TEKsystems

Our people make IT possible.

TEKsystems Education Services

presents

Intensive Advanced Python

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

COURSE CODE IN1469 / 2.5.2024

This course covers Intensive Advanced Python

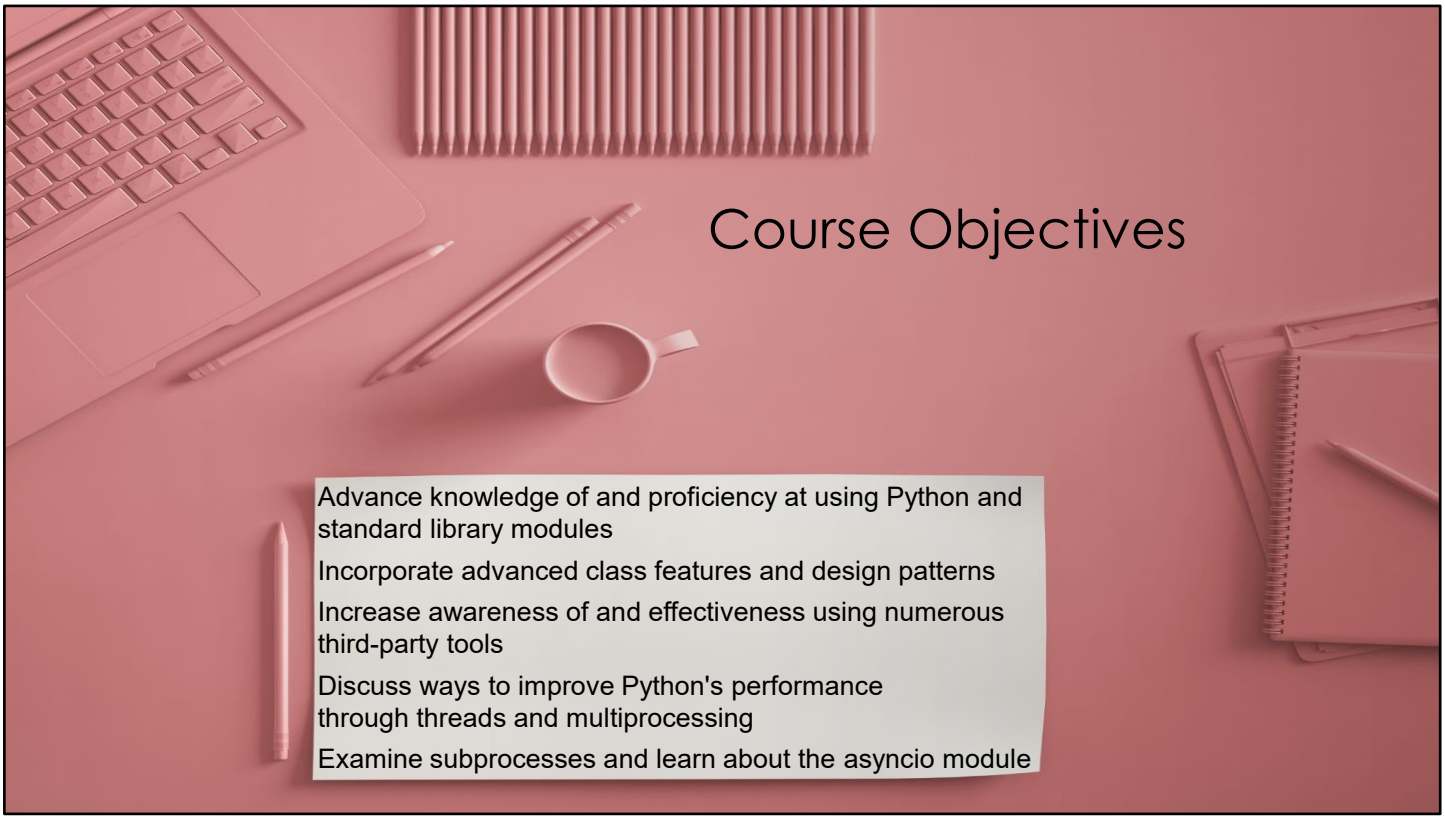
This version updated: 2/5/2024

Notes

Chapters at a Glance

Chapter 1	Review Intermediate Topics	*17*
Chapter 2	Advanced Classes	*39*
Chapter 3	Advanced Language Concepts	*70*
Chapter 4	Automation and Standard Library Modules	*91*
Chapter 5	Subprocesses, Threads, and Multi-processing	*122*
Chapter 6	Introduction to AsyncIO	*149*
Chapter 7	Intro to Machine Learning and FastAPI	*160*
Chapter 8	Django	*177*
	Course Summary	*202*
Appendix A	Python GUI Systems	*206*

Notes



Course Agenda - Day 1

Day 1

Review of Intermediate Topics

Class Design / Design Patterns

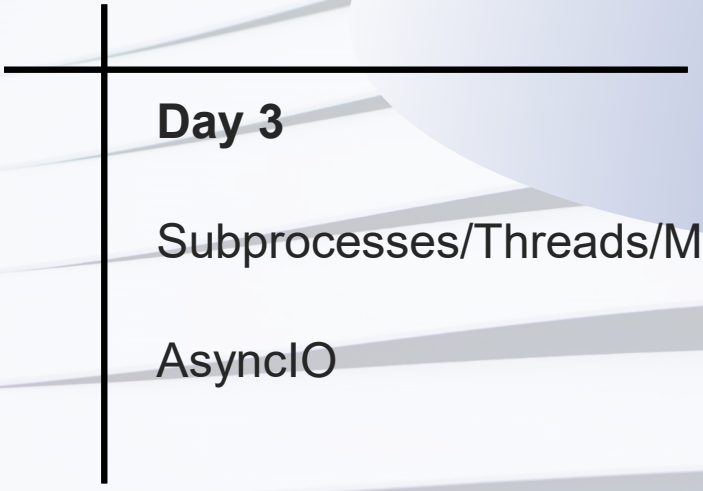
Course Agenda - Day 2

Day 2

Advanced Language Topics

Automation and Standard Library Modules

Course Agenda - Day 3

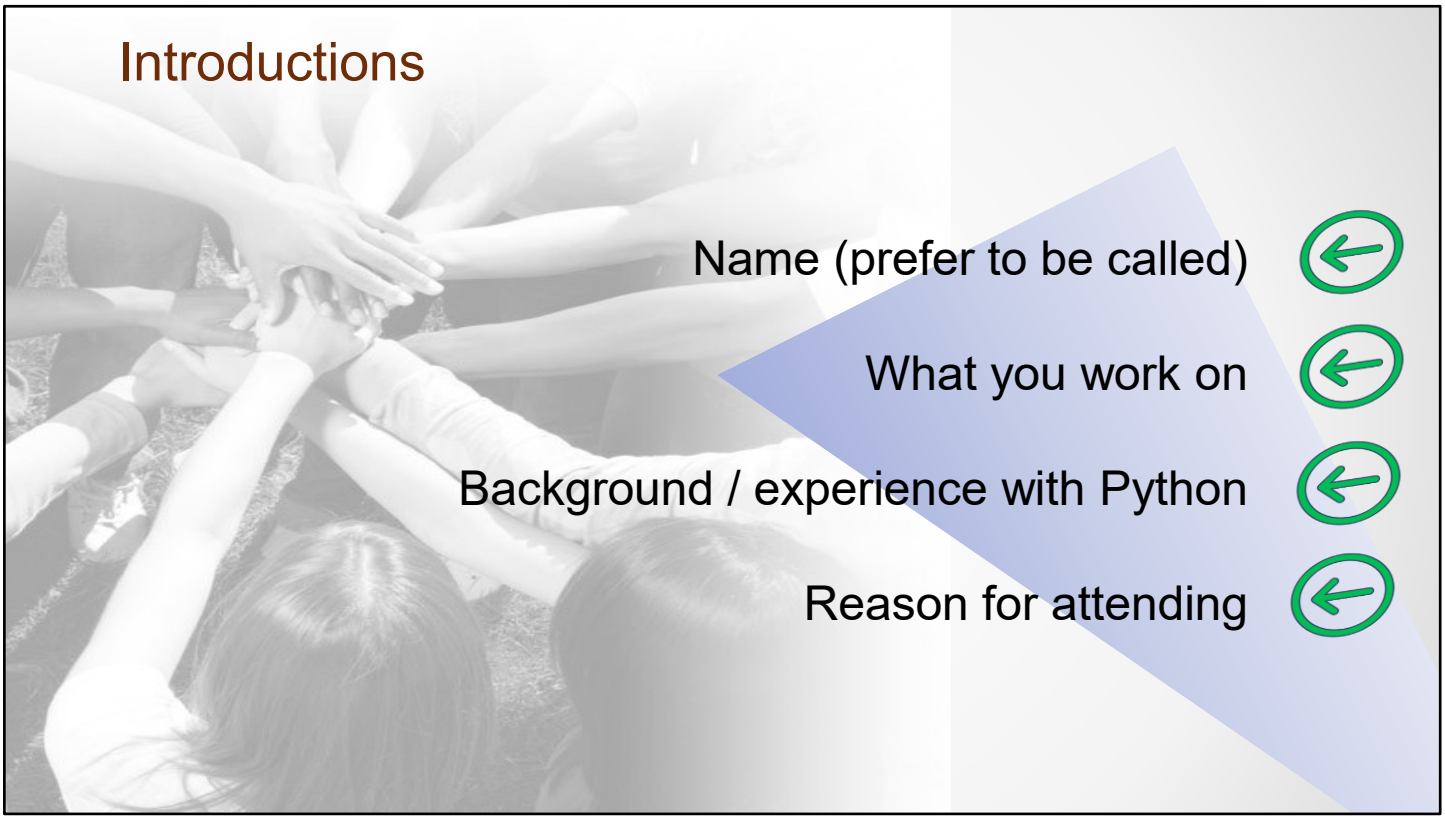


Course Agenda - Day 4

Day 4

Intro to Machine Learning and FastAPI

Django Web Application Server



Typical Daily Schedule*

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

** Your schedule may vary, timing is approximate*

Get the Most from Your Experience

Ask Questions

Setup and Installation

Python 3.12.x or later

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

PyCharm Community Edition (free)

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

Student resources

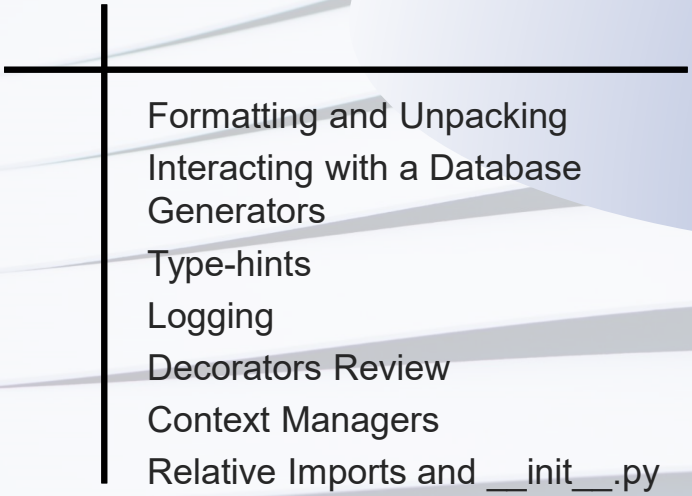
A link (usually found in an email related to this course) to download the student files will be provided to you

If a pre-established workstation was not provided for you, you may need to perform the setup instructions found in the exercise workbook in the back of the student manual, entitled "Setup and Test of Python Environment"

Chapter 1
Review of Intermediate Topics

Review of Prior Course Topics *and* Some New Stuff

Chapter 1 - Overview



Formatting and Unpacking
Interacting with a Database
Generators
Type-hints
Logging
Decorators Review
Context Managers
Relative Imports and __init__.py
argparse
Coroutines

Formatting and Unpacking

- * and ** can be used to "unpack" (expand) iterables and dictionaries

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

print('Top two: {0}, {1}'.format(winners[0], winners[1], winners[2]))
print('Top two: {0}, {1}'.format(*winners))
```

* unpacks items into individual elements

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

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

More values than are needed can be provided

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

** only works for dictionaries

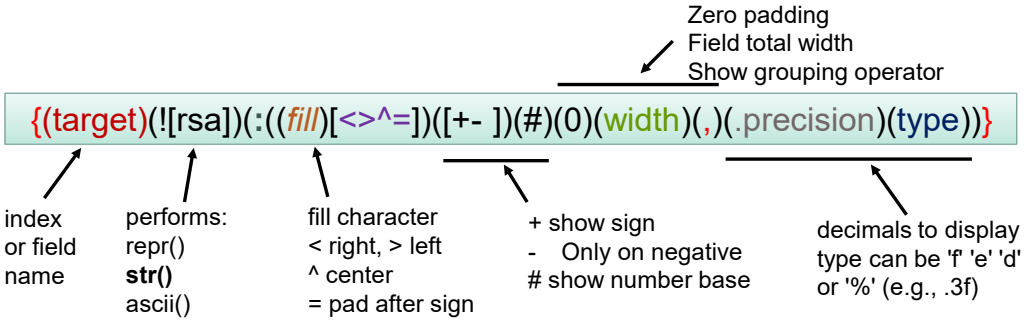
ch01_review/01_formatting.py

The str type's format() method has been written to accept most any type of parameter. As a result, we can pass into it as many items as desired.

The examples show the use of the unpacking operators (* and **). When used within a call to a function they will unpack variables. The * will unpack an iterable while the ** will unpack a dictionary. Unpacking merely means expanding it into a set of "loose" or individual values separated by commas.

format() Syntax and f-strings

- Format() supports a syntax for rendering output values



- f-strings have become popular and support the same *format()* syntax

```
distance = 240_000

print('Distance: {0:,.3f}'.format(distance))
print('Distance: {dist:_^+20,}'.format(dist=distance))
print(f'Distance: {distance:_^+20,.3f} miles')
```

Positions

Keywords

```
Distance: 240,000.000
Distance: ______+240,000______
Distance: ____+240,000.000____ miles
```

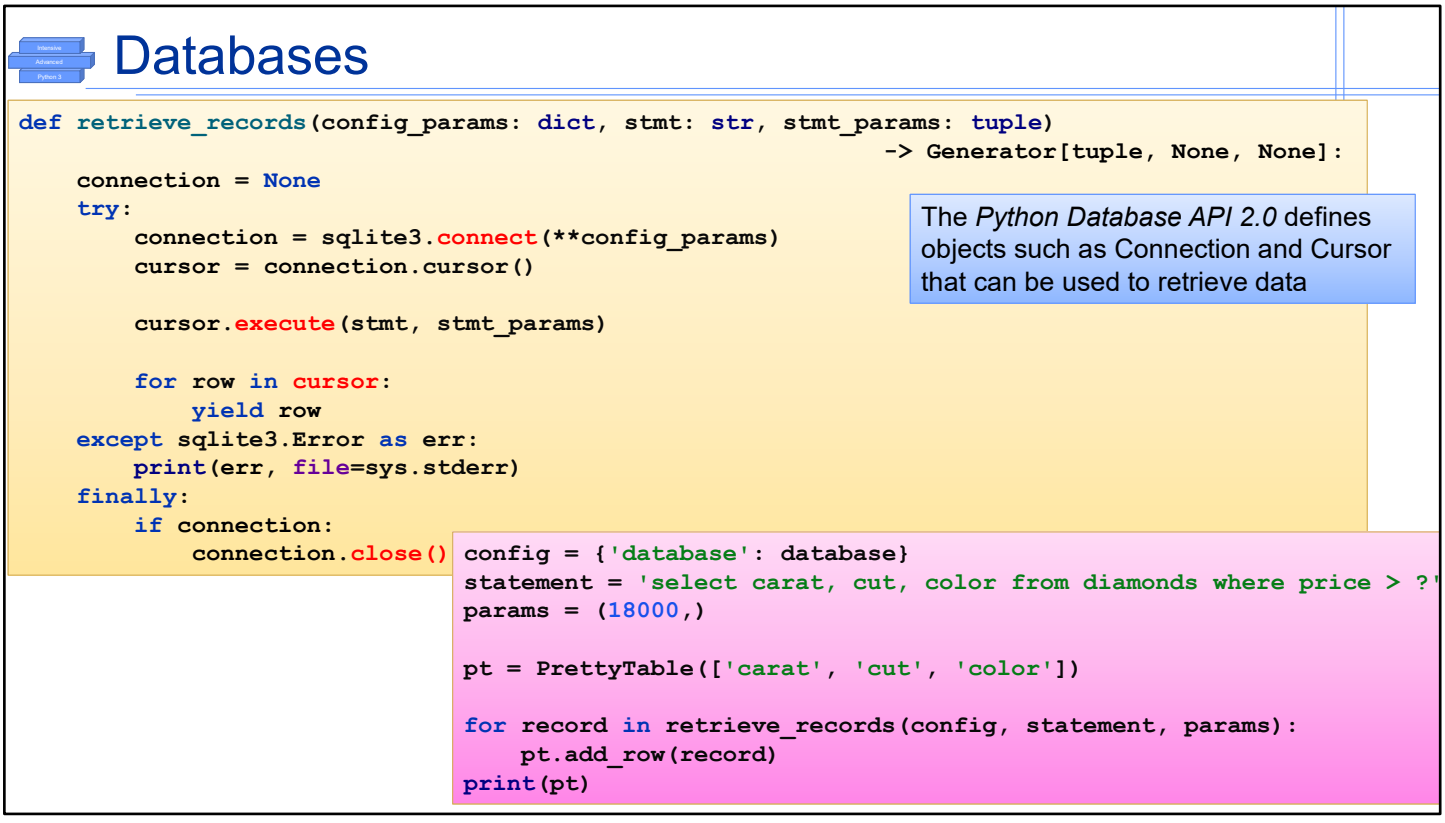
ch01_review/01_formatting.py

Precede f-strings with an "f" as shown. F-strings are preferred over format() in most situations. There are some cases that cannot support the use of f-strings, however.

It is possible to have nested "curlies" within f-strings:

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

```
Distance: ______+240,000.000______
```



ch01_review/02_database.py

The DB-API (or Python Database API 2.0, PEP 249) identifies the objects needed to interact with a relational database. Vendors conforming to the DB-API should create a module-level connect() method that returns a Connection object. This, in turn, returns a Cursor object that is used for manipulation and retrieval of data within the database.

Not shown are fetchone(), fetchmany(), and fetchall() methods that retrieve a specified number of records.

The example retrieves data from the "diamonds" table in our course database and outputs the results in a formatted table layout.

Generators

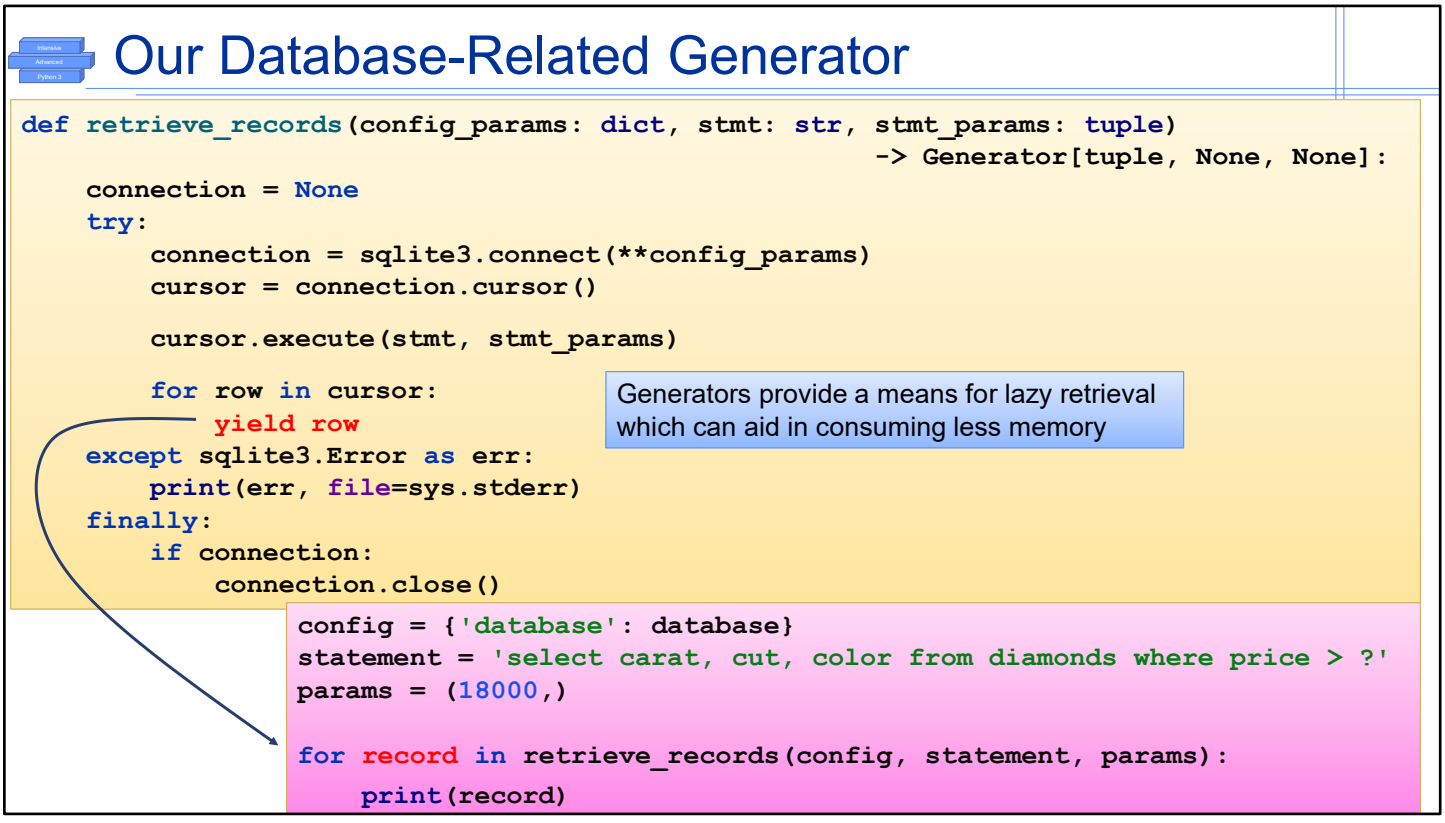
- Generators provide lazy (on-demand) retrieval of items
- Ideal to help avoid creating large data structures
- Invoking a generator does not begin executing it
- Special yield keyword "pauses" the generator
- "Picks up where it left off" when __next__() or next() is called

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

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

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

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

ch01_review/03_generators.py

The use of generators can help ensure we don't create an overall data structure in memory to hold all the returned values from the database. This means that we can return a large dataset and process it immediately with less concern for memory utilization.



ch01_review/02_database_print.py

Our generator is used to retrieve a single record from the database and then return that record. In this case, that means the connection to the database remains open.

Generators will resume where they left off, so here, the database connection will be left open for longer. The tradeoff is often acceptable, but each situation should be evaluated separately. In other words, holding onto the database connection for longer may need to be looked at to see if this doesn't cause other problems.

Type Hints (Annotations)

```
def retrieve_records(config_params: dict, stmt: str, stmt_params: tuple)
                                                    -> Generator[tuple, None, None]:
    connection = None
    try:
        connection = sqlite3.connect(**config_params)
        cursor = connection.cursor()

        cursor.execute(stmt, stmt_params)

        for row in cursor:
            yield row
    except sqlite3.Error as err:
        print(err, file=sys.stderr)
    finally:
        if connection:
            connection.close()
```

The use of annotations have become popular in Python and is recommended as projects get larger

```
config = {'database': database}
statement = 'select carat, cut, color from diamonds where price > ?'
params = (18000,)

for record in retrieve_records(config, statement, params):
    print(record)
```

ch01_review/02_database_print.py

Type hints help make projects more robust and stable by detecting type inconsistencies during development. While the annotations have no effect at runtime, they do enable large teams of developers to better understand each other's code and provide a form of self-documentation.

Since a generator is used in the example above, there are a couple of options for the return type annotation. The first is the version shown which uses the format *Generator[yield_type, send_type, return_value]*. The send_type refers to a coroutine (discussed later), so it's None in this example. The other way to annotate the generator is to simply use the typing module's Iterator definition stating *Iterator[tuple]* for the return type.

Logging Module

- The standard library *logging* module can be configured using basicConfig() which supports numerous options

This basicConfig() call defines a root logger with a handler that will output log info to the console

```
import logging

logging.basicConfig(level=logging.DEBUG, format='%(message)s',
                    handlers=[logging.StreamHandler(stream=sys.stdout)])

def retrieve_records(config_params: dict, stmt: str, stmt_params: tuple) -> ...:
    connection = None

    ...

    except sqlite3.Error as err:
        print(err, file=sys.stderr)
    finally:
        if connection:
            connection.close()
            logging.info('Connection closed.')
```

Logging Best Practices
Submodules (libraries) should not set handlers (except a NullHandler). The following is a good way to declare a logger in a library module:

```
logger = logging.getLogger(__name__)
           .addHandler(logging.NullHandler())
```

ch01_review/02_database_print.py

In Python, loggers are hierarchical where the root logger is the parent to all of them. Loggers pass messages on to their parent. The root logger (here) is created using logging.basicConfig(). It can later be retrieved using logging.getLogger() without arguments. Any call using the module-level methods (e.g., logging.info(), warning(), debug(), error(), or critical()) will invoke the root logger.

To create a separate logger with different behavior than the root in a module, use logger = logging.getLogger(__name__). Then use this logger's methods instead of the root logger. For more on this, see the file *04_child_loggers.py*. If creating a library that others will import and use, ensure this logger doesn't create its own handler (place to send log messages). Let the user define the handler (usually in the root) and the library will send messages to the root logger's handler. To be safe, it is okay to define a NullHandler() in the library just in case a root logger isn't defined by the user. This ensures the library doesn't generate "No handlers found" error messages.

More in-depth specialized logging circumstances can be found here: https://docs.python.org/3/howto/logging-cookbook.html.

Decorators

- Decorators enable the creator to gain control of others' code

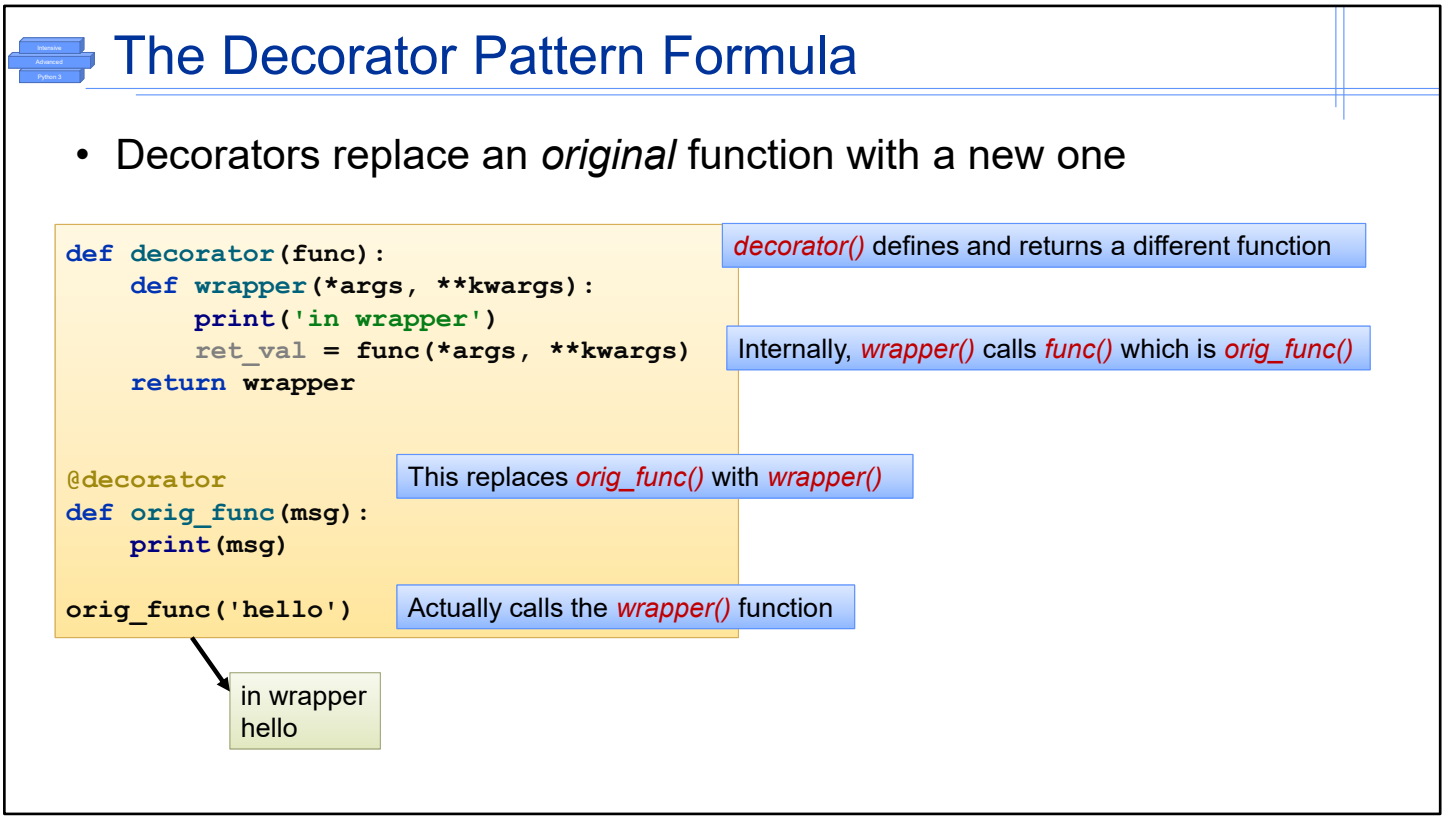
```
from pydantic import validate_call

@validate_call(validate_return=True)
def read_definitions(filepath: str | Path,
                     size: int = 10,
                     encoding: str = 'utf-8') -> list[tuple[str, str]]:
    results = []
    with open(Path(filepath), encoding=encoding) as f:
        for line in islice(f, size):
            try:
                word, definition = line.strip().split(maxsplit=1)
                results.append((word, definition))
            except ValueError:
                pass
    return results
```

`print(read_definitions(datafile, 7))`	Okay
`print(read_definitions(size='7', filepath=datafile))`	Okay, Pydantic will coerce string to int safely
`print(read_definitions(datafile, 'hello'))`	Not okay, Pydantic can't convert 'hello' to int

ch01_review/05_decorators.py

Pydantic is a popular third-party data validation framework. It relies on type hints within Python to work properly. In the example, a decorator gains control of the user function (read_definitions()) and checks function call parameters *before* the function actually gets called. If the parameters are invalid types, a ValidationError is raised.

Pydantic can convert string types to int if desired. This happens in the second call to read_definitions(). If it can't make that conversion, a ValidationError will occur.



ch01_review/05_decorators.py

Shown above is a relatively generic pattern for implementing a decorator in Python. In effect, the wrapper() function replaces the original function (orig_func()) which is what gives the decorator creator control over the original function. The decorator creator can now examine or modify parameters in the call to the original function as well as examine the return value. It can even opt not to call the original function at all!

To make this pattern work, replace *orig_func()* with the function you want to decorate and replace wrapper with a function that has the desired behavior you are trying to create.

Context Managers

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

__enter__ is called at the beginning,
return value becomes 'as-clause' object
__exit__ is called *no matter what happens*

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

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


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

Suppresses the exception

```
in enter
foo
in exit
```

A context manager defines both an *__enter__* and an *__exit__* method in a class. When the *with* control 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__() will be called no matter whether an exception is raised or not.

In the example above, the __exit__() method exhibits a rarely known (and used) feature. If __exit__() returns anything non-False, any exceptions from the *with* control body will be suppressed.

Context Manager Utilities: @contextmanager

- The *contextlib* module provides several tools for the *with* control
  - *@contextmanager* is a decorator that enables customizing the *with* control without the need for classes

```
from contextlib import contextmanager

@contextmanager
def read_utf8(filepath: str | Path):
    f = open(filepath, encoding='utf-8')
    try:
        yield f
    finally:
        f.close()
```

Perform unique initialization before the *yield*

yield the resource

Perform cleanup after the *yield*

```
filepath = Path(__file__).parents[2] / 'resources/poe.txt'
try:
    with read_utf8(filepath) as f:
        for line in islice(f, 81, 89):
            print(line.rstrip())
except IOError as err:
    print(err, file=sys.stderr)
```

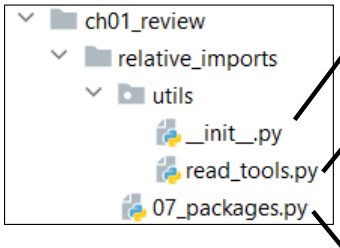
Use your resource in the *with* control

ch01_review/06_context_utils.py

This example merely simplifies reading from text files by setting the encoding to *utf-8* for us automatically. Now a Path or str can be provided to read_utf8() in a with control which reads from a text file without having to declare an encoding each time. It is always set to utf-8 no matter what (of course, don't use this if the encoding shouldn't be utf-8). When done, the file is still automatically closed.

__init__.py

- Defining a file called *__init__.py* within a package and then referencing that package invokes the __init__.py file




```
from .read_tools import read_utf8
```

```
@contextmanager
def read_utf8(filepath: str | Path):
    f = open(filepath, encoding='utf-8')
    try:
        yield f
    finally:
        f.close()
```

```
from ch01_review.relative_imports.utils import read_utf8

with read_utf8(filepath) as f:
        for line in islice(f, 81, 89):
            print(line.rstrip())
```

ch01_review/relative_imports

The example above illustrates the use of __init__.py, a specially named Python file that is automatically invoked whenever an import passes over the directory where the __init__.py exists. In this example, the __init__.py exists in the utils folder, but our main file (07_packages.py) passes over (imports) the ch01_revew-relative_imports.utils location. This causes the __init__.py to be automatically invoked.

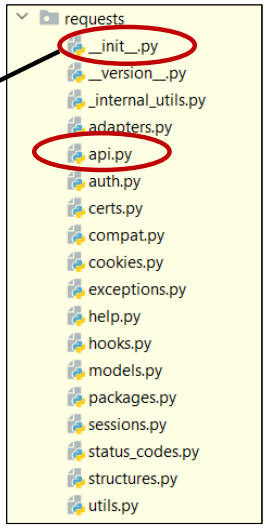
The loading of the __init__.py sets up a relative import which is discussed next.

Relative Imports

- Relative imports make references relative to the current file's location
  - Relative imports begin with "dot" notation and must follow two rules:
    1. They must use the *from ___ import ___* syntax

```
from .api import delete, get, head, options,
                 patch, post, put, request
```

    2. The file where they are imported must be an absolute (non-relative) import



This relies on the shortcut defined in the __init__.py

```
import requests
requests.get(...)
```

This relies doesn't use a shortcut definition, but now requires us to be more familiar with the many files that make up the requests tool

```
import requests
requests.api.get(...)
```

Relative imports provide a way for package distributions (containing many files, perhaps) to reference other files internally without having to be concerned with absolute paths.

For example, consider the popular Python third-party *requests* module. The physical structure of this module is shown on the right above. Some of the most important methods are defined in the file called *api.py*. But due to the shortcut defined in the __init__.py, these items are now accessible as if they resided at the module level. This makes it easier to invoke methods by stating requests.get(), for example, and users don't have to know about the underlying file structure of the distribution.

What else can be in an __init__.py?

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

```
__version__ = '0.4.6'
```

From colorama/__init__.py

- Initialization code

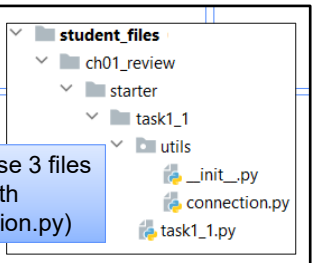
What else can be placed into the __init__.py? Anything from an empty file to lots of source code; however, commonly what is placed in here will be a docstring containing a description of the entire module, any import shortcuts, and perhaps version information.

__all__ = [item1, item2, …] will import only these items if the following syntax is used: from animals import *

In a way, __all__ defines the "public" interface of that module. The __all__ is not required and is only implicit not literal. But the use of __all__ can be helpful for those that are unfamiliar with the module layout.

Your Turn! - Task 1-1

Generators and the Database (Part 1 of 2)



Edit these 3 files (start with connection.py)

- Read the data from the countryinfo database table
  - Retrieve all country names, ISO symbols, and neighbors

iso	iso3	country	capital	neighbours
AD	AND	Andorra	Andorra la Vella	ES,FR
AE	ARE	United Arab Emirates	Abu Dhabi	SA,OM
AF	AFG	Afghanistan	Kabul	TM,CN,IR,TJ,PK,UZ
AG	ATG	Antigua and Barbuda	St	
AI	AIA	Anguilla	Th	
AL	ALB	Albania	Tirana	MK,GR,ME,RS,XK

Partial listing of countryinfo table

- Create a *@contextmanager decorator* that yields a connection to a 'with' control
  - Query the database saving the results in a dictionary as shown:

```
{
   'AD': ['AND', 'Andorra', 'Andorra la Vella', 'ES,FR'],
   'AE': ['ARE', 'United Arab Emirates', 'Abu Dhabi', 'SA,OM'],
   'AF': ['AFG', 'Afghanistan', 'Kabul', 'TM,CN,IR,TJ,PK,UZ'],
   'AG': ['ATG', 'Antigua and Barbuda', "St. John's", ''],
   'AI': ['AIA', 'Anguilla', 'The Valley', ''],
   'AL': ['ALB', 'Albania', 'Tirana', 'MK,GR,ME,RS,XK'],
   ...
}
```

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

argparse

argparse makes it easier to define and parse command line arguments

```
from argparse import ArgumentParser
from itertools import islice

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

args = get_args()

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

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

python 08_argparse.py *../resources titanic.csv -r -c 5*

ch01_review/08_argparse.py

The example reads a set number of lines from a file. It uses the command line to provide parameters such as the directory (dir), filename, number of lines to read (count), and whether to discard the header (header). The function returns an object, called *args* in this case, that contains dir, filename, count, and header attributes.

