



Advanced Python Programming



MASTERING PYTHON
Getting Started with Python

Overview of Python

What is Python?

- Python is a general purpose, high-level, interpreted programming language created by Guido van Rossum in February 1991
- Currently, it is maintained by the Python Software Foundation
- Python is dynamically-typed and strongly-typed
- Python is multi-paradigm programming language
 - Object-oriented, Structured
 - Functional (some support)
 - Aspect-oriented (some support)
- For many developers, Python is a first language

Source: [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))

Overview of Python

What is Python?

- Learning the history of a language and the common philosophies of the community around it help developers to better use the language and to correctly follow the accepted best practices
- This concept of correctness and best practices is especially important when first learning a language to avoid common pitfalls
- To understand and connect with the Python community, the developer must be aware of the concept of something (especially code) being "pythonic"

Overview of Python

What is Python?

- The community drives the meaning of what "pythonic" means, and its principles should be followed when writing code
- A list of 20 "pythonic" principles is captured in the "Zen of Python"
- Zen - contemplation of something's essential nature to the exclusion of everything else¹

1. <http://www.dictionary.com/browse/zen>

Overview of Python

What is Python?

- The "Zen of Python" is the essential nature of Python
- Among the set of 20 principles, are the following aphorisms
 - Beautiful is better than ugly.
 - Explicit is better than implicit.
 - Simple is better than complex, readability counts.
 - Special cases aren't special enough to break the rules.
 - Errors should never pass silently.
 - There should be one — and preferably only one — obvious way to do it.
- To read the whole list, run the command 'import this' from the Python REPL

Source: https://en.wikipedia.org/wiki/Zen_of_Python

Overview of Python

What is Python?

- In addition to the "Zen of Python", Python comes with a style guide
- The name of the style guide is PEP 8
- PEP stands for Python Enhancement Proposal
- The style guide can be accessed from here:
 - <https://www.python.org/dev/peps/pep-0008/>
- Following the style guide is a best practice and helps to write "pythonic" code
- Using an editor with a Python linter configured for PEP 8 makes following the style guide much easier
- Limitations of PEP 8 – primarily focuses on code syntax, not construction of algorithms to solve problems

Overview of Python

What is Python?

- The concept of "pythonic", the Zen of Python, and the PEP 8 style guide are a big reason why Python is popular especially with new programmers or programmers who only code as tool to their primary occupation such as science or engineering
- In other languages, such as C or JavaScript, the emphasis is not on readable code
- Sure, developers want readable code in all languages, but most other languages do not have a this strong ethos dedicated to writing code that is readable and supportable

Overview of Python

What is Python?

- In the worst cases, complex but cool code is the norm making it hard for new developers or less experienced developers to understand what is going on
- In other languages, readability and the style guide is the last step, rarely taken, in Python it's the first step
- Python is known for its use with science, machine learning, etc... In these disciplines, programming is a tool not the primary purpose (unlike software development)

Overview of Python

What is Python?

- Unlike C-style languages, Python does not use curly braces to denote code block, instead it uses indentation which means whitespace is significant
- Python rejects the more cryptic syntax of Perl and Perl's philosophy the there are many ways to do something, and prefers a more readable syntax and an approach of having one obvious (and best) way to do something
- All data is an object, there are no primitives
- Because all data is an object, a variable does not have a type, but the data it points to is typed (all data is stored on the heap, not the stack)
- The type a variable points to can change at run-time
- Python data is strongly-typed, there is very little implicit type coercion

Source: [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))

Overview of Python

What is Python?

- Python is interpreted, but several options exist for compilation
 - Portions of programs can be written in C and compiled and called from their Python program
 - A Python program itself can use Just-In-Time (JIT) Compilation using PyPy
 - Cython can be used to cross-compile a Python program to C
 - Jython compiles to Python to Java bytecode
 - IronPython does for .NET what Jython does for Java

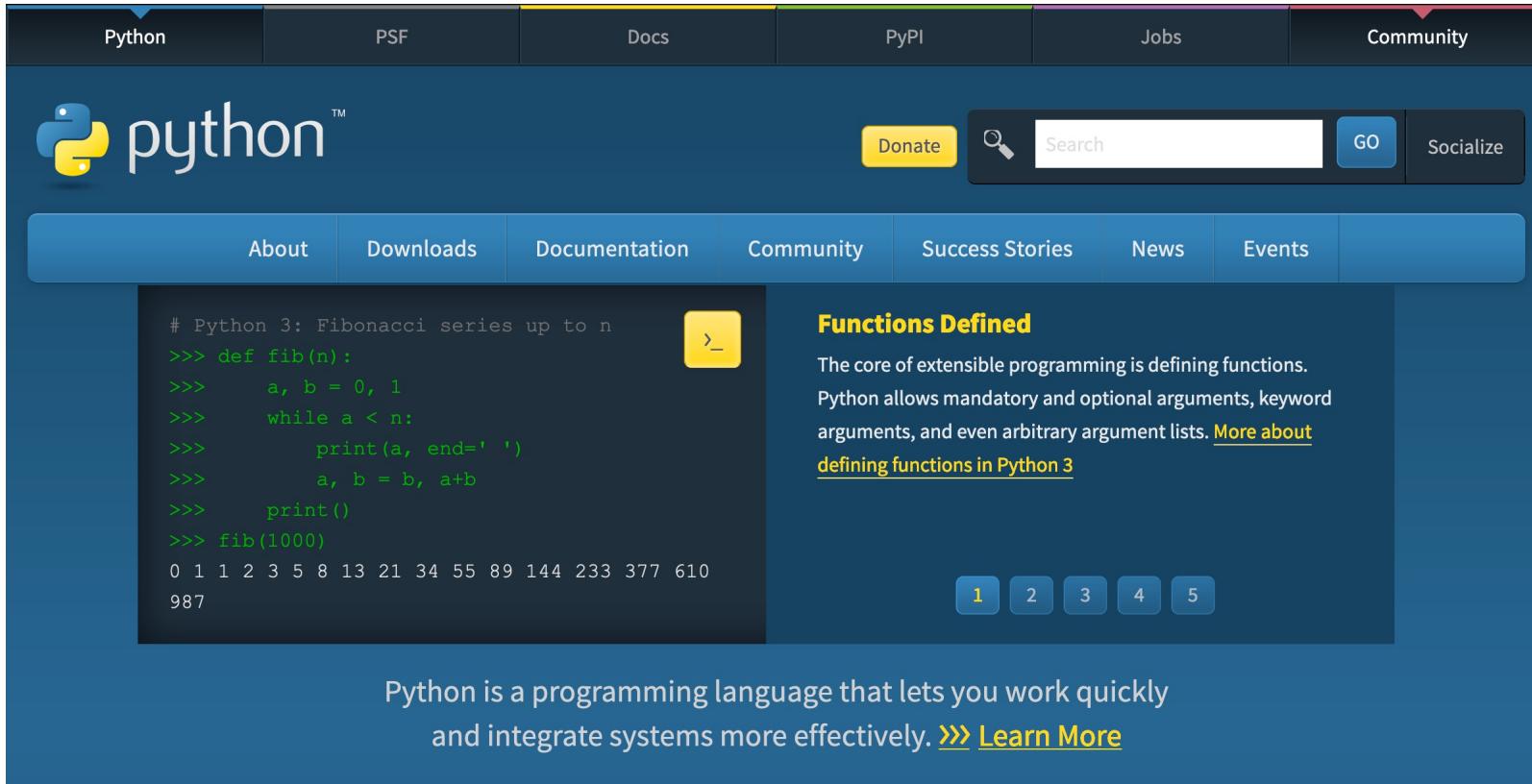
Overview of Python

What is Python?

- Python runs on all major platforms including Windows, Mac and Linux
- Python is an open source, and the source code can be downloaded for any platform for which there is not a distribution
- Two major versions of it are supported: 2.7 and 3
- Python 3 should be used for all new projects, but because of the extensive code base for version 2.7, it was supported until 2020
- Downloading and installing Python on Mac and Windows is covered in the section

Overview of Python

What is Python?



The screenshot shows the official Python website homepage. The header features a dark blue navigation bar with tabs for "Python", "PSF", "Docs" (which is highlighted), "PyPI", "Jobs", and "Community". Below the header is a large banner with the Python logo and the word "python" in white. To the right of the banner are buttons for "Donate", a search bar with a magnifying glass icon, a "GO" button, and a "Socialize" link. A secondary navigation bar below the banner includes links for "About", "Downloads", "Documentation", "Community", "Success Stories", "News", and "Events". The main content area contains a code snippet demonstrating a Fibonacci series generator in Python 3:

```
# Python 3: Fibonacci series up to n
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
987
```

To the right of the code, a section titled "Functions Defined" explains the core of extensible programming. It states: "The core of extensible programming is defining functions. Python allows mandatory and optional arguments, keyword arguments, and even arbitrary argument lists." It includes a link to "More about defining functions in Python 3". At the bottom of the content area, there are five numbered buttons (1, 2, 3, 4, 5) and a call-to-action message: "Python is a programming language that lets you work quickly and integrate systems more effectively. [»» Learn More](#)".

Screenshot: Python Home Page (<https://www.python.org/>)

Overview of Python

Getting Packages

- Like other programming language platforms, Python comes with a full package management system
- PyPI (Python Package Index) is the official package management tool for Python
- The package management application `pip` is used to manage packages
 - Installed packages can be listed
 - Packages can be installed and uninstalled
 - Packages can be upgraded

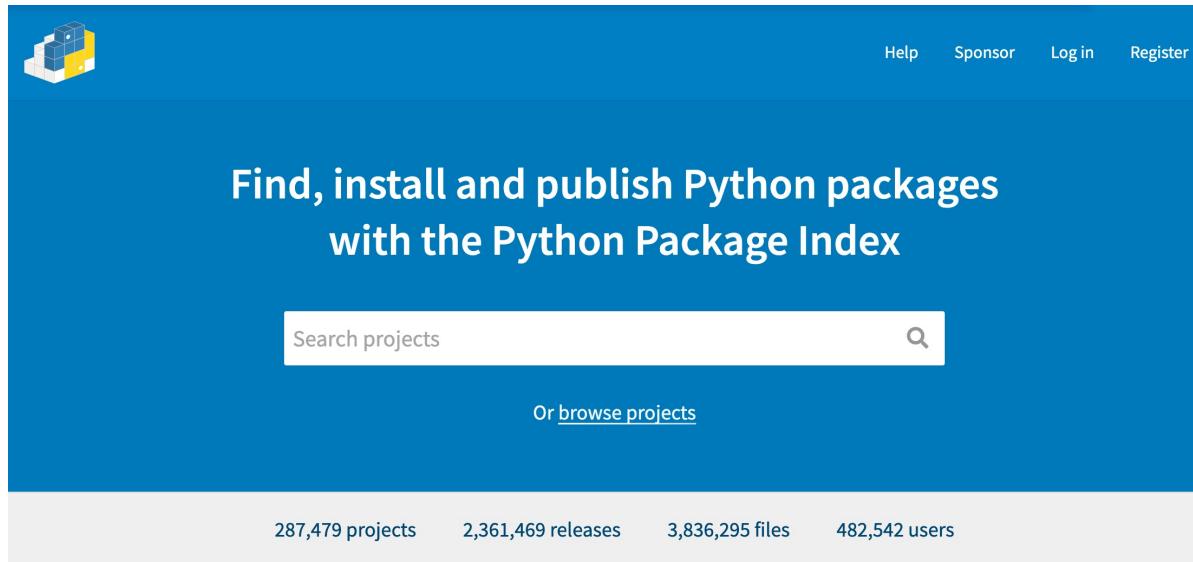
Overview of Python

Getting Packages

- Packages are stored globally and per user
 - Global packages are stored as part of the Python installation
 - User packages are stored within the user's home folder path
- Package dependencies for projects are stored in a `requirements.txt` file

Overview of Python

Getting Packages



The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages ↗](#)

Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI ↗](#)

Screenshot: PyPi Home Page (<https://pypi.org/>)

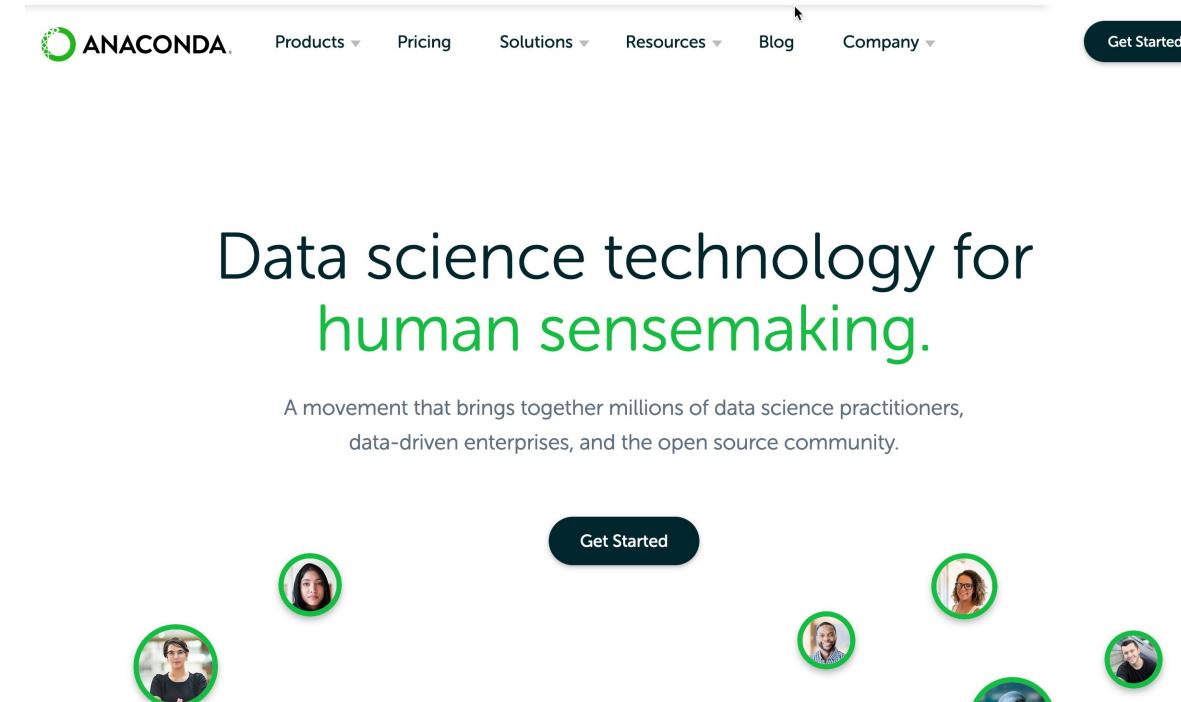
Overview of Python

Python and Data Science

- While this course focuses on Python programming in general, Python itself is very popular in the scientific programming community
- There is a specialized distribution of Python named Anaconda
- Anaconda is a specialized distribution of Python and PyPI packages related to data science
- Anaconda is free to use, but also provides commercial support for enterprises
- Some of the biggest corporate names in data science are part of the Anaconda project
- Anaconda can be downloaded for Windows, Mac, Linux
- Anaconda is the quick and easy way to get started with Python for Data Science applications

