) as conn:
        cursor = conn.cursor()
        cursor.execute(SELECT_SCHOOLS_SQL,
                                    ('%' + 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):
    results = []
    try:
        with closing(sqlite3.connect(data_sourcefile)) as conn:
            cursor = conn.cursor()
            cursor.execute(SELECT_SCHOOLS_SQL,
                                    ('%' + 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
```

Task6-1
Python Classes

Overview

This exercise creates two classes: a School and a SchoolManager class. Both classes are to be created within the task6_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 ch06_oo/task6_1_starter.py create a School class that contains a school_id, fullname, city, state, and country attributes.

```
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 two SQL-related 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.

```
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

Now comes the heart of the code. Within find(), check the column name provided by the user that we want to search on and 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:
        self.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 task5_1_starter.py:

```
def find(self, value, column='fullname',
         sort_by='fullname'):

    results = []

    if column in self.PERMITTED_SEARCH_COLUMNS:
        self.SELECT_SCHOOLS_SQL =
           self.SELECT_SCHOOLS_SQL.replace('column', column)

    with closing(sqlite3.connect(self.db_filename)) as conn:
        cursor = conn.cursor()
        params = ('%' + value + '%',)

        cursor.execute(self.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:
            self.SELECT_SCHOOLS_SQL =
               self.SELECT_SCHOOLS_SQL.replace('column', column)

        with closing(sqlite3.connect(self.db_filename)) as conn:
            cursor = conn.cursor()
            params = ('%' + value + '%',)

            cursor.execute(self.SELECT_SCHOOLS_SQL, params)

            for record in cursor:
                results.append(School(*record))

            results.sort(key=lambda s: vars(s).get(sort_by))

        return results
```

Test it Out!

Run the module. The if block at the bottom should execute successfully.

Task7-1
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 uses the search() function from the previous exercise.

Import the schools.py Module

At the appropriate location within **`ch07_frontend/starter/app.py`**, import the SchoolManager class from the schools.py module.

```
from ch07_frontend.starter.schools import SchoolManager
```

Call the find() Method

At the location specified for step 2 within the app.py file, instantiate the SchoolManaer 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

Just before the return statement of the get_schools() method, invoke the jsonify() method passing two keyword arguments into it: school_name and schools.

```
@app.route('/school/<school_name>', methods=['GET'])
def get_schools(school_name):
    results = None
    try:
      results = [str(school) for school in
                      SchoolManager(database).find(school_name)]
    except Exception as err:
        results = err.args

    resp = jsonify(schools=results, school_name=school_name)
    return Response(resp.data, status=200,
mimetype='application/json')
```

To test out your solution, first run app.py and then browse to http://localhost:8051 in your browser.

Task8-1
requests and JSON

Overview

This exercise requires making an HTTP GET request to our running Flask server, retrieving JSON data and parsing it. The parsed data will be in the form of a Python dictionary. We'll take that dictionary and convert it to a list of data classes. Finally, we'll take the data classes and persist them to a file in the form of JSON data.

Some steps, such as the Encoder, have been accomplished already.

Make Sure the Server is Running

It's a short and simple step, but necessary. Within the **`ch08_network_apis/server`** folder, right-click on **`wsgi.py`** and select Run...

The server should now be running.

Build the Data Class

Open `ch08_network_apis/task8_1_starter.py`. Follow the template provided on the slide to build the data class at the location for step 2 in the source file.

```
@dataclasses.dataclass
class Character:
    name: str
    actor: str
    role: str
    original_air_date: datetime.date
```

Make a Request to the Server

The URL for making requests has been given to us. Embed it into a requests.get() call. Don't forget to convert the returned JSON response into a dict type using the .json() method:

```
char_name = input('Enter partial Simpsons character name: ')
url = f'http://localhost:8051/simpsons?char_name={char_name}'

data = requests.get(url).json()
```

Write the List of Data Classes Out to a File

You may choose the filename, but .json is a good file extension. Place the filename into an open() call. Set the write mode flag 'w' and put the open() call into a with control.

Now, within the with control, invoke json.dump() passing the data, the file handle, and the encoder class.

```
with open('simpsons_data.json', 'wt') as f:
    json.dump(results, f, cls=CharacterEncoder)
    print('Write complete.  Open file to view results.')
```

That's it!

Test it Out!

To test it, run the starter file and watch for a file to get created in the same directory where the starter file is running.

Task9-1

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).

Add the Country Code to the cities dict

Read records from the file adding the country column for each city record to a list:

```
cities = []

with open(datafile, encoding='utf8') 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!