The *dest* parameter allows a different name to be used in the args object than what was provided on the command line while action defines the type of value used for the *remove_header* switch (bool in this case). The *nargs* parameter allows the directory to be optional. If it is not specified, the current working directory will be used.

coroutines

- Coroutines are generators that can "receive" values repeatedly

```
def write_msg(filepath: Path | str) -> Generator[None, str, None]:
    with open(filepath, 'wt', encoding='utf-8') as f:
        msg = yield
        while msg:
            print(msg, file=f)
            msg = yield

filename = 'coroutine_messages.txt'
write_coroutine = write_msg(filename)
next(write_coroutine)

message = click.prompt('Enter message to save (enter to quit)',
                       default='', show_default=False)
while message:
    write_coroutine.send(message)
    message = click.prompt('Enter message to save (enter to quit)',
                           default='', show_default=False)
```

Coroutines receive input from a send() call. Notice the yield is on the right-hand side here.

Creates the coroutine (which is a generator)

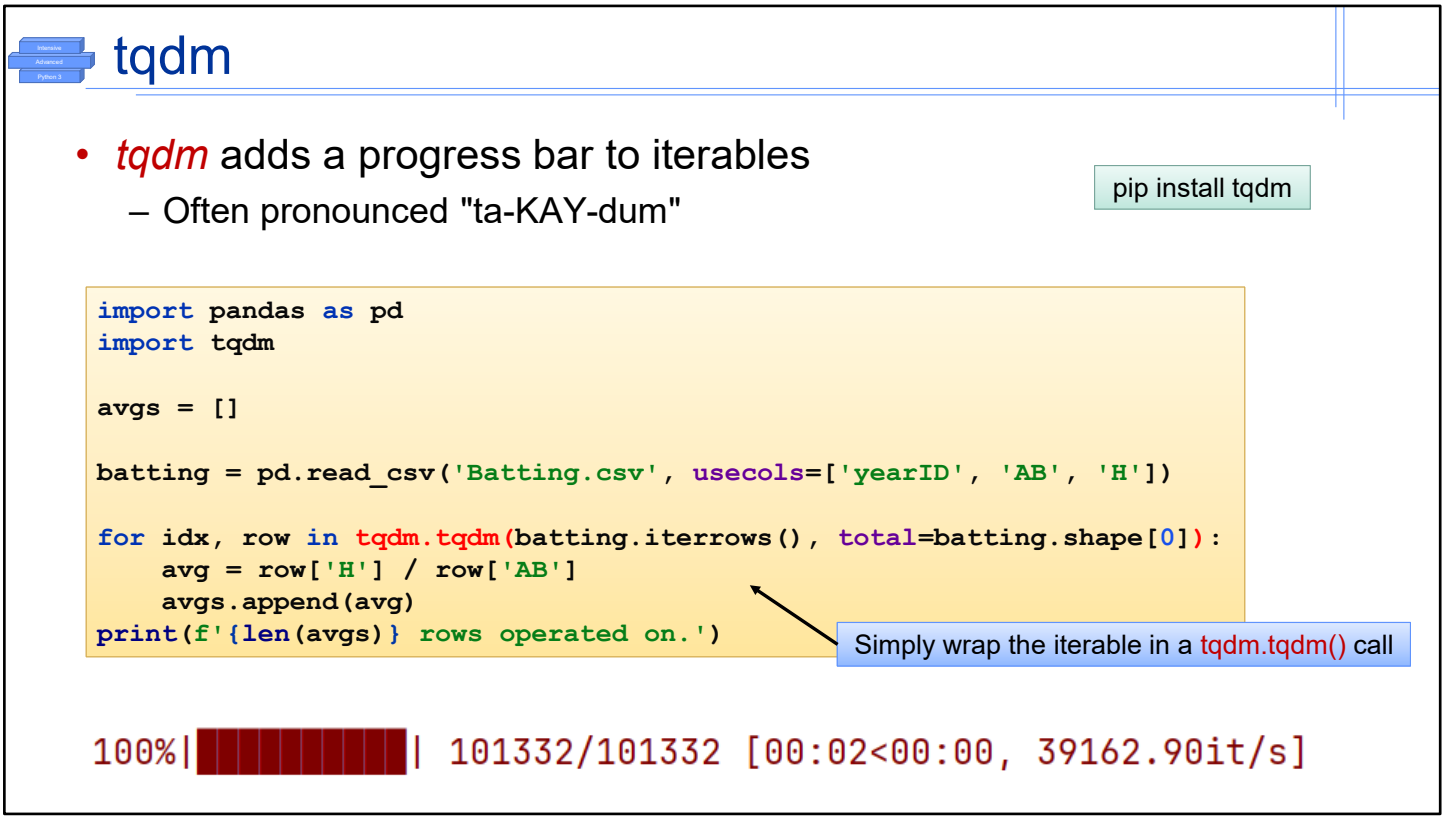
Advances ("primes") the coroutine up to the yield

Example uses the *click* library to receive user input

ch01_review/09_coroutines.py

In the example, messages are repeatedly sent to the coroutine from the send() method to the *yield* within the generator. Coroutines are just generators that receive input. Our type hint indicates this by stating that the Generator receives a str input (Generator[None, **str**, None]).

Coroutines can be useful if there is a need for repeated operations that may require a bit of initialization first (such as connecting to a database). The coroutine can perform this initialization once but then repeatedly send values into the coroutine, which pauses in-between calls, but doesn't have to perform the initialization over-and-over again.



ch01_review/10_tqdm.py

tqdm can wrap an iterable and add a progress bar output to the console results. Pass the iterable into the tqdm.tqdm() constructor and indicate a total number of counts it is expected to perform. There are a few variations for configuring the progress bar, but its simplicity makes it a great choice. Also, it may not always work well if intermediate output occurs in the middle of the progress bar. So, tqdm is best for long-running, uninterrupted tasks.

tqdm is short for "taqadum," which means "progress" in Arabic.

Your Turn! - Task 1-2

Co-routines, Command-line Args, and Progress Bars (Part 2 of 2)

Locate the instructions for this exercise in the back of the student manual.
Work from ch01_review\starter\task1_2.

- Determine all of a country's neighbors
  - Use a coroutine to receive values
  - Read country values from the command-line using argparse
  - Use tqdm to create a progress bar for each country processed

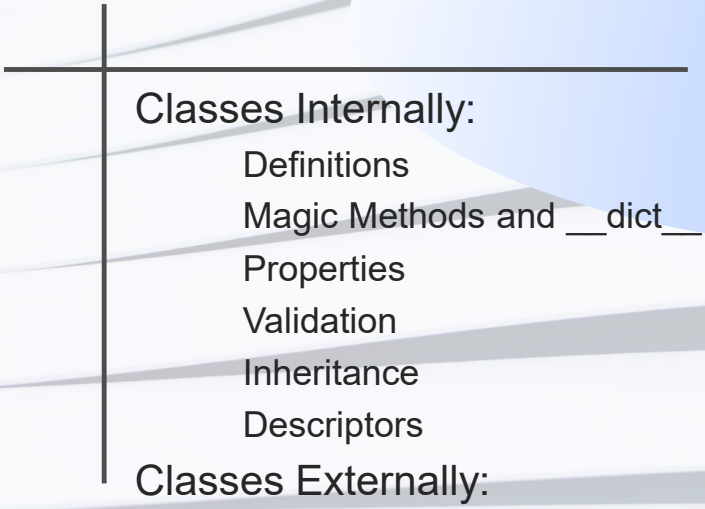
Sample Output:

```
(venv) C:\student_files\ch01_review\starter\task1_2>python task1_2.py us de in br
100%|████████████████████████████████████████████████████| 4/4 [00:04<00:00,  1.00s/it]
['Canada', 'Mexico', 'Cuba']
['Switzerland', 'Poland', 'Netherlands', 'Denmark', 'Belgium', 'Czech Republic',
 'Luxembourg', 'France', 'Austria']
['China', 'Nepal', 'Myanmar', 'Bhutan', 'Pakistan', 'Bangladesh']
['Suriname', 'Peru', 'Bolivia', 'Uruguay', 'Guyana', 'Paraguay', 'French Guiana',
 'Venezuela', 'Colombia', 'Argentina']
```

Chapter 1 Summary

- Context managers provide a way for effectively managing resources
  - Strategic use of the *with* control can result in elegant, easy-to-read solutions
  - Most DB vendors support the context manager protocol in one way or another
- *Argparse* takes the grunt work out of command line processing
  - Flexible module allows for numerous variation for handling input options
- Incorporate relative importing in special circumstances and __init__.py to simplify imports
- Decorators have become common in Python
  - They help library developers gain control over your code in order to process values
- The logging module provides the best way to record what happened in your code with better flexibility than simple output to the stdout

Chapter 2 - Overview



Classes Internally:

- Definitions
- Magic Methods and __dict__
- Properties
- Validation
- Inheritance
- Descriptors

Classes Externally:

- Design Principles / Patterns
- Abstract Classes

Classes

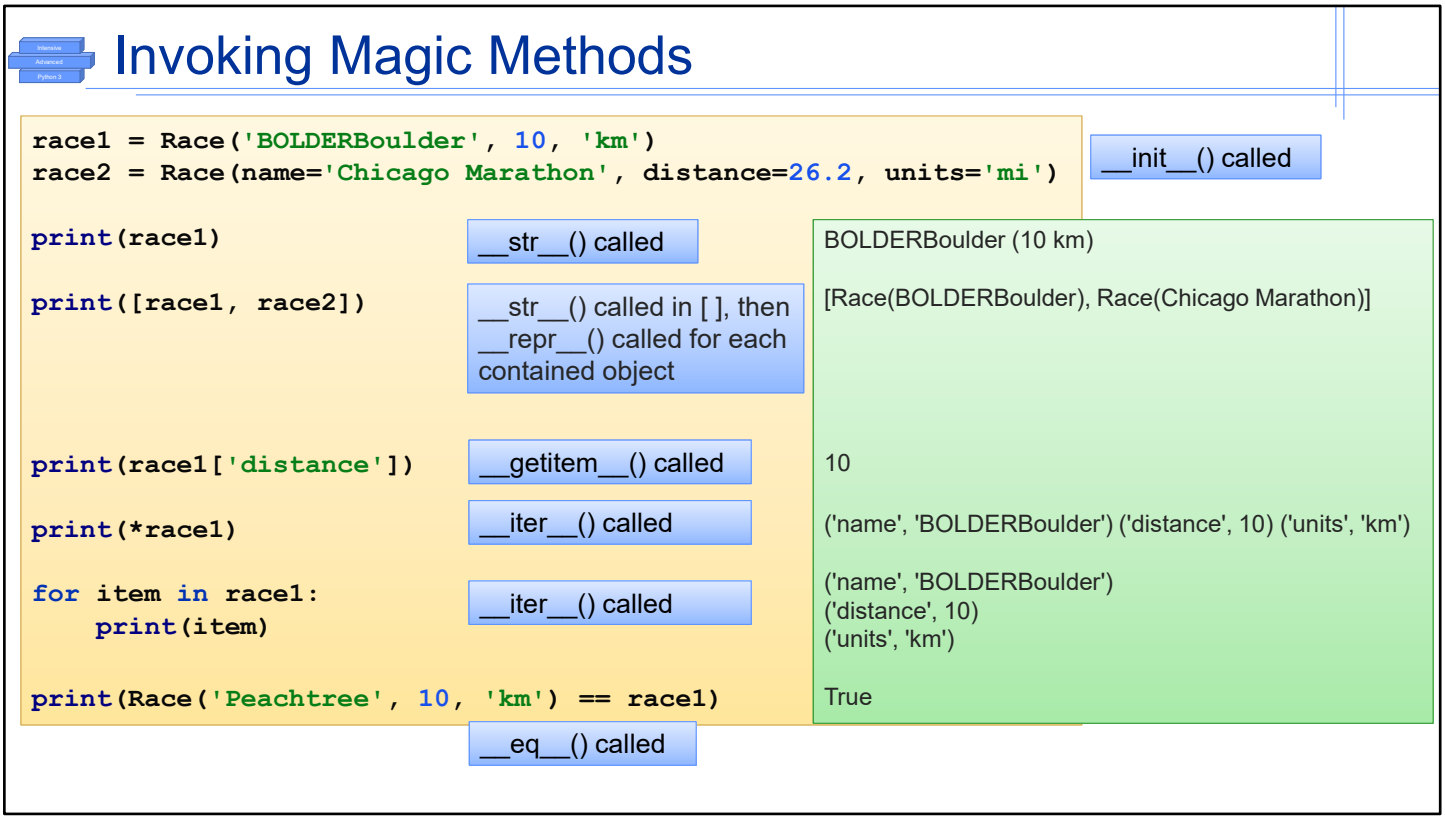
```
class Race:
    def __init__(self, name: str = '', distance: float = 0.0, units: str = 'km'):
        self.name = name
        self.distance = distance
        self.units = units

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

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

    def __getitem__(self, item):
        return self.__dict__[item]

    def __iter__(self):
        return iter(self.__dict__.items())

    def __eq__(self, other):
        if type(other) is not type(self):
            return False
        return True if self.distance == other.distance
                    and self.units == other.units else False
```

Classes consist of attributes, user-defined methods and magic-methods

continued…

ch02_classes/01_classes.py

This example continues onto the next slide. Here, we see a class definition for a Race (such as a running race, for example). In addition to both class and instance attributes, classes support both user-defined methods and predefined methods called "magic methods" or special methods. Magic methods are invoked under special circumstances.



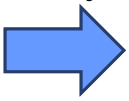
ch02_classes/01_classes.py

The statements above invoke the various magic methods of the defined Race class. Each type of statement invokes a different magic method depending on what the statement is trying to do.

__dict__

- Most classes are backed by a dictionary

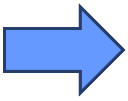
```
print(race1.__dict__)
```



```
{'name': 'BOLDERBoulder', 'distance': 10, 'units': 'km'}
```

  - Use vars(obj) to access the __dict__ from *outside* the class:

```
print(vars(race1))
```



```
{'name': 'BOLDERBoulder', 'distance': 10, 'units': 'km'}
```

```
vars(race1).get('distance')
```

Accesses *race1* via its dictionary representation returning the value of the *distance* attribute

  - Classes implementing __slots__ have no __dict__ (e.g., datetime class)

In Python, both classes and objects are backed by a dictionary. This means that under the hood an object has a __dict__ attribute that can be accessed.

A special attribute, called __slots__ removes the underlying dictionary that holds object data. This can be seen in the datetime module's datetime (or date or time) classes. These classes define __slots__which means a __dict__ will not exist. Originally, dictionaries consumed more resources than they do today, and thus, they were more wasteful. __slots__ was designed to allow objects *not* to have a dictionary in case a smaller memory footprint was desired. As a result of the improvements in dictionaries, __slots__ is less common today. However, classes within libraries like Click, Pydantic, Django, and others can be found using __slots__.

Properties

- Properties provide a way to access data before values are directly set on attributes

```
class Race(object):

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

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

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

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

Invokes the setter

Why use underscore in _distance?
By stating *return self.distance*, the property would invoke itself (recursion).

Will be filtered by the setter

ch02_classes/02_properties.py

Properties provide a way in Python to achieve a level of encapsulation by allowing access to values before actually trying to assign them to attributes. They behave in some ways like getters() and setters() from other object-oriented programming languages.

The underscore in distance is used to create a different variable to actually hold the distance value. While a different variable name could be chosen, such as "foo" for example, there are two reasons for using an underscore here: 1) _distance sounds more related to the distance() property than anything else, and 2) Underscore is an accepted syntax to mean "private" in Python.

Pydantic Field Validation vs DataClasses

```python
from dataclasses import dataclass
from dataclasses_json import dataclass_json

@dataclass_json
@dataclass
class Race:
    name: str = ''
    _distance: float = 0.0
    units: str = 'km'

    def __post_init__(self):
        if self.units == 'mi':
            self.units = 'km'
            self.distance =
                 self.distance * 1.60934

    (property present, but not shown)

race1 = Race('BolderBOULDER', 6.214, 'mi')
print(race1)
print(race1.to_json())
```

Provides serialization (to_json()) for dataclasses

Called after __init__

```python
from pydantic import BaseModel,
         ValidationError, field_validator

class Race(BaseModel):
    name: str = ''
    distance: float = 0.0
    units: str = 'km'

    @field_validator('distance')
    def validate_distance(cls, distance):
        if distance < 0:
            raise ValueError(
                      'Invalid distance.')
        return distance

try:
    race3 = Race(name='BolderBoulder',
                 distance=-5)
    print(race3)
except ValidationError as err:
    print(err, file=sys.stderr)
```

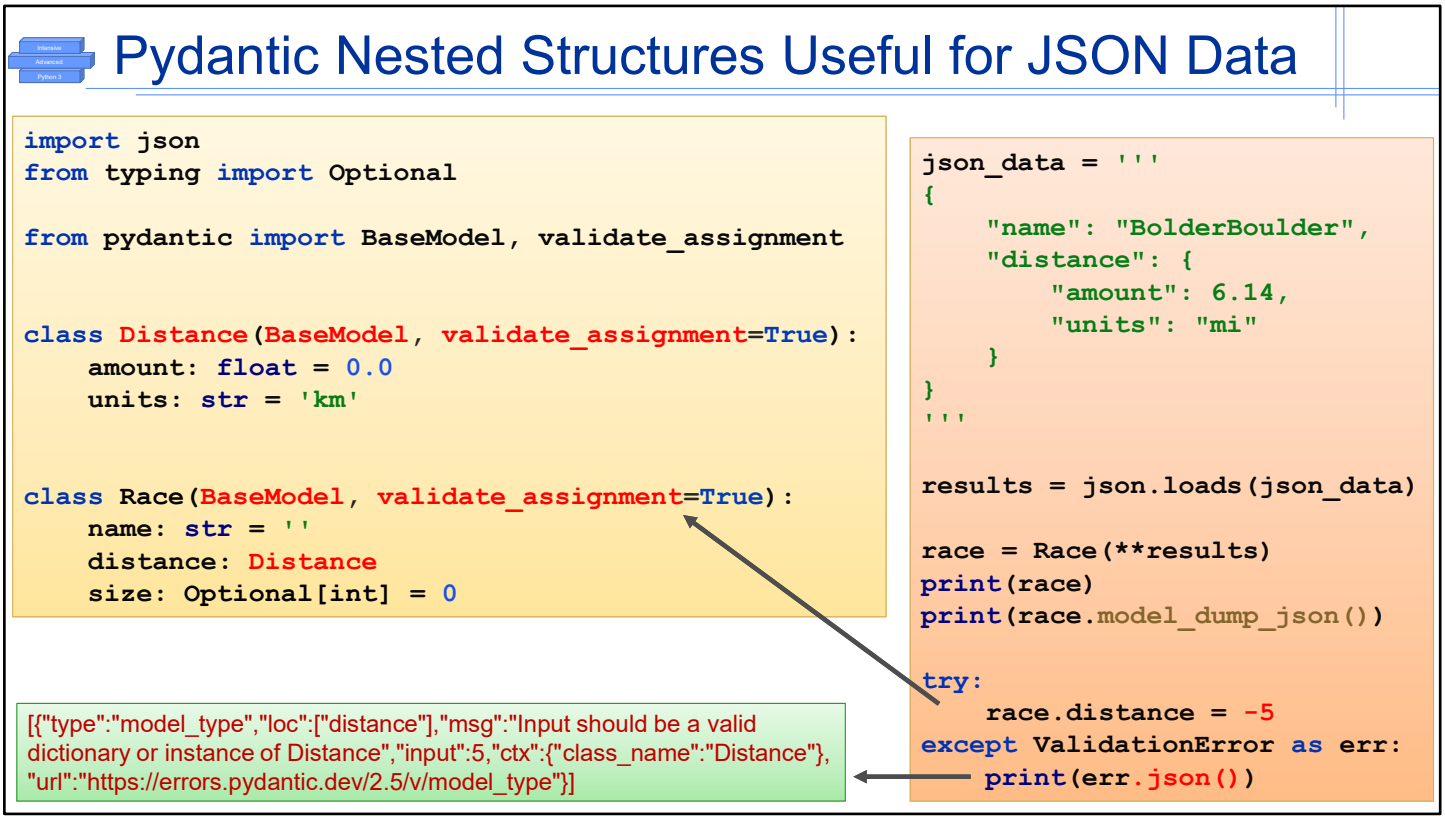
Called when setting *distance* attribute

ch02_classes/03_pydantic_vs_dataclasses.py

A single slide can't completely capture all the features of Pydantic, so a quick side-by-side reference to a dataclass is shown. Refer to the source file for additional notes on the use of dataclasses vs. Pydantic models.

Dataclasses are simple and easy to implement but don't provide any runtime type of checking. They can be useful for development environments or code quality checkers (more on these later). In the example above, the __post_init__ would be called after the __init__ to do further work. The *@dataclass_json* decorator comes from a third-party tool that provides adds a to_json() method to dataclass objects.

The use of BaseModel declares the class as a Pydantic model. This means features such as automatic field validation will be enabled. Note, by default this will require you to pass params into the __init__ using keyword arguments, but by overriding the Pydantic-provided __init__, you can enable the use of positional params as well. Pydantic models can also have a model_post_init() method that acts like the dataclass' __post_init__ method.



ch02_classes/04_pydantic_json.py

Here, our objects form a nested relationship. This relationship can be matched to data, such as JSON data, when brought in. On the right, sample JSON data is converted to a dictionary using the Python `json` module and then passed into the Race() constructor for instantiation.

Pydantic's default behavior is to validate only *on creation*. If validation is desired when changes are made, `validate_assignment = True` must be turned on as shown. The *model_dump_json()* method is available to any Pydantic model to convert from object to JSON format. In addition (though not shown), converting from JSON to object format can be done using *PydanticClassModel.parse_obj(dict),* or Race.parse_obj(results) above.

Inheritance

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

    def __str__(self):
        return f'{self.name} ({self.event_date.strftime('%Y-%b-%d')})'

    __repr__ = __str__

class Race(Event):
    def __init__(self, name: str, location: str = '', event_date: date = None,
                 distance: float = 0.0, units: str = 'km'):
        Event.__init__(self, name, location, event_date)
        self.distance = distance
        self.units = units

    def __str__(self):
        return f'Race: {Event.__str__(self)}'

events = [Event('Citywide Garage Sale', event_date=date(2024, 4, 14)),
          Race('BolderBOULDER', 'Boulder, CO', date(2024, 5, 27), 10, 'km'),
          Event('Presidential Election', 'USA', date(2024, 11, 5))]
print(events)
```

Explicit way to call parent __init__()

Invoke base class methods using: *Class*.method(self)

[Citywide Garage Sale (2024-Apr-14), BolderBOULDER (2024-May-27), Presidential Election (2024-Nov-05)]

ch02_classes/05_inheritance.py

The example takes our Race class and inherits from Event. To ensure the __init__() methods are properly invoked, the Event class __init__() gets called from within the Race class __init__() method.

Prefer *super()*

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

    def __str__(self):
        return f'{self.name} ({self.event_date.strftime('%Y-%b-%d')})'

    __repr__ = __str__

class Race(Event):
    def __init__(self, name: str, location: str = '', event_date: date = None,
                 distance: float = 0.0, units: str = 'km'):
        super().__init__(name, location, event_date)
        self.distance = distance
        self.units = units

    def __str__(self):
        return f'Race: {super().__str__()}'

events = [Event('Citywide Garage Sale', event_date=date(2024, 4, 14)),
          Race('BolderBOULDER', 'Boulder, CO', date(2024, 5, 27), 10, 'km'),
          Event('Presidential Election', 'USA', date(2024, 11, 5))]
print(events)
```

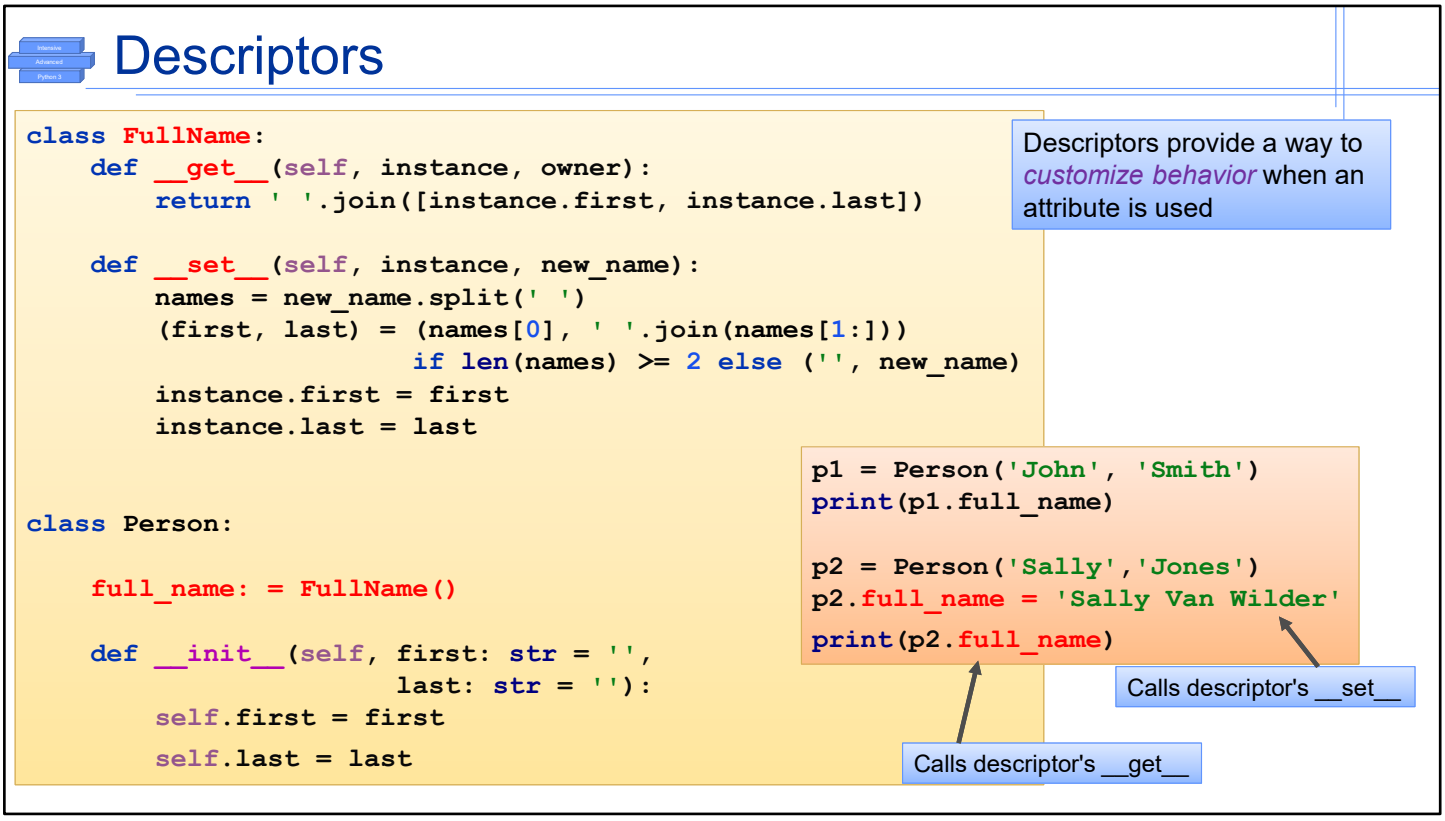
With super(), no *self* is passed

Invoke base class methods using super().method() - no *self* passed

[Citywide Garage Sale (2024-Apr-14), BolderBOULDER (2024-May-27), Presidential Election (2024-Nov-05)]

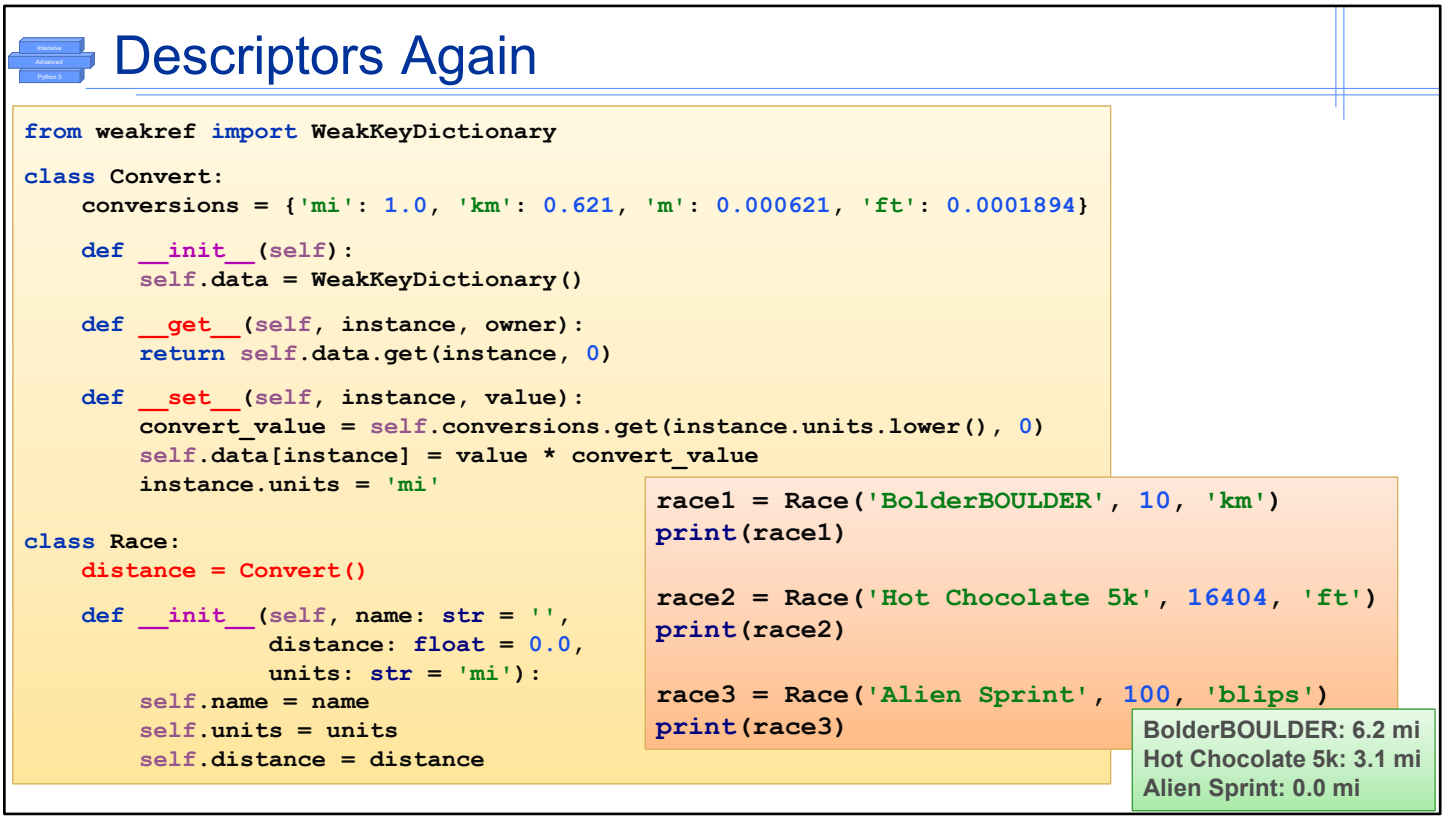
ch02_classes/06_prefer_super.py

The example shows the preferred use of super() instead of the explicit __init__() call from the previous example.



ch02_classes/07_descriptors.py

The descriptor class here is called FullName and exists separate from the class where it is used. The descriptor is used in the Person class. To use it correctly, it must be declared at the class level, and it must instantiate an object of the descriptor type. Now object instances can ask for the descriptor value which in turn calls the __get__() magic method of the descriptor class. Modifying the descriptor value calls the __set__() method of the descriptor class.



Descriptors Again

```
from weakref import WeakKeyDictionary

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

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

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

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

class Race:
    distance = Convert()

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

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

race2 = Race('Hot Chocolate 5k', 16404, 'ft')
print(race2)

race3 = Race('Alien Sprint', 100, 'blips')
print(race3)
```

```
BolderBOULDER: 6.2 mi
Hot Chocolate 5k: 3.1 mi
Alien Sprint: 0.0 mi
```

ch02_classes/08_descriptors_again.py

In this example, several race instances are created. One provides units in kilometers, another in feet, and another in an unrecognized unit. In each case a descriptor is used to track set or get calls to the *distance*. When the distance attribute is set, because of the distance = Convert() class-level argument, the __set__ method of the descriptor class will be invoked. This method checks the *units* attribute of the instance. If it is 'km' then the distance field value is converted to miles (as well as the units value). Later when the distance is displayed, the __get__ method of the descriptor class is invoked.

The use of the WeakKeyDictionary ensures that reference counting doesn't count the reference to instances held within this dictionary so that instances can be properly cleaned up later if desired.

For the Exercise: Using *requests*

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

pip install requests

```
import requests

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

print(r.text)

print(r.json())
```

Returns a string representation of the response

Converts a JSON response into a dict

The requests module is a popular third-party module used to simplify HTTP requests.

Most of the exceptions generated by the *requests* module come from RequestException and can be handled by this class as follows:

```
import requests
try:
    requests.get('http://yahoo.com')    # generates error
except requests.RequestException as err:
    print(err)
```

For the Exercise: Using *jinja2*

- *Jinja2* is a *templating library* for quick rendering of data

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

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

try:

    tmpl = env.get_template('example_tmpl.jinja')

    print(tmpl.render(data=races))

except TemplateError as e:
    print('Error: {0} {1}'.format(type(e), e))
```

example_tmpl.jinja

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

data= refers to variable within the template
races refers to value passed into the template

ch02_classes/tmpl/example_tmpl.jinja and 09_jinja2.py

This renders the provided template using the list of races passed to the *data=* keyword which is used within the template file.

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

- Create a script to query an API that returns JSON-based movie-related information
- Work from ch02_classes/task2_1_starter

Note: if no internet access is available, uncomment the **import mocklab** line in task2_1_starter/task2_1.py

```
Enter a movie phrase: titanic
 1 - Titanic
 2 - Titanic
 ...
 12 - Titanic Disaster
 13 - Titanic 2000
 14 - Titanic: The Complete Story
 15 - Asia's Titanic
 16 - Saving the Titanic
 17 - Titanic 2
 18 - Words of the Titanic
 19 - Titanic Arrogance
 20 - Beyond Titanic
Enter number for details: 1
Title:              Titanic
Released:           1997-11-18
Runtime:            194
Budget:             $200,000,000
Earnings:           $2,264,162,353
Tagline:            Nothing on Earth
could come between them.
```

Data for this exercise is acquired from the APIs at **http://api.themoviedb.org/**.

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

- Sample search results:

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

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

- Sample detailed title results:

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

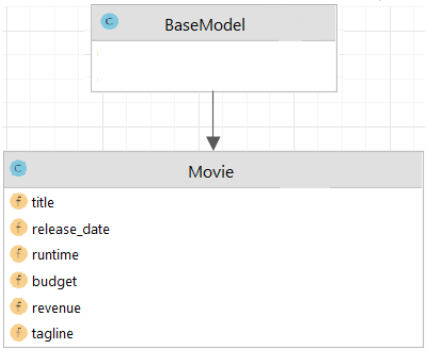
```
{
    "runtime":143, "title":"The Avengers", "budget":220000000,
    "id":24428,
    "overview":"When an unexpected enemy emerges...",
    "release_date":"2012-04-25",
    "revenue":1519557910,
    "tagline":"Some assembly required.",
}
```

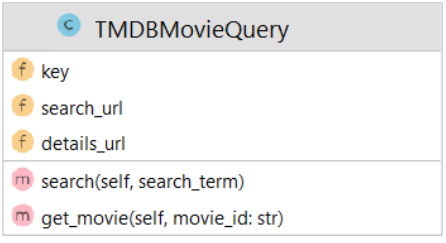
The above examples illustrate the format of the two returned JSON data structures.

The URLs used for this exercise are shown and provided in the source code. A key must be inserted into the URL.
This key can also be found in the task2_1_starter/task2_1.py file

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

- Create a Pydantic model, called Movie, to hold movie data
- Create a second class, TMDBMovieQuery to perform the queries to the API
- Prompt the user for a search phrase
- Display a list of these results
- Prompt to select a specific title
- Retrieve specific details for the movie
- Render the results using the provided Jinja2 template
- Edit task2_1_starter/movie/**movie.py**
  task2_1_starter/movie/**query.py**
  task2_1_starter/task2_1.py





Software Architecture: Class Design Principles

- Software architecture deals with the overall *structure* of the system
  - Python is repeatedly being used to build larger and larger systems making it more important to consider architecture and design
- Architecture patterns are used to
  - Break large, complex systems into manageable pieces
  - Create components that communicate with each other
  - Limit design choices that may promote a chaotic system
  - Simplify testing
  - Promote team-based development

Software architecture is common in enterprise application development due to the large, complex nature of systems created. Often, applications are built with many employees across distributed locations. This makes it important to use component architectures so that parts of the system can be worked on by different groups. Usually, enterprise systems have many subtasks to perform making it difficult to grasp the whole solution in its entirety. Component architectures help break systems into smaller manageable pieces.

Architecture deals with the system-level design structure. How do the components fit together and communicate with each other? What are the layers within the system? Security issues, performance factors, maintainability, scalability, reliability, and much more are all part of the architectural considerations.

Abstract Classes

- Abstract base classes are designed to enforce behavior (methods) in child classes
  - The abstract base class *cannot be instantiated*
  - The child class inherits methods from the abstract base class
  - Concrete class should implement the unimplemented base class methods

```
import abc
import sys


class AbstractBase(abc.ABC):
    def __init__(self, item):
        self.item = item
        self.do_action()

    @abc.abstractmethod
    def do_action(self):
        pass


class Concrete(AbstractBase):
    def do_action(self):
        print('in concrete do_action()')
        super().do_action()

c = Concrete('item')
```

Python 3.4+

Calls child class' do_action()

Or *raise NotImplementedError* or put actual code in here

in concrete do_action()

ch02_classes/10_abstract_classes.py

While abstract classes have been around in Python for a while, the syntax changed a couple of times. The approach shown here is valid since Python 3.4 through the current version of Python.

To have an abstract class, two things are needed: 1) Inherit from abc.ABC, and 2) mark with a decorator any methods that should be considered abstract. These methods are intended for force the child class(es) to define specifically what the method should do.

Abstract Classes *(continued)*

Subclasses must define this method

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

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

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

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

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

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

```
def show_message(msg: TextMessage) -> None:
    print(msg.get_text())


value = 'Meeting today at 3pm.'
show_message(HtmlMessage(value))
show_message(JsonMessage(value))

try:
    show_message(
       TextMessage('No Meeting today.'))
except TypeError as err:
    print(err, file=sys.stderr)
```

```
<span>Your meeting is at 3pm.</span>
{ "text": "Your meeting is at 3pm." }
TypeError: Can't instantiate abstract class
```

ch02_classes/10_abstract_classes.py

In this example, the TextMessage class has been made abstract by inheriting from abc.ABC. An abstract class must have at least one method marked as @abc.abstractmethod or else it won't be abstract. In other words, if the TextMessage class didn't mark the get_text() method as abstract, it would be able to be instantiated even if it still inherits from abc.ABC.