Overview of Python

Python and Data Science



Screenshot: Anaconda Home Page (<https://www.anaconda.com/>)

Overview of Python

Editors

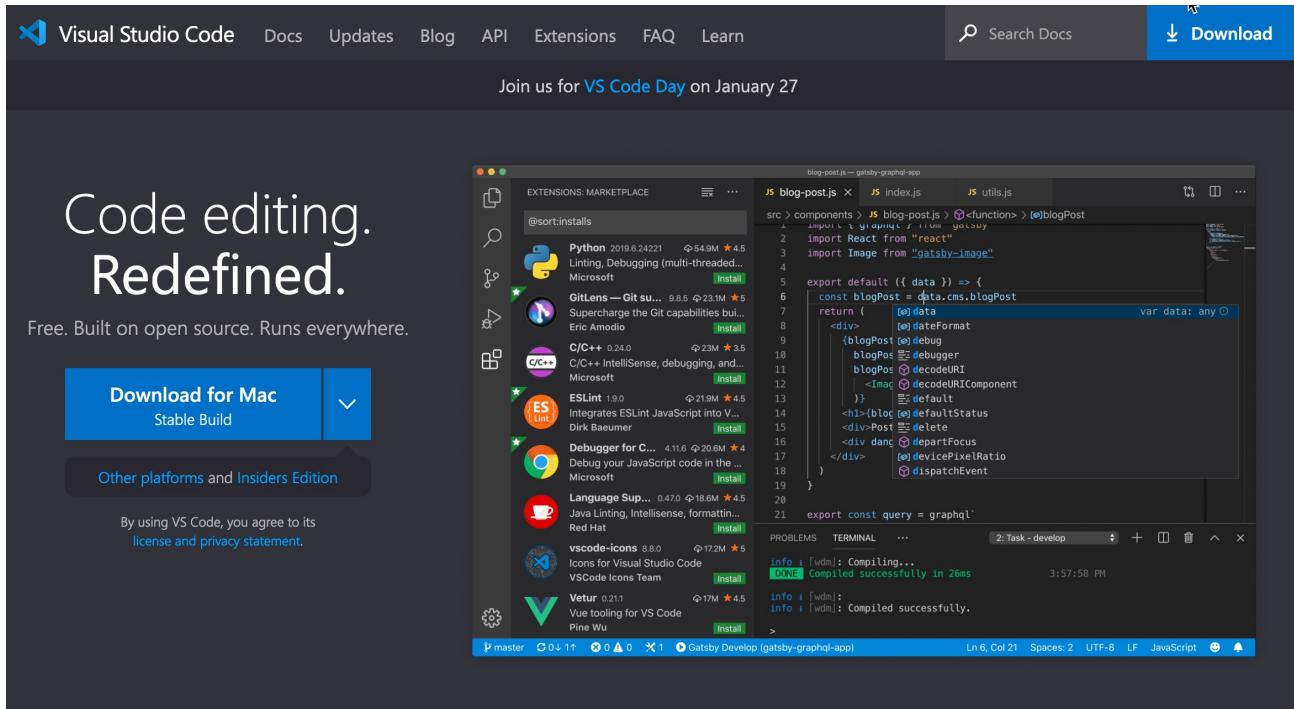
- There are many editors which natively support Python or support it through extensions
- Popular editors include:
 - PyCharm (JetBrains - Community and Professional Editions)
 - Visual Studio Code w/ Python Extensions (Microsoft)
 - Visual Studio (Microsoft - Community and Paid Editions)
 - Atom (GitHub), Sublime
 - VI, Emacs
- PyCharm, Visual Studio Code and Visual Studio will be demonstrated for doing Python development and debugging

Overview of Python Editors



Screenshot: PyCharm Home Page (<https://www.jetbrains.com/pycharm/>)

Overview of Python Editors



IntelliSense



Run and Debug



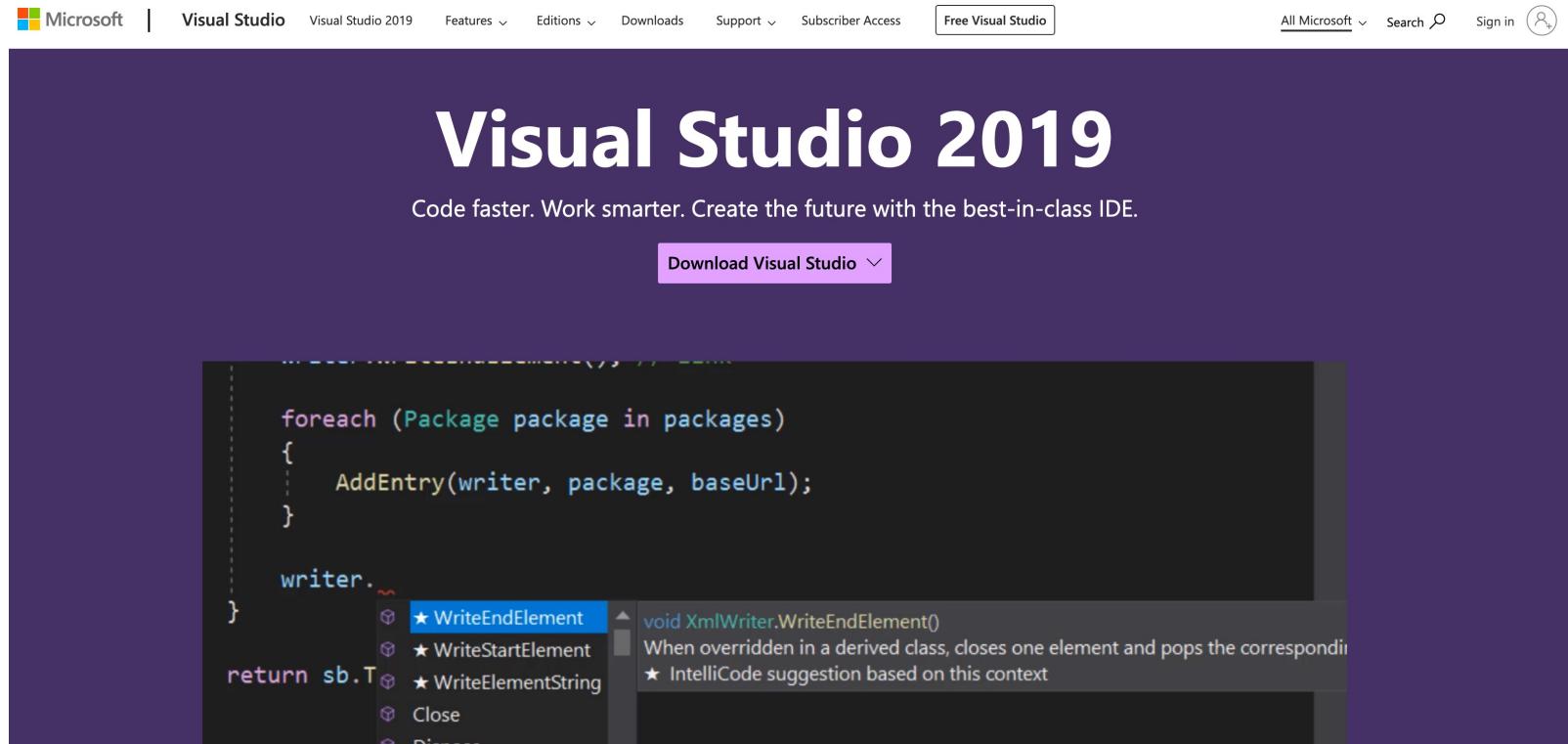
Built-in Git



Extensions

Screenshot: Visual Studio Code Home Page (<https://code.visualstudio.com/>)

Overview of Python Editors



Screenshot: Visual Studio Home Page (<https://visualstudio.microsoft.com/vs/>)

Getting Python

Installing Python

- This course will be using Python 3, but will cover how to use Python 3 with Python 2 on the same machine
- There are several ways to install Python
 - Official Python Installers
 - Third Party Package Managers
 - HomeBrew - <https://brew.sh/> (Mac)
 - Chocolatey - <https://chocolatey.org/> (Windows)
 - Various Linux Package Managers
 - Build from Source Code
- We will discuss Windows and Mac

Getting Python

Installing Python on Mac

- Mac computers come with Python 2 pre-installed, its located at:
 - `/usr/bin/python`
- This version of Python should not be used for development purposes, its an older version of 2 (not the latest version of 2)
- The easiest way to install Python is to download the package installers from the official Python web site
 - <https://www.python.org/downloads/>
- Generally, down and install both version 2 and version 3
- Let version 2 be the default version in the system path, and run version 3 with the command named '`python3`'

Getting Python

Installing Python on Mac

- Having version 2 as the default can make it easier for other tools which use Python 2 to find it
- For Python coding, the editor will be configured with a specific Python interpreter, it will not rely upon the path environment variable
- After installing both version 2 and version 3 packages, Python will be installed in the following folders:
 - /Library/Frameworks/Python.framework/Versions/2.7/bin/python
 - /Library/Frameworks/Python.framework/Versions/3.6/bin/python3
- Each will have a symlink from /usr/local/bin:
 - /usr/local/bin/python (points to version 2)
 - /usr/local/bin/python3 (points to version 3)

Getting Python

Installing Python on Mac

- From the command line, Python 2 can be invoked by running `python`, and Python 3 can be invoked by running `python3`
- To install packages for Python 2, the `pip` command is used
- To install packages for Python 3, the `pip3` command is used
- To see the paths where Python for each version is installed (including where global and user packages will be stored) run the following commands from the terminal:
 - For Python 2, `python -m site`
 - For Python 3, `python3 -m site`
- By default, all user packages are installed in `/Users/<username>/Library/Python`

Getting Python

Installing Python on Windows

- Windows does not come pre-installed with Python
- The Python website provided official installers for Windows, in 32-bit and 64-bit flavors
- Download the installers for both the latest version 2 and the latest version of 3
- Install version 2, and have it added to the path to be the default Python version
 - Making version 2 the default will make it easier for other tools on your system to use the older version of Python
- Install version 3, do not configure it to add environment variables

Getting Python

Installing Python on Windows

- A cool thing about the Python installers for Windows is they come with a Python launcher application
- The Python launcher allows the developer to easily specify the version of Python to run

Getting Python

Installing Python on Windows

- To run Python 2, open a command prompt and run `python`
- To use the Python Launcher:
 - To run Python 2, `py -2`
 - To run Python 3, `py -3`
- With the launcher, the system path is not used to select the python executable, and no long path names to the executable have to be supplied
- To run the `pip` package installer for each version, run the following commands
 - To install packages for Python 2, `py -2 -m pip install <package name>`
 - To install packages for Python 3, `py -3 -m pip install <package name>`

Getting Python

Installing Python on Windows

- To see the folder configuration for each Python version on Windows, run the following commands:
 - For Python 2, `py -2 -m site`
 - For Python 3, `py -3 -m site`

Getting Python

Verifying Installation

- To verify the Python version for commands on Mac, run the following commands in a terminal window and verify the output:

```
~&nbsp; $ python -V  
Python 2.7.14  
~&nbsp; $ python3 -V  
Python 3.6.3
```

Command Line: Verify Python Installation and Versions on macOS

Getting Python

Verifying Installation

- To verify the Python version for commands on Windows, run the following commands in a command prompt and verify the output:

```
C:\>python -V  
Python 2.7.14  
C:\>py -2 -V  
Python 2.7.14  
C:\>py -3 -V  
Python 3.6.3
```

Command Line: Verify Python Installation and Versions on Windows

Python Interactive Mode

Writing and Running Python Code

- Python code can be executed from interactive mode, and from a Python script file
- To run Python in interactive mode, run the Python interpreter without a file name
- To run a Python script, run the Python interpreter and passed the file name of the script as a command line argument
- Interactive mode is known as a REPL – Read, Evaluate, Print, Loop
- To exit interactive mode, run the exit function 'exit()'

Python Interactive Mode

Writing and Running Python Code

- The default Python interactive environment is not very useful, but there are packages which can be installed to make it better
- One such package is IPython, IPython (and fork of the project named Jupyter Notebook) is hugely popular with the scientific Python community