Task9-2
Using Generators

Overview

This exercise will refactor the previous exercise (Task 9-1). 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 task9_1_starter.py file or you may work from the provided task9_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 the 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='utf8') as cities_file:
        for line in cities_file:
            yield line.strip().split('\t')[8]
```

Invoke the generator and run it to conclusion 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!

Task9-3
Using argparse

Overview

This exercise will incorporate the argparse module for use with your Task 9-2 solution. It adds the ability to retrieve a specified number of "mosts" from the Counter by indicating 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 task9_3_starter.py -c 5

or

python task9_3_starter.py --count 5

Note: you may work from your task9_2_starter.py solution if you wish.

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='The number of countries to retrieve')
    return parser.parse_args()
```

Invoke the 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.

Task10-1
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 Task6-1 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 **ch10_adv_functions/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
```

Decorate the find() Method within schools.py

As our last step, we'll apply the decorator we created. First, we need to import it. Open the **ch10_adv_functions/starter/task10_1_starter.py file**. Up near the top of the file, add the import as shown:

```
from ch10_adv_functions.solution.logger import log
```

Next, we'll apply our decorator as shown:

```
@log
def find(self, value: str, column: str = 'fullname', sort_by:
str = 'fullname'):
    results = [
    ...
```

Test out your solution by starting the server and running the task10_1_starter.py file (this is the client). Look for a log file created, called logfile.log in the same directory. Open and view the log file.

That's it!

Task11-1
A Ping Subprocess

Overview

This task uses the operating system ping utility via a subprocess within Python. It detects your operating system and selects the appropriate command format. You'll need to add the address to your command and then run subprocess.run().

Begin by opening `task11_1_starter.py` from the ch11_subprocesses folder within PyCharm. In the following steps, place your code at the locations marked in the source code.

Split the Command String

The source code already pulls the appropriate OS ping command out of the dictionary for you. You should split that command into a list of strings:

```
command = command.split()
```

Add the Ping Address to the Command

Add the address to the command list of strings and then display your command list before using it.

```
command.append(address)
print('Using: ', command)
```

Call subprocess.run(), Display Results

As our last step, invoke the subprocess.run() method. Provide the args=, stdout=, stderr=, and text= arguments.

```
result = subprocess.run(args=command, text=True,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE)
```

Display your results. Ping should send all results to stdout but we can check for errors by looking at the result.returncode.

```
if not result.returncode:
    print(result.stdout)
else:
    print('Error: ', result.stdout, file=sys.stderr)
    print(result.stderr, file=sys.stderr)
```

That's it! Test it out!

Task12-1
A Multi-threaded Pinger

Overview

This task continues from the subprocess task (Task 11-1). This time, however, you will modify the solution to become a multi-threaded pinger. To ensure instructions flow properly, work from the provided task12_1_starter.py file within the ch12_threads folder.

Begin by opening `task12_1_starter.py` from the ch12_threads folder within PyCharm. In the following steps, place your code at the locations marked in the source code.

Create the Function Definition

At the location mentioned for step 1, write the function stub.

```
def ping(address):
  # to be filled in on the next step
```

Complete the Contents of the Function

The contents of this function are nearly the same as the code we wrote for Task 11-1. As such, this code has been given to us in the comments section at the top of the module (within the Step 2 section). Obtain this code and place it within our new function. For reference, the completed function is shown below.

```
def ping(address):
    ping_command = command.copy()
    ping_command.append(address)
    print(f'Ping using: {ping_command}')
    result = subprocess.run(args=ping_command, text=True,
                stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    if not result.returncode:
        print(result.stdout)
    else:
        print('Error: ', result.stdout, file=sys.stderr)
        print(result.stderr, file=sys.stderr)
```

Create the Threads

Now it's time to call the ping() function repeatedly (once for each thread we create). We'll create one thread for each address to work with. Remove the pass statement at the bottom of the file and create the thread in the for loop as shown below:

```
addresses = ['www.google.com', 'www.cisco.com',
             'www.im_a_fake_address.com']
for addr in addresses:
    threading.Thread(target=ping, args=(addr,)).start()
```

That's it! Test it out!