How can you prevent instantiation of a class without creating any abstract methods? You can mark the __init__() as abstract.

SOLID and Class Design in Python

- SOLID is a set a design principles aimed at creating software that is easy to maintain and understand

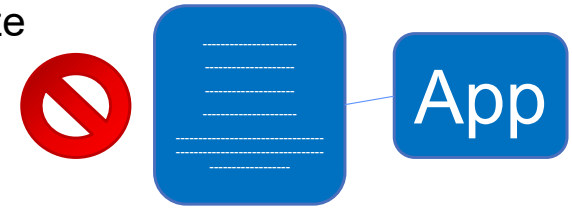
Does *SOLID* apply in Python?

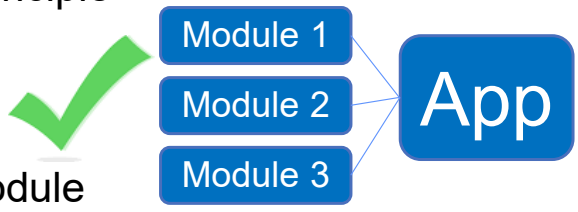
SRP - Single responsibility principle
OCP - Open/closed principle
LSP - Liskov substitution principle
ISP - Interface segregation principle
DIP - Dependency inversion principle

SOLID represents a set of guiding principles used in numerous languages (particularly statically typed languages) such as Java that have helped shaped large system design over the years. Does SOLID apply to Python? In some cases, yes. In others, no. Let's explore.

Single Responsibility Principle (SRP)

- *A module should have only one reason to change*
- Modules (and classes) should specialize in doing one thing
  - It should only have one action, or
  - It should only focus on one topic
- Standard library modules follow this principle
  - datetime, subprocess, thread
- Create modules with a single focus
- Violating this principle may cause a module to be highly coupled across the system





The Single Responsibility Principle (SRP) does apply to Python. In fact, it applies at both the class level as well as the module level. Having smaller modules allows apps to load only what is needed. At the class-level, classes in Python are the same as in other languages--keep them singular in concept/focus which ultimately improves reusability.

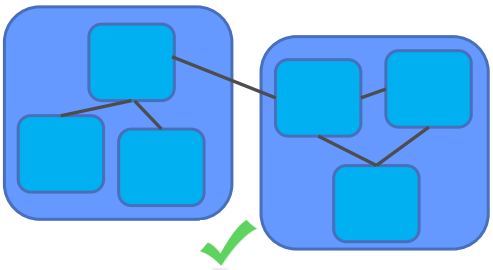
When a class has multiple responsibilities, it increases the likelihood that it will need to be changed more frequently over time. This change can cause parts of a system that rely on it to become broken.

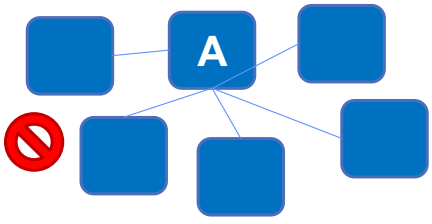
Cohesion and Coupling

This is SRP!

- Favor high cohesion
  - How well elements within a module belong together
  - This builds better reusability, robustness, and understandability
- Favor loose (low) coupling
  - How much elements from different modules refer to each other
  - Changes in one module won't affect another module as much

High cohesion: Related items are contained together decreasing interdependencies

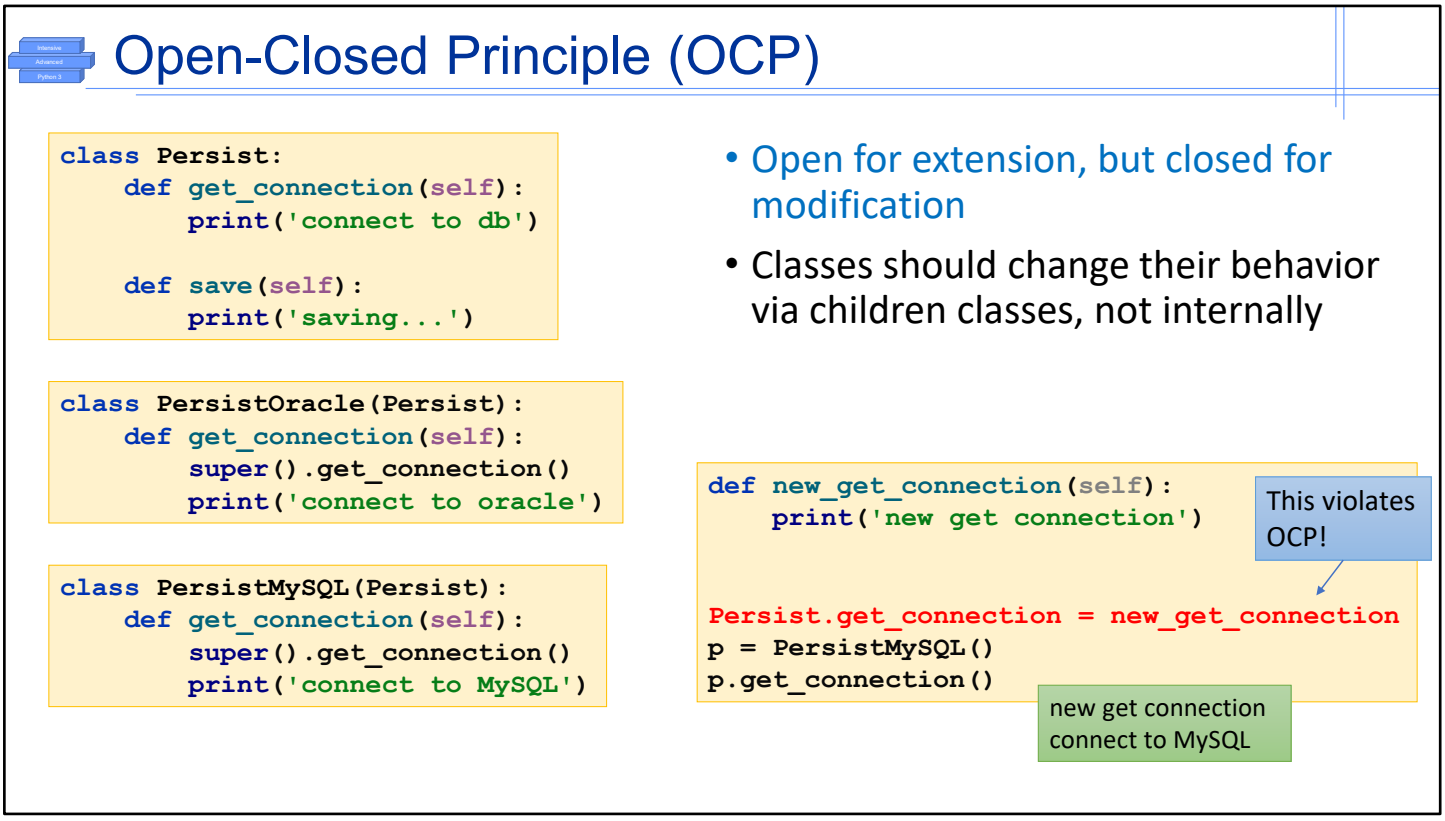




High coupling: If module **A** changes, it may affect all other modules

High cohesion simply means that the things that need to talk to each other (functions, classes) are located with each other. In other words, put related items together in a module.

Coupling means that classes and functions that reference other items (classes, functions, and variables) create a dependency (a line between boxes, as in the lower picture). The desire is to have the least number of lines between boxes in the lower diagram. Fewer lines represents lower coupling.



ch02_classes/11_ocp.py

This principle states that classes (since modules can't be extended in Python only classes will pertain to this rule) should only be extended from, they should not be modified directly (sometimes referred to as monkey patching).

As a good rule of thumb and best practice, extending classes should be the primary approach considered. However, because of the ease with which methods and attributes can be substituted, monkey patching does occur in Python. A degree of monkey patching is acceptable, such as in testing; however, overuse of this technique can lead to fractured code and can lead to unexpected behaviors in children classes.

Liskov Substitution Principle (LSP)

- Subclasses should be substitutable with base classes

While a human is not a Quacker, it still works when passed into the interact method

```
class Pond:
    def interact(self,
                 quacker: Quacker):
        quacker.quack()
        quacker.swim()

pond = Pond()
pond.interact(Duck())
pond.interact(Frog())
pond.interact(Human())
pond.interact(Fish())
```

These 3 work!

AttributeError!

- This principle is *mostly* valid in Python
  - Subclasses ***should*** be usable where a parent class is used
  - In Python, any object with the proper methods will work
    - Duck typing

```
class Quacker:
    def quack(self):
        print('quack!')

    def swim(self):
        print('splash!')

class Duck(Quacker):
    pass

class Frog(Quacker):
    pass

class Human:
    def quack(self):
        print('quack!')

    def swim(self):
        print('splash!')

class Fish:
    def swim(self):
        print('splash!')
```

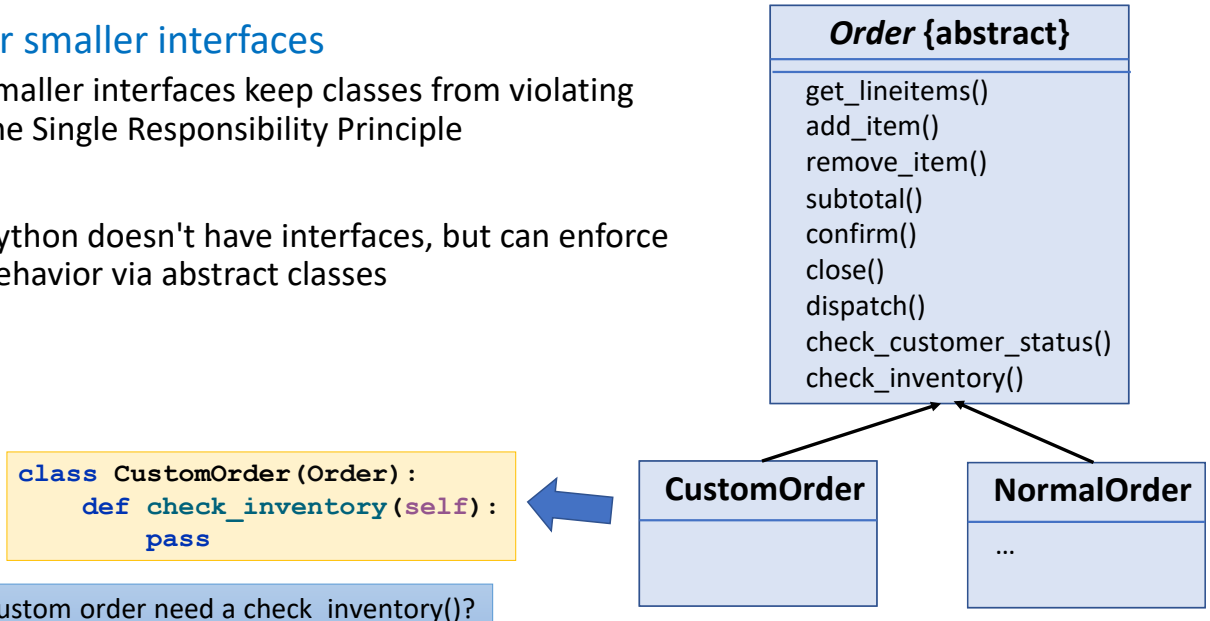
ch02_classes/11_ocp.py

If a method has a dependency of a given type, you should be able to provide an instance of that type or any of its subclasses without introducing unexpected results and without the method knowing the actual type provided (i.e., a Frog object can be passed into the interact() and it should work because Frog can do everything a Quacker can do).

And yes, some frogs can quack.

Interface Segregation Principle (ISP)

- Favor smaller interfaces
  - Smaller interfaces keep classes from violating the Single Responsibility Principle
  - Python doesn't have interfaces, but can enforce behavior via abstract classes




```
class CustomOrder(Order):
    def check_inventory(self):
        pass
```

Does a custom order need a check_inventory()?
What do we do with it now?

When concrete classes are dependent upon a specific set of interfaces, then the concrete class must implement all of the methods of those interfaces. If interfaces have many methods, then the concrete classes also must have many (typically unneeded) methods. This leads to using adapters to implement stubs for the unneeded methods.

These concepts are rarely encountered in Python; therefore, this principle gets less consideration in Python.

Dependency Inversion Principle (DIP)

- Depend upon abstractions, not concretions
- Possibly one of the most important of the SOLID principles to architecture
- You should adhere to this principle
- Python supports this principle (though it can be bypassed)

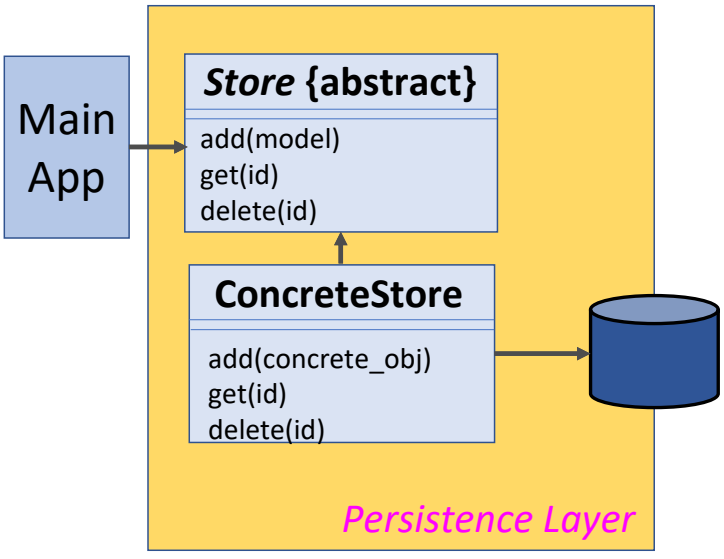
```
class Pond:
    def interact(self,
                 quacker: Quacker):
        quacker.quack()
        quacker.swim()

pond = Pond()
pond.interact(Duck())
pond.interact(Frog())
pond.interact(Human())
```

We did not specifically identify which type of object to pass into interact(), but rather, we indicate passing in a more generalized type. This way, we can easily pass different object types and we can easily mock it for testing purposes.

Though Python supports passing in "non-Quacker" types (as long as they swim and quack), at the design level we can ensure that our object hierarchy is valid and supported properly.

A Repository Pattern

- Repositories *encapsulate the logic* for working with data sources
- They decouple the service or business layer from the persistence infrastructure
- For each domain aggregate there should be one repository class
  - An aggregate is an overall business object, such as our Order or Customer object

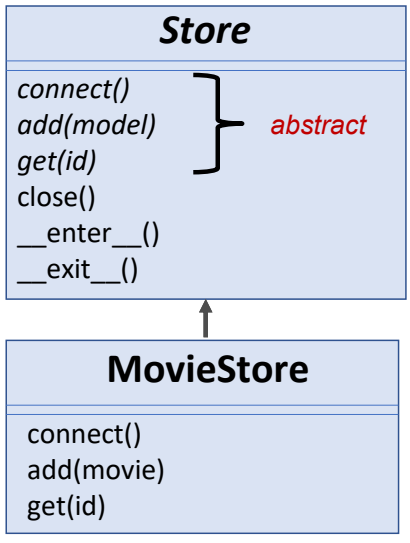


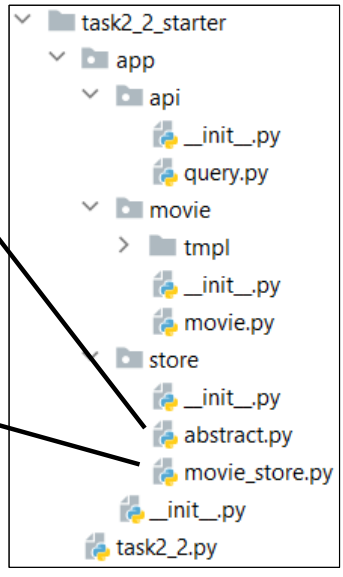
The repository pattern begins by creating an abstract base class that defines methods needed to work with the model against the data source. The concrete store will contain the actual logic for working with our data source. In this case, it will be an SQLite database. Other methods, such as delete(), find_by_XXX(), update(), etc., might be defined as needed, but keep the interface small (ISP) .

Your Turn! - Task 2-2

A Repository

- Use the repository pattern to persist Movie objects retrieved from the API in Task 2-1





course_data.db *(**movies** table):*

id	title	release_date	runtime	budget	revenue	tagline
1	Batman	1989-06-21	126	35000000	411348924	Justice is always darkest

Class Generics <T>

```python
from typing import Generic, TypeVar

T = TypeVar('T')

@dataclass
class Race:
    name: str = ''
    distance: float = 0.0
    units: str = 'km'

race1 = Race('BOLDERBOULDER', 6.214, 'mi')

class GenericListing(Generic[T]):
    items: dict[str, T] = {}

    def add(self, key: str, item: T) -> None:
        self.items[key] = item
        print(f'{type(item).__name__} added.')

    def get(self, key: str) -> T:
        return self.items.get(key)

class RaceList(GenericListing[Race]):
    pass
```

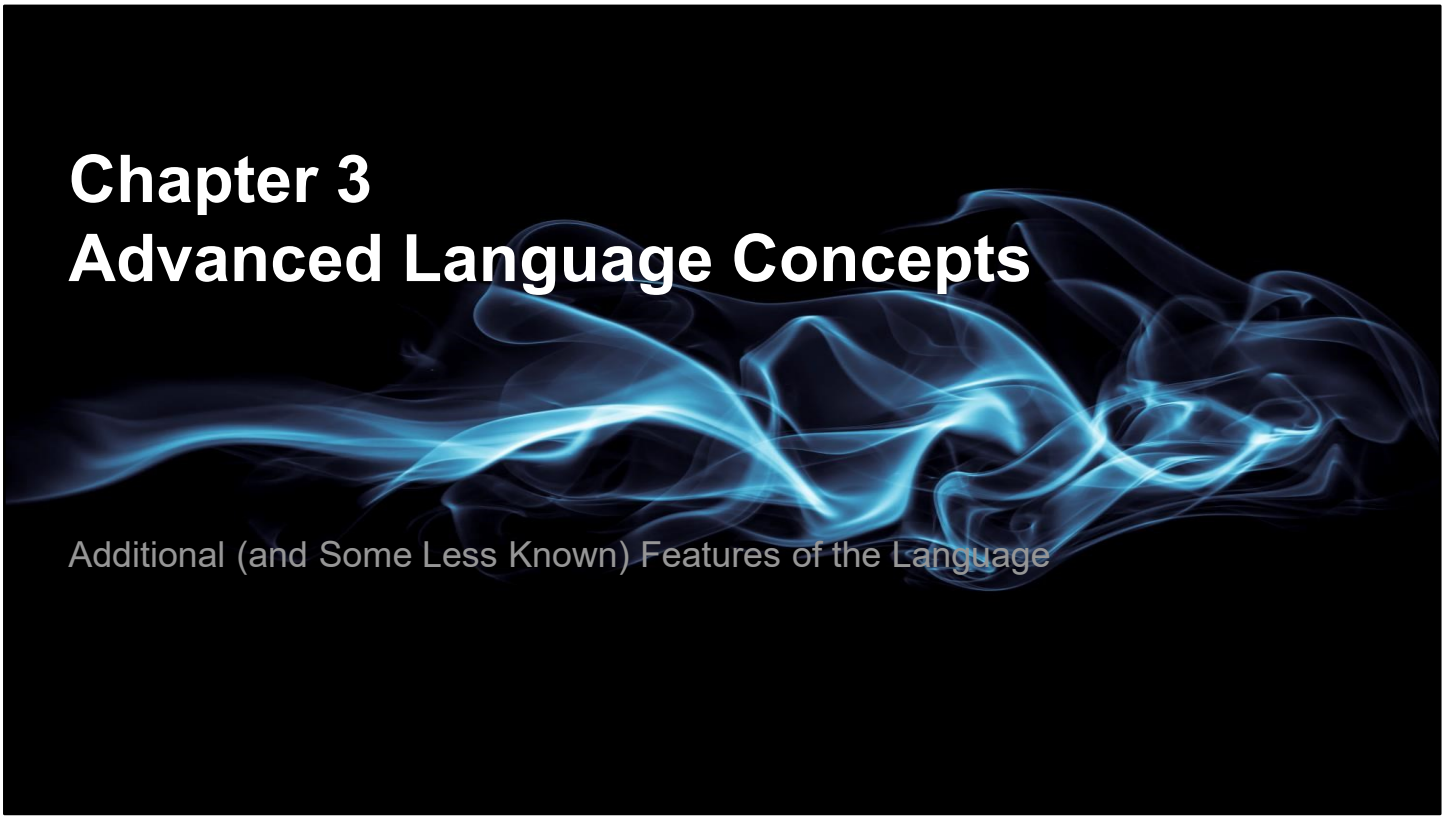
```python
rlist = GenericListing[Race]()
rlist.add('BOLDERBoulder', race1)
rlist.add('Unknown', 10)


rlist2 = RaceList()
rlist2.add('Peachtree',
           Race('Peachtree', 'Georgia'))
```

ch02_classes/11_generics.py

Sometimes in larger applications, numerous classes can arise that are very similar to each other, even differing by only the specific type of object they deal with. For example, above a RaceList deals with Race objects and even wants to provide an add() and get() method to operate on them. But a RaceList class might be like another class in the application, let's say a ParticipantList. So generically, a class could be created without reference to the specific type it deals with and instead replaces that type with the generic type 'T' created by using TypeVar. T can be any class type. Our derived class specifically implements T as a Race.

Chapter 2 Summary

- Classes have numerous magic methods that help implement the behavior of objects
- Pydantic classes aid in validating object data types which increase robustness of solutions
- Architecture deals with the structure of the overall system
  - Python has its own rules and guidelines for how best to write code
  - The SOLID principles deal with best practices for designing classes



Chapter 3
Advanced Language Concepts

Additional (and Some Less Known) Features of the Language

Chapter 3 - Overview

- Scope Rules
- Positional/Keyword-Only Rules
- Enums
- Literal Types
- Overloading
- StringIO
- Code Quality Checking
- Packaging

Python Scope Rules

- ***LEGB*** defines variable scope rules in Python
  - L look in the local function
  - E look at an enclosing function
  - G look to the global scope
  - B look for a built-in variable

```
db_url = 'database1'

class MovieStore:
    db_url = 'database2'

    def connect(self):
        db_url = 'database3'
        print(f'Connecting to: {db_url}')



ms = MovieStore()
ms.connect()
print(db_url)
```

How can you modify this value from within the connect() function?

Connecting to database3

Local scope

Comment the *db_url* variable out in connect(). What does this print now? Why?

database1

Global scope

ch03_adv_language/01_scope_rules.py

The LEGB acronym comes from *Learning Python* by Mark Lutz.

In this example, the local value for db_url is used to display the "local" variable. In the bottom print() statement, the global value (the first db_url variable defined on the top line) will be applied.

Using the global Keyword

- The *global* keyword indicates the desire to access ***and possibly modify*** a global variable from within a function

```
db_url = 'database1'

class MovieStore:
    db_url = 'database2'

    def connect(self):
        global db_url
        db_url = 'database3'
        print(f'Connecting to: {db_url}')



ms = MovieStore()
ms.connect()
print(db_url)
```

The global keyword here allows us to modify the variable

Connecting to database3

database3

The global variable was modified inside of connect()

ch03_adv_language/02_using_global.py

In this example, all of the red-colored db_url variables are the same variable. They all relate to the globally declared db_url. By using the keyword global in the connect() method, db_url is brought into the function. It can be modified within the function now.

Keyword/Positional-Only Functions

```
with Path('../resources/airports.dat').open(encoding='utf-8') as f:
    f.readline()
    data = [line.strip().split(',')[1:9] for line in f]


def get_location(search_name: str = None, max_results: int = 5) -> list[str]:
    if max_results < 1:
        max_results = len(data)
    return [record[0].strip('"') for record in data if search_name.lower()
                                        in record[0].lower()][:max_results]
```

The get_location() function can be successfully called in each of these ways

```
results = get_location('chicago', 3)
results = get_location('chicago', max_results=3)
results = get_location(search_name='chicago', max_results=3)
results = get_location(max_results=3, search_name='chicago')
print(results)
```

['Chicago Midway Intl', 'Chicago Ohare Intl', 'Chicago Rockford International Airport ']

ch03_adv_language/03_position_keyword_only.py

The function above returns partial matches of airport names. Each of the four examples shown above are perfectly valid and yield equivalent results.

Functions with Positional-Only Params

Positional-only operator

```
def get_location(search_name: str = None, /, max_results: int = 5) -> list[str]:
    if max_results < 1:
        max_results = len(data)
    return [record[0].strip('"') for record in data if search_name.lower()
                                                    in record[0].lower()][:max_results]


results = get_location('chicago', 3)
results = get_location('chicago', max_results=3)
results = get_location(search_name='chicago', max_results=3)
```

Okay
Okay
TypeError!

```
def get_location(search_name: str = None, max_results: int = 5, /) -> list[str]:
    if max_results < 1:
        max_results = len(data)
    return [record[0].strip('"') for record in data if search_name.lower()
                                                    in record[0].lower()][:max_results]


results = get_location('chicago', 3)
results = get_location('chicago', max_results=3)
```

Positional-only operator

Okay
TypeError!

Params declared ***before*** / can only use the positional approach

ch03_adv_language/03_position_keyword_only.py

All items ***before*** the forward-slash (**/**) must use the positional parameter technique.

Functions with Keyword-Only Params

Keyword-only operator

```
def get_location(search_name: str = None, *, max_results: int = 5) -> list[str]:
    if max_results < 1:
        max_results = len(data)
    return [record[0].strip('"') for record in data if search_name.lower()
                                                    in record[0].lower()][:max_results]

results = get_location('chicago', max_results=3)
results = get_location(search_name='chicago', max_results=3)
results = get_location('chicago', 3)
```

Okay
Okay
TypeError!

Keyword-only operator

```
def get_location(*, search_name: str = None, max_results: int = 5) -> list[str]:
    if max_results < 1:
        max_results = len(data)
    return [record[0].strip('"') for record in data if search_name.lower()
                                                  in record[0].lower()][:max_results]


results = get_location(search_name='chicago', max_results=3)
results = get_location('chicago', max_results=3)
```

Okay
TypeError!

Params declared *after* * can only use the keyword approach

ch03_adv_language/03_position_keyword_only.py

All items ***after*** the asterisk (splat) (*****) must use the keyword parameter technique.

Both Positional and Keyword-Only Params

Positional-only operator

Keyword-only operator

```
def get_location(search_name: str = None, /, *, max_results: int = 5) -> list[str]:
    if max_results < 1:
        max_results = len(data)
    return [record[0].strip('"') for record in data if search_name.lower()
                                                    in record[0].lower()][:max_results]

results = get_location('chicago', max_results=3)
results = get_location(search_name='chicago', max_results=3)
```

Okay
TypeError!

- Why use these?
  - Simply put, to create easier-to-read solutions

```
val = int('3')
val = int(x='3')
```

```
for item in range(3, 5):
    print(item)
for item in range(step=1, stop=5, start=3):
    print(item)
```

```
data = (['c', 'a', 'b'])
data.sort(key=lambda x: x, reverse=True)
data.sort(lambda x: x, True)
```

ch03_adv_language/03_position_keyword_only.py

Both operators may be used at one. If you define parameters in between the / and * then they can be provided as *either* positional *or* keyword arguments.

At the bottom are three examples where either keywords may only be used (as in the sort function) or positions may only be used (as in the range and int functions).

Enumerations (Enums)

- Enums are classes containing groups of enumerated values
  - Often useful when objects can exist in or have different states

```
from enum import Enum


class Direction(Enum):
    NORTH = 1
    EAST = 2
    SOUTH = 3
    WEST = 4

    @staticmethod
    def turn_right(faces):
        return Direction.NORTH \
            if faces == Direction.WEST \
            else Direction(faces.value + 1)
```

name

value

Attributes are called Enum *members*

Should be uppercased, can't be reassigned

Enums can have methods

```
for member in Direction:
    print(member, member.name, member.value, sep=', ')

facing = Direction.NORTH
for turn in range(4):
    facing = Direction.turn_right(facing)
    print(facing)
```

```
Direction.NORTH, NORTH, 1
Direction.EAST, EAST, 2
Direction.SOUTH, SOUTH, 3
Direction.WEST, WEST, 4
```

```
Direction.EAST
Direction.SOUTH
Direction.WEST
Direction.NORTH
```

ch03_adv_language/04_enums.py

Python introduced Enums in version 3.4 and as expanded their capabilities in Python 3.11 and 3.12. Python Enums are not regular classes. Asking for a value from an enum, doesn't return the value, it returns the member. Its value can be obtained from the notation: *member*.value (see above). Don't define repeated values for enums or it won't function properly. You can optionally let Python create values for you (starting at 1) by using the auto() function (e.g., NORTH = auto()). Enums can also inherit from IntEnum and StrEnum to create enums that override the __str__() magic method.

Not shown above is the ability to define a bitwise flag system as follows:

```
from enum import Flag

class Style(Flag):
    ITALIC = 1
    BOLD = 2
    UNDERLINE = 4
    UPPERCASE = 8

fmt = Style.ITALIC | Style.UPPERCASE | Style.UNDERLINE
print(fmt, fmt.name, fmt.value, sep='\n')
```

```
Style.ITALIC|UNDERLINE|UPPERCASE
ITALIC|UNDERLINE|UPPERCASE
13
```

Type Literals

- Type Literals (somewhat similar to Enums) allow type checkers to ensure variables can only be of a limited type

```
from typing import Literal, get_args

Size = Literal['Small', 'Medium', 'Large', 'Extra-Large']
CakeType = Literal['Chocolate', 'White', 'Carrot', 'Red Velvet', 'Lemon']

class CakeOrder:
    prices = [20, 40, 80, 100]

    def __init__(self, typ: CakeType, size: Size = 'Medium'):
        idx = get_args(Size).index(size)
        self.values = (self.prices[idx], typ, size)

print(CakeOrder('Chocolate').values)
print(CakeOrder('Red Velvet', 'Extra-Large').values)
print(CakeOrder('Carrot', 'Big').values)
```

```
(40, 'Chocolate', 'Medium')
(100, 'Red Velvet', 'Extra-Large')
ValueError: tuple.index(x): x not in tuple
```

ch03_adv_language/05_type_literals.py

Type literals were introduced in Python 3.8. They are handy when only a discrete set of values is desired for use. These can generate ValueErrors when a non-allowed value is provided as shown above.

Overloading with @multimethod

- Overloading is the ability to create functions that take different parameters but have the same name
  - It can be useful to simplify work for the caller of your functions
- Strictly speaking, Python doesn't support overloading
  - @multimethod can dispatch to a registered function based on annotated types

```
from multimethod import multimethod

@multimethod
def payment(employee: Manager) -> float:
    return employee.calc_pay()

@multimethod
def payment(employee: HourlyEmployee,
            hours: float) -> float:
    return employee.calc_pay(hours)

m = Manager(salary=120_000)
h = HourlyEmployee(rate=48.00)
print(f'The manager gets: {payment(m)}.')
print(f'The employee gets: {payment(h, 33.0)}.')
```

Note: *multimethod*, is a third-party tool. Python provides a standard library option, called *singledispatch* that only considers the first parameter to a function

ch03_adv_language/06b_multimethod.py

For brevity, the HourlyEmployee and Manager classes aren't shown here. Refer to the source file for these classes.

The @singledispatch decorator from the functools module only looks at the **first parameter** to determine which registered subsequent function to call. It isn't exactly the true concept of overloading, but it can provide a way to organize functions and provide a simplified interface to the user. Here, the multimethod decorator is used and considers *all* parameters of a function. So, the hours parameter of the second payment() function is looked at. If the hours is an int instead of a float, for example, an error will arise.

Ellipsis and @Overload

- @overload makes it possible to specify multiple type *signatures*
  - Methods using this decorator should not provide implementations
  - A *non-decorated* function must appear last

```
from typing import overload, Optional

@overload
def get_location(search: str) -> list[str]:
    ...

@overload
def get_location(search: Iterable[str]) -> list[list[str]]:
    ...


def get_location(search: str | Iterable[str]) -> list[str] | list[list[str]]:
    if isinstance(search, str):
        not shown for brevity
    elif isinstance(search, Iterable):
        not shown for brevity
```

Two versions of get_location() illustrate the possible type combinations that are allowed

... and pass are largely equivalent

The last function *should not* have the decorator, but *should* have the implementation

ch03_adv_language/07_overload.py

@overload is merely for type declarations. The first two definitions are not given implementations. They merely identify a possible type signature.

@overload allows for defining functions that may accept differently typed parameters. The last function **must** provide a body and define what it will do with *all* implementation versions.

Note: for handling different numbers of parameters in overloaded, see the lower example of the source file.

StringIO

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

```
import io

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

print(buffer_results)
```

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

This is the initial buffer value.
Added to buffer.

StringIO is supported within a *with* control also

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

ch03_adv_language/08_stringio.py

io.StringIO objects create file-like behavior in memory for quick read access or write manipulation. Larger StringIO objects can benefit in performance over the rebuilding of strings in memory since strings are immulable and have to be re-written to memory each time.

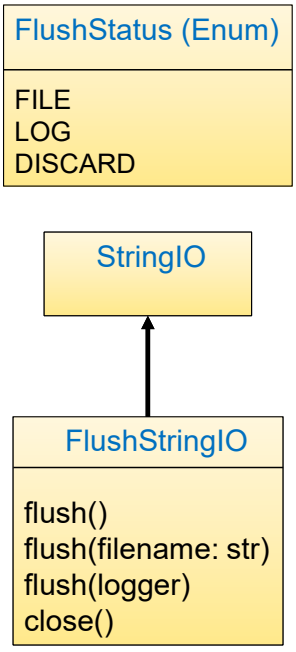
Note: the slide above shows how to use the print() statement to redirect the standard out to the StringIO object.

A BytesIO object exists for binary data as well.

Your Turn! - Task 3-1

A Self-Determined StringIO

- Create (via Inheritance) a StringIO object that determines how its value will be used
  - Use an *enum* to identify how it will be used
  - Use *multimethods* to invoke the behavior on the custom StringIO type
  - Work from *task3_1_starter.py*



PyCharm Code Quality Analyzing vs *mypy*

- *mypy* is a static type checker that can catch many programming errors without running your scripts

pip install mypy

> *mypy* file1.py file2.py or > *mypy* directory

```
Command Prompt

(venv) c:\student_files>mypy ch03_adv_language python_IN1469
ch03_adv_language\02_type_literals.py:16: error: Incompatible default for argument "size" (default has type "Literal['Normal']", argument has
type "Literal['Small', 'Medium', 'Large', 'Extra-Large']")  [assignment]
ch03_adv_language\02_type_literals.py:25: error: Argument 2 to "CakeOrder" has incompatible type "Literal['Big']"; expected "Literal['Small',
'Medium', 'Large', 'Extra-Large']"  [arg-type]
Found 2 errors in 1 file (checked 2 source files)
```

```
class CakeOrder:
    ...
    def __init__(self, typ: CakeType, size: Size = 'Normal'):
        idx = get_args(Size).index(size)
        self.values = (self.prices[idx], typ, size)

print(CakeOrder('Carrot', 'Big').values)
```

> *mypy --strict* directory

Stricter checking will catch nearly any possible runtime type error

Here we modified the default argument "Medium" used in the CakeType __init__() to "Normal". Tools like PyCharm, while powerful, are not able to catch this error condition (at least at the time of this writing). On the other hand, tools such as *mypy* provide stronger type checking capabilities. It can catch these conditions as well as numerous other related to possible type checking problems.

For a quick overview of mypy type checking, visit:
https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Black, and Flake8, Autopep8

- *black, flake8, autopep8* are code formatters
  - Use in addition to *mypy* to add format syntax checking to your code analyses

pip install black

pip install flake8

pip install autopep8

```
(venv) c:\student_files> black ch03_adv_language
reformatted C:\student_files\ch03_adv_language\02_type_literals.py
reformatted C:\student_files\ch03_adv_language\01_enums.py
2 files reformatted.
```

Black is very uncompromising and may reformat code in a way you do not like

```
(venv) c:\student_files> flake8 ch03_adv_language
ch03_adv_language\01_enums.py:17:80: E501 line too long (89 > 79 characters)
ch03_adv_language\02_type_literals.py:17:10: E111 indentation is not a multiple of 4
ch03_adv_language\02_type_literals.py:17:10: E117 over-indented
ch03_adv_language\02_type_literals.py:18:10: E111 indentation is not a multiple of 4
ch03_adv_language\02_type_literals.py:25:80: E501 line too long (104 > 79 characters)
```

Flake8 only makes suggestions

```
(venv) c:\student_files> flake8 --ignore E501,E111 ch03_adv_language
ch03_adv_language\02_type_literals.py:17:10: E117 over-indented
```

Turn off some rules

```
(venv) c:\student_files> autopep8 --in-place --ignore E501 02_type_literals.py
(no output)
```

Formats file(s) directly, ignores line too long rule

It's worth noting that the PyCharm code quality checking feature performs other kinds of checks, particularly code formatting checks that *mypy* does not. For a more complete code analysis, use *mypy* in conjunction with *black* to enable code formatting checks as well. Black is very unforgiving and doesn't support a large range of options for turning features on and off. For this, consider using flake8. Flake8 can ignore certain rules if desired. To allow modifications, use autopep8 --in-place. Autopep8 also uses the same error codes as Flake8.

Tools for Packaging: Creating a Source Distribution

- Python code can be packaged and made available for pip installs
- Different formats exist for creating package distributions
  - The type used is partially up to you and partially the project type
    - A single module
    - Multiple .py files
    - .py files and other (potentially compiled) resources
- A source distribution, or *sdist*, is a common way to create a library
- To create an *sdist* distribution:
  1. Establish a project structure
  2. Define the *.toml* file
  3. Define the Readme.md
  4. Define a license
  5. Build the distribution

.toml files require the **build** package to be installed

.toml files represent a newer version over the classic *setup.py*

Depending on the type of project you have will determine the approach used. A large majority of Python projects are simply made up of .py files or other static resources. Projects that do not require compiling code into native formats can use the sdist (source distribution) approach for building distributions. This is discussed further on the subsequent slides.