Python Interactive Mode

IPython

- To install IPython, run the `pip` installer program using the `install` command with the `ipython` package name
- On Mac, open a terminal window and run the following command:

```
pip3 install ipython
```

Command Line: Install IPython on macOS

- On Windows, open a command prompt and run the following command:

```
py -3 -m pip install ipython
```

Command Line: Install IPython on Windows

- Once the installation is complete, run `ipython` from the command prompt

Python Interactive Mode

IPython

- Just like the Python REPL,
 - single-line and multi-line expressions can be entered and evaluated
 - Command history (including commands from previous sessions) can be navigated using the arrow keys
- Unlike the Python REPL:
 - Each entry is associated with a line number, which increments with each command
 - Commands for specific line numbers can be saved to a file
 - Tab, Object completion
- There are numerous other features including plotting and even using it as a command line replacement

Python Interactive Mode

IPython

- One useful feature available with IPython is the ability to save the current session to file, and the ability to load the file again in the future
- To save a session:

```
%save my-session 1-10
```

Command Line: Save IPython Session

- To load a session:

```
%load my-session
```

Command Line: Load IPython Session

- The session will be saved to `my-session.py`
- The `my-session` name can be any valid file name, and the 1-10 the desired lines to save to the file

Python Interactive Mode

Hello, World Python

- To output data to the console, the print function is used
- With Python either single quotes or double quotes can be used for string literals, there is no difference, and no strong preference one way or the other in the Python community

```
print("Hello, World!")
```

Python: Print String with Double Quotes

```
print('Hello, World!')
```

Python: Print String with Single Quotes

Python Interactive Mode

Hello, World Python

- User input can be captured from the console with the `input` function

```
message = input("Please type a message: ")  
print(message)
```

Python: Capture User Input with the Input Function

- In the code sample, the captured input value is stored in a variable named `message`. Variables are covered later in the series
- Python's `print` and `input` are the basic output and input functions for a console application

Coding and Debugging

Creating Apps

- Creating an entire application starts with a single file to run the first lines of code
- Building complete Python application is beyond the scope of this course in the series, but there are agreed upon best practices (remember we want to be "pythonic") to organizing the various files which comprise an application
 - One approach is: <http://docs.python-guide.org/en/latest/writing/structure>
 - In future courses of this series, this structure will be explained and followed
- In this first course, the emphasis will be creating a module with a main function and successfully running it and debugging it using:
 - PyCharm, Visual Studio Code, Visual Studio

Coding and Debugging

Purpose of Functions

- Functions are the building blocks of programs
- Functions enable:
 - Organization
 - Reusability
 - Testability
 - Separation of Concerns

Coding and Debugging

Purpose of Functions

- Almost all languages provide syntax for creating functions
- In some languages functions are a primary building block (JavaScript) and other languages functions play a more limited role overshadowed by other structures such as classes (Java)
- Because Python is multiple paradigm in nature, application can be coded with functions alone, or functions can members of classes

Coding and Debugging

Creating a Main Function

- To create a function in Python, the `def` keyword is used
- The `def` keyword is used to define a function
- The name of the function follows the `def` keyword
- After the name of the function, any parameters for the function are defined within parentheses
- Finally, a colon is used to denote the start of the function body (the code implementing the logic of the function)
- Python uses a colon followed by a series of indented lines to mark code blocks, so no curly braces are needed, but the function implementation will need to be indented relative to the `def` keyword

Coding and Debugging

Coding a Main Function

- The token `def` is a keyword in Python
- Comments start with a hashtag
- The indentation of the function body must match the consistent indentation of the file, the usual indentation in Python is 4 spaces (do not use tabs)
- When invoking the function, the argument parentheses must be used even when no arguments are passed
- When defining a function, always include the parameter parentheses even when there are no parameters

```
# defines the function
def main():

    print("Hello, World!")

# invokes the function
main()
```

Python: Hello, World! Example

Conclusion

What we've learned...

- Quick introduction to getting up and running with Python
- Learned how to setup Python and configure a development environment
- Explored how to use the Python and IPython REPL environments
- Performed coding and debugging with modern IDEs
 - Visual Studio Code
 - Visual Studio
 - PyCharm



MASTERING PYTHON
Getting Started with Python

Overview of Variables, Classes, Types and Control Flow

Working with Data

- Software programs are comprised of two essential elements: code and data
- Data is the information the software program manages
- To help structure the data, types are used to describe what kind the data is such as numeric, string, boolean, lists, etc...
- To create complex types, classes are used
- Classes are a combination of data and the code used to manipulate the data
- To reference the typed data in memory, variables are used
- Variables themselves do not contain data, they just point to or reference the data
- These variables and the data they point to are used by the code to perform useful operations for the end user such as changing the data or reporting on the data
- Also, the code uses the data to control the execution of the software

Working with Variables

Creating Variables

- A variable is an identifier (symbolic name) that points to a value
- All values are stored as objects including numbers and booleans
 - In some languages, there is a distinction between values which are primitives (stored on stack) and which are objects (stored on heap), there is no distinction in Python, all values are objects
- Unlike other languages (and because all values are stored on the heap), variables themselves do not have types, but the objects they point to have types
- A variable can point to different values and thereby different types as the program executes
- This known as dynamic typing (the type the variable points to can change during run-time)
- Sometimes the term binding is used, a variable is nothing more than a name bound to value
- The name itself has no type, it just allows the value (which has a type) to be accessed
- The name can be rebound (reassigned) to another value as the program executes

Working with Variables

Data Types

- The `type` function returns the type of the value that the variable points to

```
some_var = "Hello, World!
print(type(some_var)) # Outputs <class 'str'>

some_var = 42
print(type(some_var)) # Outputs <class 'int'>

some_var = 3.14
print(type(some_var)) # Outputs <class 'float'>

some_var = True
print(type(some_var)) # Outputs <class 'bool'>

some_var = Person("Bob", "Smith")
print(type(some_var)) # Outputs <class '__main__.Person'>
```

Python: Data Types

Working with Variables

Example of Name Binding

- The same name (identifier) is used for both assignments
- In the first assignment, the name is bound to the integer object with a value of 10
- In the second assignment, the name is re-bound to the string object with a value of "Test"
- If a value has no names (or other references) bound to it, then it can be garbage collected by Python

```
some_data = 10  
  
# outputs: <class 'int'>  
print(type(some_data))  
  
some_data = "Test"  
  
# outputs: <class 'str'>  
print(type(some_data))
```

Python: Name Binding

Working with Variables

Type Conversion

- Python uses dynamic typing (type changes at run-time), but uses strong-typing, there is very little implicit type coercion
- To convert a value from one type to another requires the use of a conversion function
 - `str` – converts a value to a string
 - `int` – converts a value to an integer
 - `float` – converts a value to a float
 - `bool` – converts a value to a bool And many more...
- The `input` function returns value of type string, to capture numeric values from the user, the string value must be converted to a numeric type using one of the functions above

Working with Variables

Global vs. Local

- There is no formal keyword for declaring a variable, simply assign to a variable, then it becomes declared in the local scope of the function
- If the variable is not assigned within a function it is global
- Variables assigned to outside the bounds of a function are global
- Inside a function, if a variable with the same name as a global variable is retrieved, the global value is returned
- Inside a function, if a variable with the same name as a global variable is assigned to, a new local variable is created within the function
- The new variable will hide the global variable within the scope of the function
- To prevent a new variable from being created locally in the function, the variable must be marked as global with the `global` statement at the top of the function

Working with Variables

Global vs. Local

- The variable `message` is assigned to outside of a function, it will be global
- The variable `message2` is assigned to inside a function, it will be local
- Both `message` and `message2` can be accessed inside of `main`

```
message = "This is a global!"  
  
def main():  
    message2 = "This is a local!"  
  
    print(message)  
    print(message2)  
  
main()
```

Python: Global vs. Local Variable

Working with Variables

Global vs. Local

- The variable `message` is assigned to outside of a function, it will be global
- The variable `message`, which is assigned to inside the `main` function, will be local and shadow the global variable within the function (the global variable cannot be accessed)

```
message = 'This is a global!'

def main():

    # new local variable created
    message = 'This is a local!'

    # because of shadowing
    # outputs 'This is local!'
    print(message)

main()

# outputs 'This is global!'
print(message)
```

Python: Variable Shadowing

Working with Variables

Global vs. Local

- The variable `message` is assigned to outside of a function, it will be global
- The `global` keyword will allow the global variable `message` to be assigned to within the `main` function without producing a new local variable

```
message = "This is a global!"  
  
def main():  
    global message  
  
    message = "This is a local!"  
  
    # outputs 'This is local!'  
    print(message)  
  
main()  
  
# outputs 'This is local!'  
print(message)
```

Python: Global Statement

Working with Classes

Short Introduction to Classes

- As mentioned before, Python is a multi-paradigm language
- Python applications can be coded with variables and functions only, this would be the structured programming approach
- Python also supported object-oriented programming
- One part of object-oriented programming is the use of objects
- Objects combine attributes (data) and methods (functions) into a single structure
- In Python, object structures are defined with classes, and objects are created by invoking the class and instantiating a new object
- As part of the instantiating process, classes provide a special constructor function to allow the new object to be initialized with data
- Using classes in Python is not required, but it is common and many third-party Python libraries use them extensively

Working with Classes

Short Introduction to Classes

- The `class` keyword is used to define a class
- The constructor function is named `__init__`
- All methods receive an instance variable as their first argument
- The instance variable should be named `self`
- New instances are created by invoking the class as a function without a `new` keyword

```
class Person:  
    def __init__(self, fname, lname):  
        self.fname = fname  
        self.lname = lname  
  
    def get_full_name(self):  
        return self.fname + " " + self.lname  
  
p = Person("Bob", "Smith")  
print(p.get_full_name())
```

Python: Class Example

Working with Classes

Duck-Typing

- An important feature of dynamic typing is duck-typing
- With duck-typing the program cares less about the specific type of a value, and cares more about whether or not the value can be used as desired
- Using special method names known to Python, duck-typing is implemented by adding custom implementations to custom classes which mimic well-known Python operations
- For example, adding two numbers yields their sum, but adding two Person objects generates an error unless the special method `__add__` is implemented
- Then, based upon the implementation details of `__add__` two Person objects can be added together
- It's important that the operation make sense, and any operations which are explicitly not needed can be set to `None`
- We will not cover the details of duck-typing here, just be aware that duck-typing is caring more about capability than the specific type being used

Control Flow: If

Controlling the Flow of Code

- Python supports controlling the flow of logic based upon a truthy/falsy values
- The boolean type supports two values: True & False (in Python, "True" and "False" keywords start with capital letters)
- The value of None is equivalent to "null" in other languages
- The following values are falsy:
 - None
 - 0
 - False
 - Empty sequence or mapping
 - Zero-length String

Control Flow: If

Boolean Operations

- Three boolean operations are supported
- Unlike other language which use symbols, Python uses real words for each operation

Operation	Result
or	if x is false, then y, else x
and	if x is false, then x, else y
not	if x is false, then True, else False

Table: Python Boolean Operators

- Both the `or` and `and` operations use short-circuiting to optimize the boolean operations
- The order of operations is `or`, `and`, `not`
- `not` has a lower priority than non-boolean operators
- "`not a == b`" is evaluated as "`not (a == b)`"
- "`a == not b`" generates an error

Source: <https://docs.python.org/3/library/stdtypes.html>)

Control Flow: If

Comparison Operations

- Python supports the following comparison operators

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Table: Comparison Operators

- Different numeric types will compare
- All other different types compare as false
- <, >, <=, >= cannot be used between complex numbers and other numeric types
- <, >, <=, >= for all other types, the comparison operators can only be used between values of the same type

Source: <https://docs.python.org/3/library/stdtypes.html>

Control Flow: If

Controlling the Flow of Code

- Python's `if` statement will execute code in the first block if the expression returns a truthy value, and it will execute code in the second block if the `else` statement is present and the expression is falsy
- Python's `if` and other control flow statements such as looping constructs are known as compound statements

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

if len(nums) > 3:
    print("list length is greater than 3")
else:
    print("list length is equal to or less than 3")
```

Python: If Statement

Working with Lists

A Sequence Type

- Lists are a mutable sequence of values
- The values do not have to be the same type
- Lists can be created by using square brackets with the initial values

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

Python: Create a List with 5 Numbers

- Empty lists can be created too

```
nums = []
```

Python: Create an Empty List

- A number of functions are provided for working with lists such as: append, remove, insert, count, clear, etc...

Control Flow: For & While

Iterating over a List

- Python's `if` statements control branching logic, `for` statement control looping logic
- Unlike language such as C, Python does not support traditional for loops which rely upon indexes and terminating condition
- The Python `for` statement iterates over an iterable object such as a list
- An optional `else` statement can be added to execute on the loop finishes
- The `else` will execute even if the there are no items to iterate over

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

for num in nums:
    print(num)
```

Python: Looping over a List

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

for num in nums:
    print(num)
else:
    print("sum: {}".format(sum(nums)))
```

Python: Looping with an Else Statement

Control Flow: For & While

Looping Based Conditions

- In addition to iterating over a collection of items, it's possible loop based upon the truthy/falsy value of an expression
- The loop executes "while" the expression is true

```
max_value = 5  
  
current = 0  
  
while current < max_value:  
    print("current: {}".format(current))  
    current = current + 1
```

Python: While Loop

Control Flow: For & While

Looping Based Conditions

- Similar to the `for` loop, the `while` loop supports the `else` statement block as well

```
max_value = 5

current = 0

while current < max_value:
    print("current: {}".format(current))
    current = current + 1
else:
    print("final current {}".format(current))
```

Python: While Loop with an Else Statement

Control Flow: For & While

Controlling and Terminating a Loop

- In addition, the previous approach to ending the loop, it's possible to explicitly cause a loop to immediately iterate to the next value, or even exit the loop prematurely
- In this example, the `continue` keyword causes loop to immediately iterate to the next value, when the current value is even

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in nums:
    if num % 2 == 0:
        continue
    print(num)
else:
    print("sum: {}".format(sum(nums)))
```

Python: Jumping to the Next Iteration

The code outputs:

```
1
3
5
7
9
sum: 55
```

Command-Line: Loop Output

Control Flow: For & While

Controlling and Terminating a Loop

- In this code example, the `for` loop exits once "num" is greater than 5
- To exit the loop, the `break` statement is used
- When breaking out a `for` a for loop, the `else` block is not executed

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in nums:
    if num > 5:
        break
    print(num)
else:
    print("sum: {}".format(sum(nums)))
```

Python: Exiting from the Loop

The code outputs:

```
1  
2  
3  
4  
5
```

Command-Line: Loop Output

Pass Statement

Null Operations

- Sometimes, a null operation is needed to satisfy the syntax requirements of Python
- Consider a place holder function with no implementation

```
def some_placeholder():
```

Python: Python Function with no Code Block

- This code cannot standalone, Python syntax requires a function implementation
- To meet Python's syntax requirements, but not provide an implementation for the function, the `pass` statement can be used

```
def some_placeholder(): pass
```

Python: Python Function with Pass Statement

Conclusion

What we've learned...

- Variables, Classes and Python's Type System was explored
- Evaluation of Expressions and their Role in Control-Flow was explained
 - Truthy/Falsy values
 - Comparison Operators
 - Boolean Operators
- Control-flow (Compound) statements were demonstrated
 - `if` – execute code based upon a truthy/falsy expression
 - `for` – loop over an iterable
 - `while` – loop while an expression is truthy
- Learned about the `pass` statement for null operations



MASTERING PYTHON
Python Functions (Part 1 of 2)

Overview of Functions

Organize Code

- As a Python program grows from a few lines to many lines, the code needs to be organized
- Many times, the same block of code needs to be executed over and over again
- Sure, the code could be copied and pasted again and again, but how would changes to that code be efficiently made?
- Each place the code was copied to would have to be updated separately, it would be a nightmare...
- Fortunately, programming languages provide a tool called "functions" which enable programmers to re-use code over and over without having to copy and paste it

Overview of Functions

Organize Code

- A function is a block code which is "callable" from another line of code
- Functions can return results or cause side-effects
- Functions can be called with arguments that provide input data that the function uses to calculate a result
- Using the unpacking operator arbitrary positional and named arguments can be captured into parameters

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

Define a Function

Reusing Code

- Consider the following code:

```
print("Hello!")  
print("Hello!")
```

Python: Repetitive Code

- The details of the greeting is coded twice. If a third greeting was added, the details of the greeting would be copied again:

```
print("Hello!")  
print("Hello!")  
print("Hello!")
```

Python: More Repetitive Code

Define a Function

Reusing Code

- Then what if the greeting needs to be changed from "Hello!" to "Hi!". How many times would it need to be changed? Correct, three times.
- A function could help here, because the greeting could be coded in one place, then called multiple times

```
# define the function
def greeting():
    print("Hello!")

# call the function
greeting()
greeting()
greeting()
```

Python: Organize Reusable Code with a Function

Define a Function

Defining a Function

- The `def` keyword defines a function
- The name following the `def` keyword is the name of the function
- The parentheses following the name are where parameters are defined (covered later)
- The body of the function is indented

```
# define the function
def greeting():
    print("Hello!")
```

Python: Define a Function

Define a Function

Calling a Function

- A function is called by using its name followed by parentheses
- For functions which accept arguments, the arguments would be passed inside the parentheses (covered later)
- The calling of the function is also referred to as invoking the function

```
greeting()
```

Python: Call a Function

Function Outputs

Several Approaches

- Functions output their results through several approaches:
 - First, a function can cause a side-effect, it can change something external to itself
 - Second, a function can modify an object passed in as an argument (covered later in the course)
 - Third, a function can return value

Function Outputs

Side Effects

- When a function changes something outside of itself, the change is called a side effect
- For example, printing to the terminal, changes the terminal, so when a function prints to the terminal, it's a side effect of the function

```
def greeting():
    print('Hello!') # example of side effect

greeting()
```

Python: Perform a Side-Effect with Print

Function Outputs

Side Effects

- Functions can modify variables external to themselves
- To modify a variable external to a function it must be marked as `global` or `nonlocal` so that Python does not create a local variable within the function
- To use a global variable within the function, the variable is defined with the `global` statement within the function

Function Outputs

Side Effects

- The `global` statement tells the function not to create a new variable within the function
- When working with nested functions, it's possible change a variable defined in a outer function using the `nonlocal` statement (covered later)

```
total = 0

def increment_total():
    global total # use the "total" variable defined outside the function

    total += 1

increment_total()
increment_total()

print(total) # the "total" variable has changed
```

Python: Perform a Side-Effect by Mutating a Variable Outside of the Function

Function Outputs

Return a Value

- Frequently, functions will return a result
- To return a result, the `return` statement is used

```
def greeting():
    return "Hello!"

print(greeting())
```

Python: Returning a Value from the Function

Pass Arguments to a Function

Arguments and Parameters

- Generally, functions are defined with parameters
- Parameters allow the output of the function to be determined by a combination of the function's logic and input data
- The data passed into the function is called an argument, the argument is a Python expression
- The argument is evaluated, then passed into the function where it is assigned to a parameter
- Parameters are variables (local names within the function bound to the value)
- Generally parameters, are named and have one value assigned
- However, there are special parameters commonly called `*args` and `**kwargs`

Pass Arguments to a Function

Arguments and Parameters

- Parameters are defined with the parentheses following the function name in the `def` statement
- In the code below there is one parameter defined, it is called `name`

```
def greeting(name):  
    print(f'Hello, {name}')
```

Python: Python Function with a Parameter

- To call the function and populate the **parameter** with data, an **argument** is passed

```
greeting("John")
```

Python: Calling a Python Function with an Argument

Pass Arguments to a Function

Arguments are Expressions

- An easily overlooked fact which can lead to some misunderstandings later is that an argument is an expression which is evaluated then passed into the function and assigned to the parameter
- The three strings below are concatenated first, then passed in:

```
greeting("John" + " " + "Smith") # outputs "Hello, John Smith"
```

Python: Arguments are Expressions

- Python variables can be used within expressions too
- The `personName` variable itself is not passed in, it is evaluated, the result is "John Smith" and the value "John Smith" is passed in

```
personName = "John Smith"  
greeting(personName) # outputs "Hello, John Smith"
```

Python: Arguments are Evaluated

Pass Arguments to a Function

Parameters

- Some languages such as JavaScript allow an arbitrary number of arguments to be passed into a function when calling it
- If more arguments are passed than parameters defined, the additional arguments are ignored
- If less arguments are passed than parameters defined, the unassigned parameters are undefined
- Python DOES NOT WORK LIKE THIS....

Pass Arguments to a Function

Parameters

- Python is more like C, C++, Java, and C# in this regard
- If a parameter is defined on the function it must be supplied (unless configured with a default value covered later)
- Extra parameters cannot be passed unless the function is configured with `*args` or `**kwargs` (covered later)

Pass Arguments to a Function

Default Parameter Values

- One useful feature of Python is the ability to configure default values for parameters
- The argument for a parameter with a default value can be omitted when calling the function

```
def incr(value, step = 1):  
    return value + step
```

Python: Function Parameter with a Default Value

- The function can be called with or without a value being passed into the `step` parameter

```
print(incr(2,3)) # outputs 5  
  
print(incr(2)) # outputs 3
```

Python: Pass Arguments to a Function with a Default Parameter Value

Pass Arguments to a Function

Arguments and Named Parameters

- When calling functions with multiple parameters which have default values, it may be necessary to assign values to certain parameters and not others, consider the following code:

```
def display_greeting(name, greeting = 'Hello', num_of_greetings = 1):  
    for _ in range(num_of_greetings):  
        print(greeting + ", " + name)
```

Python: Function with Multiple Parameters with Default Values

- What is the `display_greeting` needs to display 10 greetings, but the text of the `greeting` does not need to change?
- Using named parameters, the `num_of_greetings` can be passed without having to pass `greeting`

```
display_greeting('John', num_of_greetings = 4)
```

Python: Assign a Parameter by Name when Calling the Function

Pass Arguments to a Function

Mutating Parameters

- Generally, mutating parameters is NOT recommended
- In programming languages such as C and C++, passing by reference allowed functions (sometimes referred to as methods) to mutate parameters
- Instead of returning a value, the parameter would be mutated and that is the output
- In modern programming, especially functional programming, this is not encouraged, it's better to return a new value than mutate a parameter
- Python does not support pass by reference, but it is possible modify arguments that evaluate to an object reference, this is known as "call by sharing"
- The object reference is assigned to the parameter and interactions with the parameter involve the actual object defined outside the function

Pass Arguments to a Function

Mutating Parameters

- In the code below, a `Person` object is created and passed into the `set_person_first_name` function
- Within the function the actual object is mutated with the value of `first_name` being assigned to object's `first_name` attribute

```
class Person:

    def __init__(self):
        self.first_name = ''


    def set_person_first_name(person, first_name):
        person.first_name = first_name


person = Person()

set_person_first_name(person, 'Bob')

print(person.first_name)
```

Python: Mutating an Object Passed as Parameter

Capture Function Arguments

Unknown Number of Arguments

- Many times functions need to accept an arbitrary number of parameters
- The caller of the function, passes as many arguments as needed, and the function needs to be able to handle all of the arguments passed in
- When working with regular parameters, one argument is assigned to one parameter
- Python provides two special parameters that are defined with the unpacking operators: * and **
- These special parameters capture all of the extra positional and named arguments

Capture Function Arguments

Unknown Number of Arguments

- The unpacking operators capture all of the "extra" arguments, and then provide those arguments through convenient Python data structures
- When using the * unpacking operator, the "extra" arguments are collected in a tuple (a kind of Python sequence)
- When using the ** unpacking operator, the "extra" arguments are captured as a dictionary (a key-value pair collection of data)
 - An argument is placed into the dictionary if it was passed as a named parameter, and the parameter name is not defined on the function

Capture Function Arguments

Capture Arguments in a Tuple

- The unpacking * operator collects any unnamed, extra arguments that are passed into the function
- The parameter is commonly named `args`, but that is only a convention, the only requirement is the * parameter prefix

```
def sum_nums(*args):  
    sum_total = 0  
  
    for arg in args:  
        sum_total += arg  
  
    return sum_total  
  
print(sum_nums(1, 2, 3, 4, 5))
```

Python: Capturing Arbitrary Arguments as a Tuple

Capture Function Arguments

Capture Arguments in a Dictionary

- The unpacking `**` operator collects any named, extra arguments that are passed into the function
- The parameter is commonly named `kwargs`, but that is only a convention, the only requirement is the `**` parameter prefix

```
def get_connection_url(**kwargs):  
  
    protocol = kwargs.get("protocol") or "http://"  
    host = kwargs.get("host") or "localhost"  
    port = kwargs.get("port") or "80"  
  
    return f"{protocol}{host}:{port}"  
  
print(get_connection_url(protocol="https://", port="3000")) # outputs https://localhost:3000  
print(get_connection_url(host="127.0.0.1")) # http://127.0.0.1:80
```

Python: Capturing Arbitrary Named Params as a Dictionary

Capture Function Arguments

Capture Parameters and Other Parameters

- When combining capture parameters with other parameters, the capture parameters come last in the parameter list
- When both the `*args` and `**kwargs` are present, the `**kwargs` must come last
- Both the `*args` and `**kwargs`, can only appear once in the parameter list

```
def func1(param, *args, **kwargs): # valid
    # def func1(*args, **kwargs, param): # invalid, *args and **kwargs must come last
    # def func1(param, **kwargs, *args): # invalid, **kwargs must come last

    print(param)
    print(args)
    print(kwargs)

func1("a") # output: a () {}

func1("a", "b", "c") # output: a ('b', 'c') {}

# output: a ('b', 'c') {'first': 'd', 'second': 'e'}
func1("a", "b", "c", first="d", second="e")
```

Python: Capture Parameters combined with Other Parameters

Conclusion

What we've learned...

- Most Python programs require structure and code organization
- Python provides functions to help structure and organize code
- Functions allow code to be parameterized and reused
- Functions output their results through side effects and returning values
- Python provides many useful syntax features for passing arguments such as default values and capturing extra arguments



MASTERING PYTHON
Python Functions (Part 2 of 2)

Overview of Functions

Organize Code

- Functions are first-class citizens, meaning they exist on their own (they do not have to be part of a larger class structure or something similar)
- Functions are objects, and can be assigned to variables, passed into other functions, and returned from functions
- Functions can be defined within other functions
- Functions can be decorated to enhance their functionality
- Inline functions, called lambdas, can be defined within an expression to simplify code
- Using the `functools` module, partial functions can be created

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

Functions are Objects

Functions are Objects

- Similar to other data types in Python, functions are objects

```
def do_it():
    print("did it")

print(type(do_it)) # outputs <class 'function'>
```

Python: Function Object Type

- Being objects, they can be bound to other names (assigned to variables)

```
def do_it():
    print("did it")

do_it2 = do_it

do_it()
do_it2()
```

Python: Binding a New Name to the Function

Functions are Objects

Function Names

- When a function is defined, the name used in the definition is the name of the function

```
def do_it():
    print("did it")

print(do_it.__name__) # outputs do_it
```

Python: Function Bound Name

- Even if a function is bound to another name, it still retains the original name