Creating the Project Structure

- Use the following project structure when building an *sdist* type of distribution

```
flushstringio/
├── LICENSE
├── pyproject.toml
├── README.md
└── src/
    └── myutil_package
        ├── __init__.py
        └── myutil.py
```

```
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "flushstringio"
version = "1.0"
authors = [
  { name="Example Author", email="author@example.com" },
]
description = "Utility for creating StringIO objects that can decide how their value is saved."
```

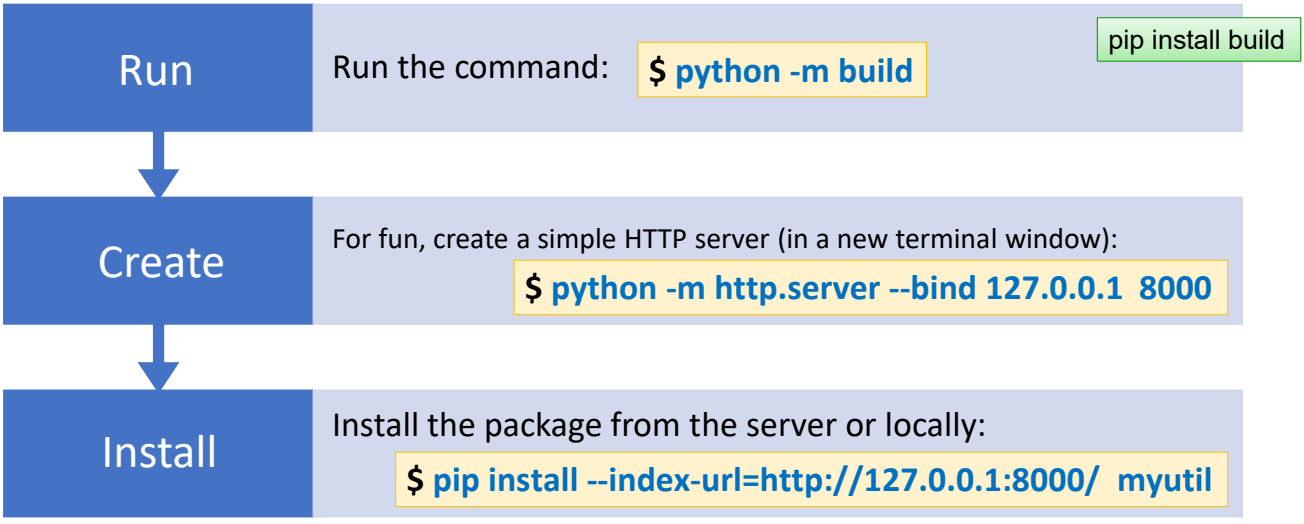
ch03_adv_language/task3_2_starter

.toml files are short for Tom's Obvious Minimal Language which defines a simple configuration file format.

Make sure to create your project by following a proper structure. Documentation files such as the README, LICENSE, and .toml files are all located at the root of the project. Your files should be placed in a src directory (usually) in the correct hierarchy.

Build and Deploy the Distribution

- Build the wheel
  - It can even be made available for installation



After the project structure is complete and the supporting documentation has been created (.toml file, readme, and license), the distribution can be built with the python -m build command. In order to do this, the build module should be pip installed first. Run the python -m build command from the directory where the .toml file exists.

As a separate step and demonstration, we included how you can start an HTTP server (in the directory where the distribution was created) and the package can be installed from the server using the pip command.

Your Turn! - Task 3-2

Packaging

- Work from the ch03_adv_language/task3_2_starter folder
- Refer to specific instructions in the exercise workbook in the back of the (PDF) manual
- Validate the project (task3_2_starter) using **mypy**
- Complete the .toml file
- Build the distribution (wheel) file
- Launch an HTTP server to server the new wheel file
- pip install it into our Python virtual environments
- Test it
- Optionally, uninstall it using pip uninstall

Chapter 3 Summary

- To make function calls easier to read, limitations can be placed on how parameters are provided
  - These are called positional-only and keyword-only parameters
- *Enums*, *type literals* and *overloading* techniques are all features in Python that have been around for quite some time since Python 3 was released
- Type checkers, such as mypy and code quality analyzers, such as flake8, are third-party tools that can be easily installed
- The newer format for building libraries is to write a .toml file

Chapter 4
Automation and Standard Library Modules

Tools to Assist in Automating Environments

Chapter 4 - Overview

File Management and Directory Access
os.environ
Email Notifications
Creating PDFs
Timeit
Dictionaries and JSON
Context Manager Utilities

File and Directory Access Modules

- The standard library provides useful modules for working with files
- These modules include:
  - pathlib - cross-platform directory-based utilities
  - fnmatch - for matching file patterns and file names
  - shelve - persisted dictionary-style objects
  - shutil - for moving/copying multiple files
  - filecmp - compares files and directories
  - zipfile - creates, reads, extracts items from zip files

os.environ

- os.environ is a dictionary that captures a user's operating system environment variables

```
import os
import pprint
from pathlib import Path

pprint.pprint(dict(os.environ))
print(os.getenv('HOME'))

print(os.path.expanduser('~'))

print(Path.home())
print(Path('~').expanduser())

print(Path('~username').expanduser())
```

Yields *None* on Windows

Works on all platforms, but uses older os module

Both return a Path() object of the user's home

Returns Path() object for a specific user

ch04_automating/01_environ.py

os.environ can be modified (since it is just a dictionary), but changes to it will only affect the Python session, not the actual operating system values.

Use Path.home() to get a Path object to the current user's home directory.

Working with Paths and fnmatch

```
from pathlib import Path
p = Path(__file__)

print(p)                    c:\student_files\ch04_automating\02_pathlib.py
print(p.parent)             c:\student_files\ch04_automating
print(p.parents[0])         c:\student_files\ch04_automating
print(p.parents[1])         c:\student_files

print(p.name)               02_pathlib.py
print(p.stem)               02_pathlib
print(p.suffix)             .py
print(p.drive)              C:
```

- Path serves as an object-oriented wrapper around os paths

Path objects have many methods that replace the os module methods

General syntax:
Path(location).method()
over os.method(path)

```
resources_dir = p.parents[1] / 'resources'

pattern = 'c*.txt'

for filepath in resources_dir.iterdir():
    if fnmatch.fnmatch(filepath.name, pattern):
        print(filepath)
```

cities15000.txt
cities15000_info.txt
countryInfo.txt

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

ch04_automating/02_pathlib.py

Python has slowly been converting many standard library methods to accept Path objects. It presents a cleaner, more object-oriented syntax than using the os module. Where a function doesn't accept a Path object, simply convert it to a string using: str(Path()).

Most os module operations can also be performed with Path objects.

fnmatch() returns True if the pattern matches the provided filename.

shelve

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

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

```
import shelve
from pathlib import Path

prefs = Path(__file__).parent / 'prefs.dat'
with shelve.open(str(prefs)) as sh:
    sh['data_dir'] = Path('..') / 'resources'
    sh['data_file'] = 'poe.txt'

with shelve.open(str(prefs)) as sh:
    file_location = sh['data_dir'] / sh['data_file']
    if file_location.exists():
        print(file_location.read_text())
```

Saves directory and file object to the file system

Reopens shelve object, retrieves persisted data

ch04_automating/03_shelve.py

The shelve module is useful for smaller, single-system, sysadmin-related apps. Shelve uses the pickle object serialization technique to persist data to the file system.

The example above must convert the Path() object to a string first before calling open() as shelve doesn't support pathlib as of Python 3.12.

This example opens a file (or creates it if it doesn't exist) using shelve and writes two entries (under the key of "data_dir" and "data_file"). This file is later opened and the information within it is retrieved.

shutil

- shutil contains utilities for copying, moving, and removing files
  - For individual file-related tasks, use the pathlib.Path class or os module

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

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

shutil provides methods for copying larger numbers of items.

filecmp

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

```
sample = Path('sample')
print(filecmp.cmp(sample / 'quotes1/thoreau.txt',
                  sample / 'quotes2/thoreau.txt'))

print(filecmp.cmp(sample / 'quotes1/thoreau.txt',
                  sample / 'quotes2/thoreau.txt', shallow=False))
```

True

True

Compares timestamp and size only

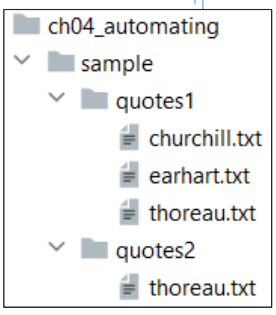
ch04_automating/04_filecmp.py

With shallow=False, cmp() will read file contents and compare them. The default value for cmp() is to perform a shallow comparison (shallow=True) which only looks at file stats (size and timestamp).

filecmp dircmp()

- dircmp() compares two directories according to a specified attribute

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




```
results = filecmp.dircmp(sample / 'quotes1', sample / 'quotes2')
results.report()
```

```
diff sample\quotes1 sample\quotes2
Only in sample\quotes1 : ['churchill.txt', 'earhart.txt']
Identical files : ['thoreau.txt']
```

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

```
Left only:   ['churchill.txt', 'earhart.txt']
```

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

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

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

```
Right only:  []
```

ch04_automating/04_filecmp.py

The dircmp() function can compare various items such as: same_files, common_files, common_directories (subdirectories), and common (both files and subdirectories).

zipfile

- zipfile provides a way to compress or decompress files

```
from pathlib import Path
from zipfile import ZipFile

zipfile = Path('quotes.zip')
filepath = Path(__file__).parent / 'sample/quotes1/thoreau.txt'

with ZipFile(zipfile, 'w') as fzip:
    fzip.write(filepath, arcname=filepath.name)

with ZipFile(zipfile) as fzip:
    fzip.extractall()
```

Opens the zipfile to be written to

Adds the specified file under the *arcname* name. In this case it will be added directly into the zipfile with no path.

Extracts all files into the current directory

ch04_automating/05_zipfile.py

The ZipFile() class allows for adding files into a zip file by using a *with* control much like a typical file would add lines into it within a similar control. The arcname parameter defines the name the file will have within the archive, including the path to it.

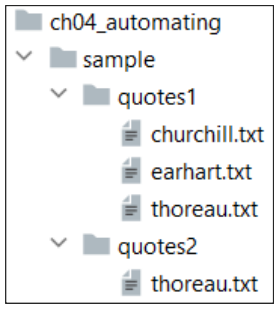
Your Turn! - Task 4-1

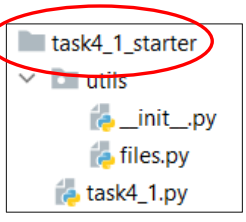
Copying Files

- Use shutil, filecmp, and pathlib to copy the files in the *sample* directory from quotes1 to quotes2

  Don't copy files that already exist in dir2

  – Display a list of the files copied





Create a shortcut import in __init__.py

Complete the copy_files function within files.py

Invoke the copy_files() function from task2_1.py

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

On your own: Create a similar function, called *delete_common()* that deletes the common files from the destination? Hint: Use Path().unlink().
Optional: Add a parameter to select whether to delete from the src or dst directory

Generating and Sending Emails

- Sending emails via Python can be useful for automation tasks
  - Modules used include: smtplib and email.message
- To test sending an email, we'll use a dummy server
  - Launch from a command line using one of these commands

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

Py3.11 or earlier

```
python -m aiosmtpd -n -l localhost:1025
```

Py3.12 or later

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

```
(venv) c:\student_files>python -m aiosmtpd -n -l localhost:1025
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: 7bit
MIME-Version: 1.0
Subject: Sample Message
From: joe@example.com
To: john@example.com
X-Peer: ('::1', 10966, 0, 0)

Hello!

This is the body for our email message sample.  It will be sent
from the example in ch04_automating.

------------ END MESSAGE ------------
```

```
pip install aiostmpd
```

Sending an email is usually done via an SMTP server. Python provides client APIs to connect to and send an email to an SMTP server using the smtplib module. It has APIs to construct the email using the email.message module. Since we do not wish to touch an on-premise, active SMTP server, we'll set one up that runs locally.

To run our own local SMTP daemon, run one of the two python commands shown above depending on the Python version you are using.

Sending the Email

- Use email.message to construct the email

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

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

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

Construct the email

Create the client capable of sending the email

Send the email to the server

ch04_automating/06_email.py

The above example sends an email to a dev SMTP server (running locally) by constructing the email message. The email.message module contains a handy class that requires the "Subject," "From," and "To" fields to be set. In our example, we added the email body by obtaining from a file. Emails can contain log entries or other related info as well.

The last 3 lines establish an SMTP client that points to our running server and sends the message to it. send_message() is a convenience wrapper that makes it easy to send emails.

The email created and sent here is received by our locally running SMTP server.

Sending Emails Through Your Own Server

- To send emails to your SMTP server, you will need:
  - Permission! Make sure you go through proper channels that grant you permission to work with your on-premise SMTP service

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

```
import smtplib
from email.message import EmailMessage

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

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

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

hostname and *port* are determined within your organization

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

Connecting to *your* internal SMTP server is often a bit more complex. Several pieces of information will be needed including hostname or address and port. It is also important to know if the server uses SSL or TLS as the connection technique is a bit different.

Even having this info can still present problems when connecting. It is generally best to see if there is any documentation within your organization showing how Python clients can connect to your SMTP server or if there are examples within your organization showing other programming languages connecting to your SMTP servers. Additionally, you'll generally need to be given permission to access your internal servers directly.

Sending Email Attachments

- Attachments (such as log files, PDFs, or other documents) can be added

```
from email.mime.multipart import MIMEMultipart
from email.mime.application import MIMEApplication

email_msg = MIMEMultipart()

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

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

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

Create the email type that supports attaching documents

Open the file to attach and read it

ch04_automating/07_attachments.py

Attach files to emails by using a MIMEMultipart() object. This object is the email, but it supports allowing files to be attached to it.

Working with PDFs

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

Tool	Description	Install
PyPDF (PyPDF2)	Reading/Modifying PDFs	pip install PyPDF2
Tabula-py	Reads PDFs into CSV files	pip install tabula-py
Pdfminer	Extracts text from PDFs	pip install pdfminer
Pdfkit	Create PDFs from Webpages	pip install pdfkit
Camelot	Extracts tables from PDFs	pip install Camelot-py[cv]
ReportLab	Creates PDFs from scratch	pip install reportlab
PyMuPDF	Extracts various content types from PDFs	pip install pymupdf
Pikepdf	PDF reader/writer	pip install pikepdf

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

There are numerous tools available in Python to work with PDFs. And, if that's not enough, for even more options exist. Check out: https://github.com/pmaupin/pdfrw#other-libraries.

A popular tool is called Pdfminer. It's useful for extracting content from a PDF.

Another tool, ReportLab, is open source and free to use. ReportLab must be pip installed as described in the table above. It generates nice PDFs. The ReportLab documentation was created into a PDF and can be viewed here: https://www.reportlab.com/docs/reportlab-userguide.pdf.

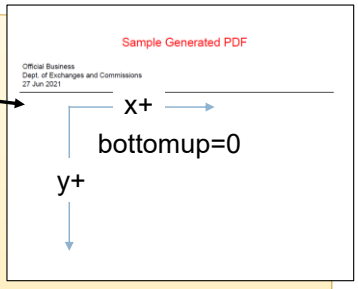
ReportLab

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

- The following can generate a PDF using vector graphics

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

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

canvas.setFont('Helvetica', 20)

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

Canvas, the main object, represents the PDF document



Renders text

Saves the PDF

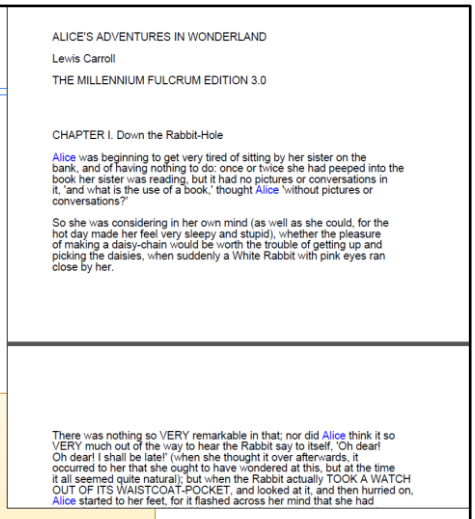
ch04_automating/08_pdfs.py

There are a couple of ways to use ReportLab. One way is to use the Canvas API and render graphics into the document. The Canvas API has a rich set of methods for doing just this. Another way to use ReportLab is to manage the items that will be placed into the document (see next slide).

The bottomup parameter in the Canvas constructor is used to define the location of the origin. With bottomup=1, the origin is at the lower left and bottomup=0 sets it at the upper left.

Rendering Text *(1 of 2)*

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


ALICE'S ADVENTURES IN WONDERLAND

Lewis Carroll

THE MILLENNIUM FULCRUM EDITION 3.0

CHAPTER I. Down the Rabbit-Hole

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

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

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

```
from pathlib import Path

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

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

doc_items = []
fontsize = 12
```

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

ch04_automating/09_pdfs.py

The example renders the file, alice.txt, into a PDF. In addition, it adds extra space for new paragraphs and colors every occurrence of the word "Alice" blue (just for fun).

Since this PDF is text-based, using a SimpleDocTemplate object is a good starting place as this sets up a template where every page is the same. There is no need to manage or control each individual page when using SimpleDocTemplate.

Rendering Text *(2 of 2)*

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

doc.build(doc_items)
```

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

Build the PDF

ch04_automating/09_pdfs.py

Here, we iterated line-by-line over alice.txt. If a line was empty, we added a Spacer() and if it wasn't, then we checked to see if "Alice" appeared within the line. If it did, we split the line and colored her name blue while the rest of the line was colored black.

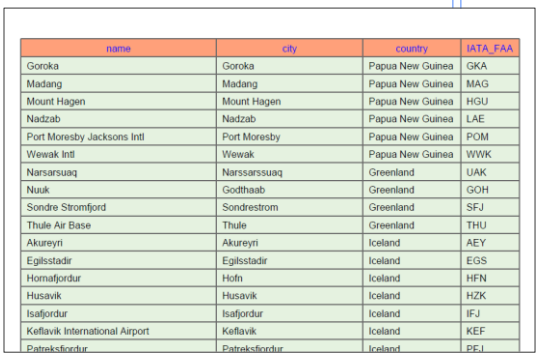
Note: the two f-strings above are wrapped on the slide for display only.

The Paragraph object wraps the text to be inserted and adds it to a list of Paragraphs that ultimately will be used to build the PDF doc. Numerous options exist via a ParagraphStyle object (not used here) that allows for modifying the look and feel of each text item added.

The line (f-strings above) are defining XML elements with HTML attributes. Be sure strings are formatted as XML.

Rendering Tables *(1 of 2)*

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



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

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

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

doc_items = []

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

airports = []
```

Continued…

ch04_automating/10_pdfs.py

Tabular data can be rendered using the ReportLab Table object. The look and feel can be controlled through the TableStyle object (see next slide). TableStyle arguments are shown below.

Rendering Tables *(2 of 2)*

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

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

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

These affect the header row

ch04_automating/10_pdfs.py

In the example shown, we read from "airports.dat" into a list of lists. The resulting data (airports) is passed to the ReportLab Table object. Before finishing, we defined the look and feel of the table using a TableStyle object. The syntax for TableStyle is to provide a list of tuples. In each tuple, a keyword is provided followed by a start cell, an end cell, and one or two parameters depending on the style to be set.

NOTE: cells are written in **(column, row)** format, NOT (row, column). So, (1, 3) refers to the fourth row, second column (both rows and columns begin at 0).

PyMuPDF

- PyMuPDF has strong capabilities for extracting text, tables, data, graphics/images from PDFs (and other file types)

```
import fitz

doc = fitz.open('lorem_ipsum.pdf')
images = []
with open('pdf_out.txt', 'w') as txt_out:
    for page in doc:
        try:
            txt_out.write(page.get_text())
            images.extend(page.get_images())
        except UnicodeEncodeError:
            pass
doc.close()
```

ch04_automating/11_pymupdf.py

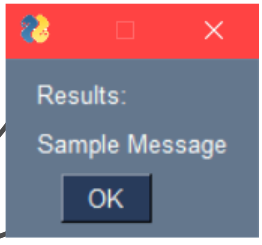
PyMuPDF has a GPL-3.0 license. For more on PyMuPDF, visit its documentation page at https://pymupdf.readthedocs.io/en/latest/. The PyMuPDF features are extensive and this example is only meant to preview what PyMuPDF looks like to use.

PySimpleGUI

- A simple, but easy-to-use GUI library that adds input dialogs and popup message boxes is called PySimpleGUI

```
import PySimpleGUI as sg

layout = [
            [sg.Text('Value')],
            [sg.InputText()],
            [sg.Submit(), sg.Cancel()]
         ]

window = sg.Window('Query', layout)
event, results = window.read()
window.close()

print(event, results[0])

sg.popup('Results:', results[0])
```

results will contain a list of inputs from the dialog. *event* is the button selected



ch04_automating/12_simple_gui.py

PySimpleGUI is designed to be deliberately easy to design GUI dialogs with. sg.Window() accepts a title string and a layout configuration. The user can type in a value and return a result including the button name that was selected. An sg.popup() can display a message and title.

Our student files defines a module in resources that can easily use these two dialogs as follows:

```
import resources.gui as gui

result = gui.simple_input()

gui.simple_popup(result)
```

A dialog appears asking the user for input

A popup shows a result

Task 4-2

- Generate a PDF, send it as an email attachment
- Work from *starter/task2_2/task2_2.py* and complete the remaining two methods

- Complete and then call the remaining two functions at the bottom of the source file

COMPLETED!

get_data()

generate_pdf()

send_email()

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

psutil

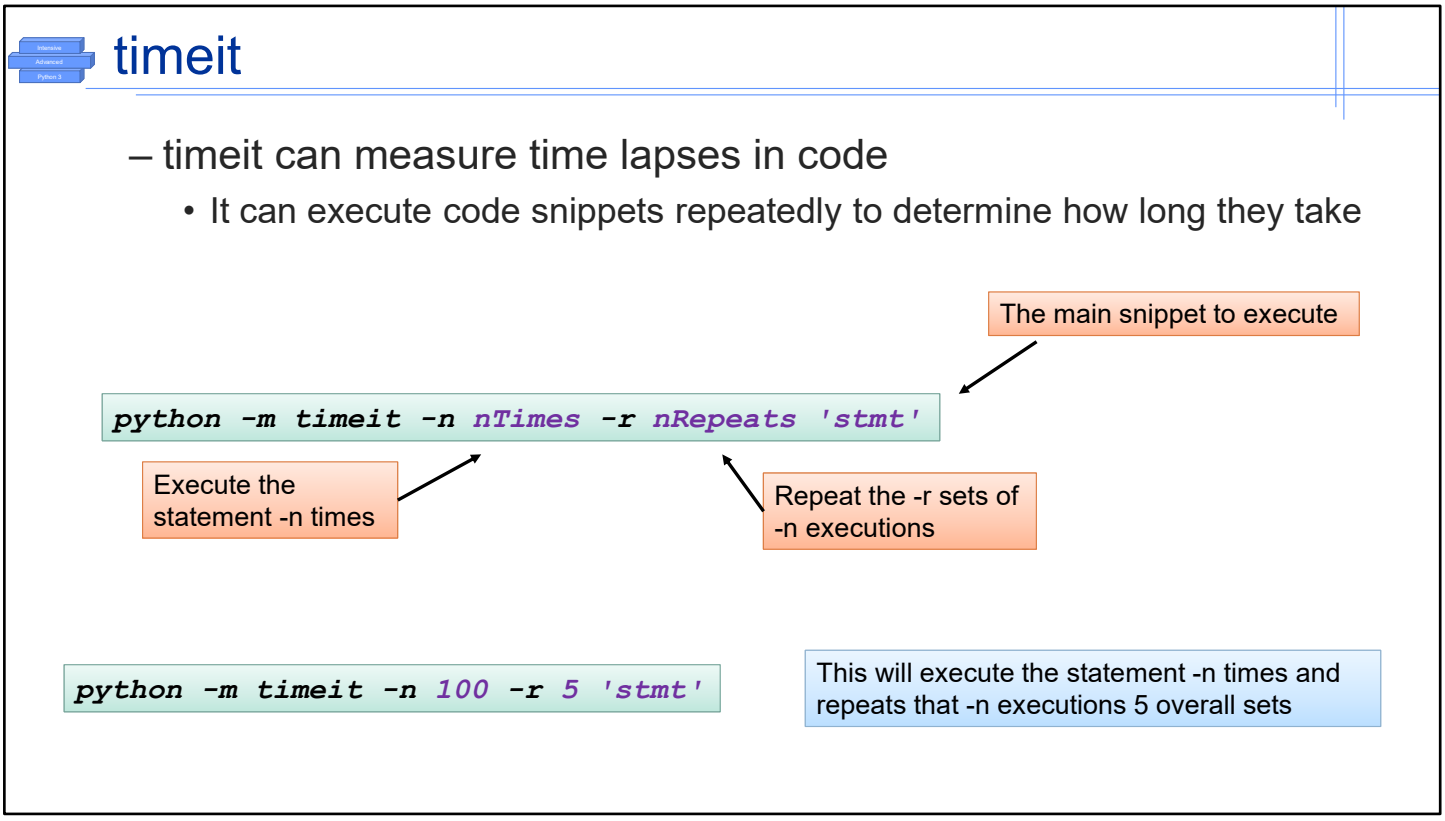
- *psutil* can provide cross-platform information about memory and cpu utilization

```
pip install psutil
```

```
import psutil

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

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

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

ch04_automating/13_psutil.py

psutil is a third-party utility that can be easily installed. *psutil* can provide all kinds of information about system resources and devices.



For the command line version, -n can control how many iterations (loops) of the statement to perform. -s provides an initialization (setup) string.

-n defines how many times to execute the statement
-r defines how many times to repeat the n loops (runs the timer again), default is 5.

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

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

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

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

This builds a str object by appending strings together 200000 times

ch04_automating/14_using_timeit.py

There are two algorithms that build a very large string. The first example builds a StringIO() buffer. The second technique repeatedly appends to a string. Keep in mind that strings are immutable and therefore, a new string must be created in memory each time.

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

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

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

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

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

timeit results

Difference is memory used: 6688768

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

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

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

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

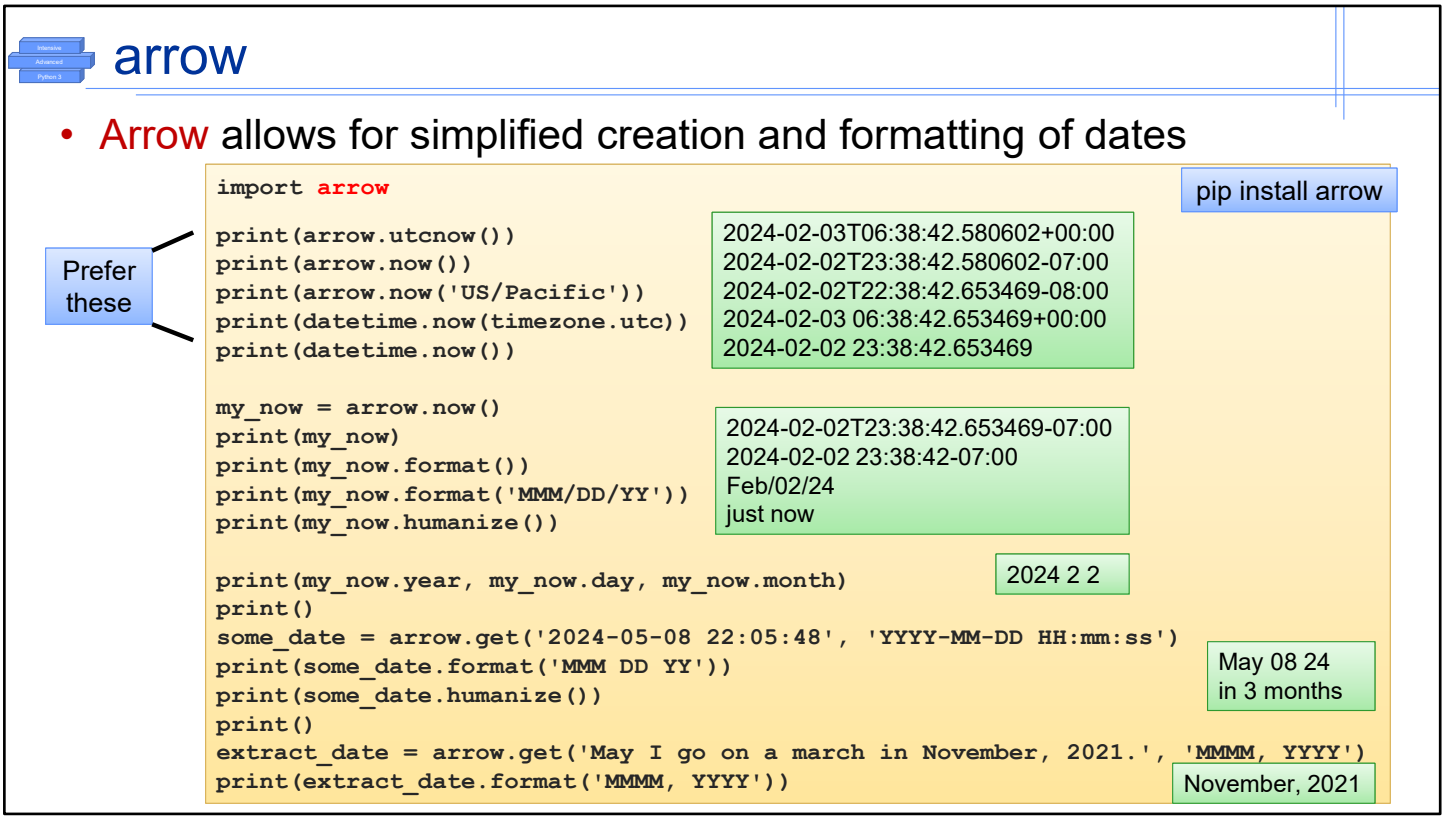
timeit results

Difference is memory used: 26714112

ch04_automating/14_using_timeit.py

The output above shows timeit and psutil results used to measure calls to sample_func1() and sample_func2() from the previous slide. The timing values shown are only samples as timing results will be machine-relative.

The globals=globals() parameter just makes it so the timeit module can "see" the function coming from the main module. As can be seen, both the length of time AND the memory utilized is considerably less for StringIO objects.



ch04_automating/15_arrow.py

Arrow is named after "arrow in time" and is based on the JavaScript moment.js library.

In the example above, various examples of using arrow are shown. It can be used to format between timezones, display human readable relative values, and even search strings for dates hidden within. Prefer the use of arrow.utcnow() or datetime.now(tz=) or the localized version arrow.now() as they are all "aware" while datetime.now() without a timezone is a naïve date time. Aware dates include timezone information. Naïve dates do not.

fire

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

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

        return self.results

fire.Fire(FileReader)
```

pip install fire

Fire processes command-line arguments calling functions or classes and passing values into them

Class method is automatically called

python 16_fire.py --filename somefile.txt --count 2 read

Args passed to __init__ should use -- notation

ch04_automating/16_fire.py

Fire is a command line argument processing tool that makes it easy to pass arguments into functions or classes. It supports numerous variations. For more on fire, visit: https://google.github.io/python-fire/.

In this example, fire instantiates a FileReader object passing command-line provided arguments into the __init__(). The read() method is called, as per the command-line switch provided, and values are used within the read() method. The return value from read() is passed back to the stdout of the command line.

Chapter 4 Summary

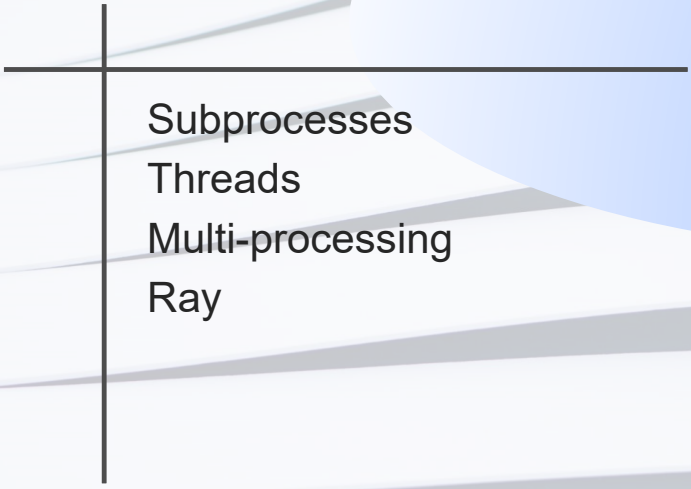
- Numerous useful modules in the standard library can aid with automating environments
  - As you advance your Python skills, these modules will become more and more familiar to you
    - pathlib, fnmatch, shutil, filecmp assist with searching, copying, and comparing files
    - email, smtplib simplify sending email notifications
    - timeit, psutil helps measure performance of code

- Countless third-party modules can assist as well
  - Modules such as ReportLab help process PDFs, arrow for dates
  - PySimpleGUI provides quick and easy GUI controls to improve the feel of scripts while fire provides an OO-based command-line interface

Chapter 5 Subprocesses, Threads and Multiprocessing

Working with Multiple Paths of Execution

Chapter 5 - Overview



Subprocesses
Threads
Multi-processing
Ray

Processes and Subprocesses

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

- *The* subprocess module allows scripts to launch *other* programs
  - The script can perform some *limited communication* with that other process
  - Two techniques exist for launching subprocesses 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)
```

ch05_multitask/01_subprocess_run.py

In Python 3.5, subprocess.run() was added which effectively performs the same functionality as all 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([sys.executable, '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

ch05_multitask/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)

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

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

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

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

```
result = subprocess.run(args=[sys.executable, '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

ch05_multitask/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)

```
result = None
try:
    result = subprocess.run(args=[sys.executable, '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}')
```

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

ch05_multitask/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

```
result = None
try:
    result = subprocess.run(args=[sys.executable, '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}')
```

sample2.py always times out

ch05_multitask/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.

Using Popen() *(1 of 2)*

```
proc = subprocess.Popen(['python', '-V'], text=True,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE)
stdout, stderr = proc.communicate()
print(f'Your python version: {stdout}')
```

Python 3.10.1

```
with subprocess.Popen(['pip', 'list', '--format=json'],
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        text=True) as proc:
    result = proc.stdout.readlines()

print(f'proc stdout closed?: {proc.stdout.closed}')
data = json.loads(result[0])
for package in data:
    print(f'{package["name"]}:
            {package["version"]}')
```

True

pip: 22.0.3
pipenv: 2022.1.8
prettytable: 3.2.0
setuptools: 58.1.0
…

ch05_multitask/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).

Using Popen() *(2 of 2)*

```
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
…
```

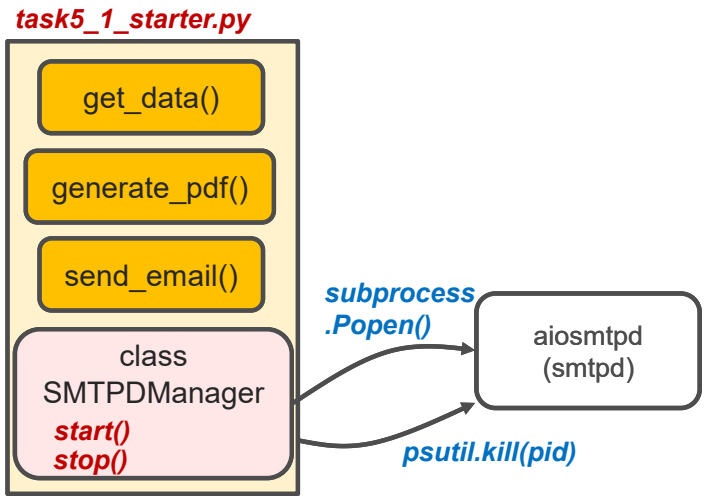
ch05_multitask/06_popen.py

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 5-1

Subprocesses

- Revisit Task 4-2: generating a PDF, sending it as an email
- Launch the *aiosmtpd* server from within our main script by creating a SMTPDManager class

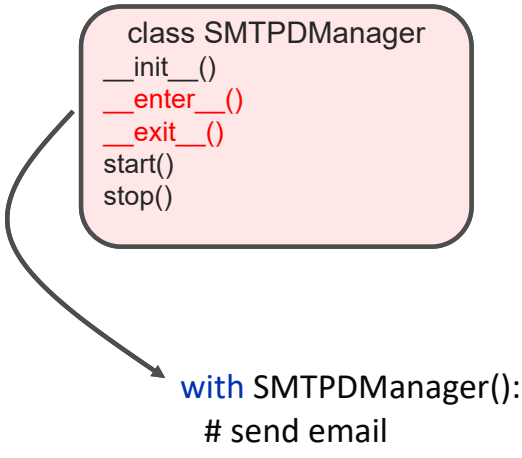


Your Turn! - Task 5-2

Subprocesses and Context Managers

- Modify Task 5-1 so that the SMTPDManager class works using a *with* control to start and stop the server

- Do this by adding an __enter__() and __exit()__ method



Threads

- Threads provide multiple paths of execution within a program
  - This means multiple tasks can often execute faster than single threads*
- The **threading** module provides a *Thread* class that can implement a thread
  - Thread class methods include:

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:
        logging.error(f'Error: {err}')

    return data

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

    def run(self):
        logging.info(f'Results: {load_data(self.url)}')


m = DataFetcher('https://eonet.gsfc.nasa.gov/api/v3/events', '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

ch05_multitask/07_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://eonet.gsfc.nasa.gov/api/v3/events '

def load_data(url):
    data = None
    try:
        r = requests.get(url)
        data = r.json()
        logging.info(data)
    except requests.exceptions.RequestException as err:
        logging.error('Error: {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

ch05_multitask/08_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

ch05_multitask/09_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 Locks

```
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

ch05_multitask/09_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.

Locks and Context Managers

```
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'

ch05_multitask/10_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.

The Global Interpreter Lock (GIL)

- A mutual exclusion lock is held by the Python Interpreter
  - Only the current thread holding the GIL may operate
  - *Threads waiting on the GIL will block*
  - The interpreter releases the lock approx. every 100 instructions or when blocking operations occur
    - sys.setswitchinterval(*seconds*) can change this value
- Applications that feature *more I/O operations* generally benefit more from multiple threads
- Apps that are *purely computational (think of for-loops)* may lose performance due to the switching required 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.