```
def do_it():
    print("did it")

do_it2 = do_it

print(do_it2.__name__) # outputs do_it
```

Python: Function Bound to Another Name

Functions are Objects

Passing Function Objects as a Function Argument

- Similar to other objects in Python, functions can be passed as arguments into other functions
- In the case of `map`, a transformation function is passed into the `map` function
- When the map iterable is iterated over, the `double` function will be executed with each item

```
def double(x): return x * 2

nums = [1,2,3,4,5]

# the function double is passed as a parameter to map
# the object reference is what is passed into the map function
doubleNums = list(map(double, nums))

print(doubleNums)
```

Python: Passing a Function as an Argument to a Function

Inner Functions and Closures

Functions within Functions

- Functions are objects, and objects can be created within Python functions, therefore, functions can be defined within functions

```
def outer():  
    print("outer")  
  
    def inner():  
        print("inner")  
  
    inner()  
  
outer()
```

Python: Define a Function within a Function

Inner Functions and Closures

Return a Function from a Function

- In addition to defining the function within a function, the inner function can be returned from the outer function just as any other object could be returned

```
def outer():  
    print("outer")  
  
    def inner():  
        print("inner")  
  
    return inner  
  
fn = outer()  
  
fn()
```

Python: Return a Function from a Function

Inner Functions and Closures

Using a Variable from the Outer Function inside the Inner Function

- Variable declared in the outer function can be used within the inner function
- This forms a closure preventing the variable defined in the outer function from being cleaned up after the outer function returns

```
def outer():
    print("outer")
    num = 2

    def inner():
        print("inner")
        print(num)

    return inner

fn = outer()
fn()
```

Python: Access a Variable via Closure

Inner Functions and Closures

Mutating the Closure Variable

- It's possible to update the closure variable, if the variable is defined as `nonlocal` in the inner function
- Changes made to the closure variable in the inner function, changes the actual variable defined in the outer function

```
def outer():
    num = 2

    def update_number():
        nonlocal num
        num = num + 1

    def print_number(): print(num)

    return (update_number, print_number)

(update_fn, print_fn) = outer()
print_fn()
update_fn()
print_fn()
```

Python: Mutate the Closure Variable

Decorator Functions

Adding Functionality to a Function

- Decorator functions are a powerful way to add new functionality to a function
- Decorators can run extra code before and after a function call
- Decorators are a form of metaprogramming, metaprogramming is programming that manipulates programming

```
def decorator(func):
    def wrapper():
        print("Start")
        func()
        print("Stop")
    return wrapper

def do_it():
    print("do it")

fn = decorator(do_it)
fn()
```

Python: Decorator Function

Decorator Functions

Syntactic Sugar

- Python provides a very useful syntax for applying decorators to functions
- Using the `@` symbol, decorator functions can be applied to the definition of the functions they wrap

```
def decorator(func):
    def wrapper():
        print("Start")
        func()
        print("Stop")
    return wrapper

@decorator
def do_it():
    print("do it")

do_it()
```

Python: Apply a Decorator with the @ Symbol

Decorator Functions

Capturing and Forwarding Wrapper Parameters

- Using the unpacking operators * and **, parameters can be captured from the wrapper function and passed to the function being wrapped by the decorator

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print("Start")
        func(*args, **kwargs)
        print("Stop")
    return wrapper

@decorator
def do_it(a, b):
    print(a, b)

do_it('A', 'B')
```

Python: Capture and Forward Wrapper Function Parameters

Decorator Functions

Capturing and Forwarding Decorator Parameters

- Wrapping the decorator function in another function allows the decorator to receive parameters and forward those to the wrapper function or the function being decorated

```
def decorator_params(*decorator_args, **decorator_kwargs):
    def decorator(func):
        def wrapper(*wrapper_args, **wrapper_kwargs):
            print("Start")
            func(*decorator_args, *wrapper_args, **decorator_kwargs, **wrapper_kwargs)
            print("Stop")
        return wrapper
    return decorator

@decorator_params(1, 2)
def do_it(*args, **kwargs):
    print(*args, **kwargs)

do_it('A', 'B')
```

Python: Capture and Forward Decorator Function Parameters

Lambda Functions

Inline Syntax for Single Expression Functions

- The lambda function comes from lambda calculus, which is the root of functional programming
- While Python is not a functional programming language (it is imperative), it includes many of the features found in functional programming languages
- Lambda functions are single expression functions
- Typically, they are defined and passed into another function in a single step (inline)

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

doubleNums = map(lambda x: x * 2, nums)

print(list(doubleNums))
```

Python: Double Function implemented in Lambda Syntax

Lambda Functions

Inline Syntax for Single Expression Functions

- To code a lambda function, the `lambda` keyword is used
- A colon separates the parameters from the expression
- Lambda functions support zero or more parameters
- For long lambda functions, the expression following the colon can be placed on the next line

```
from functools import reduce

get_nums = lambda: [1,2,3,4,5]

doubleNums = map(lambda x: x * 2, get_nums())
print(list(doubleNums))

numsSum = reduce(lambda acc, cur:
    acc + cur, get_nums(), 0)
print(numsSum)
```

Python: Lambda Function Syntax

Lambda Functions

Being Pythonic and PEP-8

- Unlike languages, such as JavaScript, where "lambda" functions are used everywhere, in Python, their actual use is quite limited
- Generally, lambda functions are only used as inline functions
- Per PEP-8, lambda functions should not be assigned to a variable, instead a normal `def` function should be used
- Even for situations such as `map` it can be more "Pythonic" to use a list comprehension instead of a `map` and a lambda function

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

# this is ok
print(list(map(lambda x: x * 2, nums)))

# but this is even better
print([x * 2 for x in nums])
```

Python: Being More Pythonic

Lambda Functions

Being Pythonic and PEP-8

- Assigning the lambda function to a name and calling it later violates the PEP-8 standard
- Instead, create the function with `def`, it will have a name, then use the name as normal
- Please note, `def` style functions can be one line

```
# this works but does not follow PEP-8
succ = lambda x: x + 1

print(succ(1))

# this follows PEP-8 and is more Pythonic
def succ(x): return x + 1

print(succ(1))
```

Python: Lambda Function Assignment

Lambda Functions

Being Pythonic and PEP-8

- If lambda functions tend to be discouraged, then why do they exist?
- The `def` style functions cannot be defined inline, when an inline function is needed, lambdas must be used

```
# does not work, def cannot be used inline
print(list(map(def double(x): x * 2, [1, 2, 3, 4])))

# works great
print(list(map(lambda x: x * 2, [1, 2, 3, 4])))
```

Python: Lambda Function Assignment

Partial Functions

Apply Arguments to Some Parameters

- Many times it can be helpful to wrap a function in a new function where the new function supplies some of the parameters of the original function
- In the code below, the `add` function accepts two parameters
- Using the `partial` function, a new function can be created that supplies the first parameter and allows the caller of the resulting "partial" function to supply the second parameter

```
from functools import partial

def add(x, y): return x + y

add2 = partial(add, 2)
add3 = partial(add, 3)

print(add2(10)) # outputs 12
print(add3(10)) # outputs 13
```

Python: Partial Functions

Conclusion

What we've learned...

- Functions are objects and can be passed into other functions as arguments
- Functions can be defined within functions and even returned from functions
- Functions can interact with variables outside of themselves through globals and closures
- Decorator functions are a form of metaprogramming and allow decorated functions to receive enhanced functionality
- Lambda functions can be defined inline and evaluate a single, parameterized expression
- Partial functions enable the creation of functions that supply some of the parameters for an inner function allowing the caller to supply the remaining parameters



MASTERING PYTHON
Getting Started with Python

Overview of Modules and Packages

Organizing Code

- A Python module is a Python source code file
- Each source code file is a module and the code from it can be imported into other Python source code files
- The "main" Python file of an application is itself a module, known as the main module, and can import the functionality of other modules
- Most Python applications span multiple files, and with each file being a module, each module's functionality can be shared across the application
- Often a module or collection of modules needs to be shared with many applications
- Sharing a module or collection of modules is done through a package
- A package is a module or modules that are "packaged" up to be shared with other applications

Overview of Modules and Packages

Organizing Code

- In reality, every Python application itself is a package, but it does not have to be distributed
- In addition to the modules and packages created locally, there are many packages which are distributed through the Python Package Index and other package management systems
- These packages can be installed within a Python application using pip
- The Anaconda distribution of Python uses the Conda package (and environment) management system
 - <https://conda.io/docs/>
 - Not covered in this course

Organizing Code with Modules

Multiple Files

- The name of a module is the file name of the Python source code file
 - Exception: the Python source code file invoked with the Python interpreter is referred to as the main module based upon its role in the invocation of the program
- Unlike other languages, nothing has to be marked as exported
- Variables, Functions and Classes defined globally in the script are automatically exported and can be imported into another module

```
def add(num1, num2):  
    return num1 + num2
```

Python: File calc.py

```
import calc  
  
print(calc.add(1, 2))
```

Python: Import the Entire Module

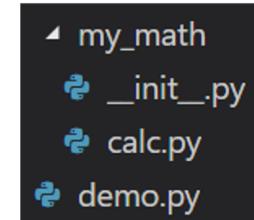
```
from calc import add  
  
print(add(1, 2))
```

Python: Import One Function from the Module

Organizing Modules into Packages

A Collection of Modules

- Packages are nothing more than a folder of Python module files
- In the package folder, a file named `__init__.py` should be present, it can be empty
 - This file can be used for package initialization
- When Python sees this file, it treats the folder as a package, and its modules can be imported
- The folder name is the package name, and can be thought of as a namespace



Screenshot: Folder Tree

```
import my_math.calc  
  
print(my_math.calc.add(1, 2))
```

Python: Import Calc as Property of Package

```
from my_math import calc  
  
print(calc.add(1, 2))
```

Python: Import Calc by Itself

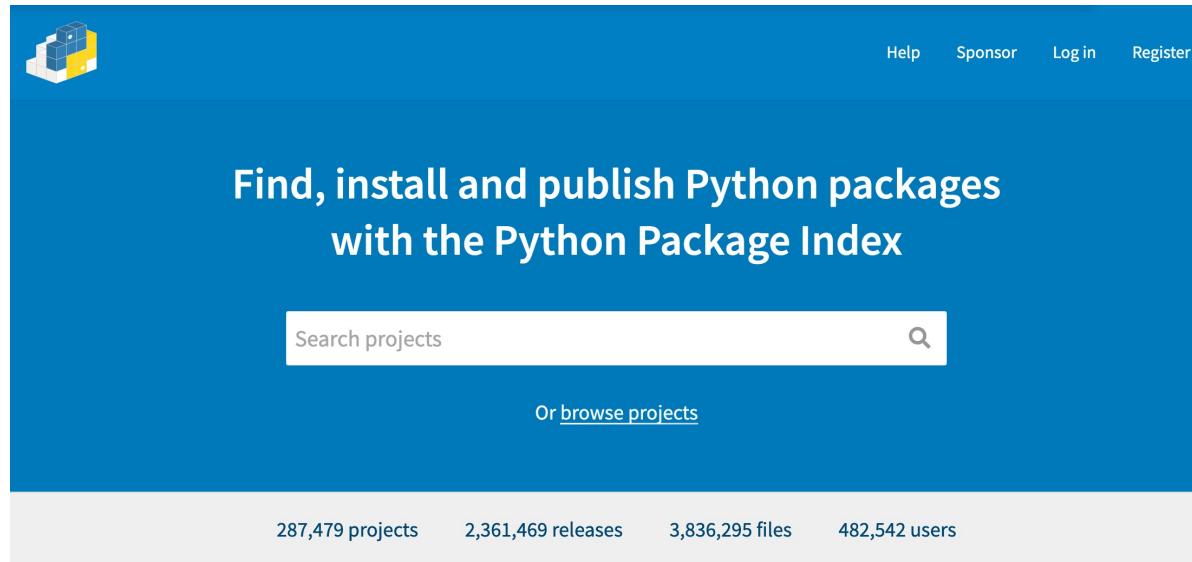
Python Package Index

Publicly Distributed Packages

- The Python Package Index is the official public package repository for Python
- There are over 120,000 packages (as of Nov 2017) which Python developers can use in their own applications
- The packages are simply shared code libraries which save developers lots of time by not requiring them to reinvent the wheel for each task they need to accomplish
- In this package repository are some of the most popular scientific packages such as NumPy that developers rely upon
- To use the repository to download and install packages, the program `pip` is used. Pip is distributed with Python.

Python Package Index

Publicly Distributed Packages



The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages ↗](#)

Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI ↗](#)

Screenshot: PyPi Home Page (<https://pypi.org/>)

Python Package Index

Installing Packages

- Packages can be installed globally or for the user
- To install a package globally, open a terminal, and run the following command:

```
pip install <package name>
```

Command-Line: Install Python Package

- To install a package for the current user only, run the following command:

```
pip install --user <package name>
```

Command-Line: Install Python Package for a User

Python Package Index

Removing Packages

- To remove a package run:

```
pip uninstall <package name>
```

Command-Line: Uninstall a Python Package

- If a package is installed locally (with --user) and globally, the local version will be uninstalled first
- Run the command again to uninstall the package globally

Python Package Index

Listing Packages

- To list all installed packages:

```
pip list
```

Python: List Packages

- To see a list of outdated packages:

```
pip list --outdated
```

Python: List Outdated Packages

- To upgrade a package:

```
pip install --upgrade <package name>
```

Python: Upgrade a Package

Installed Version	Latest Version
anaconda-project (0.8.0)	Latest: 0.8.2 [wheel]
asn1crypto (0.22.0)	Latest: 0.23.0 [wheel]
Babel (2.5.0)	Latest: 2.5.1 [wheel]
bleach (2.0.0)	Latest: 2.1.1 [wheel]
cchardet (1.1.3)	Latest: 2.1.1 [sdist]
certifi (2017.7.27.1)	Latest: 2017.11.5 [wheel]
cffi (1.10.0)	Latest: 1.11.2 [wheel]
cloudpickle (0.4.0)	Latest: 0.5.1 [wheel]

Screenshot: Package List

Python Package Index

Displaying Package Details

- To see more details of a package:

```
pip show <package name>
```

Command-Line: Show Package Details

```
Name: blaze
Version: 0.11.3
Summary: Blaze
Home-page: UNKNOWN
Author: Continuum Analytics
Author-email: blaze-dev@continuum.io
License: BSD
Location: /Users/ericwgreene/anaconda3/lib/python3.6/site-packages
Requires: flask, flask-cors, odo, psutil, sqlalchemy, toolz, dask
```

Screenshot: Show Blaze Package Details

Using Packages with Projects

Requirements File

- To ensure packages are installed for a given application, a "requirements.txt" file can be created
- The "requirements.txt" can be used by pip to install the packages listed inside of it

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

Command-Line: Install a Project's Dependencies

- To initially create the "requirements.txt" file, the pip "freeze" command can be used

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

Command-Line: Save a List of the Project's Dependencies

Conclusion

What we've learned...

- Most Python applications are long and should be divided into many files, not stored in one file
- Python modules allow an application to be divided into many files, where a file's functionality can be imported into another file
- Shared functionality can include variables, functions and classes
- A collection of modules can be stored in a package which is nothing more than a folder with Python module files
- Package folders should have a special file named `__init__.py` in order for Python to properly recognize it as a package
- In addition to local packages, packages can be distributed through the Python Package Index and used by all Python developers
- To install publicly distributed packages, the `pip` command is used



MASTERING PYTHON
The File System - Part 1

Overview

Paths, Files, and Directories

- Python provides a rich set of functionality for working with file system paths, files and directories (aka directories)
- Within Python's Standard Library there are numerous modules for working with the file system
- For this course, the `os` and `pathlib` modules will be explored
- Specifically, API functions related to working with paths and directories will be explored
- The older `os` module and the newer `pathlib` module solve many of the same problems, and their differences will be explored
- Also, operating system compatibility issues will be demonstrated

Overview

Modules for Working the File System

- The `os` module provides lower-level functions for working with the operating system directly
- The `pathlib` module is a newer module that provides a more object-oriented way of working with path, files and directories
- For most operations, `pathlib` objects can be used where traditional string-based paths are used
- These two modules can be combined together to cover just about any file operation needed

Overview

Why does the File System matter?

- Python is commonly used for tasks which involve working with file systems
- First, Python is scripting language used for system administration tasks
- System administration tasks almost always involve many interations with the file system
- Second, Python is used to script DevOps functionality that commonly includes managing files and directories on various systems
- Because, Python runs on Windows, macOS and Linux, cross-platform file system management code be written relatively easily
- Finally, many Python applications need access to the file-system to read/write data and configuration files

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

OS Module: File and Directory Paths

OS Module

- The `os` module provides numerous functions for working with paths, files, and directories
- The `os` module is a lower-level API within Python, the API is more tightly coupled to the specific APIs of the file system
- As such, not all `os` module functions work with all operating systems, so care must be taken when using this module to create Python programs that are cross platform
- Many `os` functions are Posix (Linux & macOS) only, or have similar but more limited functionality on Microsoft Windows
- The key is to double check the documentation for specific functions to ensure they will work as intended on all of the operating systems your script will run on

OS Module: File and Directory Paths

Working Directory

- The `os` module is imported like any other module (its functions are not built-in Python language functions like the `open` function; rather, they are part of the Standard Library)
- The `getcwd` function returns the current working directory
- The current working directory can be changed with the `chdir` function
- Path strings passed to the `chdir` function are operating system dependent

```
import os

# output the current working director
print(os.getcwd())

# change current working directory to "sample-files" subdirectory
os.chdir('sample-files')

# current working directory is now within sample-files
print(os.getcwd())
```

Python: Using OS Module to Manipulate the Current Working Directory

OS Module: File and Directory Paths

OS Path Module

- The `os` module provides a `path` module that contains many useful functions for working with paths
- File system paths are composed of 0 or more path segments representing the directory tree
- The path is a single string with all of the segments joined together with a path separator

OS Module: File and Directory Paths

OS Path Module

- The path separator is operating system dependent, for POSIX it is a forward slash /, for Windows it is a backslash \

```
from os import getcwd
from os.path import join

path = join(getcwd(), 'sample-files')

# POSIX: /Users/someuser/python-demos/sample-files
# Windows: C:\Users\someuser\python-demos\sample-files
print(path)
```

Python: Join Path Segments

OS Module: File and Directory Paths

OS Path Module

- Within a directory there can be files and subdirectories
- When getting a listing of a directory, both files and subdirectories are returned
- To determine if a member of a directory is a file or a directory, the `os.path` module provides the following functions: `isdir` and `isfile`

```
from os import listdir
from os.path import isdir, isfile, join

for member in listdir('.'):
    print(f"{member}: {'file' if isfile(join('.', member)) else 'directory'}")

for member in listdir('.'):
    print(f"{member}: {'directory' if isdir(join('.', member)) else 'file'}")

files = [member for member in listdir('..') if isfile(join('..', member))]
print(files)

directories = [member for member in listdir('..') if isdir(join('..', member))]
print(directories)
```

Python: Identify Files and Directories

OS Module: File and Directory Paths

Operating System Compatibility

- Operating system compatibility is easily overlooked when writing a Python program (it works on the developer's computer, right?)
- Compared the two examples below, the only difference is the backslash (favored by Windows) and the forward slash (required for POSIX file system such as Linux and macOS)
- The `r` prefix means a string literal (the backslash does not need to be escaped)

```
import os  
  
print(os.getcwd())  
  
os.chdir(r"./sample-files")  
  
print(os.getcwd())
```

Python: Works on Windows and POSIX

```
import os  
  
print(os.getcwd())  
  
os.chdir(r".\sample-files")  
  
print(os.getcwd())
```

Python: Works on Windows Only

OS Module: File and Directory Paths

Operating System Compatibility

- For all paths, use forward slashes including for the paths on Windows which use a drive letter, C:/Windows
- If a Windows path with backslashes is needed when running Python on Windows utilize the `normpath` function from the `os` module
- When an absolute path is needed from a relative path, use the `abspath` function which will combine the current working directory with the relative path and produce an absolute path with the right kind of slashes depending upon the operating system

```
from os.path import abspath  
  
path = abspath("./sample-files")  
  
# Windows: C:\Users\someuser\python-demos\sample-files  
# POSIX: /Users/someuser/python-demos/sample-files  
print(path)
```

Python: Cross-Platform Absolute Paths

PathLib Module: File and Directory Paths

Path Class

- A newer, more object-oriented approach to working with paths, files and directories is available through the `pathlib` module
- There are two kinds of path classes, pure paths and concrete paths
- Pure paths are computational only (such as constructing paths), they perform no operations on the file system itself
- Concrete paths inherit from pure paths, and perform operations on the file system as well

PathLib Module: File and Directory Paths

Path Class

- Under the hood, when a path class is created, it creates one of two kinds of objects: a POSIX path object (for Linux and macOS) and a Windows pat object (for Windows)
- The `pathlib` module is preferred to the `os` module
 - For a mapping of `os` module functions to `pathlib` module functions:
<https://docs.python.org/3/library/pathlib.html#correspondence-to-tools-in-the-os-module>

PathLib Module: File and Directory Paths

Path Class

- The `Path` class provides helpful class methods for retrieving the current working directory and the user's home directory
- The `Path` class produces an `PosixPath` instance on macOS and Linux, and a `WindowsPath` instance on Windows

```
from pathlib import Path

current_working_directory = Path.cwd()
# outputs: the current working directory from where Python was executed
print(current_working_directory)

home_directory = Path.home()
# outputs: the user's home directory on their operating system
print(home_directory)

# outputs: pathlib.PosixPath (on macOS, Linux) and pathlib.WindowsPath (on Windows)
print(type(home_directory))
```

Python: Using Path Class to Display Special Directories

PathLib Module: File and Directory Paths

Path Class

- Using a starting path, a path can be built using the `joinpath` instance method

```
from pathlib import Path

home_directory = Path.home()

bash_profile_path = home_directory.joinpath('.bash_profile')

# output on macOS: /Users/<username>/.bash_profile
print(bash_profile_path)
```

Python: Building a Path with new Segments

PathLib Module: File and Directory Paths

Path Class

- The `Path` instance provides two methods for checking if a path refers to a file or directory

```
# outputs: True
print(bash_profile_path.is_file())

# outputs: False
print(bash_profile_path.is_dir())
```

Python: Checking if a Path is a File or Directory

PathLib Module: File and Directory Paths

Path Class

- Most Python and Python Package APIs accept a `PathLike` object (such as a `Path` object or a `String`)
- Using the `str` function on the `Path` instance returns the traditional path string

```
from pathlib import Path

home_directory = Path.home()

bash_profile_path = home_directory.joinpath('.bash_profile')

# outputs: path string
print(bash_profile_path)
```

Python: Output a String from a Path Object

OS Module: Directories

List, Create and Remove Directories

- In addition to using the `pathlib` module, the `os` module can be used to work with directories as well

```
from os import path, getcwd, listdir, mkdir, rmdir

current_working_directory = getcwd()

for item in listdir(current_working_directory):
    print(item) # print each item in the directory

# make a directory at the specified path
mkdir(path.join(current_working_directory, 'temp'))

# remove a directory at the specified path
rmdir(path.join(current_working_directory, 'temp'))
```

Python: List, Create and Remove Directory with OS Module

PathLib Module: Directories

List Directory Contents

- When working with a directory, the `Path` object provides an `iterdir` function that enables iteration over the contents of a directory

```
from pathlib import Path

for item in Path.cwd().iterdir():
    print(item.name)
```

Python: Iterate over the Contents of a Directory

PathLib Module: Directories

Create and Remove Directories

- Path objects can be used to create and remove directories.

```
from pathlib import Path

# create a new directory named temp in the current working directory
Path.cwd().joinpath('temp').mkdir()

# remove a directory named temp in the current working directory
Path.cwd().joinpath('temp').rmdir()
```

Python: Make and Remove Directories with a Path Object

Conclusion

What we've learned...

- Python provides numerous APIs for working with files and directories
- Python supports both Windows and POSIX file systems
- When writing file system dependent code care must be taken to ensure it is compatible with all operating systems the code is intended to run on
- The older `os` module provides numerous independent methods for working with both POSIX and Windows file systems
- The newer `pathlib` module provides more object-oriented interface for working with the file system



MASTERING PYTHON
The File System - Part 2

Overview

Paths, Files, and Directories

- Python provides a rich set of functionality for working with file system paths, files and directories (aka folders)
- To open a file, the built-in `open` function is used (no module needs to be imported)
- The `open` function opens files for reading, writing, and appending
- Python supports both text and binary files
- In addition to the `open` function, Python provides additional functions for working with paths, files and directories through numerous modules
- For this course, the `shutil` module will be explored as well
- The `shutil` module is a high-level utility module for performing common file and directory management tasks

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

Reading Files

Opening Files for Reading

- There are many ways to read a file in Python
- First, the `open` function can be used
- Second, a `Path` instance can be used as well
- Working with the file system can result in errors so it's best to perform operations within a Try-Except block and/or a With statement
- Python can read both text files and binary files
- To set the file mode for reading, writing, appending as well as specifying text or binary files, file mode flags are used

Reading Files

Opening Files for Reading

- To read a file the `open` function is used to open the file
- The `r` flag indicates the file is being opened for reading
- Opening and reading from a file can cause errors, so a `try-except` block is needed
- In the `finally` block the file is closed
- While this approach is ok, it would be much cleaner with a `with` statement

```
file_path = 'colors.txt'
colors_file = None

try:
    colors_file = open(file_path, 'r')
    for color in colors_file:
        print(color.rstrip())

except FileNotFoundError:
    print(f'{file_path} not found.')
except:
    print(f'Error accessing file {file_path}')
finally:
    if colors_file:
        colors_file.close()
```

Python: Read from a File

Reading Files

Opening Files for Reading

- Using a `with` statement the `finally` block can be removed
- The `open` function returns an object that implements the context manager protocol
- When the `with` block ends, the file resource will be closed and cleaned up

```
file_path = 'colors.txt'

try:
    with open(file_path, 'r') as colors_file:
        for color in colors_file:
            print(color.rstrip())
except IOError as exc:
    print(exc)
except:
    print(f'Error accessing file {file_path}')
```

Python: Open a File using With

- To handle exceptions, the `try-except` block wraps the `with` statement
- While this approach to handling exceptions works, it is not very Pythonic, for a more Pythonic solution let's visit PEP 343

Reading Files

Opening Files for Reading

- Following the example of PEP 343, a more Pythonic way of writing the code is to wrap the file opening and exception handling within a context manager
- Using the `@contextmanager` decorator, a context manager function is created with the `open` function and `try-except` code blocks

```
@contextmanager
def safe_open(file_path, file_mode='r'):

    file_handle = None
    try:
        file_handle = open(file_path, file_mode)
    except IOError as exc:
        yield None, exc
    except:
        yield None, 'Error accessing file {file_path}'
    else:
        try:
            yield file_handle, None
        finally:
            file_handle.close()
```

Python: Create a Context Manager

Reading Files

Opening Files for Reading

- The `safe_open` context manager is used with the `with` statement to open the file and handle any exceptions which occur
- If an exception occurs, the `exc` variable will be populated with the error

```
file_path = 'colors.txt'

with safe_open(file_path) as (colors_file, exc):
    if exc:
        print(exc)
    else:
        for color in colors_file:
            print(color.rstrip())
```