So, Do Threads Work?

```
num_worker_threads = 20
tasks = ['https://requests.readthedocs.io/en/latest/',
         'https://automatetheboringstuff.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

ch05_multitask/11_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.

The *multiprocessing* Module

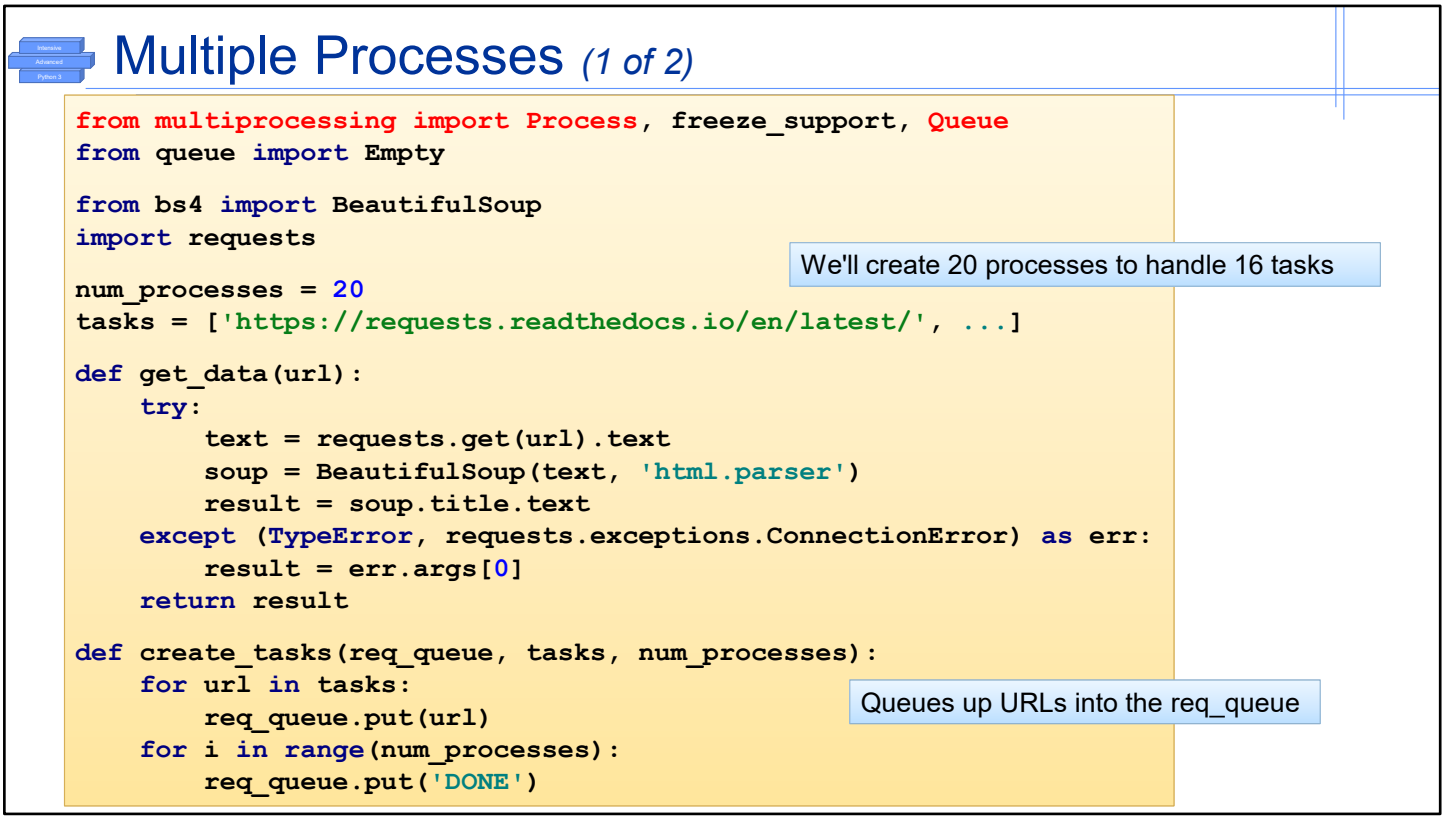
```
import multiprocessing

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!')
```

Each ***Process()*** instantiation is a separate OS-level process

```
Square result: 16
Cube result: 1000
Done!
```

- The *multiprocessing* module overcomes the limitations of the GIL
  - The multiprocessing module creates full processes rather than worker threads
  - It uses the Process() class
  - Processes create their own Python virtual machines
  - Each process can run in a separate CPU core

ch05_multitask/12_multi_processing.py



ch05_multitask/13_multi_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.

Multiple Processes *(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

```
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)
```

Spawn 20 processes

work() is the task each process will perform

Create the task and results queues

ch05_multitask/13_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.

Libraries to the Rescue

- Modern Python scripts have pushed CPU processing to the limit
  - To make easier use of multiprocessing, third-party frameworks exist such as Ray, Dask, and others

```
import ray

@ray.remote
def square(val: int):
    return f'Square result: {val * val}, pid: {os.getpid()}'

ray.init()

obj_refs = [square.remote(i) for i in range(5, 10)]
results = ray.get(obj_refs)
print(*results, sep='\n')
ray.timeline(filename='timeline.json')

ray.shutdown()
```

Marks any function as one that can be scheduled into a cluster for execution asynchronously

Connect to or create a Ray cluster

Immediately returns an object ref (a future) and creates a task to be executed on a worker process

Retrieve the results

```
2024-02-04 00:03:41,270 INFO worker.py:1724 -- Started a local Ray instance.
Square result: 25, pid: 1792
Square result: 36, pid: 8852
Square result: 49, pid: 8232
Square result: 64, pid: 11764
Square result: 81, pid: 14940
```

ch05_multitask/14_using_ray.py

Ray allows any typical function/function call to become a job in a cluster. func.remote(args) on any decorated function schedules that function to be executed, but it is not executed synchronously. In other words, the function is eventually executed, but for now an object reference is returned (a future result, if you prefer).

Calling the ray.get() method allows for retrieving those results which means get() blocks until the results are finished.

Task 5-3 Multiprocessing with Ray

Working with Ray

- In this task, you will take a standard function and execute it in a multiprocessing environment using the Ray library

```
def get_data(url: str) -> str:
    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
```

Execute this function for many URLs using Ray

  - Work from *task5_3_starter.py*

Chapter 5 Summary

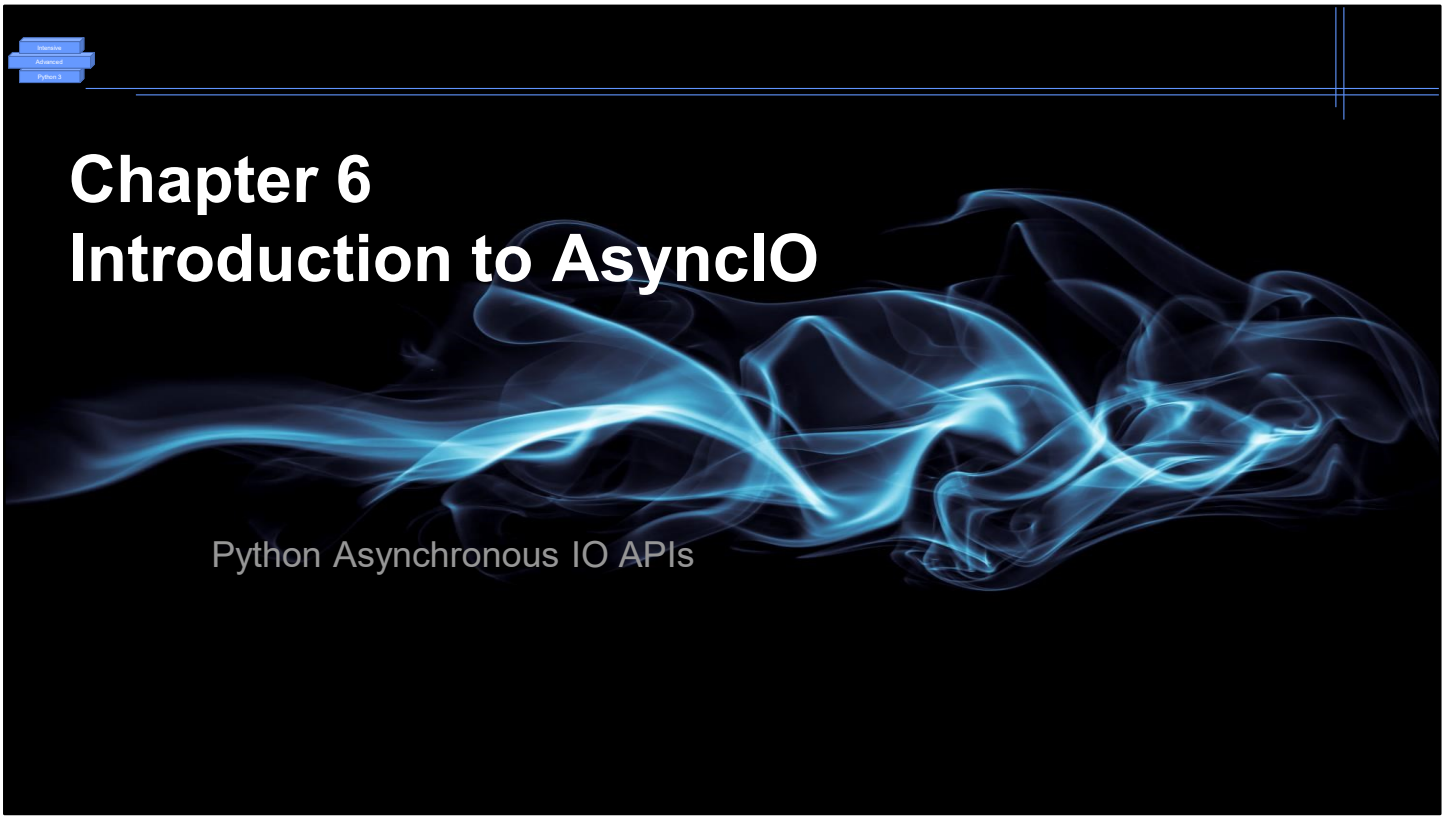
– Use the threading module and the *Thread* class to create multi-threaded apps

  • 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 shared resources used by all processes

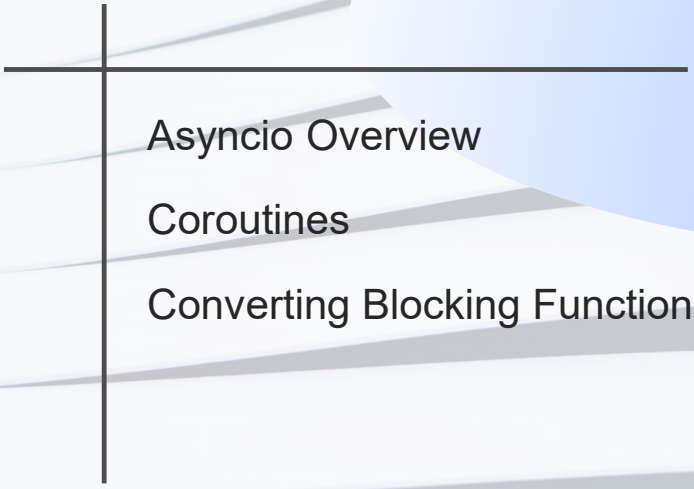
  • For this reason, newer frameworks such as Ray can hide these complexities and expose an easy to work with interface



Chapter 6
Introduction to AsyncIO

Python Asynchronous IO APIs

Chapter 6 - Overview



Asyncio Overview

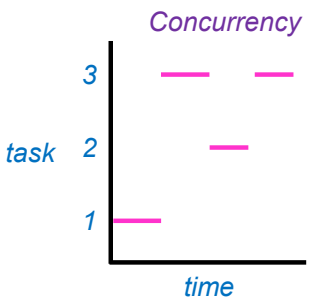
Coroutines

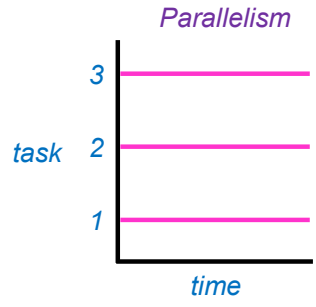
Converting Blocking Functions

Concurrency vs Parallelism

- Concurrency involves running multiple tasks in a single CPU core, switching tasks as needed
- Parallelism - can schedule multiple tasks into multiple CPU cores to run simultaneously
- Python provides modules to help improve application performance
  - threading - single-process, concurrency model, GIL
  - multiprocessing - supports parallelism OS processes
  - asyncio - supports single-threaded concurrency by switching tasks executed

There are *high-level* asyncio APIs for general use and *low-level* asyncio API for those building libraries to gain better control over the event loop



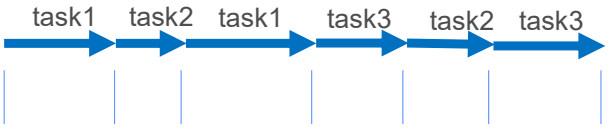


Our discussions will emphasize the high-level APIs. Low-level APIs involve the manipulation of the event loop, retrieving and canceling futures, creating asynchronous subprocesses, and more.

The high-level APIs involve the scheduling of coroutines using the methods mentioned in these notes.

Asynchronous Tasks *(Single Thread)*

- Asyncio executes multiple tasks in a *single thread* using "time-slicing" allowing tasks to run *a short time*



  - Tasks execute until they *yield* to another thread
    - Problematic if tasks are greedy
    - Works best with lots of I/O
    - Great for anything using network calls

```
import asyncio
import time


async def my_func(val: int):
    print(f'entering my_func {val}')
    await asyncio.sleep(1)
    print(f'exiting my_func {val}')


async def main():
    await asyncio.gather(my_func(1),
                         my_func(2))

start = time.time()
asyncio.run(main())
print(f'Elapsed: {time.time() - start}')
```

async def defines a "coroutine" or any function that will *yield* control

Yields control

Schedules coroutines into the event loop

Runs the event loop, passes the coroutine to it

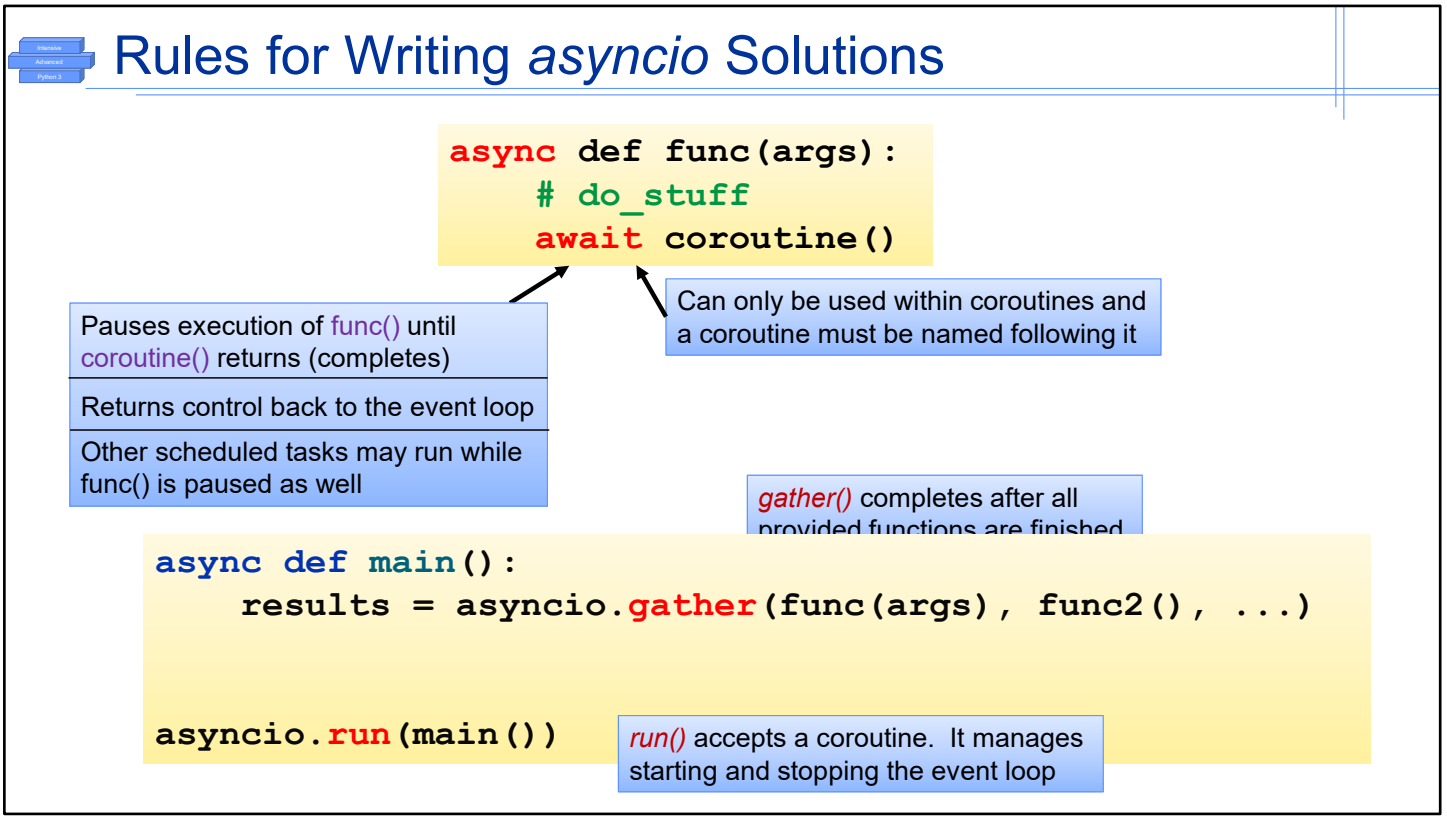
```
entering my_func 1
entering my_func 2
exiting my_func 1
exiting my_func 2
Elapsed: 1.0113542079925537
```

ch06_asyncio/01_coroutine.py

Asyncio assumes a single-thread is executing tasks but only allows tasks to run for a period. In some cases, this approach can be less complex than a multi-threaded implementation because we aren't as susceptible to thread synchronization problems. Remember, there is only one thread executing.

When compared to the single-threaded version, asynchronous solutions can be better when there is a lot of I/O blocking. Blocking occurs when the program is waiting for input, perhaps from outside the Python environment. In solutions where there is a lot of blocking, this approach can provide a performance improvement. Also, if there are a lot of tasks, it is possible that some tasks will likely involve some IO blocking.

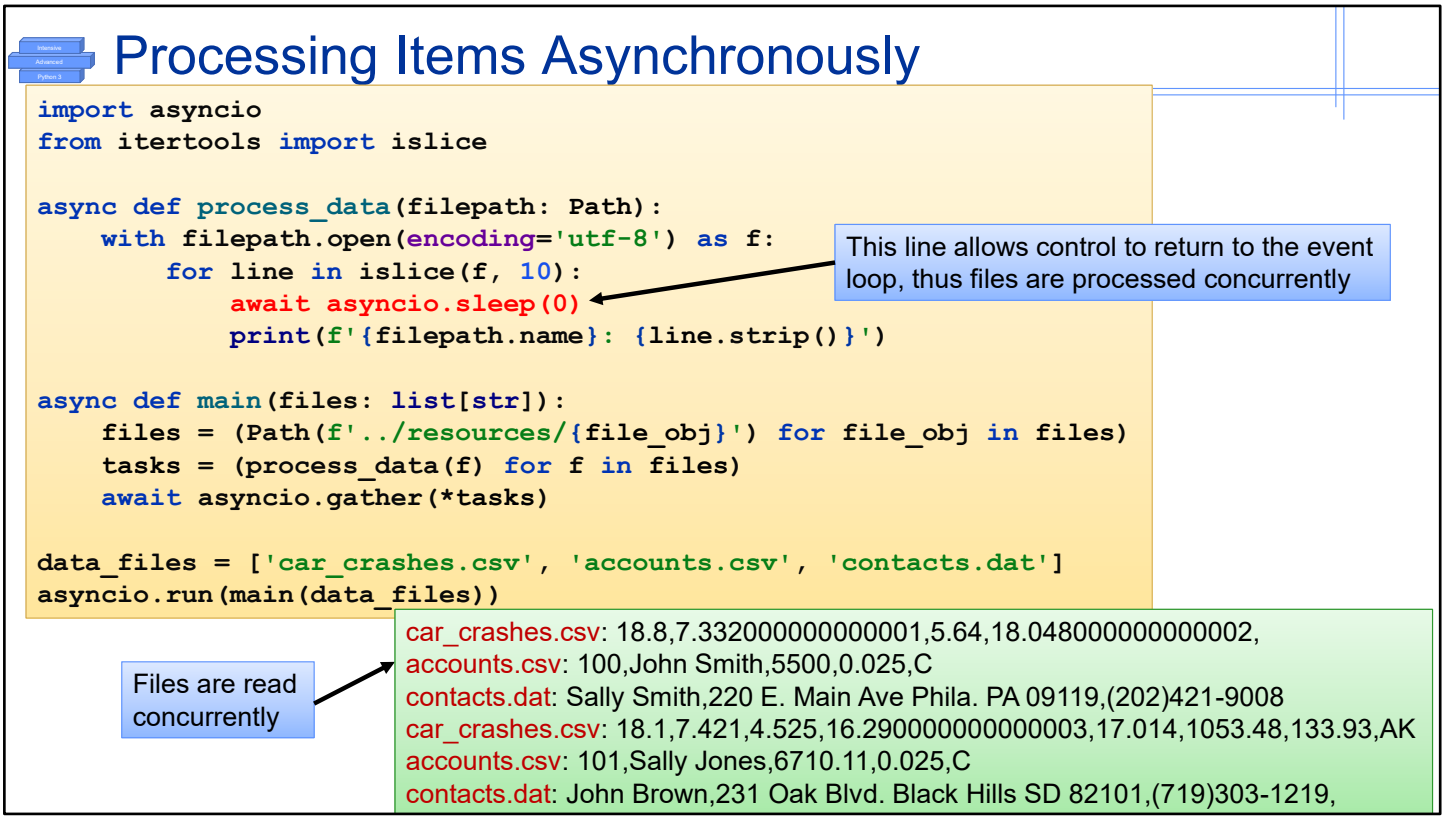
Good reference: https://github.com/timofurrer/awesome-asyncio



An event loop manages the tasks that are scheduled. It decides which tasks are executed when control is returned to it. Tasks must be coroutines--functions that are marked with the *async* definition or have the *await* keyword used within them.

gather() works with multiple coroutines and schedules them. It returns after all supplied coroutines are finished. Gather() treats all coroutines as a group, so that if one should fail, gather() finishes without executing the other tasks.

A closely related function, wait(), works much like gather but treats each task passed into it as independent tasks, so if one fails, other tasks may still return eventually.



ch06_asyncio/02_await.py

The example takes three files and reads 10 lines from each file. This happens by putting three process_data() calls into the event loop. To read a file, however, in the traditional way, would cause the function never the yield back to the event loop. Each file would be processed entirely and sequentially.

However, in this example, await asyncio.sleep(0) causes control to be returned to the event loop. The output then renders data from the files concurrently as a result.

Third-party Tools and Language Modifications

```
import aiofiles
from asyncstdlib import islice

async def process_data(filepath: Path):
    async with aiofiles.open(filepath) as f:
        async for line in islice(f, 10):
            print(f'{filepath.name}: {line.strip()}')

async def main(files: list[str]):
    files = (Path(f'../resources/{file_obj}') for file_obj in files)
    tasks = (process_data(f) for f in files)
    await asyncio.gather(*tasks)


data_files = ['car_crashes.csv', 'accounts.csv', 'contacts.dat']
asyncio.run(main(data_files))
```

aiofiles is a third-party library that provides asyncio-based methods that yield to the event loop

This is a third-party version of islice() that allows asyncio objects to be iterated over

Async with and async for are Python language modifications that support yielding control to the event loop

car_crashes.csv: 18.8,7.332000000000001,5.64,18.048000000000002,
accounts.csv: 100,John Smith,5500,0.025,C
contacts.dat: Sally Smith,220 E. Main Ave Phila. PA 09119,(202)421-9008
…

ch06_asyncio/03_aiofiles.py

Async with and Async For

- Python provides an *async for* and *async with* to support asynchronous operations

```
import aiofiles
from asyncstdlib import islice

async def process_data(filepath: Path):
    async with aiofiles.open(filepath) as f:
        async for line in islice(f, 10):
            print(f'{filepath.name}: {line.strip()}')

async def main(files: list[str]):
    async with asyncio.TaskGroup() as tg:
        for f in files:
            tg.create_task(process_data(Path(f'../resources/{f}')))

data_files = ['car_crashes.csv', 'accounts.csv', 'contacts.dat']
asyncio.run(main(data_files))
```

TaskGroups automatically schedule the task in the event loop at the end of the with-control

Adds a task to the TaskGroup (Py 3.11+)

ch06_asyncio/04_async_with.py

Python added an ***async with*** statement which yields to the event loop at the enter and exit of the with control. It does this by implementing the __aenter__() and __aexit__() methods instead. Similarly, the ***async for*** keywords implement the __aiter__() and __anext__() methods that yield back control to the event loop. This means it can yield control even while iterating.

TaskGroup was added in Python 3.11. It allows for grouping tasks together (during the with-control). These tasks are automatically added to the event loop at the end of the with-control.

Converting Blocking Operations

- Classic functions that block are not suitable for *asyncio* as they will not work properly

requests.get(url)

This blocks without returning control to the event loop and thus would not work with asyncio

- Use *asyncio.to_thread()* to cause the function to run in a separate thread that can then be scheduled to run properly

```
import asyncio

def nonblocking_func(val: int):
    for val1 in range(1, 300):
        if val1 % val == 0:
            print(f'{val1} / {val}')

async def main():
    await asyncio.gather(
          asyncio.to_thread(nonblocking_func, 3),
          asyncio.to_thread(nonblocking_func, 5),
          asyncio.to_thread(nonblocking_func, 7))
```

This is a normal function that does not yield

```
resp = await asyncio.to_thread(function, args)
```

ch06_asyncio/05_to_thread.py

In this example, nonblocking_func() never relinquishes control. Yet, when run, you will clearly see that it outputs values from each of the 3 runs in overlapping fashion such that the calls are clearly occurring concurrently. This happens because the to_thread() call converts the function to run in its own thread and that thread is awaited into the event loop.

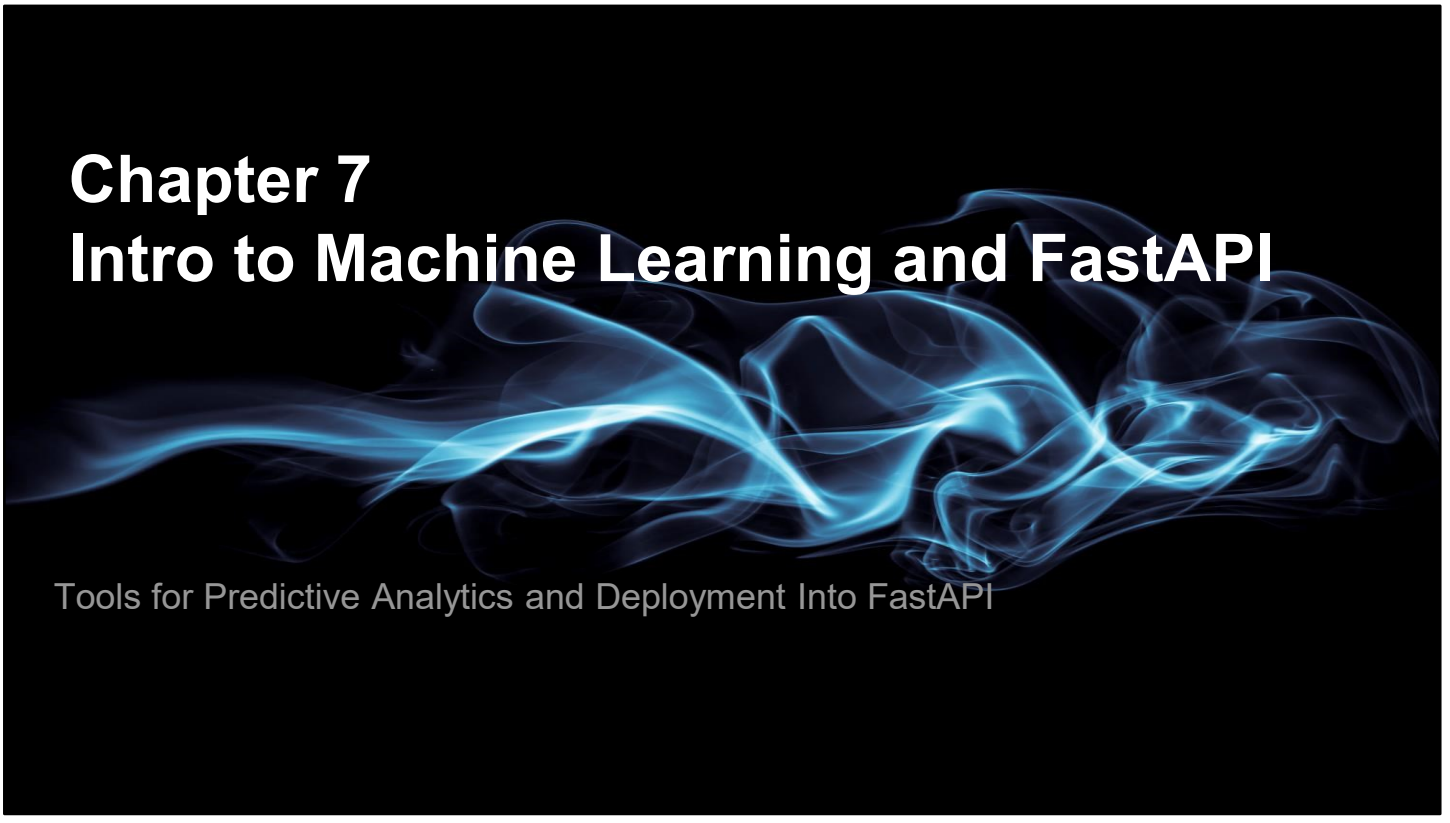
Your Turn! - Task 6-1

Using *requests* Asynchronously

- Convert the multi-threaded URL request example from the previous chapter into an asynchronous solution
- Work from task6_1_starter.py

Chapter 6 Summary

- The *asyncio* module has evolved heavily since its introduction in Python 3.4
- Asyncio represents a growing movement toward the use of asynchronous solutions and APIs
  - These APIs are in lieu of trickier thread-based models
  - Asyncio is both complex and can be somewhat steep to learn and simple when using the higher-level APIs
- The list of third-party tools incorporating asyncio is growing and continues to make it more and more accessible to use



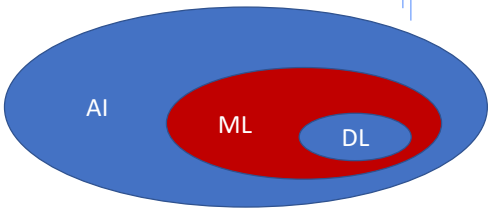
Chapter 7
Intro to Machine Learning and FastAPI

Tools for Predictive Analytics and Deployment Into FastAPI

Chapter 7 - Overview

Machine Learning Basics
Creating a Model
Introducing REST and FastAPI
Deploying a Model into FastAPI

Machine Learning



- *Machine learning* uses computers to learn about new data without being explicitly programmed
  - It does this by:
    - Fitting data into a function (supervised learning), or
    - Figuring out how to group data points together (unsupervised learning)

- *Scikit-learn* is a machine learning library for Python supporting various classification, regression, and clustering algorithms
  - Built upon SciPy (Pandas, NumPy, Matplotlib)
  - Also provides tools for:
    - Creating estimators, imputing (to fill in missing values)
    - Preprocessing, normalization, model-tuning, accuracy measurements

pip install scikit-learn

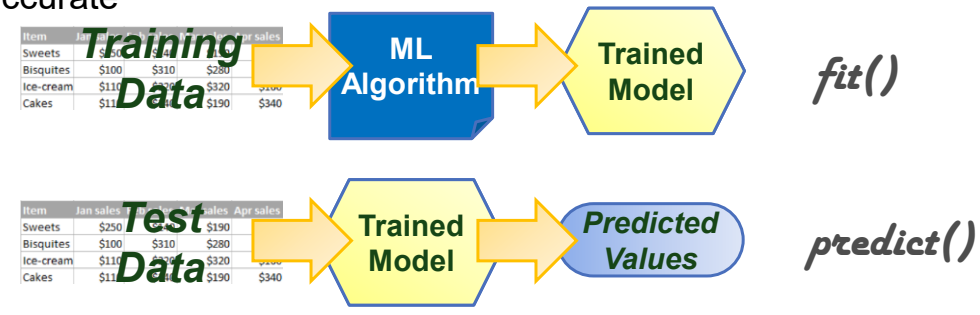
AI= artificial intelligence, ML = machine learning, DL = deep learning.

Machine learning is one of the major subfields of artificial intelligence which includes deep learning and neural networks. Other branches of AI include: evolutionary computation, probabilistic methods such as Bayesian networks, fuzzy systems, chaos theory, and computational creativity.

The term "machine learning" was coined by computer gaming pioneer Arthur Samuel in 1958.

Create Models

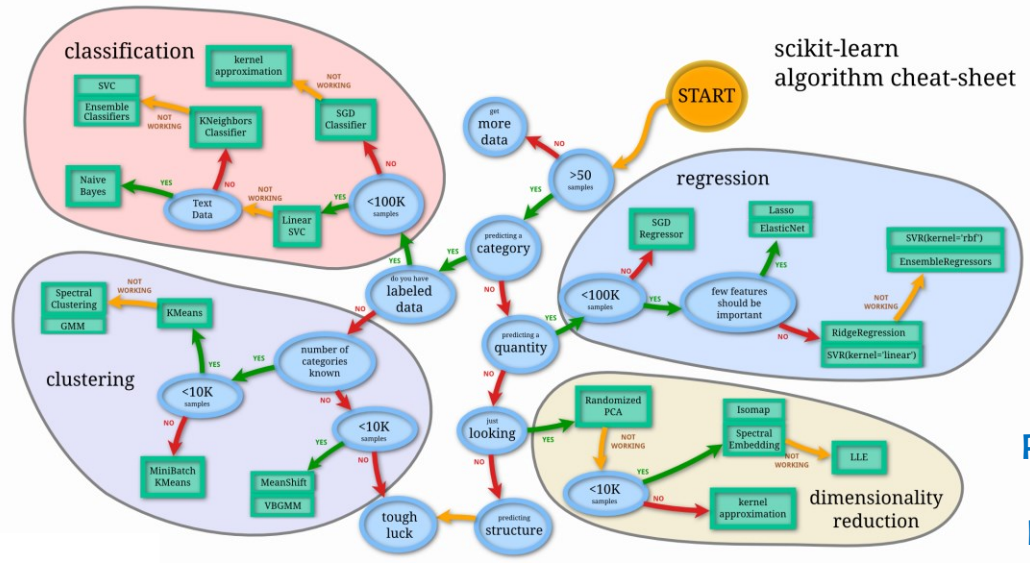
- Most ML involves creating a model by passing a dataset through the algorithm repeatedly
  - Parameters of the model, called hyperparameters are changed to make the model more accurate



  - Numerous algorithms exist and different algorithms perform better depending on the types of data used

Supervised learning algorithms require labeled data (or data containing known output values or results). The labels, along with one or more features, are used to generate the model. Once a model is created (fitted using our data) it can be applied to some test data to determine if the model was appropriate. If the model is deemed a suitable fit, we can then use it to predict values.

Determining an Estimator to Use



Neural Networks
Random Forest
Logistic Regression
Naïve Bayes
Decision Trees
Kernel SVM
SVM
Gradient & XG Boosting
Classifications

Single and Multiple
Linear Regression
Random Forest
Gradient & XG Boosting
Decision Trees
Lasso & Ridge Regression
Regressions

The chart shown above is developed and provided by the Scikit-learn development team and can be viewed online at:
https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html.

Numerous estimators exist within Scikit-learn for both supervised learning and unsupervised learning. Once it is determined the type of dataset you are dealing with, you can begin selecting one or more estimators to use with it. For example, labeled data for classification algorithms must be categorical while regressions have continuous-valued labels.

Logistic Regression

- A logistic regression is a supervised learning algorithm that produces a *discrete output*
- Logistic Regressions are the linear regressions of classifications
- The output of the logistic function is a ***probability*** that an event occurred
- For a logistic regression, the equation that will be used is called the Sigmoid or logistic function

Uses include:
- Fraud detection
- Spam detection
- Medical diagnosis
  (e.g., cancer or not)
- Customer behavior
  (e.g., will they make the purchase?)

Logistic regressions often work best with large datasets. A good rule of thumb is that your dataset should have approximately 50 samples per feature.

Predicting Survival in the Titanic Dataset

- We'll use a logistic regression to determine likelihood of survival

```
import seaborn as sns

titanic = sns.load_dataset('titanic')
titanic.replace({'who': {'man': 0, 'woman': 1, 'child': 2}},
                inplace=True)
print('Columns with missing values:')
print(titanic.isnull().sum())

print(f'Total records: {len(titanic)}') 891
print(titanic.head())
```

```
survived       0
pclass         0
sex            0
age          177
sibsp          0
parch          0
fare           0
embarked       2
class          0
who            0
adult_male     0
deck         688
embark_town    2
alive          0
alone          0
```

Should we include the age or deck columns?

? ?

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

Too many missing values

ch07_ml/01_log_reg_titanic.py

We begin by loading the dataset. Seaborn, a data visualization library, happens to have an ability to load it easily for us into a Pandas DataFrame. Using the Pandas head() method, the first five rows are displayed. Many of the columns contain redundant information. Also, there are some missing values encountered in several columns. The deck column has far too many missing values to be of use. This leaves just a few useful columns for our model.

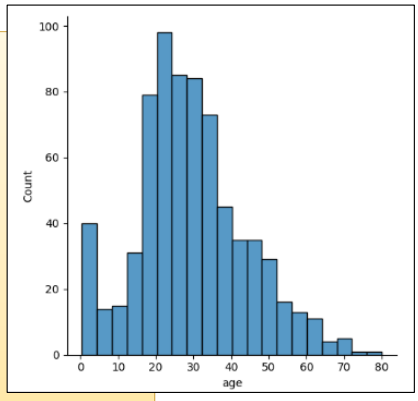
Imputing Values: The Age Column

- While Pandas can impute values for us, so can Scikit-learn

```
from sklearn.impute import SimpleImputer

print('Age mean: ', titanic.age.mean())
print('Age median: ', titanic.age.median())
sns.displot(titanic.age[~pd.isnull(titanic.age)])
plt.show()

imputer = SimpleImputer(strategy='median')
titanic['age'] =
imputer.fit_transform(titanic.age.to_numpy().reshape(-1, 1))

print(f'Total records: {len(titanic)}')
print(titanic.head())
```

29.7

28.0



Converts the data from this format: [22 38 26]
to this format: [[22][38][26]], suitable for Scikit-learn

Impute Strategies
constant - fill with a non-legal value or other desired value
most_frequent (mode) - use with classifications, uses most common
mean - when dealing with normal distributions
median - when distributions are skewed

ch07_ml/01_log_reg_titanic.py

Before imputing values into the age column, we did a distribution plot check (we can also do a histplot the same way). The curve skews low. For this reason, we chose the median value to impute. We ran a fit_transform() on the age column.

Creating, Predicting, Fitting the Model

- With no missing values remaining, the model can be created

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

features_matrix = titanic[['pclass', 'sibsp', 'parch', 'fare', 'who', 'age']]
response = titanic['survived']
X_train, X_test, y_train, y_test =
                train_test_split(features_matrix, response, random_state=42)

print('Train-to-test sizes: ', len(X_train), len(X_test))
lgreg = LogisticRegression()

lgreg.fit(X_train, y_train)
y_pred = lgreg.predict(X_test)
print(lgreg.score(X_test, y_test))
```

```
Train-to-test sizes:  668 223
```

```
0.7937219730941704
```

ch07_ml/01_log_reg_titanic.py

Six features are selected that aren't redundant or don't have missing values. These, along with the response, are broken into four parts: A training set that consists of approximately 75% of the total records and a test set that consists of the remaining 25%. This occurs for both the features and labels.

A fit() is run using the logistic regression. The predict() method is used against the test values (these weren't used to create the model). The score shown is the accuracy of the model. Our model predicts with about 79.3% accuracy.

Recursive Feature Engineering (RFE)

- It's possible to determine the most important features to the model by performing an RFE

```
from sklearn.feature_selection import RFE

rfe = RFE(estimator=lgreg, n_features_to_select=4)
rfe = rfe.fit(X_train, y_train)
print(f'RFE features: {list(X_train.columns[rfe.support_])}')
```

Determine the top 4 most important features to the model

```
RFE features: ['pclass', 'sibsp', 'parch', 'who']
```

ch07_ml/01_log_reg_titanic.py

Performing an RFE, it is determined that the top four features are pclass, sibsp, parch, and who. They way RFE works is that it tests all combinations of features and examines the scores. It then returns the top features that had the highest score combinations.

The Final Model and Saving the Model

- With the final feature set determined, the model is fit again and persisted to the file system using joblib

```
import joblib

features_matrix = titanic[['pclass', 'sibsp', 'parch', 'who']].values
response = titanic['survived'].values
X_train, X_test, y_train, y_test = train_test_split(features_matrix,
response, random_state=42)

lgreg.fit(X_train, y_train)
y_pred = lgreg.predict(X_test)
print(lgreg.score(X_test, y_test))

joblib.dump(lgreg, 'model.joblib')
```

ch07_ml/01_log_reg_titanic.py

Other techniques can be used when modeling. For example, cross validation can be employed which is essentially the rotating of data (into folds) so that it can be used in multiple combination sets and therefore can be used to create multiple models which are then scored. The average score should be higher than the version that doesn't use folding (kfold, etc.).

Afterward, the joblib library is used to persist the model to the file system.

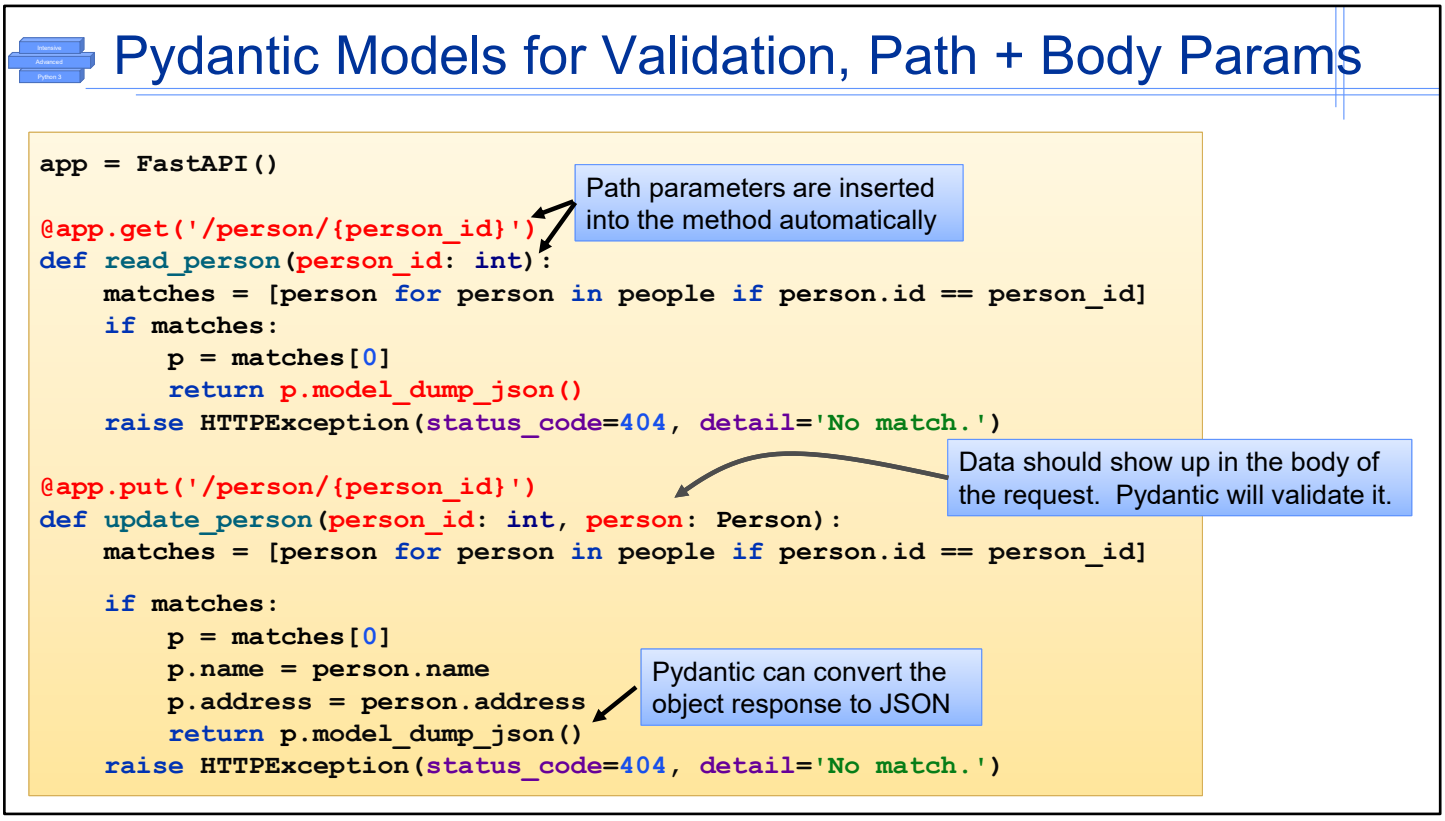
Introducing FastAPI

- FastAPI is a newer (2018) Python-based WSGI-compliant server designed specifically to support, high-performance APIs
  - It's Open API compliant
  - Capable of generating schema automatically
  - Compatible with Pydantic Models

```
import uvicorn
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

class Person(BaseModel):
    id: int = 0
    name: str = ''
    address: str = ''

people = [
    Person(id=1, name='Sam Smith', address='123 Main St'),
    Person(id=2, name='Tina Turtle', address='456 Elm Rd'),
    Person(id=3, name='Rocky Road', address='789 Birch Ave'),
]

app = FastAPI()

@app.get('/person/{person_id}')
def read_person(person_id: int):
    matches = [person for person in people
                                    if person.id == person_id]
    return p.model_dump_json()

uvicorn.run(app, host='127.0.0.1', port=8000)
```

ch07_ml/03_simple_fastapi.py

There's a lot going on in this small example. When a HTTP GET request comes to the server, if it maps to the url of localhost:8000/person/2, then the read_person() method will be called. This code locates the person and returns the object (converted to JSON using Pydantic's model_dump_json() method. If the object is not found, an HTTPException is raised (shown on the next slide).



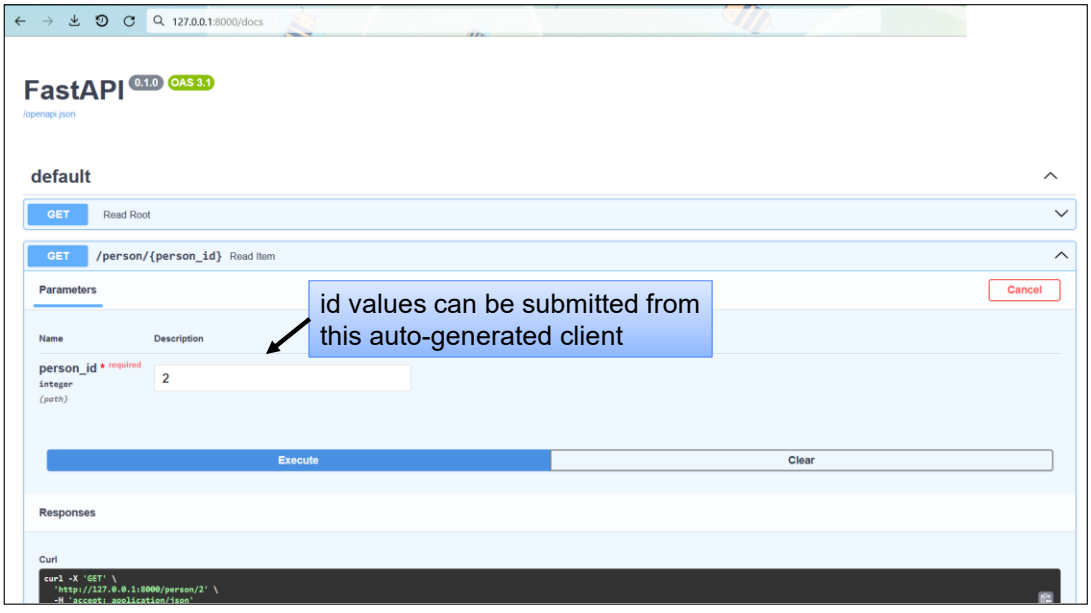
ch07_ml/03_simple_fastapi.py

Here, we can see a more complete picture of our simple FastAPI server. Exceptions are raised when an error occurs. These are converted to JSON responses. In addition, we can see a PUT implementation now. When a PUT request comes in, values are extracted from the path (person_id in this case) and inserted into the update_person() method. In addition to that, the parameters embedded into the body of the HTTP PUT request will be put into the Person parameter. They will be validated by Pydantic.

The remainder of the PUT route merely finds a person, modifies their data values or returns an HTTPException if no person is found.

Auto-Generated Test Client

- The client for the FastAPI server can be found at localhost:8000/docs



ch07_ml/03_simple_fastapi.py

The client is a Swagger-generated client that supports all of the operations built into the server code.

Titanic Logistic Regression Model in FastAPI

- Returning to the logistic regression model saved earlier, now it can be deployed into FastAPI and accessed via a request

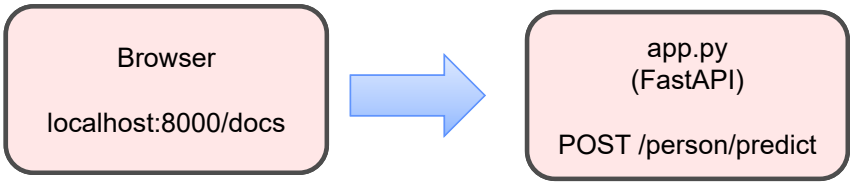
```
import joblib
from fastapi import FastAPI

class Passenger(pydantic.BaseModel):
    pclass: int
    sibsp: int
    parch: int
    who: int

app = FastAPI()
joblib_file = open('model.joblib', 'rb')
model = joblib.load(joblib_file)

@app.post('/titanic/predict')
def predict(data: Passenger):
    prediction = model.predict([[data.pclass,
          data.sibsp, data.parch, data.who]])
    return {'result': 'survived' if prediction[0]
                              else 'did not make it'}

uvicorn.run(app, host='127.0.0.1', port=8000)
```

The model we created earlier and saved using the joblib library, can now be recalled and used in the deployed FastAPI environment. To test it, visit localhost:8000/docs.

Your Turn! - Task 7-1

Logistic Regression and FastAPI

- Given the provided dataset related, create and deploy a model (based on a logistic regression) into FastAPI that predicts gender



	Weight	Height	Gender	T-Shirt	Gender_enc	T-Shirt_enc
0	180	6.0	male	XL	1	3
1	190	5.9	male	XL	1	3
2	210	5.7	male	XXL	1	5
3	220	5.9	male	XXL	1	5
4	170	5.6	male	L	1	0
5	165	5.9	male	L	1	0
6	100	5.0	female	XS	0	4
7	120	5.2	female	S	0	2
8	130	5.3	female	M	0	1
9	150	5.6	female	M	0	1
10	155	5.7	female	L	0	0
11	165	5.6	female	L	0	0
12	170	5.5	female	XL	0	3

- Create a Pydantic model containing weight (int), height (float), tshirt(int), gender (int)
- Work from the task7_1_starter/create_model.py file first and then app.py next

Chapter 7 Summary

- The short time spent on both machine learning and FastAPI do not do either topic proper justice
- FastAPI supports many additional features worth exploring
- Scikit-learn is quite extensive and requires additional time as well
- However, even in the span of a single chapter, you should be able to get a feel for both products and how they can be used elegantly together as well as each on their own

Chapter 8 - Overview

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

Django Features

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

django The web framework for perfectionists with deadlines.

https://www.djangoproject.com/

Version Info	
5.0	Dec 2023
4.1	Aug 2022 (some breaking changes)
4.0	Dec 2021
3.2	Apr 2021
3.1	Aug 2020
3.0	Dec 2019
2.0	Dec 2017
1.11	Apr 2017 Py 3.6 support
1.10	Aug 2016
1.9	Dec 2015
1.8	Apr 2015 Multiple Templating Engines
1.7	Sep 2014 Db Migrations
1.5	Feb 2013 Python 3 support
1.2	May 2010
1.0	Sep 2008
0.9	Nov 2005

Documentation URL

https://docs.djangoproject.com/en/5.0/

Django is one of the most popular Python-based full-featured web application frameworks. It has a strong user community and fantastic documentation.

Installing and Test Django

- Django is third-party and must be installed

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

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

```
> django-admin --version
5.0.1
```

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

OS X users may need to include the sudo argument when installing Django.

Before continuing, ensure the *django-admin --version* command works.

If you have an older version of Django installed, simply uninstall it with pip: pip uninstall Django and then reinstall it.

Creating a Project: django-admin.py

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

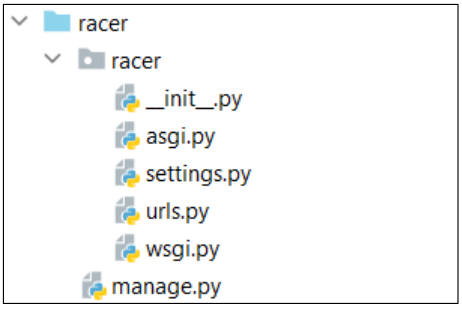
Execute from the directory where the project root folder should be created

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

Our Task 8-1 performs the following command from within the student_files directory:

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

The following project is generated



The django-admin.py script is a starting point for working with Django. This utility supports a number of subcommands, one of which creates the project skeleton. While using the utility is not necessary and you can build your own project structures, it is a good way to get started when beginning with Django development.

Note: You may perform Task 8-1 in parallel to walking through these slides as they align directly with the task.

django-admin.py and manage.py

- django-admin.py is a utility installed with Django initially
  - Used primarily to generate the initial project

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

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

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

Many manage.py commands

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

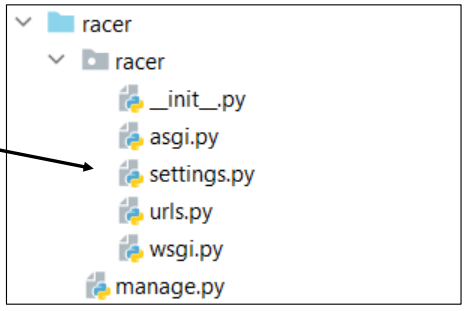
Type django-admin at a command prompt to get this list

manage.py becomes the main utility used once the initial project is created.

Establishing a Database

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

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



  - The following section exists within settings.py

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

settings.py defines 100+ configuration options. When the project was generated, SQLite was established as the default database to be used. This, of course can be modified.

Officially Django works with the above cited databases, but numerous additional databases can be unofficially used as well. For more on these, visit: https://docs.djangoproject.com/en/5.0/ref/databases/#third-party-notes

Django Backend Support

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

Officially supported Databases:
PostgreSQL
MySQL
Oracle
SQLite
MariaDB

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

Django database documentation:
https://docs.djangoproject.com/en/5.0/ref/databases/

Django Backends are easily established within settings.py (next slide). You may even configure multiple databases, though this is not common.

Example Setting Up Other Backends

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

settings.py

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

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

For more on the settings and configuration values for changing the database backend, visit:
https://docs.djangoproject.com/en/5.0/ref/settings/#std:setting-DATABASES

Creating Apps

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

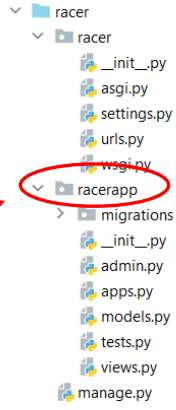
Projects can be viewed as a module containing settings. Apps are holders of models (yet to be discussed)

- To create an app issue the command

python manage.py startapp <app_name>

The following command was executed from our racer project directory:

> python manage.py startapp racerapp

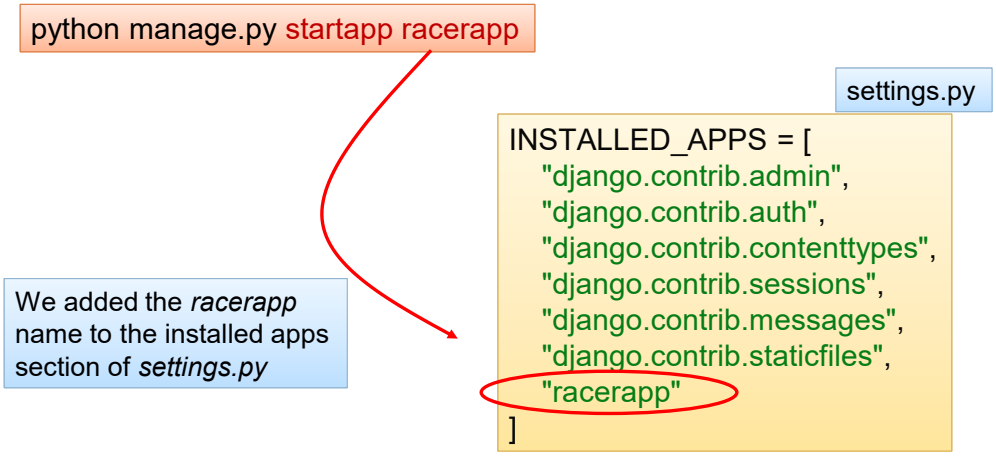


Projects and apps can always be manually created and configured in the settings.py. Generally, apps are somewhat independent of each other, usually having different models, for example. A project may support many apps. Apps may differ in URL by their top-level domain names (e.g., www.mysite.com and shop.mysite.com)

Above, the app is generated from the command and app-specific files are created.

Adding Apps to Settings

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



The settings.py file identifies the various apps that are installed and available for this application.

Models

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

racerapp/models.py

```
from django.db import models


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

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

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

All models that will use Django database object relational mapping must inherit from models.Model and then define predefined fields (at the class level) such as CharField or IntegerField.

Integrating the Database

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

The following command was executed:

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

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

The initial migrate command takes settings from settings.py and pushes them to the database. The migrate command without arguments is issued initially. This creates the tables to manage the app and authentications.

Defining and Creating Model Tables

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

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

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

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

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

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

The *makemigrations* command examines any changes to your models.
The *migrate <app>* command actually performs the migrations determined by makemigrations.

Making changes is a two-step process: first, the makemigrations identifies changes to be made and migrate pushes them. Prior to Django 1.7 this was the syncdb command.

A migration is a slow process that alters a DB schema. It can add rows to tables (by dropping the table and re-adding).

Optionally, you may use the *sqlmigrate* subcommand to display the SQL used to create the database tables. Use **python manage.py sqlmigrate racerapp 0001** to do this.

Checking Models and the Shell

- An interactive shell works with the database directly

This launches an interactive shell within your project root

The following command launches the shell:

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

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

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

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

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

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

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

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

The Django shell is a typical Python interactive shell, the only difference is that it will add your project directory to the PYTHONPATH. This enables easier imports such as the one above where we easily import Race and Participant from racerapp.models.

Within the shell, you may alternatively see a >>> prompt.

Using the Admin Features

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

racerapp/admin.py

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

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

We added these to the existing file

- To use the tool, first create a superuser:

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

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

The admin.py in the racerapp app allows for registering models. Before testing it out in a browser, however, you will need to create a superuser as shown here.

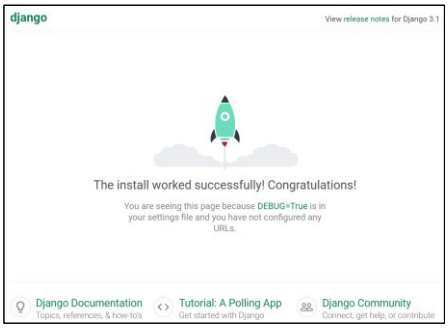
Running the Server

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

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

```
Performing system checks...

System check identified no issues (0 silenced).

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

- Navigate to localhost:8000 to view the default page

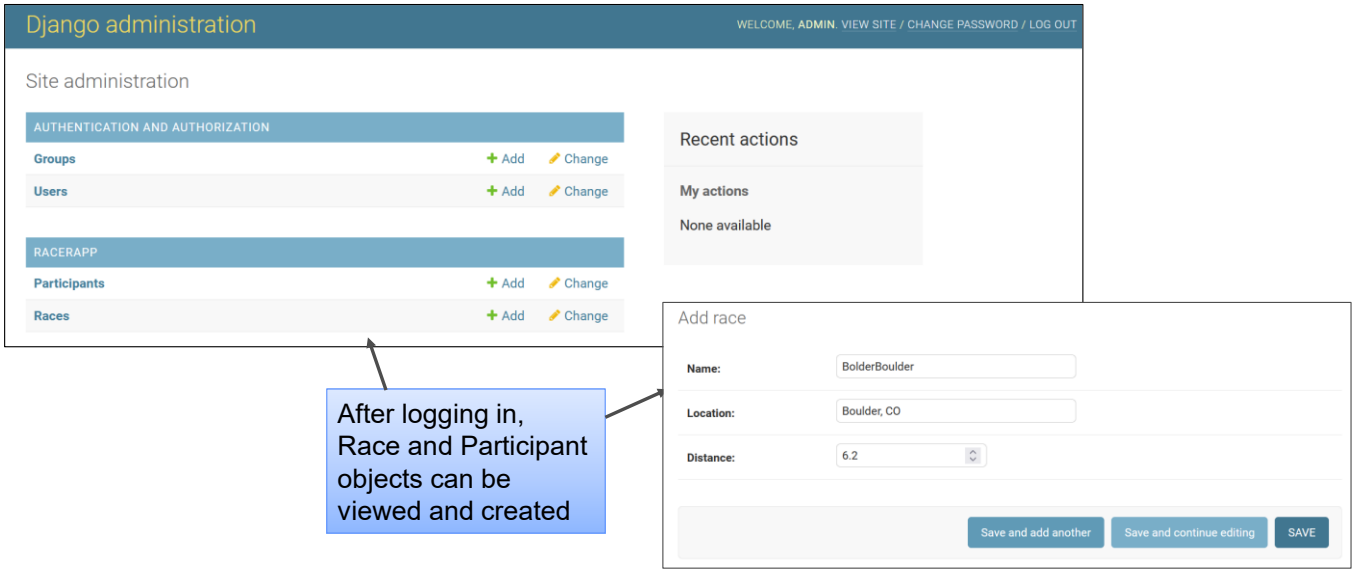


python manage.py runserver [ip_address:port]

where ip_address is optional and defaults to 127.0.0.1
and port is optional and defaults to 8000

Web-based Admin Interface

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



The admin web application assumes you launched the server using python manage.py runserver. Once authenticated using the super user credentials formerly created, objects and users may be managed directly with the database.

Creating Django Views

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

racerapp/views.py

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

from racerapp.models import Race, Participant


def list_races(request):
    races = Race.objects.values()
    return render(request, 'list.html', {'races': races})
```

This function (a view) is called for the URL: *http://localhost:8000/races*

But where is this defined?

This retrieves the races from the database

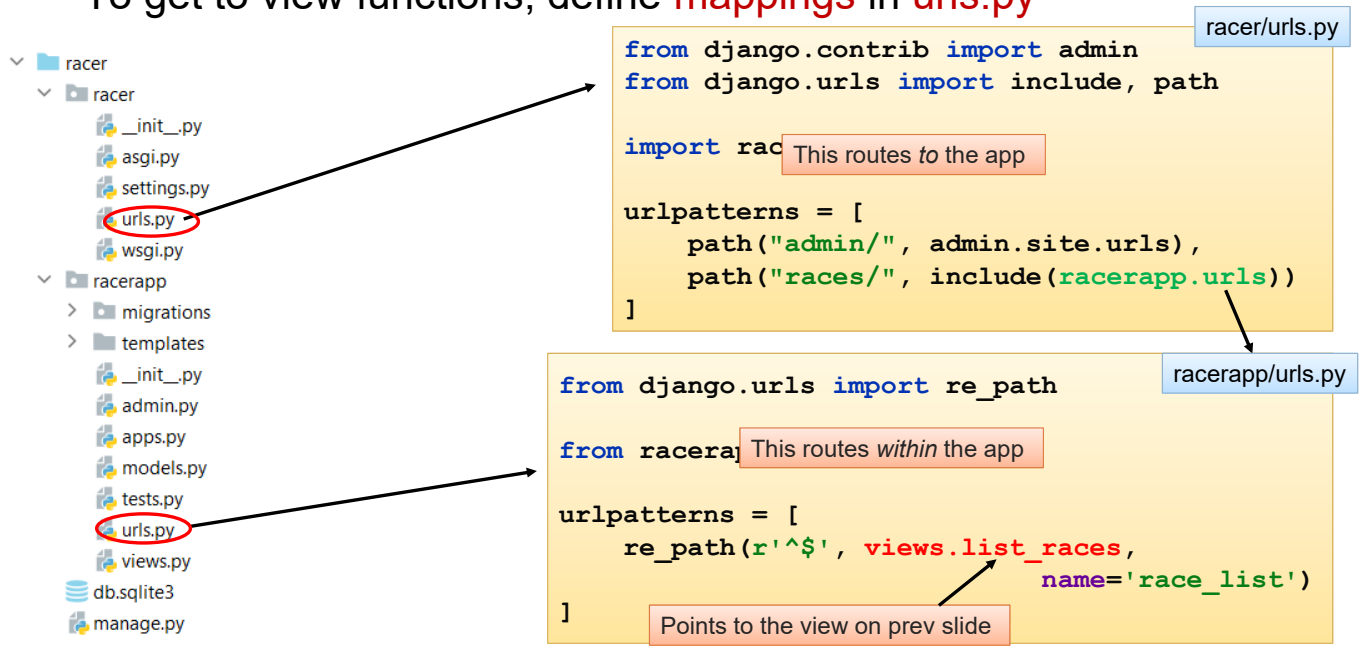
This forwards the request and races to a template called *list.html*

A function can be used to respond to web requests that come in. This function should be placed in the app's views.py file.

Django allows for class-based views as long as the class is callable. Additionally, Django has several built-in views to facilitate tasks such as working with database objects and forms.

Linking urls.py Files

- To get to view functions, define mappings in urls.py



Understanding the urls.py mapping files is essential to ensure the views get called. There are two urls.py files. The "master" urls.py file is in the racer folder and belongs to the project. It gets evaluated first.

In this example, a url of http://localhost:8000/races will first be evaluated in the racer urls.py file. The entry here is path('races/', include(racerapp.urls)) which means it will load the racerapp's specific urls.py file for the races/ mapping.

How URLs Map

- URLs first map to the urls.py in racer and then to the urls.py in racerapp

racerapp/urls.py

```
from django.urls import re_path
from racerapp import views

urlpatterns = [
    re_path(r'^$', views.list_races, name='race_list'),
]
```

A regex that is matched against the incoming URL *after* the match in racer/urls.py

The function in views.py to be invoked

An optional name to help distinguish this mapping

The r in front of the mapping implies a raw string, which is common when working with regular expressions. The ^ and $ in the regex indicate "from the beginning" and "up to the end" respectively.

When mapping URLs, be sure to place more specific regexes first as they are evaluated in order.

views.list_races refers to the list_races() function in views.py.

Rendering a Template from a View

- Views can render templates via a *render()* call

```
def list_races(request):
    races = Race.objects.values()
    return render(request, 'list.html', {'races': races})
```

racerapp/templates/list.html

```
<html lang="en">
    <head>
        <title>Race List</title>
    </head>
    <body>
        <h1>Race List</h1>
        <ul>
            {% for race in races %}
                <li>
                    <a href="/races/register/{{ race.name }}/">
                        {{ race.name }}
                    </a>
                </li>
            {% endfor %}
        </ul>
    </body>
</html>
```

This is Django templating syntax

Race List

Chicago Marathon
BolderBoulder

The list_races() view function matches: http://<server_name>/races. Refer to the next slide to understand the URL mapping further.

list_races() calls the list.html template which is rendered by passing the races retrieved from the database into it.

Regex Named Groups

- More complex mappings can use *regex named groups*

http://localhost:8000/races/register/Chicago Marathon

racerapp/urls.py

```
from django.urls import re_path
from racerapp import views

urlpatterns = [
    re_path(r'register/(?P<racename>[^/]+)', views.register),
    re_path(r'^$', views.list_races, name='race_list')
]
```

We added this new mapping to urls.py

racerapp/views.py

```
def register(request, racename=''):
   return render(request, 'register.html',
                 {"racename": racename})
```

The syntax (P<name>pattern) is called a regex named group. It allows naming the matching pattern. Here, we named it racename. The pattern that comes up is "Chicago Marathon" and this value is passed into the register() view function into the variable called racename.

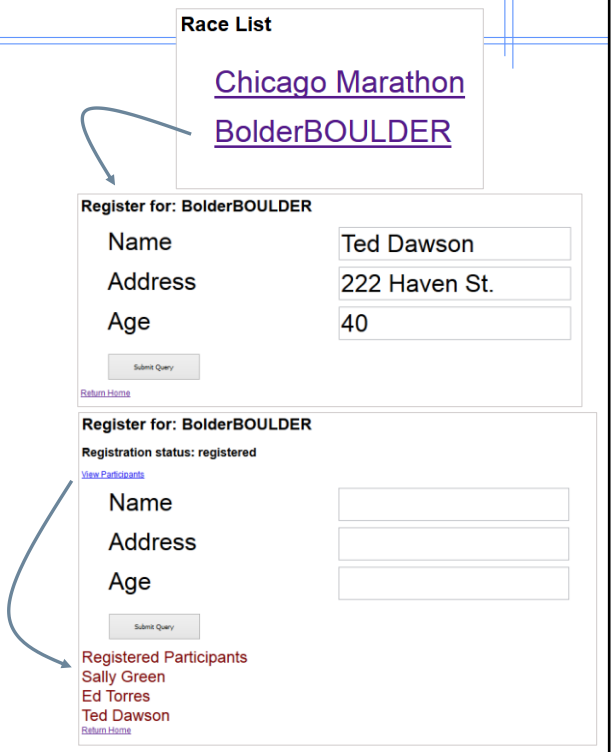
The register() view function then invokes the register.html template passing the racename into it.

Your Turn! - Task 8-1

Creating a Django App

- Build the Racer Django application
- Follow the steps within the workbook

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



Chapter 8 Summary

- Django has quite a bit of depth to it, making its learning curve somewhat steep
- It has several advantages over other frameworks, however
  - Most features are all integrated into the tool
  - Built-in templating, admin, and ORM, security, and more
- Django can be used in cloud environments such as Google App Engine or AWS

Course Summary

What did we learn?

Review of Intermediate Topics

- ***Databases***
- ***Logging***
- ***Context Managers***
- ***Decorators***

Relative Imports and __init__.py

Type Hints (Annotations)

Validators: mypy and pydantic

Coroutines (Generators)

Third-party tools: tqdm, fire, arrow, psutil, pysimplegui, black, flake8, click, multimethod, ray, aiofiles, asyncstdlib

Class Descriptors

Class Design Principles: SOLID

StringIO

Abstract Classes

Repository Pattern

Class Generics

Enums, String Literals

Techniques for Overloading

Keyword/Positional-only Parameters

Automating Tools

File Directory and Access Modules

PDF Creation, ReportLab, pymupdf

Email Creation / Sending

Introduce Scikit-Learn Modeling

Introducion to FastAPI

Django

Packaging and Distribution: wheel files

Subprocesses: run() and Popen()

Threads and Multiprocessing

Asynchronous I/O

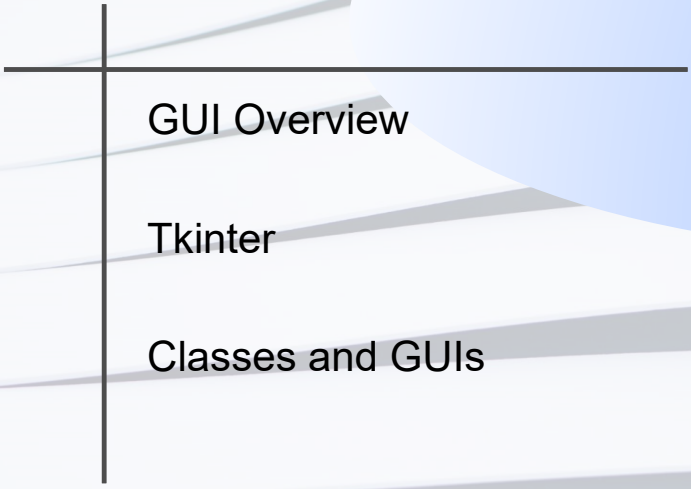
Evaluations

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

Thank you for your feedback
which is critical to this process.

Questions

Chapter 9 - (Apdx A) Overview



GUI Overview

Tkinter

Classes and GUIs

Python GUI Intro

- Python provides several options when developing Graphical User Interface (GUI) applications
  - Tkinter
    - Part of the Python Standard Library, typically no install needed
    - Python-based interface to the Tk GUI framework
    - No special license needed
  - PyQt (PyQt4, PyQt5, PyQt6) `pip install PyQt5` `pip install PyQt6`
    - A Python-based interface to the Qt C++ GUI framework
    - Support for Windows, OS X, Linux, Android, iOS
    - Can use Qt Designer, a GUI Drag-and-drop editor, to build the app and generate Python code from it
    - Dual license: commercial and GPLv3
  - Other GUI systems include: Kivy, wxPython, PySide2

While there are numerous GUI frameworks available within Python, the two most commonly used are: Tkinter (due to its ease of use and availability within the standard library) and PyQt (due to its nice-looking widgets and drag-and-drop interface capabilities).

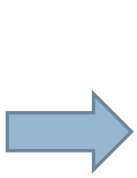
For more on PyQt, visit its homepage: https://www.riverbankcomputing.com/software/pyqt/

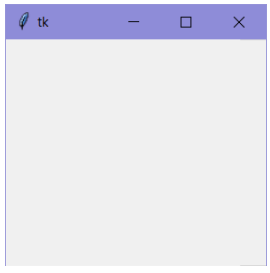
Tkinter Intro

- Tkinter creates *widgets* and places them into *windows*
- The main window is created from the Tk class

Try it now!