Python: Open a File using With

Reading Files

Opening Files for Reading

- The `Path` class from the `pathlib` module can be used with the `safe_open` context manager to open a file

```
file_path = Path('colors.txt')

with safe_open(file_path) as (colors_file, exc):

    if exc:
        print(exc)
    else:
        for color in colors_file:
            print(color.rstrip())
```

Python: Open a File using With a Path Object and Safe Open

Reading Files

Opening Files for Reading

- Using the `isinstance` function, a string path can be converted to a `Path` object
- `Path` objects have an `open` function to open the file

```
from pathlib import Path
from contextlib import contextmanager

@contextmanager
def safe_open(file_path, file_mode='r'):

    isinstance(file_path, Path)
    if not isinstance(file_path, Path): file_path = Path(file_path)
    file_handle = None

    try:
        file_handle = file_path.open(file_mode)
    except IOError as exc:
        yield None, exc
    except:
        yield None, 'Error accessing file {file_path}'
    else:
        try:
            yield file_handle, None
        finally:
            file_handle.close()
```

Python: Using a Path Object and the With Statement

Writing Files

Writing Data to a File

- To write to a file, the `w` file mode flag is passed into the `safe_open` function which passes it into the `open` function
- The file object has a `write` function that writes content to the file

```
from file_utils import safe_open

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

with safe_open('newcolors.txt', 'w') as (new_colors_file, exc):

    if exc:
        print(exc)
    else:
        for color in colors:
            new_colors_file.write(f"{color}\n")
```

Python: Open a File for Writing with Safe Open

High-Level File and Directory Tasks

Copy, Move and Remove Files

- The `shutil` provides many helpful, high-level directory operations

```
from shutil import copy, move, copytree, rmtree

# copy a file
copy('./colors.txt', './colors2.txt')

# move a file
move('./colors2.txt', './colors3.txt')

# move a directory
move('./sample-files2', './sample-files3')

# copy an entire directory tree
copytree('./sample-files', './sample-files2')

# remove an entire directory tree
rmtree('./sample-files')
```

Python: Copy, Move, and Remove Files and Directories

Conclusion

What we've learned...

- Python provides numerous APIs for working with files and directories
- Python supports both Windows and POSIX file systems
- With Python files can be read and written to, and using the context manager simplifies the code to open/close files and handling exceptions
- In addition to the `open` built-in function, `Path` objects from the `pathlib` module can be used to read and write files to the file system
- Also, Python provides with `shutil` module to perform many common, high-level file system operations



MASTERING PYTHON

Common File Formats

Overview

Common File Formats

- One common use of Python is to automate system administration tasks such as DevOps systems
- Also, many Python programs utilize some kind of configuration to run in the desired way
- When managing system administration tasks reading, writing, modify common configuration file formats is very common
- Also, when storing configuration for a Python program, using common file formats makes managing the configuration easier

Overview

Common File Formats

- As part of Python's "batteries included" approach, there is rich support within the Python Standard Library for common file formats such as Configuration, JSON and CSV
- Also through PIP packages, support for YAML files can be added as well

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

Configuration Files

Read a Configuration File

- The configuration file format is a list of simple key-value pairs, with the key-value pairs organized into sections
- Python supports reading configuration files using the `configparser` module

```
[WebServer]
port=8080
path=www
```

Config: Input File web.cfg

```
from pathlib import Path
from configparser import ConfigParser

cfg = ConfigParser()
cfg.read(Path("./web.cfg"))

# outputs: 8080
print(cfg["WebServer"]["port"])

# outputs: www
print(cfg["WebServer"]["path"])
```

Python: Read a Config File

Configuration Files

Write a Configuration File

- In addition to reading configuration files, they can be written to as well

```
from pathlib import Path
from configparser import ConfigParser

cfg = ConfigParser()
cfg.add_section("WebServer")
cfg.set("WebServer", "port", "8080")
cfg.set("WebServer", "path", "www")

with open(Path("./web2.cfg"), "w") as cfg_file:
    cfg.write(cfg_file)
```

Python: Write a Config File

```
[WebServer]
port=8080
path=www
```

Config: Output File web2.cfg

CSV Files

Read a CSV File

- Python is used for data science tasks that involve large amounts of data
- A common format for this data to be distributed in is the CSV file
- A CSV file is like a spreadsheet, and in fact, spreadsheet programs offer the options to save spreadsheets as CSV files instead of their native formats
- Python provides support for CSV files within its Standard Library through the `csv` module

```
id,name,qty,price
1,apple,2,1.2
2,chicken,4,4.99
3,ice cream,1,4.49
4,soda,12,3.99
5,bacon,1,6.79
```

CSV: Food Data

```
import csv

with open('./foods.csv') as csv_file:
    reader = csv.DictReader(csv_file)

    for food in reader:
        print(food['name'])
```

Python: Read a CSV File

CSV Files

Write a CSV File

- Python can write CSV files too

```
import csv

foods = [
    {"id": 1, "name": "apple", "qty": 2, "price": 1.20},
    {"id": 2, "name": "chicken", "qty": 4, "price": 4.99},
    {"id": 3, "name": "ice cream", "qty": 1, "price": 4.49},
    {"id": 4, "name": "soda", "qty": 12, "price": 3.99},
    {"id": 5, "name": "bacon", "qty": 1, "price": 6.79},
]

with open('./foods.csv', 'w') as csv_file:
    writer = csv.DictWriter(
        csv_file, fieldnames=list(foods[0].keys()))
    writer.writeheader()
    for food in foods:
        writer.writerow(food)
```

Python: Write a CSV File

id	name	qty	price
1	apple	2	1.2
2	chicken	4	4.99
3	ice cream	1	4.49
4	soda	12	3.99
5	bacon	1	6.79

CSV: Food Data

JSON Files

Read a JSON File

- JSON is a popular format for interacting with the web applications or Node.js applications

```
[  
  {  
    "id": 1,  
    "name": "apple",  
    "qty": 2,  
    "price": 1.2  
  },  
  {  
    "id": 2,  
    "name": "chicken",  
    "qty": 4,  
    "price": 4.99  
  },  
  ...additional data omitted...  
]
```

JSON: Food Data

Note: "...additional data omitted..." is added for presentation purposes only, it is not valid JSON

```
import json  
  
with open('foods.json') as json_file:  
    foods = json.load(json_file)  
  
for food in foods:  
    print(food['name'])
```

Python: Read a JSON File

JSON Files

Write a JSON File

- Most REST APIs today return JSON string instead of XML
- When building Python REST APIs, its important for Python to be able to read/write an JSON

```
import json

foods = [
    {"id": 1, "name": "apple", "qty": 2, "price": 1.20},
    {"id": 2, "name": "chicken", "qty": 4, "price": 4.99},
    {"id": 3, "name": "ice cream", "qty": 1, "price": 4.49},
    {"id": 4, "name": "soda", "qty": 12, "price": 3.99},
    {"id": 5, "name": "bacon", "qty": 1, "price": 6.79},
]

with open('foods.json', 'w') as json_file:
    json_file.write(json.dumps(foods, indent=2))
```

Python: Write a JSON File

```
[
{
    "id": 1,
    "name": "apple",
    "qty": 2,
    "price": 1.2
},
...additional data omitted...
]
```

JSON: Food Data

Note: "...additional data omitted..." is added for presentation purposes only, it is not valid JSON

YAML Files

Read YAML File

- YAML is an acronym for "YAML Ain't Markup Language"
- YAML files are very popular for configurations
- DevOps tools such as Kubernetes and Azure DevOps Pipelines use YAML files for their configuration

YAML Files

Read YAML File

- Python does not provide YAML support within its Standard Library, so the `pyyaml` package needs to be installed

```
- id: 1
  name: apple
  price: 1.2
  qty: 2
- id: 2
  name: chicken
  price: 4.99
  qty: 4
...additional data omitted...
```

YAML: Food Data

*Note: "...additional data omitted.." is added for presentation purposes only,
it is not valid YAML*

```
import yaml

with open("./foods.yml") as yaml_file:
    foods = yaml.load(yaml_file)

    for food in foods:
        print(food["name"])
```

Python: Read a YAML File

YAML Files

Write YAML File

- Using the `yaml` module, Python can write YAML files as well

```
import yaml

foods = [
    {"id": 1, "name": "apple", "qty": 2, "price": 1.20},
    {"id": 2, "name": "chicken", "qty": 4, "price": 4.99},
    {"id": 3, "name": "ice cream", "qty": 1, "price": 4.49},
    {"id": 4, "name": "soda", "qty": 12, "price": 3.99},
    {"id": 5, "name": "bacon", "qty": 1, "price": 6.79},
]

with open('./foods.yml', 'w') as yaml_file:
    yaml_file.write(yaml.dump(foods))
```

Python: Write a YAML File

```
- id: 1
  name: apple
  price: 1.2
  qty: 2
- id: 2
  name: chicken
  price: 4.99
  qty: 4
...additional data omitted...
```

YAML: Food Data

Note: "...additional data omitted..." is added for presentation purposes only, it is not valid YAML

Conclusion

What we've learned...

- Python is commonly used for tasks such as system administration and data science
- Both tasks require working with configuration and data files
- Python's Standard Library provides modules for working with Configuration, JSON and CSV files
- Another format which is becoming increasingly popular for configuration files is YAML

Conclusion

What we've learned...

- Through the `pyyaml` package and its `yaml` module, YAML files can be read and written
- Python's rich support for the file system and popular file formats makes it a great language for managing files within larger applications or through system administration scripts



MASTERING PYTHON
Compression and Archiving

Overview

Compression and Archiving

- Compressing and archiving files are common operations performed when automating processes
- Python's Standard Library provides extensive support for many kinds of file compression such as GZip, BZip2, LZMA, and Zip
- For archiving files, both Tar and Zip are supported
- Python's extensive support for compression schemes and archive formats makes it ideal for writing system administration and file management scripts

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

GZip Compression

Compressing and Decompressing with GZip

- GZip is one of the most commonly used compression algorithms
- It's very popular on Linux for compressing archives and it is used to compress HTML requests/responses
- The GZip API is available through the `gzip` module

```
import gzip
import shutil

with open('colors.txt', 'rb') as source_file:
    with gzip.open('colors.txt.gz', 'wb') as target_file:
        shutil.copyfileobj(source_file, target_file)

with gzip.open('colors.txt.gz', 'rb') as compressed_file:
    file_content = compressed_file.read()
    print(file_content.decode('UTF-8'))
```

Python: Using GZip to Compress and Decompress a File

BZip2 Compression

Compressing and Decompressing with BZip2

- Another popular compression scheme is BZip2
- To use BZip2, the `bz2` module is used

```
import bz2
import shutil

with open('colors.txt', 'rb') as source_file:
    with bz2.open('colors.txt.bzip2', 'wb') as target_file:
        shutil.copyfileobj(source_file, target_file)

with bz2.open('colors.txt.bzip2', 'rb') as compressed_file:
    file_content = compressed_file.read()
    print(file_content.decode('UTF-8'))
```

Python: Using BZip2 to Compress and Decompress a File

LZMA Compression

Compressing and Decompressing with LZMA

- Another compression algorithm, LZMA, has become more popular because it gets a higher levels of compression

```
import lzma
import shutil

with open('colors.txt', 'rb') as source_file:
    with lzma.open('colors.txt.xz', 'wb') as target_file:
        shutil.copyfileobj(source_file, target_file)

with lzma.open('colors.txt.xz', 'rb') as compressed_file:
    file_content = compressed_file.read()
    print(file_content.decode('UTF-8'))
```

Python: Using LZMA to Compress and Decompress a File

Tar Archiving

Create an Archive

- Using the Python Standard Library `tarfile` module, a tar file can be created in Python code
- Using a `with` statement the `open` function, add each file or directory to the archive

```
import tarfile
from pathlib import Path

with tarfile.open(Path("./data-files.tar"), "w") as tar:
    tar.add(Path('./data-files'))
```

Python: Create a Tar Archive

Tar Archiving

Read and Extract an Archive

- Existing tar archives can be read and extracted

```
import tarfile
from pathlib import Path

with tarfile.open(Path("./data-files.tar"), "r") as tar:
    tar.extractall()
```

Python: Extract a Tar Archive

Tar Archiving

Archive and Compress

- The `tarfile` module support archiving and compressing in one single operation
- As part of the file mode, add `:gz` and the archive will be compressed as a GZip file

```
import tarfile
from pathlib import Path

with tarfile.open(Path("./data-files.tar.gz"), "w:gz") as tar:
    tar.add(Path('./data-files'))
```

Python: Create a Compressed Tar Archive

Tar Archiving

Read and Extract a Compressed Archive

- By adding the `:gz` to the file mode, a compressed tar archive can be read and extracted

```
import tarfile
from pathlib import Path

with tarfile.open(Path("./data-files.tar.gz"), "r:gz") as tar:
    tar.extractall()
```

Python: Extract a Compressed Tar Archive

Zip Archiving

Create a Zip Archive

- The Python Standard Library has support for Zip files as well
- Zip files produce compressed archives
- Using the `zipfile` module, a `zipFile` object is created and files are written to it

```
from pathlib import Path
from zipfile import ZipFile

data_files = Path("./data-files").iterdir()

with ZipFile(Path("./data-files.zip"), "w") as data_files_zip:
    for data_file in data_files:
        data_files_zip.write(data_file, arcname=data_file.name)
```

Python: Create a Zip Archive

Zip Archiving

Extract a Zip Archive

- Existing zip files can be decompressed and extracted as well

```
from pathlib import Path
from zipfile import ZipFile

with ZipFile(Path("./data-files.zip")) as data_files_zip:
    data_files_zip.extractall()
```

Python: Extract a Zip Archive

Zip Archiving

List a Zip Archive

- The `zipfile` module enables the inspection of zip file contents with almost file system directory type operations
- Using the `namelist` function, the member names of the archive can be retrieved and listed

```
from pathlib import Path
from zipfile import ZipFile

with ZipFile(Path("./data-files.zip")) as data_files_zip:
    for member_name in data_files_zip.namelist():
        print(member_name)
```

Python: List a Zip Archive

Zip Archiving

Create Zip Archive from a Directory Tree

- Using the `Path` class' `glob` function, a whole tree of files can be selected and added to the zip archive

```
from pathlib import Path
from zipfile import ZipFile, ZIP_BZIP2

sample_files = Path("./sample-files").glob('**/*')
sample_files_zip = Path("./sample-files.zip")

with ZipFile(sample_files_zip, "w", compression=ZIP_BZIP2, compresslevel=9) as zip_file:
    for sample_file in sample_files:
        zip_file.write(sample_file)
```

Python: Write a Directory Tree of Files to a Zip Archive

Zip Archiving

Get Files from Part of the Archived Directory Tree

- It's possible to get files from within a path of the zip archive
- Using the `ZipPath` class, a subtree within the zip file, a list of the contents can be retrieved

```
from pathlib import Path
from zipfile import ZipFile, Path as ZipPath

with ZipFile(Path("./sample-files.zip")) as sample_files_zip:

    directory_two_zip_path = ZipPath(
        sample_files_zip, at="sample-files/directory-two/")

    sorted_files = sorted([f.name for f in directory_two_zip_path.iterdir()
                           if f.is_file()])

    for directory_two_member_name in sorted_files:
        print(directory_two_member_name)
```

Python: Extract and Sort Files from a Specific Directory in the Archive

Conclusion

What we've learned...