```
import tkinter as tk

window = tk.Tk()
window.mainloop()
```





Starts the window object's event loop which will handle user interactions such as button or key presses

Creates a top level widget. A Tcl interpreter is started (behind the scenes)

Type the code shown above into temp.py in your student files and then run it.

Tkinter Widgets

- There are numerous widgets within Tkinter
  - Frame — Intermediate widget containers to help with layout
  - Label — Widget containing text (or an image) to be placed
  - LabelFrame — A grouping of Labels
  - Button — Clickable interactive widget
  - Checkbutton — A toggle-type widget
  - Radiobutton — A selectable choice between two or more items
  - Entry — Single line input field
  - Text — Multiline input field
  - PanedWindow
  - Listbox — List of selectable items
  - Menubutton — Menu widgets
  - Scrollbar — Scrollbar widget
  - Scale — A slider widget
  - Spinbox — Arrows to increment/decrement values

A submodule, called **ttk** contains these and a few additional widgets including: *Combobox*, *Progressbar*, *Notebook*, *Separator*, *Sizegrip*, and *Treeview*
(from tkinter import ttk)

Tkinter provides a core set of commonly used GUI widgets. A submodule, called ttk, implements all of these plus a few additional widgets. The ttk widgets have a similar interface but provide a slightly better look and feel. Ttk widgets tend to be a little nicer option over the tkinter widgets. To use the ttk widgets instead, simply import the submodule as follows: from tkinter import ttk.

Adding Widgets

- Before the mainloop(), widgets can be added to our window

Continue by trying it now!

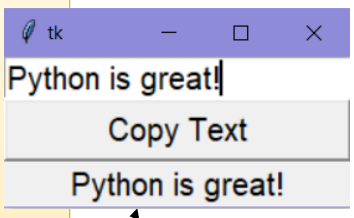
```
import tkinter as tk

root = tk.Tk()

label = tk.Label(text='Hello World',
                 foreground='#223344',
                 background='#fcfeee')
label.pack()

root.mainloop()
```

You can also replace foreground= with fg= and background= with bg=. A height and width parameter may also be specified. For labels, units are measured with the number "0," where width=1 would be 1 zero-width and width=2 would be 2 zero-widths, etc.

Event Handling

- Events and handlers can be created to enhance user interactivity

```
def copy_text():
    label.configure(text=entry.get())

window = tk.Tk()

entry = tk.Entry(font=16)
entry.pack(fill='x')

button = tk.Button(text='Copy Text', font=16,
                   command=copy_text)
button.pack(fill='x')

label = tk.Label(text='', font=16)
label.pack(fill='x')


window.mainloop()
```



ch09_apdx_a_gui/03_event_handling.py

The configure() method exists for changing any attributes of a widget. Here we used *configure()* to change the value of the text contained within the Label widget. The label widget's text is changed in the copy_text() function which is called when the button is pressed. The command= parameter assigned the function to be called when the button is pressed. Python's scoping rules allow the label and entry widgets to be visible within the event handling function.

The fill='x' option causes the widgets to expand fully in the x direction. Other options include 'y' and 'both'.

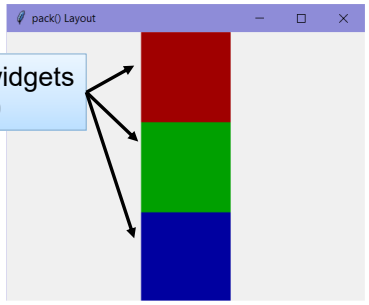
Layouts

- Widgets are placed into the view using one of three layout managers:
  - pack() — Tk sizes the window as small as it can while still displaying all the widgets
  - grid() — Widgets are laid out using a table of rows and columns
  - place() — Widgets are placed at specified locations

There are three ways in which widgets can be laid out. Each of these techniques is examined on the next few slides.

Using the pack() Layout

```
import tkinter as tk

root = tk.Tk()
root.title('pack() Layout')
root.geometry('400x300')

frame1 = tk.Frame(width=100, height=100, bg='#a00000')
frame1.pack()

frame2 = tk.Frame(width=100, height=100, bg='#00a000')
frame2.pack()

frame3 = tk.Frame(width=100, height=100, bg='#0000a0')
frame3.pack()

root.mainloop()
```



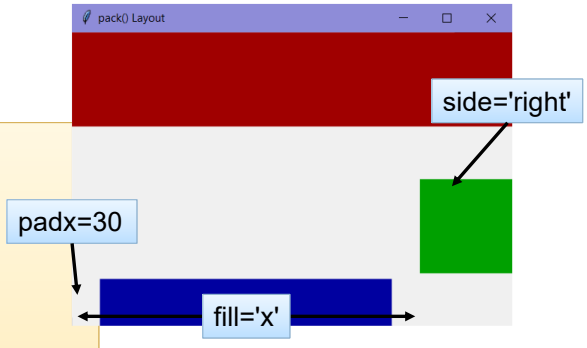
pack() stacks widgets on top of each other in order they are packed. Remaining space is left empty.

ch09_apdx_a_gui/04_pack_layout.py

A pack() layout stacks widget (unless they are placed onto a side. Remaining space not filled by the sizing of the widget will remain empty.

pack() Options

```
import tkinter as tk

root = tk.Tk()
root.title('pack() Layout')
root.geometry('400x300')

frame1 = tk.Frame(height=100, bg='#a00000')
frame1.pack(fill='x')

frame2 = tk.Frame(width=100, height=100, bg='#00a000')
frame2.pack(side='right')

frame3 = tk.Frame(height=50, bg='#0000a0')
frame3.pack(padx=30, fill='x', side='bottom')

root.mainloop()
```



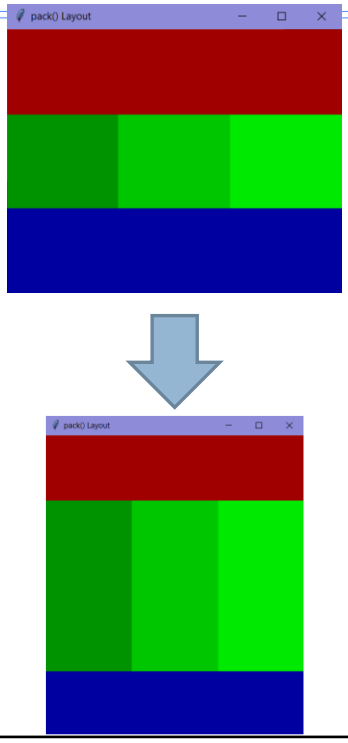
ch09_apdx_a_gui/04b_pack_fill.py

Using *side=* mounts the widget to that location and it stays there even if the window is resized. In the example above, because frame2 is mounted to the side, this reduces the remaining space for the bottom widget to be able to fill left and right.

The padx option adds a padding on the left and right sides of frame3 while fill causes it to expand fully (minus the padding).

More pack() Options

```
root = tk.Tk()
root.title('pack() Layout')
root.geometry('400x300')

frame1 = tk.Frame(height=100, bg='#a00000')
frame1.pack(fill='x', side='top')

frame2 = tk.Frame(height=100, bg='#0000a0')
frame2.pack(fill='x', side='bottom')

frame3 = tk.Frame(bg='#009300')
frame3.pack(expand=True, side='left')

frame4 = tk.Frame(bg='#00c600')
frame4.pack(expand=True, side='left',
            fill='both')

frame5 = tk.Frame(bg='#00e900')
frame5.pack(expand=True, side='left',
            fill='both')

root.mainloop()
```



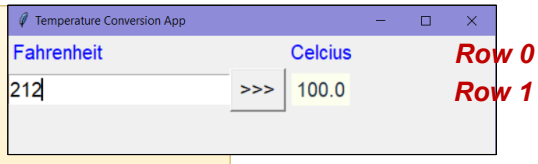
ch09_apdx_a_gui/04c_pack_side.py

Expand will cause the widget to fill any space not used. Run this example and resize the window. Note the resizing of the green frames. This is the effect of the expand= option. Also note that the red and blue frames will remain at size 100 and are fixed to the top and bottom respectively.

Using the *grid()* Layout *(1 of 2)*

- A grid layout allows for defining rows and columns

```
import tkinter as tk

root = tk.Tk()
root.title('Temperature Conversion App')
root.geometry('500x120')

# row 0 definitions
lbl_fahr = tk.Label(font=16, fg='blue', text='Fahrenheit')
lbl_fahr.grid(row=0, column=0, ipadx=2, ipady=2, sticky='W')

lbl_celcius = tk.Label(font=16, fg='blue', text='Celcius')
lbl_celcius.grid(row=0, column=2, ipadx=2, ipady=2, sticky='W')

# row 1 definitions
entry = tk.Entry(font=16)
entry.configure(text='0')
entry.grid(row=1, column=0, ipadx=2, ipady=2)
```



ch09_apdx_a_gui/05_mini_app.py

ipadx and *ipady* add a padding inside of the grid cells. *sticky* causes the contained widget to stick to that wall ('N', 'S', 'E', 'W'). Multiple values can be used.

Using the *grid()* Layout *(2 of 2)*

```
...continued from previous...
def enter_key(event):
    if event.keycode == 13:
        convert()

def convert():
    try:
        value = int(entry.get())
        conversion = 5.0 * (value - 32.0) / 9.0
        lbl_output.configure(text=str(conversion))
    except ValueError:
        lbl_output.configure(text='Must be a number!')
```

Calls *convert()* when the button is pressed

```
button = tk.Button(font=16, text='>>>', command=convert)
button.grid(row=1, column=1, ipadx=2, ipady=2)

lbl_output = tk.Label(master=root, font=16, text='0',
                      foreground='#223344', background='#fcfeee')
lbl_output.grid(row=1, column=2, ipadx=2, ipady=2)

root.bind('<Key>', enter_key)
root.mainloop()
```

bind() assigns our root window to handle <Key> presses and calls the enter_key() event handler

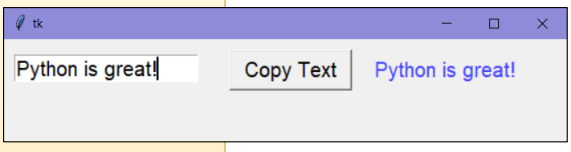
ch09_apdx_a_gui/05_mini_app.py

The Label() widget definition shows the assignment of master=root which assigns the parent control to the main window for the Label. This is the default behavior, so the master=root argument isn't really needed but has been placed here to illustrate that we can assign a different parent widget if desired.

Using the *place()* Layout

- *place()* allows for specifying x= and y= values to place widgets exactly where you want them
  - The origin (0, 0) is the upper left

```
def copy_text():
    label.configure(text=entry.get())

root = tk.Tk()
root.geometry('550x100')


entry = tk.Entry(width=16, font=16)
entry.place(x=10, y=15)

button = tk.Button(text='Copy Text', font=16, width=10,
                   command=copy_text)
button.place(x=220, y=10)

label = tk.Label(text='', font=16, width=16, fg='#4444ff')
label.place(x=340, y=15)
```

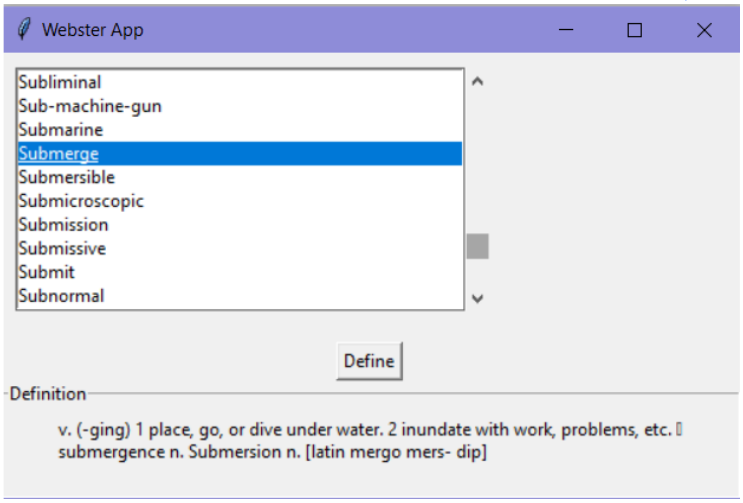


ch09_apdx_a_gui/06_place.py

Here, we demonstrate the use of the .place() layout. While it may seem easy to use, it isn't as flexible as the grid() or pack() options as Window sizes changes while the values specified here are absolute (regardless of Window size).

Your Turn! - Task 9-1

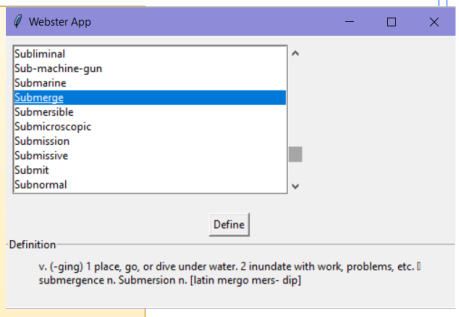
- Creating a GUI with Tkinter



- Create the "Webster App" shown above by following the instructions in the student workbook and working from the *task9_1_starter.py*

Organizing Code into Classes

```
class WordFrame(tk.Frame):
    ...

class DefinitionFrame(tk.LabelFrame):
    ...

class WebsterApp:
    def __init__(self, parent):
        parent.title('Webster App')
        parent.geometry('500x300')
        self.webster = self.load_data()

        self.word_frame = WordFrame(parent, self.webster)
        self.definition_frame = DefinitionFrame(parent)
        self.button = tk.Button(text='Define',
                                command=self.get_selection)
        self.button.pack(side='bottom')

    def load_data(self):
        ...

    def get_selection(self):
        ...
```



Sometimes it is easier to manage solutions using classes

ch09_apdx_a_gui//07_class_based.py

In this example, we've broken up the app into individual components. WebsterApp is the main class that assembles the higher-level components. Notice how the Webster __init__() is used to compose a WordFrame(), then a DefinitionFrame(), and finally a Button(). Data is also loaded in the main WebsterApp() class and event handlers have been placed here as well.

For the complete listing, refer to the source file in the student files.

Appendix A Summary

- Tkinter is an easy-to-use tool that creates GUIs
  - Its major disadvantage is a lack of creative, flexible widgets
  - It also doesn't have a native OS look-and-feel
- Tkinter widgets integrate nicely into Python code, however
- Though discussions have focused on Tkinter, PyQt is also a popular Python GUI option and deserves its own consideration

Intensive Advanced Python

Exercise Workbook

Setup and Test of *Python Environment*

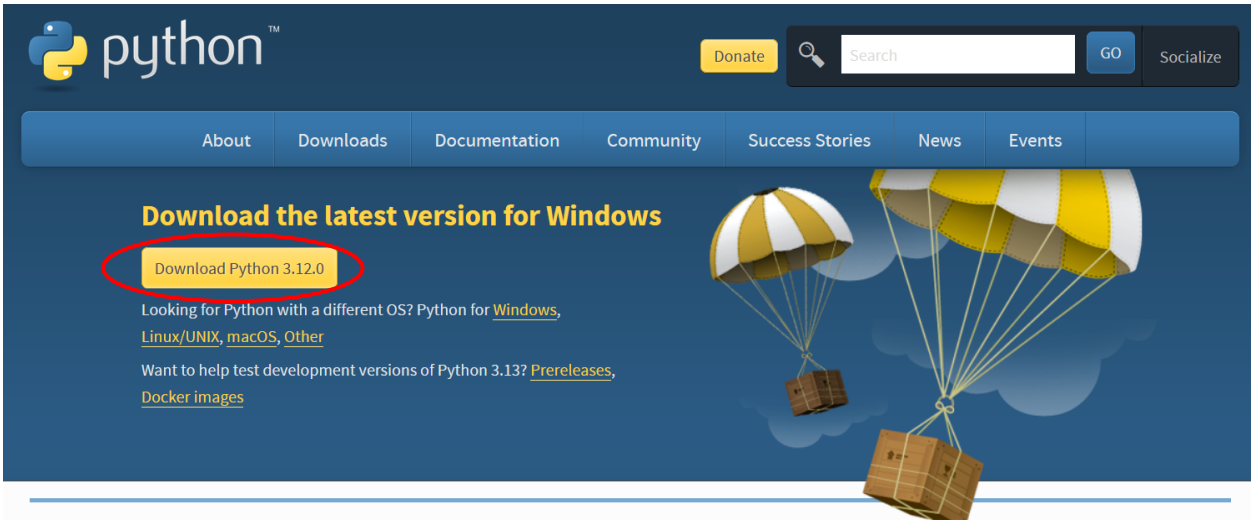
Overview

This exercise will help ensure that your Python environment for this course is properly set up. If you have been provided with a pre-configured machine or virtual desktop, you should be able to skip this task, however, it may be useful to review it to better understand what has already been done for you.

Install Python

If you have not already done so, install Python by visiting **`https://www.python.org/downloads/`**. If you already have Python installed, skip to the next step.

Click the link to download Python 3.x (3.12 is the preferred version).



Click the link (circled in the previous diagram) to download the appropriate version for your platform.

Run the installer and select the custom install option.

Install Python ensuring you ***Add Python to the Path*** (watch for this option during the installation process). Also, if your permissions will allow it, select the ***Install for all users*** option.

Afterwards, **open a console or terminal** window and type:

```
python -V
```

If the command is not recognized or the wrong python version is displayed, you will need to modify your PATH environment variable to include the <PYTHON_HOME> directory. Setting the PATH varies on operating systems, so ask for help if assistance is needed.

Note for Macs

Replace the above command with **`python3 -V`** *or* **`python3.12 -V`** *instead. Substitute 3.11, 3.10, 3.9, etc., for the version you installed version if different from 3.12.*

One of these commands should work for you. If not, you will need to check your PATH environment variable and double-check the location that Python was installed.

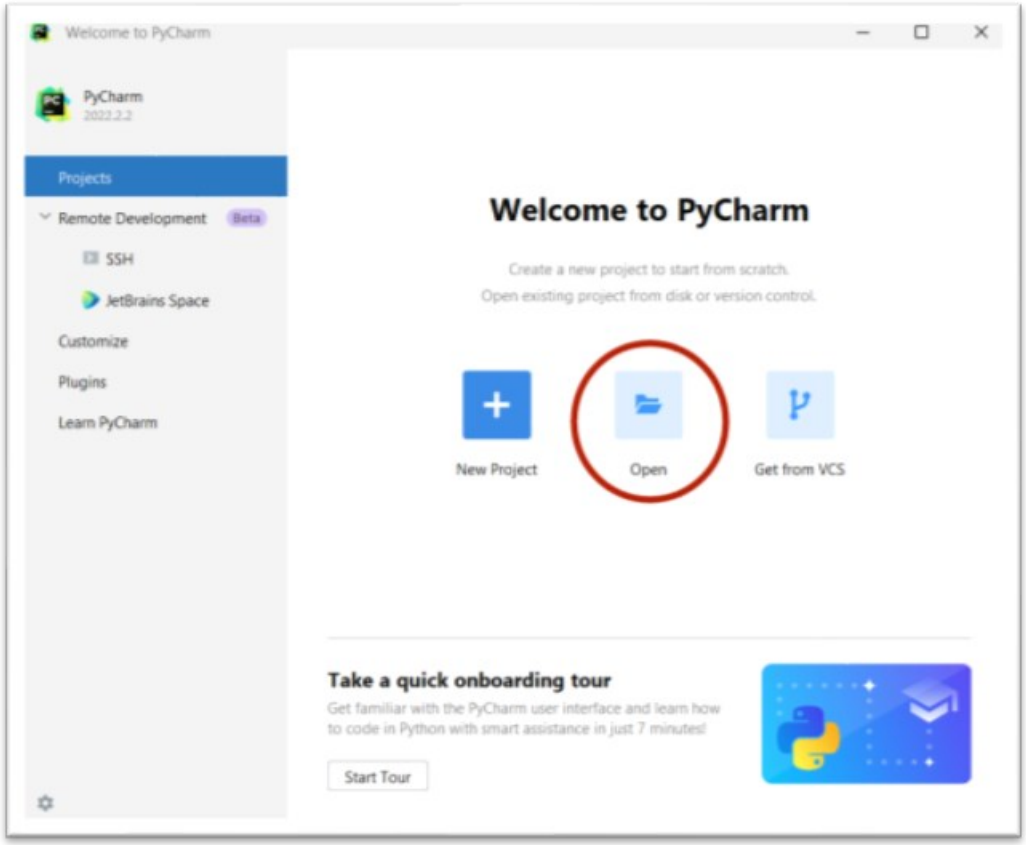
In this course, replace any Python command-line references (e.g., python, python3, python3.12, etc.) with the syntax that you just determined works for you!

Launch PyCharm, Set Up Student Files

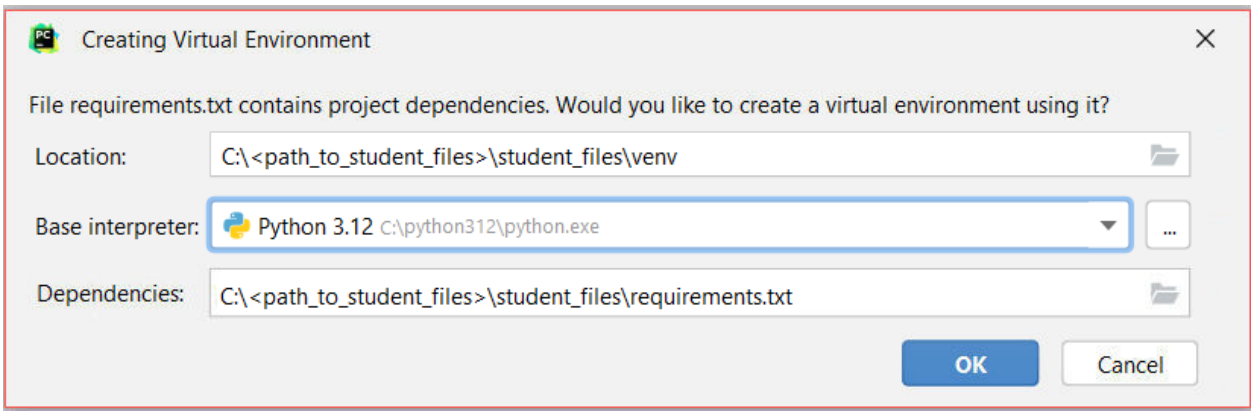
PyCharm Community Edition is a free Python IDE created by JetBrains and recommended for this course.

Note: if instead of the "Welcome" dialog your PyCharm launches directly into the IDE and you can see your project files, you may skip the rest of this step.

Launch PyCharm, arriving at the "Welcome" dialog (which will be similar though may have differences to the one shown below). Select "Open..." and browse to the student_files directory.



Once the new project opens, PyCharm should automatically prompt to create a virtual environment (like the screenshot below). If so, complete this step and then go to step 4. If you didn't see the dialog below, go to step 3 now.



Select the suggested defaults that PyCharm prompts you with (unless the *Base Interpreter* option looks wrong) and click OK. Environment creation takes several minutes which includes installing additional modules defined in *requirements.txt*.

If any information looks incorrect, fix it before clicking OK.

Manually Creating the Virtual Environment

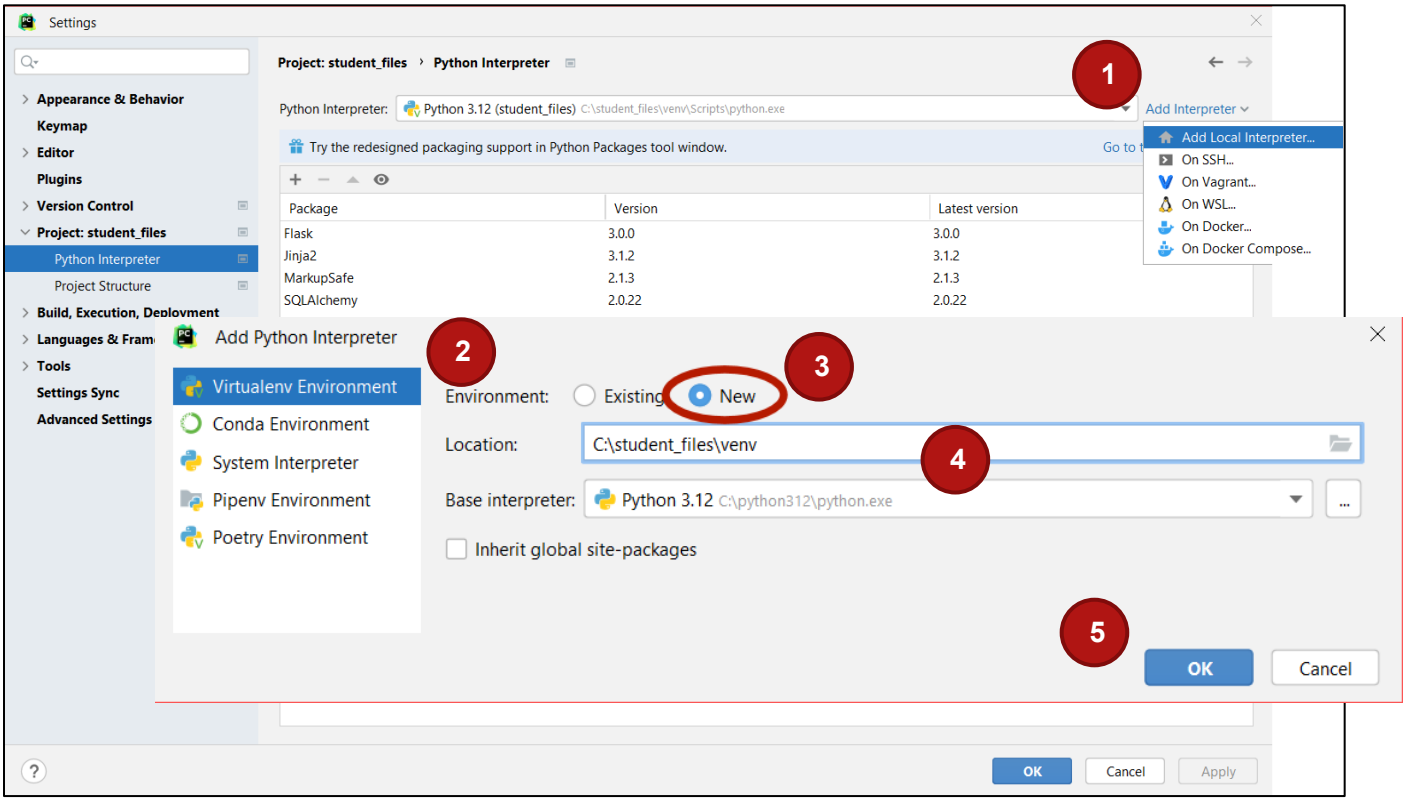
In some cases, PyCharm will not prompt to create a virtual environment for you automatically. You will need to do it manually yourself. Perform step 3 only if you couldn't complete step 2 above as directed.

Once you are within the main PyCharm project, select the *settings* menu based on operating system as follows:

OS X: **`PyCharm > Settings`**
(some versions: `PyCharm > Preferences`)

Windows: **`File > Settings`**

Within the settings, expand the **Project: student_files** menu item. Select **Project Interpreter**. On the right side, we'll have to point PyCharm to create a virtual environment as shown below:



1- Select the "Add Interpreter" item (may look like a gear symbol).
2- Leave "Virtualenv Environment" option selected.
3- Select **"New"** Environment radio button.
4- Ensure Location points to the **student_files** directory and ends with **venv** after that (your student_files location may vary). Also, ensure the correct base interpreter is selected.
5- Click OK twice to return to the packages screen.

Return to the IDE and the project view. Double-click the requirements.txt file to open it. Then select the blue underlined link that reads "Install dependencies" in the yellow bar that appears.

Running Scripts

Test out the ch01_review/01_formatting.py file to ensure it works.

Open the file, right-click in the source code and select **Run 01_formatting.py**. You should see results from running this script.

This concludes the setup.

NOTE: In this workbook, occasionally lines of code will be written like this:

```
people_filepath =
        os.path.join(working_dir, people_filename)
```

These long lines should be written all on one line.

Task 1-1
Generators and the Database (Part 1 of 2)

Overview

The following provides step-by-step instructions for completing Task1-1. In this task, you will create a context manager utility using the @contextmanager decorator. You will then use this utility to gain access to a database using a *with* control. You will query the database in the *with* control and retrieve information related to countries from around the world. You will place the results into a dictionary. The table we are using is called *countryinfo* and looks partially like the following:

iso	iso3	country	capital	neighbours
AD	AND	Andorra	Andorra la Vella	ES,FR
AE	ARE	United Arab Emirates	Abu Dhabi	SA,OM
AF	AFG	Afghanistan	Kabul	TM,CN,IR,TJ,PK,UZ
AG	ATG	Antigua and Barbuda	St. John's	
AI	AIA	Anguilla	The Valley	
AL	ALB	Albania	Tirana	MK,GR,ME,RS,XK

You will work from the ch01_review/starter/task1_1 folder. You will edit three files in the following order:

- connection.py

- __init__.py

- task1_1.py

Begin creating get_connection()

Locate and open the file ch01_review/starter/task1_1/utils/**connection.py**.

Complete the method signature and add the decorator as shown (in bold below). The connect_params will be a dictionary that passes all needed database connection info into it.

```
@contextmanager
def get_connection(connect_params: dict) ->
                        Generator[sqlite3.Connection, None, None]:

    connection = None

    try:

    except sqlite3.Error as err:

    finally:



if __name__ == '__main__':
    path = Path(__file__).parents[4] /'resources/course_data.db'
```

Connect to DB and Yield the Connection

Add logic to connect to our database. Yield this connection. Also add logging statements to display that we connected. Finally, add logic to handle any exceptions and close the connection in the `finally` block.

```
@contextmanager
def get_connection(connect_params: dict) ->
                        Generator[sqlite3.Connection, None, None]:

    connection = None

    try:
        connection = sqlite3.connect(**connect_params)
        logging.debug('DB connection obtained.')
        yield connection
    except sqlite3.Error as err:
        raise Exception('Invalid connection parameters.',
                        connect_params) from err
    finally:
        if connection:
            connection.close()
            logging.debug('DB connection closed.')
```

At the bottom of the file, test the solution in a with control as follows:

```
if __name__ == '__main__':
    path = Path(__file__).parents[4] /'resources/course_data.db'
    with get_connection({'database': path}) as conn:
        assert isinstance(conn, sqlite3.Connection)
```

Create an Import Shortcut

Open the **starter/task1_1/utils/__init__.py** file and add an import shortcut for the get_connection method:

```
from .connection import get_connection
```

Import the get_connection() Shortcut

Open the **starter/task1_1/task1_1.py** file and import the get_connection() shortcut.

```
from utils import get_connection
```

Use the get_connection() Method

Within the try-block of the countryinfo() method, use the *with* control to make a query to the database and retrieve records:

```
def countryinfo(db_params: dict) -> dict[str, tuple]:
    info = {}
    sql = 'select iso, iso3, country, capital, neighbours from countryinfo'

    try:
        with get_connection(db_params) as connection:
            cursor = connection.cursor()
            cursor.execute(sql)
```

Retrieve and Save the Data

Complete the with control by saving the returned records in a dictionary:

```
def countryinfo(db_params: dict) -> dict[str, tuple]:
    info = {}
    sql = 'select iso, iso3, ... from countryinfo'

    try:
        with get_connection(db_params) as connection:
            cursor = connection.cursor()
            cursor.execute(sql)

            for iso, *remaining in cursor:
                info[iso] = remaining
```

Call countryinfo()

Call the countryinfo() method at the bottom of the module. Pass the params object into it. Display the len() of the returned dictionary.

```
database = Path(__file__).parents[3] /'resources/course_data.db'
params = {'database': database}
country_info = countryinfo(params)
print(f'{len(country_info)} countries found.')
```

That's it! Test it out but running task1_1.py.

Task 1-2 *Co-routines, Command-line, Progress Bars (Part 2 of 2)*

Overview

The following provides step-by-step instructions for completing Task1-2. In this task, you will create a coroutine utility that accepts as many parameters as desired by sending them in. The parameters will be provided on the command-line and retrieved using argparse. The coroutine simply displays (returns) the neighboring countries of the provided input country.

The file called connection.py contains the completed get_connection() method from the previous exercise.

You will work from the ch01_review/starter/task1_2 folder. You will edit three files in the following order:

- countryinfo.py

- __init__.py

- task1_2.py

Examine the File Structure, Begin Writing the Coroutine

Look over the files located within the ch01_review/starter/task1_2 folder. Notice connection.py contains a completed get_connection() method.

Open countryinfo.py. Notice the function called countryinfo() is our completed function from the previous exercise. Below the countryinfo() function is the coroutine that needs to be completed, called find_neighbors().

In the coroutine, call the countryinfo() function above it which generates a dictionary containing all the country neighbors. Then below that, create a while loop that loops indefinitely. Retrieve that (country) value that will eventually be sent in. Use this country value to look up the neighbors in the dictionary. We can either print the neighbors immediately, or for fun, we can yield them. Do this as follows:

```
def find_neighbors(db_params: dict) -> Generator[list, str, None]:
    empty_set = ['', '', '', '']
    country_info = countryinfo(db_params)

    while True:
        next_country = yield
        if not next_country:
            break
        info = country_info.get(next_country, empty_set)
        neighbors = info[3].split(',')
        yield
[country_info.get(neighbor, empty_set)[1] for neighbor in neighbors]
```

Note: the list comprehension should come after the yield on the same line.

Add Shortcut Import to __init__.py

As done in Task1_1, open the __init__.py file and add a shortcut for the find_neighbors() function.

```
from .countryinfo import find_neighbors
```

Import find_neighbors() into task1_2.py

Open the **starter/task1_2/task1_2.py** file and add an import for find_neighbors by using the shortcut previously defined:

```
from utils import find_neighbors
```

Complete and Call the get_args() Method

Open the **starter/task1_1/task1_1.py** file and import the get_connection() shortcut.

```
def get_args():
    parser = ArgumentParser()
    parser.add_argument('names', nargs='*', default=['US'],
                        help='Example: US DE IN')

    return parser.parse_args()


...

database = Path(__file__).parents[3] /'resources/course_data.db'
params = {'database': database}

countries = get_args()
```

Call and Retrieve From the Coroutine

Invoke the coroutine to "prime" it. Then call next() on the coroutine within a for-loop for each command-line argument (country) provided. Print the results of sending a value in each time:

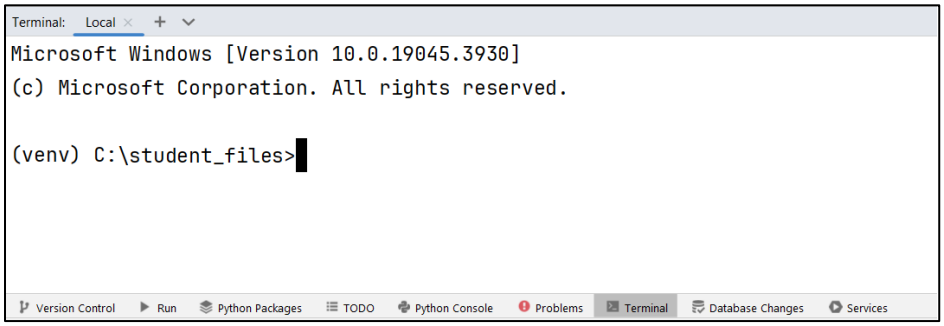
```
countries = get_args()

neighbor_coroutine = find_neighbors(params)
for name in countries.names:
    next(neighbor_coroutine)
    print(neighbor_coroutine.send(name.upper()))
```

Incorporate tqdm()

Add tqdm() into the solution above as follows:

```
results = []
for name in tqdm(countries.names):
    time.sleep(1)
    next(neighbor_coroutine)
    results.append(neighbor_coroutine.send(name.upper()))
pprint.pp(results)
```

That's it! Test it out by opening a command window (preferred) or selecting the Terminal tab at the bottom of PyCharm.



Type: **`cd ch01_review\starter\task1_2`**.

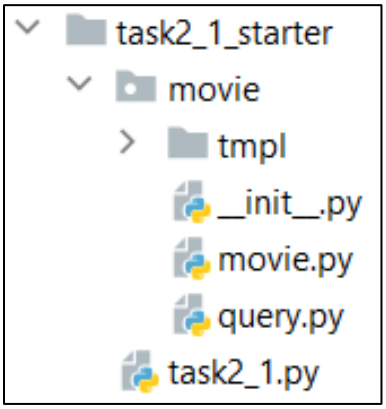
Type: **`python task1_2.py US CN BR`**

Task 2-1 *Movie Query and Classes*

Overview

This exercise requests movie data from an external source (TheMovieDB) and receives JSON-based data in response. The user enters a search term and receives a list of results in return. You'll display this list allowing a user to select one. A second API request occurs returning detailed information about the movie. The detailed information is captured into a Movie class instance and results are rendered in a Jinja2 template. You will also create the Movie class and another class to execute the network requests.

Locate the **`ch02_classes/task2_1_starter`** folder. You will begin by creating the two classes (Movie and TMDBMovieQuery).



Create the Movie (Pedantic Model) Class

Create a Movie class within movie.py. Have it inherit from BaseModel since it is a Pedantic model class. Then add a title, release_date, runtime, budget, revenue, and tagline.

```
logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())


class Movie(BaseModel):
    title: str = '(no title)'
    release_date: str = ''
    runtime: int = 0
    budget: int = 0
    revenue: int = 0
    tagline: str = ''
```

Add a render() Method to the Class

Below the code added from step 1, add a render() method. Insert the already provided Jinja2 code as shown:

```
class Movie(BaseModel):
    title: str = '(no title)'
    release_date: str = ''
    runtime: int = 0
    budget: int = 0
    revenue: int = 0
    tagline: str = ''

    def render(self) -> str:
        try:
            tmpl = env.get_template('movie.jinja')
            results = tmpl.render(movie=self)
        except jinja2.exceptions.TemplateError as err:
            logger.error(f'{type(err)}: {err}')
            results = f'Error rendering movie.  See error log.'
        return results
```

Implement the __len__() Magic Method

Implement the __len__() magic method as shown. Optionally add the __str__() magic method. The complete class is shown below:

```
class Movie(BaseModel):
    title: str = '(no title)'
    release_date: str = ''
    runtime: int = 0
    budget: int = 0
    revenue: int = 0
    tagline: str = ''

    def render(self) -> str:
        try:
            tmpl = env.get_template('movie.jinja')
            results = tmpl.render(movie=self)
        except (jinja2.exceptions.TemplateNotFound,
                jinja2.exceptions.TemplateError) as err:
            logger.error(f'{type(err)}: {err}')
            results = f'Error rendering movie.  See error log.'
        return results

    def __len__(self):
        return self.runtime

    def __str__(self):
        return f'{self.title} [${self.revenue:,}]'
```

Test the Movie Class (Within the Module)

Create a short code snippet at the bottom of the movie.py module to test the class. Do this as follows:

```
if __name__ == '__main__':
    data = {'title': 'Avengers'}
    m = Movie(**data)
    print(f'Movie: {m}')
    print(m.render())
```

Add a Movie Shortcut to __init__.py

Open __init__.py and add a relative import shortcut import that points to the Movie class.

```
from .movie import Movie
```

Import Movie into query.py

Open the query.py file. At the top, import the Movie class using the import created from the previous step.

```
from . import Movie
```

Create the search() Method

Create the search() method within the class. It accepts a search_term parameter that is a string. It returns a list of dicts. Within the method, make a request to the API using requests.get(). Add the provided exception handling around it as follows:

```
class TMDBMovieQuery:
    key = '...'
    search_url = '...'
    details_url = '...'


    def search(self, search_term: str) -> list[dict[str, str]]:
        search_dict = {}
        try:
            search_dict =
               requests.get(self.search_url.format(key=self.key,
                            title=search_term)).json()
        except requests.RequestException as rex:
            logger.error(
                       f'Request exception occurred: {rex.args}')

        return search_dict.get('results', [])
```

Create a TMDBMovieQuery Shortcut

Open the __init__.py file once again and add a shortcut entry into it for the TMDBMovieQuery class as shown:

```
from .query import TMDBMovieQuery
```

Import TMDBMovieQuery into Task2_1

Open the main file, task2_1.py, and import the TMDBMovieQuery class from the shortcut created in the previous step:

```
from movie import TMDBMovieQuery
```

Make a Movie Query

Within task2_1.py, instantiate a TMDBMovieQuery class object, prompt the user (using the click library) for a search term, and call the search() function created earlier:

```
logging.basicConfig(level=logging.INFO, format='%(message)s',
                    handlers=[...])

query = TMDBMovieQuery()
search_val = click.prompt('Movie title search phrase',
                          default='thor', show_default=True)
movie_list = query.search(search_val)
```

Display the Movie List

Taking the returned results from the previous step, display them. Optionally, you can color the output using the colorama library as shown:

```
query = TMDBMovieQuery()
search_val = click.prompt('Movie title search phrase',
                          default='thor', show_default=True)
movie_list = query.search(search_val)

if movie_list:
    for idx, movie in enumerate(movie_list, 1):
        print(colorama.Fore.RED,
              f'{idx} - {movie.get('title', '(no title)')}')
```

Get the User's Selection for Detailed Results

Referencing the previous output, prompt the user to enter a specific number to retrieve detailed results. Use the click library to prompt for input. Then, use that value by accessing the movie_list from step 10 and retrieve the dictionary for the movie selected. Get the 'id' of this movie as follows:

```
if movie_list:
    for idx, movie in enumerate(movie_list, 1):
        print(colorama.Fore.RED,
              f'{idx} - {movie.get('title', '(no title)')}')

    selected = click.prompt(text='Enter a number for details',
                            default=1, clamp=True,
                        type=click.IntRange(1, len(movie_list)))

    movie_id = movie_list[selected - 1].get('id')
```

Add get_movie() to TMDBMovieQuery

Return to the query.py file. Within the TMDBMovieQuery class, add a new method called get_movie() that accepts a movie_id string and returns a movie. This method makes a request to the API using the details_url but we'll have to insert the passed in movie_id into the url. Do this as follows:

```
def get_movie(self, movie_id: str) -> Movie:
    m = Movie()
    try:
        details = requests.get(self.details_url
                               .format(key=self.key, id=movie_id)).json()
        m = Movie(**details)
    except requests.RequestException as rex:
        logger.error(f'Request exception occurred: {rex.args}')
    return m
```

Render the Movie

Return to the task2_1.py file. Below the previous code you wrote, call the TMDBMovieQuery object's get_movie() method then render the movie by calling the movie's render() method.

```
if movie_list:
    for idx, movie in enumerate(movie_list, 1):
        print(...)
    selected = click.prompt(...)

    movie_id = movie_list[selected - 1].get('id')

    print(colorama.Fore.GREEN)
    m = query.get_movie(movie_id)
    logging.info(m.render())
```

Test out the Runtime

To test the __len__() magic method created earlier, pass the movie into a call to len() as shown:

```
print(colorama.Fore.GREEN)
m = query.get_movie(movie_id)
logging.info(m.render())
logging.info(f'Runtime: {len(m)}')
```

That's it! Fix any errors and test it out by running task2_1.py.

Task 2-2
A Repository Pattern

Overview

For this task, you will develop a repository pattern using abstract classes. This task takes the detailed movie data retrieved from the previous task and persists it into a database. The MovieStore class will represent the concrete class that inherits from the abstract Store class. The abstract class defines an add(), get(), and connect() method.

Create the Abstract Class and Methods

Open abstract.py in the task2_2_starter/app/store directory. Have the Store class inherit from ABC and add the three methods to the class as shown:

```
class Store(ABC):
    def __init__(self):
        ...

    def __enter__(self):
        return self

    def __exit__(self, typ, msg, tb):
        self.close()

    @abstractmethod
    def connect(self):
        pass
```

```
@abstractmethod
def add(self, model):
    pass

@abstractmethod
def get(self, id):
    pass

def close(self):
    ...
```

Create the Concrete Store Methods

Open **movie_store.py** in the same directory as store.py. Implement the three methods from the base class. In Python, it is legal to put type hints in these methods even if the base class did not. To reduce writing a lot of error handling and logging, the code that goes into each method is provided in the source file. You do not have to write all of this yourself from scratch. Your class should look like this:

```
class MovieStore(Store):
    db_url = Path(__file__).parents[4] /
                                        'resources/course_data.db'
    logger.debug(f'Using db location: {db_url}')

    def connect(self) -> sqlite3.Connection:
        return sqlite3.connect(MovieStore.db_url)

    def add(self, movie: Movie) -> int:
        try:
            cursor = self.conn.cursor()
            cursor.execute('INSERT INTO movies (title,
                 release_date, runtime, budget, revenue,
                 tagline) VALUES(?, ?, ?, ?, ?, ?)',
                (movie.title, movie.release_date, movie.runtime,
                 movie.budget, movie.revenue, movie.tagline))
            row_id = cursor.lastrowid
            logger.debug(f'Insert with id: {row_id}')
        except Exception as err:
            raise StoreException('Error adding movie.') from err

        return row_id

    def get(self, movie_id) -> Movie:
        try:
            cursor = self.conn.cursor()
            sql = f'SELECT title, release_date, runtime, budget,
                    revenue, tagline FROM movies WHERE id = ?'
            cursor.execute(sql, (movie_id,))
            results = cursor.fetchone()
            results_dict = {k:v for k, v in
                            zip(Movie.model_fields, results)}
            logger.debug(f'Movie retrieved with values:
                         {results_dict}')
        except Exception as err:
            raise StoreException('Error retrieving movie.')
                                                    from err

        return Movie(**results_dict)
```

Create Shortcut Imports

Open the **app/__init__.py**. Note: Be sure it is the correct __init__.py (there are several in this project)! Add the following shortcut import into it:

```
from .store import Store, MovieStore
```

Use the Store

Open the task2_2_starter/task2_2.py file next. Near the top (where indicated) add the following import to bring our classes in:

```
from app import Store, MovieStore
```

Then, within the provided persist() method, complete the method signature and implement the method body as shown:

```
def persist(store: Store, persist_obj: Any) -> int:
    with store:
        obj_id = store.add(persist_obj)
        logging.info(f'Object saved with id: {obj_id}')

        obj = store.get(obj_id)
        logging.info(f'Found object in db: {obj}')

    return obj_id
```

Call persist()

Near the bottom of the file (where indicated for step 5), add a **`click.confirm()`** call which prompts the user to save the object. Within the if-statement, call the persist method passing a new MovieStore instance and our movie, m, retrieved earlier. That's it—test it out!

```
print(colorama.Fore.GREEN)
m = query.get_movie(movie_id)
logging.info(m.render())
logging.info(f'Runtime: {len(m)}')
print(colorama.Fore.BLACK)

if click.confirm('Persist movie?'):
    persist(MovieStore(), m)
```

Task 3-1
A Self-Determined StringIO

Overview

In this task, you will create a StringIO object via inheritance that also knows what to do with itself. There are three options: it can be saved to a file, logged (to the console), or discarded (nothing happens with it). An Enum will be used to identify these states.

Import Resources

Import the Enum and StringIO classes.

```
import logging
import sys
from enum import Enum
from io import StringIO
from logging import Logger
from pathlib import Path
```

Create the Enum

As described in the overview, create an Enum with three states.

```
logging.basicConfig(level=logging.INFO,
          format='--> %(message)s',
          handlers=[logging.StreamHandler(stream=sys.stdout)])

class FlushStatus(Enum):
    FILE = 1                    # save to file
    LOG = 2                     # log it
    DISCARD = 3                 # discard it
```

Begin Building the FlushStringIO Class

Create the FlushStringIO class such that it inherits from the StringIO class. Provide an __init__() that takes the three arguments as shown. Save the status on the self object.

```
class FlushStringIO(StringIO):
    def __init__(self, status: FlushStatus, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.status = status
```

Add MultiMethods to the FlushStringIO Class

Within the FlushStringIO class, add three methods each called flush() that take differing parameters and are each decorated with @multimethod.

```
class FlushStringIO(StringIO):
    def __init__(self, status: FlushStatus, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.status = status


    @multimethod
    def flush(self, filename: str):
        with Path(filename).open(mode='wt',
                                 encoding='utf-8') as f:
            f.write(self.getvalue())

    @multimethod
    def flush(self, logger: Logger):
        logger.info(self.getvalue())

    @multimethod
    def flush(self):
        pass
```

Implement the close() Method

The close() method has been provided. Bring to life the provided close() method by uncommenting it and ensuring that it is indented within the class.

Check for any errors and test it out!

Task 3-2
Packaging

Overview

In this task, you will create a distributable Python package (in a wheel format). Optionally, you will install the resulting package as a library into your Python virtual environment.

Perform the following steps in succession:

1. Within PyCharm, open the .toml file in the project in the task3_2_starter/flushstringio directory.
2. In the [project] section, fill in the name field with "**flushstringio**".
3. Open a command (terminal) window. **`cd`** to the `student_files` directory.
4. Type **`venv\Scripts\activate`** (on Windows) or **`venv/bin/activate`** (on OS X) to activate the virtual environment.
5. **`cd`** to the `ch03_adv_language` directory next.
6. Run the command **`mypy task3_2_starter`**.
7. It should report to errors with the flush() method being a duplicate name. Optionally, you can fix this by rewriting the second and third flush() functions as follows:

```
@flush.register
def _(self, logger: Logger):
    logger.info(self.getvalue())

@flush.register
def _(self):
    pass
```

Run **mypy** again. It should have no errors this time.

8. **cd** to the `ch03_adv_language/task3_2_starter/flushstringio` directory next.

9. Build the distribution by running the command
   `python -m build`

10. cd to the `dist` directory next.

11. Run an HTTP server (on port 8000) with the following command:
    `python -m http.server –bind 127.0.0.1 8000`

12. In the `dist` directory create a folder called **`flushstringio`**. Then, move the .gz and .whl files into this folder.

13. In a new command (terminal) window (since the current window is now running a server), **`cd`** to the `student_files` directory again.

14. Activate the virtual environment (as done in step 4 earlier) (e.g., **`venv\Scripts\activate`**).

15. Run the command to pip install the new module:
    `pip install`
    `--index-url=http://127.0.0.1:8000/ flushstringio`

16. To test it out, within PyCharm, open temp.py in the student_files root directory and run this code:
17.

```
from flushstringio.flushstringio import FlushStatus,
FlushStringIO

with FlushStringIO(FlushStatus.FILE) as fs:
    fs.write('Test')
```

18. Optionally, uninstall the package after testing it:
    pip uninstall flushstringio

That's it!

Task 4-1
Copy Files Using shutil and filecmp

Overview

The goal of this task is simple: copy files from one directory to another. We'll compare the directories. Files in the src directory that are different from the dst directory will need to be copied.

As we've done previously, you will work from three files as follows:

1. Work from the **`files.py`** file in the **`task4_1_starter/utils/files.py`** file. Locate this in the **`ch04_automating`** directory. You will complete the **`copy_files()`** function.
2. Finish **`__init__.py`** with a shortcut import.
3. Finally, create the driver **`task4_1.py`** to invoke **`copy_files()`**.

Check Existence of Destination Directory

To copy files using the shutil module, we'll need to ensure the destination directory exists. Open task4_1_starter/utils/files.py. Locate the copy_files() function. Within the copy_files() function, write code to check that the destination directory exists. Also, raise an error if the src directory doesn't exist.

```
def copy_files(src, dst):
    src = Path(src)
    dst = Path(dst)

    if not src.exists():
        raise OSError('Source directory not found.')

    if not dst.exists():
        dst.mkdir()
```

Compare Directories

Using filecmp.dircmp(), compare the two directories. The results should be a diff of only what exists on the left side (src directory):

```
def copy_files(src, dst):
    src = Path(src)
    dst = Path(dst)

    if not src.exists():
        raise OSError('Source directory not found.')

    if not dst.exists():
        dst.mkdir()


    left_only = filecmp.dircmp(src, dst).left_only
```

Copy the Files

Iterate over the list of files on the left (src) side only. Use Path() / filename to join the src directory name and filename. Use shutil.copy2() to make a copy to the destination directory.

```
def copy_files(src, dst):
    src = Path(src)
    dst = Path(dst)

    if not src.exists():
        raise OSError('Source directory not found.')

    if not dst.exists():
        dst.mkdir()
```

```
left_only = filecmp.dircmp(src, dst).left_only

copied = []
for filename in left_only:
    fullpath = src / filename
    if fullpath.is_file():
        shutil.copy2(fullpath, dst)
        copied.append(filename)
```

Return copied from the function.

Create a Import Shortcut in __init__.py

Open the __init__.py and add a shortcut import into it.

```
from .files import copy_files
```

Import the New Function

Open task4_1.py and import the copy_files() function you related to the shortcut in the previous step.

```
from utils import copy_files
```

Invoke the copy_files() Function

Invoke the copy_files() function and display the results or the len() of the results.

```
results = copy_files(quotes1, quotes2)
print(f'{len(results)} files copied: {results}.')
```

Test it out!

Create delete_common()

This time, on your own, create a function called delete_common() that deletes common files from either the dst or src directory (depending on the value of a Boolean third parameter). This function will be nearly identical to copy_files(). You should wrap the src and dst, check if the two exist, use dircmp().common to see which files are in, iterate through the common files and call **.unlink()** on each one, which removes them. Return a list of the files removed.

Attempt this on your own and then compare your solution to the one presented on the next page.

```
def delete_common(src: str | Path, dst: str | Path, from_dst:
                  bool = True) -> list[str]:
    """
        Deletes common files from the dest (def.) or
        the source if from_dst = False.
    """
    src = Path(src)
    dst = Path(dst)

    if not src.exists():
        raise OSError('Source directory not found.')

    if not dst.exists():
        raise OSError('Source directory not found.')

    common = filecmp.dircmp(src, dst).common

    removed = []
    for filename in common:
        fullpath = dst/filename if from_dst else src/filename
        fullpath.unlink()
        removed.append(fullpath.name)
    return removed
```

Create a Shortcut for delete_common()

Return to the __init__.py and add a shortcut for the delete_common() function:

```
from .files import copy_files, delete_common
```

Invoke delete_common()

In task4_1.py, import delete_common() from the shortcut. Call the delete_common() function:

```
from utils import copy_files, delete_common

results = delete_common(quotes1, quotes2)
print(f'{len(results)} files deleted: {results}.')
```

Task 4-2

Generate a PDF, Send as Email Attachment

Overview

For this task, you must complete two parts: generate a PDF, using ReportLab, and send an email to a "dummy" SMTP server. To accomplish this task, we will need to complete and invoke two functions: generate_pdf() and send_email().

For this task, you will work from **`ch04_automating/task4_2_starter.py`**.

Start the SMTP "dummy" Server

From a command-line (terminal window), launch the server using:

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

or

```
python -m aiosmtpd -n -l localhost:1025      (>=Python 3.12)
```

Examine get_data()

There isn't much to do here. Examine the get_data() function. It is already written for you and should successfully return the data from the file. At the bottom of this file, the function is called. You can optionally print out the data if you wish to verify its structure.

Complete the generate_pdf() Function

Add the TableStyle object to the Table (hint: call setStyle()). Beneath that statement append the Table into the PDF doc_items. Finally, render (build) the PDF.

```
def generate_pdf(data: list[tuple], filename: str | Path):
    doc_items = []
    doc = SimpleDocTemplate(str(filename), pagesize=letter,
                            rightMargin=72, leftMargin=72,
                            topMargin=72, bottomMargin=72)

    table = Table(data, colWidths=(250, 200, 60))

    table_style_format = [
        ('ALIGN', (0, 0), (-1, 0), 'CENTER'),
        ('TEXTCOLOR', (0, 0), (-1, 0), green),
        ('BACKGROUND', (0, 0), (-1, 0), lavenderblush),
        ('ALIGN', (0, 1), (-1, -1), 'LEFT'),
        ('GRID', (0, 0), (-1, -1), 1, dimgray),
        ('BACKGROUND', (0, 1), (-1, -1),Color(0.94, 0.92, 0.90))
    ]

    ts = TableStyle(table_style_format)
    table.setStyle(ts)
    doc_items.append(table)
    doc.build(doc_items)
    logger.info(f'{filename} generated.')
```

Call generate_pdf()

At the bottom of the source file, call generate_pdf() passing the data and PDF filename into it.

```
pdf_file = Path(data_file.stem + '.pdf')
file_data = get_data(result)
generate_pdf(file_data, pdf_file)
```

Complete the send_email() Function

There are two pieces to finish within the send_email() function. These are described in a and b below:

a) At the location indicated as Step 5a. within the send_email() function, establish the Subject, From, and To fields for the email:

```
email_msg = MIMEMultipart()

email_msg['Subject'] = subj
email_msg['From'] = sender
email_msg['To'] = receiver

try:
    attachment = Path(attachment_fn)
    ...
```

b) Within send_email() at the location indicated as Step 5b., start the SMTP client, send your message, and quit (the server).

```
email_msg.attach(part)

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

logger.info('Email and attachment sent.')
```

Call send_email()

At the bottom of the source file, call send_email() passing the needed email parameters.

```
generate_pdf(file_data, pdf_file)

if send_email(sender='joe@example.com',
              receiver='dave@example.com',
              subj='Task4-2 with PDF Attachment',
              attachment_fn=str(pdf_file)):

    logger.info('Email sent.')
```

That's it--test it out!

Task 5-1
Subprocesses

Overview

In this task, you will launch the aiosmptd (or smtpd) server that was used in the previous task. However, for this task you will have the script launch the server automatically using a subprocess. We'll use psutil to kill the server.

Create the Subprocess Command List

Working with the strings at the top of the class, you must select the correct one based on the Python version you are using. Then parse (split) the string into a list of strings. Do this as follows:

```
def __init__(self, delay: int = 2):
    self.proc = None
    self.delay = delay
    command_str = self.aiosmtpd_cmd
                  if sys.version_info[1] >= 12 else self.smtpd_cmd
    self.command = [sys.executable, *command_str.split()[1:]]
```

Launch the Subprocess

Use Popen() to launch the subprocess. Pass in the command from step 1.

```
def start(self):
    logger.info(f'Starting server using {self.command}')
    self.proc = subprocess.Popen(self.command)
```

Kill the Process using psutil

Within the stop() method, kill the process using psutil.

```
def stop(self):
    if self.proc:
        proc = psutil.Process(self.proc.pid)
        proc.kill()
```

Instantiate the SMTPDManager

The SMTPDManager must be instantiated and then its start() method is called.

```
sm = SMTPDManager()
sm.start()

if send_email(...):
    logger.info('Email sent.')
```

Shut Down the *aiosmtpd* Server

Call the SMTPDManager's stop() method to shut down the server.

```
    if send_email(...):
        logger.info('Email sent.')

    sm.stop()

else:
    logger.info('Action canceled.')
```

That's it—test it out! Run your script directly. As part of the run process, it should start the server, send the email, and then shut down the email server.

Task 5-2
Subprocesses and Context Managers

Overview

In this task, we will convert the SMTPDManager class into a context manager by implementing the __enter__() and __exit__() method.

Add an __enter__() Method to the Class

Add the __enter__() magic method to the class. Have it call the start() method. The *with* control will start the server. There's no real need to **`return self`** since it's not used.

```
def __enter__(self):
    self.start()
    return self
```

Add an __exit__() Method to the Class

Similarly, add an __exit__() method. It merely calls stop() which shuts the server down at the end of the *with*-control.

```
def __exit__(self, typ, msg, tb):
    self.stop()
```

Implement the Context Manager

Replace the instantiation, start() and stop() calls with the with control as shown:

```
    generate_pdf(file_data, pdf_file)

    with SMTPDManager():
        if send_email(...):
            logger.info('Email sent.')
else:
    logger.info('Action canceled.')
```

That's it—test it out! Run your script directly again.

Task 5-3
Multiprocessing Using Ray

Overview

In this task, you will convert the simple but working example to perform asynchronously in a Ray (local) cluster .

Mark the get_data() Function

Use the ray.remote() decorator and mark the work function, get_data(),

```
@ray.remote
def get_data(url: str) -> str:
```

Queue the Jobs

In a loop, call the remote() method of each function providing the url to make a request to. The returned results will eventually be valid, but initially, they are not.

```
print('Starting jobs...')
obj_refs = [get_data.remote(url) for url in tasks]
```

Retrieve the Results

Call the get() blocking function. When the results are ready, they will be returned.

```
obj_refs = [get_data.remote(url) for url in tasks]
results = ray.get(obj_refs)
```

Test it out!

Task 6-1
Asyncio

Overview

In this task, you will take the multi-threaded script from the previous chapter and make it an asyncio version. Begin by locating and opening the task6_1_starter.py file.

Mark the Function as async

Modify the get_data() function two ways:

1. Mark it as async.
2. Rename it to async_get_data() just to make it easier to understand.

```
import asyncio

import requests
from bs4 import BeautifulSoup


async def async_get_data(url: str) -> str:
```

Invoke requests.get() in a Separate Thread

Within the async_get_data() function, invoke requests.get() by first converting it to run in a separate thread:

```
async def async_get_data(url: str) -> str:
    try:
        resp = await asyncio.to_thread(requests.get, url)
        text = resp.text
```

Build the main() Function

Create the main() function and mark it as async. Within the function, call asyncio.gather(). Add await to the front of the function call—this schedules the provided coroutines into the event loop.

```
async def main():
    print('Starting jobs...')
    results = await
         asyncio.gather(*[async_get_data(url) for url in tasks])
    for result in results:
        print(result)
```

Call run(), Test It Out

Call the asyncio run() method passing a call to main() into it. That's it—test it out!

```
asyncio.run(main())
```

Task 7-1
FastAPI and Logistic Regression

Overview

In this task, you will edit two files within the task7_1_starter folder. The first file is create_model.py. It is responsible for doing exactly what its name says, creating the model. The second file is our FastAPI server. You will complete it and then test it out by visiting the auto-generated Swagger UI interface in the browser at localhost:8000/docs.

Display the DataFrame

Open **`task7_1_starter/create_model.py`**. It's almost too easy, but we got to start somewhere!

```
personal_data['tsize_enc'] = personal_data.tsize.cat.codes
print(personal_data)
```

Create the Logistic Regression Object

Use Scikit-learn to create the logistic regression. Run a fit on the features_matrix and response object.

```
features_matrix =
        personal_data[['weight', 'height', 'tsize_enc']].values
response = personal_data.gender_enc.values

lgreg = LogisticRegression()
lgreg.fit(features_matrix, response)
```

Run a Predict and Score It

Use the model's predict() method against the entire dataset. We didn't break it into a train/test set due to its small size.

```
lgreg = LogisticRegression()
lgreg.fit(features_matrix, response)

y_pred = lgreg.predict(features_matrix)
print(lgreg.score(features_matrix, response))
```

Persist the Model

Use the joblib dump() method passing the model and filename.

```
joblib.dump(lgreg, 'model.joblib')
```

Create the Pydantic Model

Create the Pydantic Model. It will be used to serialize HTTP body-provided data and do validation.

```
class Person(pydantic.BaseModel):
    weight: int = 0
    height: float = 0.0
    gender: int = 0
    t_size: int = 0
```

Create FastAPI Server and Load Model

Instantiate the FastAPI server object and use joblib to load the model.

```
class Person(pydantic.BaseModel):
    weight: int = 0
    height: float = 0.0
    gender: int = 0
    t_size: int = 0
```

```
app = FastAPI()
joblib_file = open('model.joblib', 'rb')
model = joblib.load(joblib_file)
```

Create the Route

Use the app.post() decorator. Supply '/person/predict' as the path argument to the decorator. Create a function called predict() that accepts a person: Person parameter. Perform a predict() and return a result.

```
app = FastAPI()
joblib_file = open('model.joblib', 'rb')
model = joblib.load(joblib_file)

@app.post('/person/predict')
def predict(data: Person):
    prediction =
        model.predict([[data.weight, data.height, data.t_size]])
    return {'result': 'male' if prediction[0] else 'female'}
```

Create FastAPI Server and Load Model

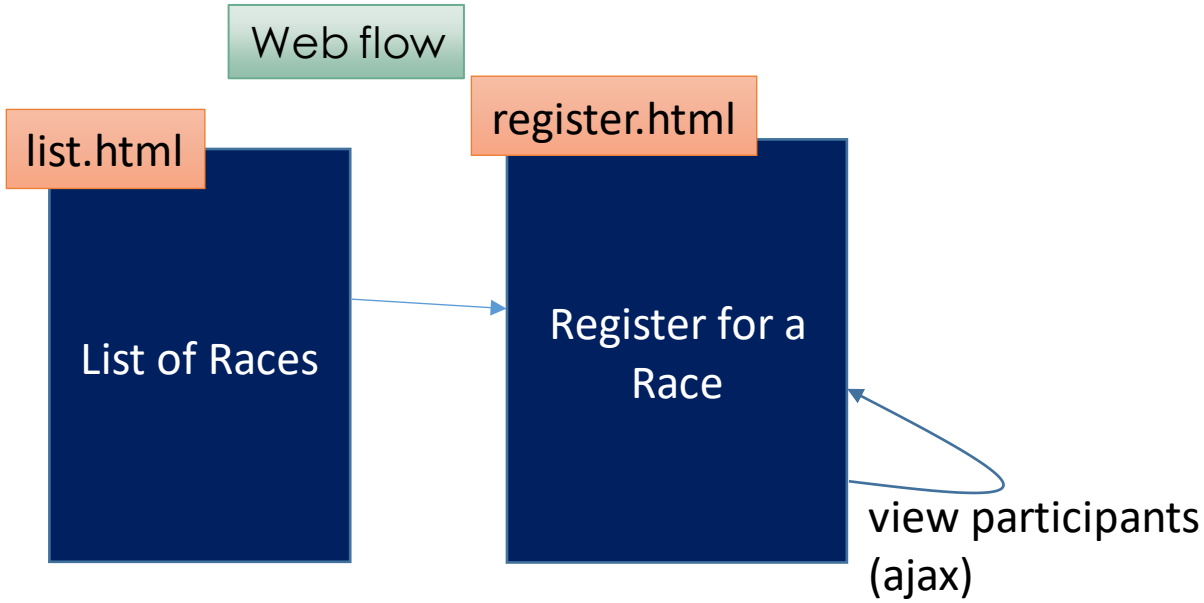
Instantiate the FastAPI server object and use joblib to load the model.

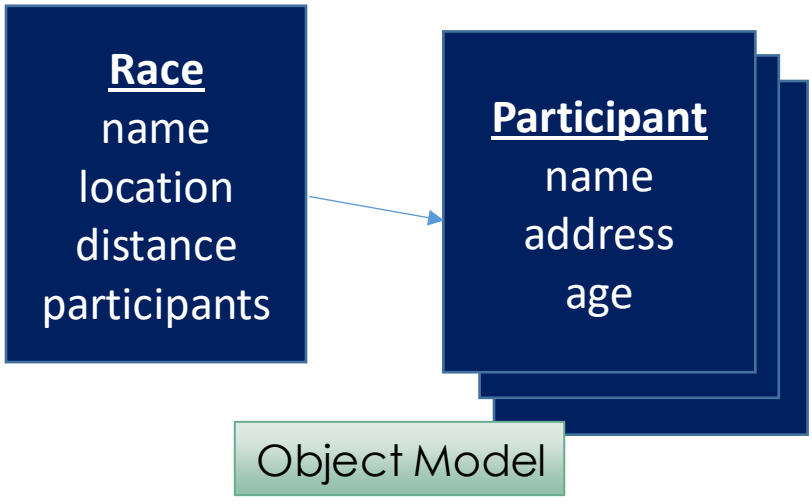
```
uvicorn.run(app, host='127.0.0.1', port=8000)
```

Task 8-1
Web Application Development Using Django

Overview

This exercise will allow for installation, installation test, configuration, development, and application test of a web-based application created using Django. This app will create the project from scratch, the database tables, models, and views.





Set up Django and Create a Project

If you haven't yet done so, install Django using [sudo] **pip install Django**. If you have an older (< 2.0) version, pip uninstall Django first and then install Django with the latest version. *This exercise was created with Django version 4.1.3 but should still run within the latest version (5.0.1 at the time of this writing)*

Check that your path can see the Django admin utility by typing the command: **django-admin** from a command line. Did it recognize your command? If it did not, the likely reason is that you are not running your Python virtual environment. To activate the virtual environment, from the student_files directory type:

venv\Scripts\activate (on Windows), or

venv/bin/activate (on OS X)

Check it again by typing:

```
django-admin --version
```

If it worked, you may proceed.

View the Finished Web App First

Before building the app, first view the completed version of the app you are attempting to build. Do this by changing (cd) to the ch08_django\solution\racer folder and then typing:

- **cd student_files\ch08_django\solution\racer**
- **python manage.py runserver**

Open a browser and browse to http://localhost:8000. This is the app you are trying to build.

Once you have tested the app, shut your server down (usually hit CTRL-C a couple of times in your open terminal window where the server is currently running.

Build the Initial Project Structure

Begin by changing to the root of the student_files directory. Then create the project as discussed in the slides.

- **`cd <location_of_student_files>`**
- **`django-admin startproject racer`**

Within PyCharm, the "racer" folder should automatically appear. Right-click on this folder and choose **Mark Directory As** > **Sources Root**. The folder should turn blue.

Finally, create the app within the project directory using the following command:

- **`python manage.py startapp racerapp`**

Edit the **racer/settings.py** file by adding **"racerapp"** to the **INSTALLED_APPS** section as shown here:

```
INSTALLED_APPS = [
    "django.contrib.admin",
```

```
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "racerapp"
]
```

Establish the Database Models

Next, we'll create the Race and Participant classes needed for the app. To save time, these have been provided for you in the **ch08_django/starter/models.txt** file. Copy these into the **racerapp/models.py** file. For reference, the code below is what you should have copied:

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

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

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


class Race(models.Model):
```

```
id = models.AutoField(primary_key=True)
name = models.CharField(max_length=100)
location = models.CharField(max_length=255)
distance = models.FloatField()

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

Migrate Models to the Database

Next, we'll establish the tables and data within our database using the models from the previous step.

Perform the following commands (in bold below):

- **python manage.py migrate**
  (creates the initial app and admin tables)

- **python manage.py makemigrations racerapp**
  (creates the initial migration file, 0001_initial.py that defines the Race and Participants tables)

- **python manage.py migrate racerapp**
  (Creates the Race and Participant tables in the DB defined from the migration file previously created)

You can optionally view the SQL that was used to generate the tables in the database using the following command:

```
python manage.py sqlmigrate racerapp 0001
```

Add Data to the Database, View Admin Interface

Next, we'll establish some race and participant data in the database. Then, we'll view this data through the provided administrative tool. Execute the following commands:

```
python manage.py shell
```

```
>>> import racerapp
```

```
>>> from racerapp.models import Race, Participant
```

```
>>> r = Race(name='Chicago Marathon',
             location='Chicago, IL', distance=26.2)
```

```
>>> r.save()
```

```
>>> p = Participant(name='John Smith',
            address='123 Main St.', age=35, race=r)
```

```
>>> p.save()
```

```
>>> Participant.objects.all()
```

```
>>> Race.objects.get(name='Chicago Marathon')
```

Open **racerapp/admin.py** and add the following to allow web-based configuration of models:

```
from racerapp.models import Race, Participant

admin.site.register(Race)
admin.site.register(Participant)
```

Exit the Django shell by typing exit().

Next, create a superuser that can view and access the admin interface.

```
python manage.py createsuperuser
```

username: admin

email address: admin@fake.com

password: admin_password

Open a web browser and browse to **localhost:8000/admin**. Log in using the credentials created above.

To the right of **Races**, click on **+Add** race:

Add a race with the following data:
Name: **BolderBOULDER**
Location: **Boulder, CO**
Distance: **6.2**

Click save.

Click the Racerapp link at the top (next to Home).

Click the "**+Add**" button next to **Participants** and click to add a participant. Fill in the name, address, age, and select a race. Click to save the participant.

Create the Views and Templates

Open the racer/urls.py file and add the following:

racer/urls.py

```
from django.contrib import admin
from django.urls import include, path
from django.views.generic.base import RedirectView

import racerapp.urls

urlpatterns = [
    path("admin/", admin.site.urls),
    path("races/", include(racerapp.urls)),
    path("", RedirectView.as_view(pattern_name="race_list"))
]
```

To save time, **copy the urls.py** from the ch08_django/starter file into the racerapp directory. *Note: don't overwrite the urls.py in the racer folder! Be sure to place this file into the **racerapp** folder.* This file looks like the following:

racerapp/urls.py

```
from django.urls import re_path

from racerapp import views

urlpatterns = [
    # more paths placed here in a moment
    re_path(r'^$', views.list_races, name='race_list')
]
```

There is nothing to change here. Simply add starter/urls.py into racerapp.

Replace the **views.py** file in the racerapp directory with the **views.py** in the **ch08_django/starter** directory. This new file looks like the following:

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

from racerapp.models import Race, Participant


def list_races(request):
    races = Race.objects.values()
    return render(request, 'list.html', {'races': races})
```

Finally, **copy the *templates* directory** from ch08_django/starter into the racerapp directory.

Restart your server by going to the terminal where it should be running and pressing CTRL-C. Start the server again (**python manage.py runserver**).

Test it out--it should work!

Open a browser and browse to **http://localhost:8000/**

It should redirect you to `http://localhost:8000/races` and you should see the list of races.

Render the Registration Page

Copy the 3 mappings from *ch08_django/starter/remaining_paths.txt* into the ***racerapp/urls.py*** **BEFORE** the existing one, as shown:

```
from django.urls import re_path

from racerapp import views

urlpatterns = [
    re_path(r'register/(?P<racename>[^/]+)', views.register,
            name='racerapp_register'),
    re_path(r'submit/', views.submit,
            name='racerapp_submit_reg'),
    re_path(r'participants/(?P<racename>[^/]+)',
            views.participants, name='racerapp_get_participants'),
    re_path(r'^$', views.list_races, name='race_list')
]
```

Copy the 3 views (functions) from ***remaining_views.txt*** into ***racerapp/views.py***.

Copy the **static** folder into the racerapp directory.

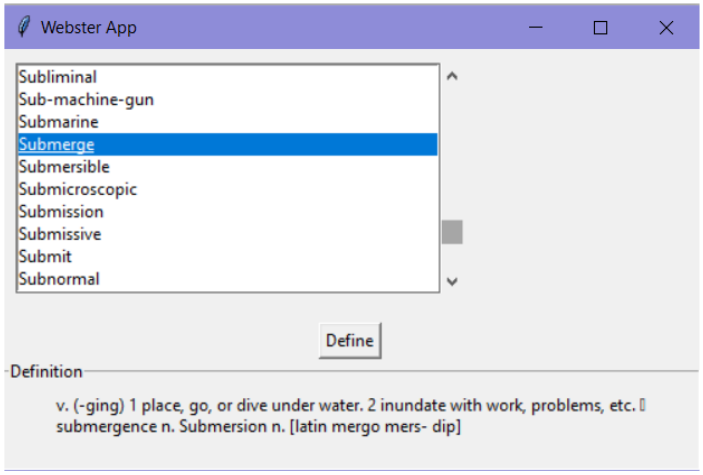
Test out the registration by browsing to http://localhost:8000 and then clicking on a race link. You should be able to register new participants and then view the list of participants!

Test it out! Everything should be working now.

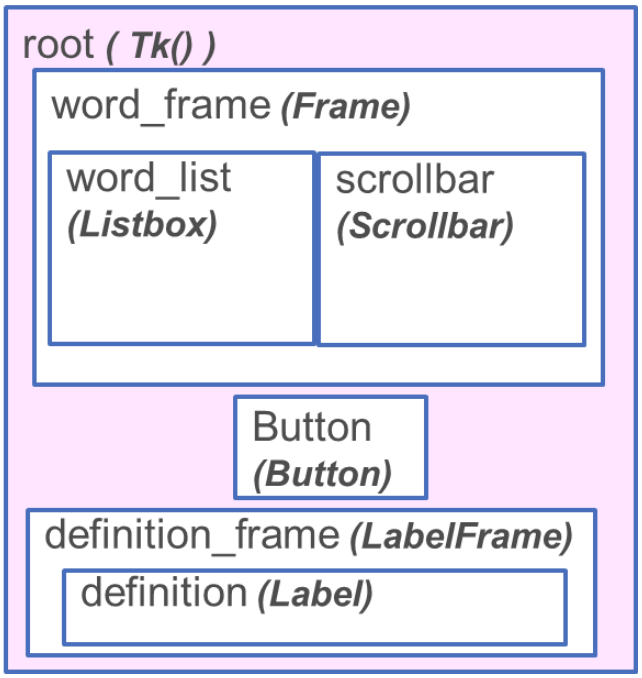
Task 9-1 *(Appendix A)*
Creating a GUI with Tkinter

Overview

This exercise builds the *Webster App* using the Tkinter framework within the Python Standard Library. The app looks like the following:



The layout of the widgets used in this exercise as as follows:



Open the Starter File, Build the Frame

Look over the working file:
ch09_apdx_a_gui/task9_1_starter.py.

We will build the various widgets over the next few steps. Refer to the diagrams on the previous page to get a feel for which widget we are creating. We'll begin with the word_frame.

The word_frame (Frame) widget will contain the Listbox and Scrollbar. Build it as follows (at the location marked as Step 1 in the source file):

```
word_frame = tk.Frame(master=root, padx=10, pady=10)
word_frame.pack(side='top', anchor='w')
```

Create the Listbox, Insert It into the Frame

The Listbox will contain a list of the words. The list of the words was already obtained in the earlier line:

```
webster = Webster.read_data()
```

We must now instantiate the Listbox and then iterate over our words and insert them into the listbox. Do this as follows:

```
word_list = tk.Listbox(master=word_frame, width=50)

for idx, item in enumerate(webster.keys()):
    word_list.insert(idx, item)

word_list.pack(side='left', fill='both')
```

Create the Scrollbar, Make it Work with the Listbox

A Scrollbar must be attached to the Listbox. Here, we'll instantiate a scrollbar and place it on the right side of the Listbox. Then, we'll attach the interaction of the scrollbar to make the Listbox change.

```
scrollbar = tk.Scrollbar(master=word_frame)
scrollbar.pack(side='right', fill='both')
word_list.config(yscrollcommand=scrollbar.set)
scrollbar.config(command=word_list.yview)
```

Create the Label and LabelFrame

The Label holds the word definition and the LabelFrame holds the Label (just for ease of positioning). Create and lay out both of these widgets as follows:

```
definition_frame = tk.LabelFrame(master=root,
                                 height=200, borderwidth=1,
                                 text='Definition')
definition_frame.pack(side='bottom', fill='x')

definition = tk.Label(master=definition_frame,
                      wraplength=470, justify='left')
definition.pack(expand=True, padx=5, pady=5,
                side='left', fill='both')
```

Create the Button

The button will be clicked after a word has been selected from the Listbox. And event handler for the button will be set up in the next step, however, for now, define the command= parameter and point it to our (as yet, unfinished) function, get_selection(). Do this as follows:

```
button = tk.Button(text='Define',
                   command=get_selection)
button.pack(side='bottom')
```

Add a Button Press Event (Command) Handler

The following code should be placed inside the get_selection() function after removing the **pass** statement.

The code below gets the index of the currently selected item, then it uses that to get the word that is selected. We take that word and look it up in our webster dict object. Finally, we update the Label widget (definition). That's it!

```
idx = word_list.curselection()
if idx:
    idx = idx[0]
```

```
word = word_list.get(idx)
definition.configure(text=webster[word])
```

Test it out!

That's it! Fix any typos and test it out by running the script. Any failures before the GUI appears may be due to the student_files directory not being on the sys.path variable. This can happen if you do not run this within PyCharm. Either run it within PyCharm or add the student files directory to the PYTHONPATH environment variable.