- Compressing and archiving files are common operations performed when automating processes
- Python's extensive support for compression schemes and archive formats makes it ideal for writing system administration and file management scripts
- Python provides native support for many file compression schemes such as GZip, BZip2, LZMA, and Zip
- Also, Python provides APIs for creating and extracting files from Tar and Zip archives
- Python is a great language for managing files within larger applications or through system administration scripts



MASTERING PYTHON Sequences

Overview of Sequences

Sequence Types

- The three fundamental collection data type groups are sequences, sets and maps
- In this course, sequences will be explored
- There are 5 kinds of sequences in Python: strings, tuples, bytes, lists and byte arrays
- Of the 5 kinds: strings, tuples and lists are the most commonly used
- Sequences are finite, ordered sets indexed by non-negative numbers
- Sequences can be mutable or immutable
- Sequences support
 - Accessing items via index
 - Iteration
 - Slicing

Overview of Sequences

Sequence Types

- Three Immutable Sequences
 - Strings
 - Tuple
 - Bytes
- Two Mutable Sequences
 - Lists
 - Byte Arrays
- In this course, tuples, lists, their APIs, and related functions will be explored

Using Tuples

Small Sets of Arbitrary Objects

- Tuples are an immutable, ordered list of comma separated values
- Different data types can be stored in a tuple
- Tuples are a useful data structure on their own, and they form the basis of other more complex structures
- Tuples are created by constructing a comma separated list of literal values or variables
- Tuples are iterable, they can be iterated over with a for loop
- Tuples are useful for returning a small list of values from a function
- Extracting the individual values from a tuple is known as unpacking

Using Tuples

Small Sets of Arbitrary Objects

- The assignment of the values "1, 2, 3" creates a tuple
- The commas between the values is the operator which determines the value to be a tuple
- Optionally, parentheses can be wrapped around the tuple, but the parentheses do not define it to be a tuple
- For empty tuples, empty parentheses are used

```
items = 1, 2, 3  
  
# Outputs: <class 'tuple'> (1, 2, 3)  
print(type(items), items)
```

Python: Create a Tuple

```
items = ()  
  
# Outputs: <class 'tuple'> ()  
print(type(items), items)
```

Python: Create an Empty Tuple

Using Lists

Indexed Collection of Items

- Lists are the primary collection data type for managing a set of related items
 - The items do not have to be related, but commonly they are
 - Example: A List of People, A Matrix of Numbers
- Lists can be mutated including appending, inserting and removing items
- Lists can be a list of lists allowing the creation of matrices and other table structures
 - Working with matrices of numbers is typically done using the NumPy package (not covered here)
 - The NumPy package is hugely popular in data science
 - Learning Python lists is a good stepping stone to learning NumPy arrays

Using Lists

Indexed Collection of Items

- To create a list, a sequence of comma separated values is wrapped in square brackets
- The values can be any Python object including integers, floats, booleans, strings, other objects and even other lists
- Empty lists can be created simply using the square bracket with no sequence of comma separated values

```
# creates a list of three items
colors = [ 'red', 'blue', 'green' ]

# creates an empty list
colors = []
```

Python: Create a List

Mutating Lists

Mutation Functions

- Items can be appended, inserted and removed from the list
- `append(value)` – add the value to the end of the list

```
# add 'yellow' to the end of the list
colors.append('yellow')
```

Python: Append a String to the end of the List

- `insert(index, value)` – insert the value at the following index location

```
# insert 'orange' into the list at index 2
colors.insert(2, 'orange')
```

Python: Insert a String in the middle of the List

Mutating Lists

Mutation Functions

- `remove(value)` – removes the value from the list

```
# remove the first occurrence of 'blue' from the list  
colors.remove('blue')
```

Python: Remove a String from the List using the Item Value

- `pop()` – return and remove the value from the end of the list

```
# last item is removed and returned from pop  
popped_color = colors.pop()
```

Python: Remove a String from the end of the List

Mutating Lists

Mutation Functions

- `clear()` - remove all items from the list

```
# remove all items from the list  
colors.clear()
```

Python: Clear all of the Strings from the List

Ordering List Items

Sorting and Reversing

- In addition to changing the contents of a list, the order of items can be changed too
- `sort()` - sorts the list in-place

```
nums = [1, 4, 5, 3, 2]  
  
# nums is [1, 2, 3, 4, 5]  
nums.sort()
```

Python: Sort the List Items

- `reverse()` - reverses the items in the list (not a reverse sort)

```
nums = [1, 4, 5, 3, 2]  
  
# nums is [2, 3, 5, 4, 1]  
nums.reverse()
```

Python: Reverse the List Items

Ordering List Items

Sorting by Object Attributes

- When sorting objects with attributes, a **key** function can be passed to the **sort** function to return the value from the object to be used for sorting
- Commonly, the value is a particular attribute on the object as demonstrated here
- Also, the **sort** function supports a **reverse** parameter to sort a list in **reverse** (do not confuse this with the list **reverse** function)

```
class Person:  
    def __init__(self, fname, lname):  
        self.fname = fname  
        self.lname = lname  
    def __repr__(self):  
        return "{0} {1}".format(self.fname, self.lname)  
  
people = [  
    Person("Bob", "Smith"),  
    Person("Jane", "Hughes"),  
    Person("Mike", "Collins")  
]  
  
people.sort(key=lambda p: p.lname, reverse=False)  
print(people)
```

Python: Sort Person Objects

Functional Programming with Lists

Transforming Lists

- Python comes with several functions which can iterate over lists and produce new lists by applying a transformation or predicate function to each item
- A common transformation function is **map**
- With **map**, a transformation is passed into **map** along with an array
- The **map** function iterates over the list, passing each item into the transformation function
- The transformation function returns a new item which is added to a new list

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

def square(x):
    return x**2

squared_nums = list(map(square, nums))

for num in squared_nums:
    print(num)
```

Python: Transforming Lists

Functional Programming with Lists

Filtering Lists

- The **filter** function uses a predicate function to produce a new list of items which return true from the predicate function
- A predicate function is a function which receives a set inputs (commonly only one), and returns true or false based upon the logic applied to the input(s)
 - [https://en.wikipedia.org/wiki/Predicate_\(mathematical_logic\)](https://en.wikipedia.org/wiki/Predicate_(mathematical_logic))
- In the case of **filter**, a new array list values which return true for the predicate is produced

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

def greaterThan3(num):
    return num > 3

# 'filtered_nums' is all numbers from 'nums' greater than 3
filtered_nums = list(filter(greaterThan3, nums))
```

Python: Filtering List Items

Functional Programming with Lists

Reducing a List

- The **reduce** function is one of the most important functions for working with lists
- The **reduce** function is the one iterative function which can be used to implement all of the other iterative functions
- The basic idea of **reduce** is that it reduces the list to a single value
- The single value could be sum as shown below, or any other kind object including a new list

```
# imports 'reduce' function from 'functools' module
from functools import reduce
nums = [1, 2, 3, 4, 5]

# calculates the sum of the nums list
sum = reduce(lambda acc, current: acc + current, nums)
```

Python: Calculate the Sum of a List with Reduce

Working with Sequences

Sequence Slicing

- Copies part of an existing sequence into a new sequence
- Supports a special syntax when accessing an element in the list via an index value
- One index value returns a single item
- One index value followed by a colon, returns all the items in the list starting with the index value to the end of the list
- Two index values separated by a colon returns the items starting from an index up to but not including the second index value
- A colon followed by one index value returns the items from the first element up to but not including the second index value

```
nums = [1, 2, 3, 4, 5, 6, 7]
# outputs 3
print(nums[2])

# outputs [3, 4, 5, 6, 7]
print(nums[2:])

# outputs [3, 4]
print(nums[2:4])

# outputs [1, 2, 3, 4]
print(nums[:4])
```

Python: Slicing

Working with Sequences

Sequences and For Loops

- Python does not support traditional counter-based looping
- Instead, the **for-in-loop** centers around iterating over a collection of items
- When iterating over sequence, the iteration starts with the first item in the list and iterates over the list through the last item
- The **for-in-loop** populates an iterator variable with each item in the list
- The **for-in-loop** has a statement body where the iterator variable can be accessed

```
colors = ['red', 'yellow', 'green']

for color in colors:
    print(color)
```

Python: Iterate over a List with a For-In-Loop

Working with Sequences

Sequences and For Loops

- To access the index of each item, the **enumerate** function can be used
- The **enumerate** function returns a tuple containing the index and item
- The **enumerate** function supports an optional **start** parameter to start the index value at a non-zero value
- The default value for the **start** parameters is **None**, and the index starts at 0

```
colors = ['red', 'yellow', 'green']

for index, color in enumerate(colors):
    print(index, color)
```

Python: Iterate over a List extracting an Index

Conclusion

What we've learned...

- Python has a very rich type system with support for a number of types which are used to manage collections of data
- The three groups of collection types are sequences, sets and maps
- Sequences include the immutable strings, tuples and bytes, and include the mutable lists and byte arrays
- Strings, Tuples and Lists are commonly used in all Python programs
- Tuples are a list of arbitrary values stored under a single variable
- Lists are used to manage a list of arbitrary values (although the values are commonly related such as a list of people)
- Tuples and Lists support the index accessing capability of sequences
- Functional programming can be done using functions such as **map**, **filter**, and **reduce**



MASTERING PYTHON Dictionaries

Overview

What is a Dictionary?

- A dictionary is a mutable mapping type in Python
- Also known as associative arrays or maps in other languages such as PHP and JavaScript
- Dictionaries map hashable values to arbitrary objects
- Hashable values, known as keys, can be any immutable type (for example: strings, numbers, tuples, but not lists)
- The key can be mapped to any object (for example: strings, numbers, tuples, and lists)
- There are numerous ways to create a dictionary including JSON like syntax and the `dict` constructor where the arguments are key-value pairs (`**kwargs`) and each key-value pair is added to the dictionary

Overview

What is a Dictionary?

- The dictionary is mutable and the dictionary object has a full API to add items, retrieve items, make a clone of itself, etc...
- Also, the dictionary supports views which update themselves when the dictionary changes
- Finally, the keys and items of the dictionary are iterable

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

Create a Dictionary

dict Constructor

- Using the `dict` constructor, a new dictionary object is created
- Named parameters are passed into the `dict` constructor, captured with `**kwargs`, and are added as key-value pairs to the dictionary object
- The value assigned to the key is retrieved (`getitem`) and updated (`setitem`) using the indexer operator "[]" - square brackets

```
person = dict(  
    first_name="Sally",  
    last_name="Thompkins"  
)  
  
# outputs: Sally  
print(person["first_name"])
```

Python: Create a Dictionary with Named Parameters

Create a Dictionary

Curly Braces Like JSON

- In addition to the `dict` constructor, curly braces "{}" (similar to JSON), can be used to create a dictionary

```
person = {  
    "first_name": "Sally",  
    "last_name": "Thompson"  
}  
  
# outputs: Sally  
print(person["first_name"])
```

Python: Create a Dictionary with Curly Braces

Create a Dictionary

List of Tuples

- In addition to the `dict` constructor, an list of tuples can be used to create a dictionary
- The first item of the tuple will be the key, and the second item of the tuple will be the value

```
person = dict([
    ("first_name", "Sally"),
    ("last_name", "Thompson"),
])

# outputs: Sally
print(person["first_name"])
```

Python: Create a Dictionary with a List of Tuples

Create a Dictionary

Zipped Lists

- A dictionary can be constructed from two zipped lists
- The first list will be the key names, the second list will be the values

```
person = dict(zip(  
    ["first_name", "last_name"],  
    ["Sally", "Thompkins"],  
))  
  
# outputs: Sally  
print(person['first_name'])
```

Python: Create a Dictionary from Zipped Lists

Create a Dictionary

From Keys

- A dictionary can be constructed from a list of keys
- An optional default value can be specified, otherwise the value stored for each key will be null

```
contest_places = dict.fromkeys(["first", "second", "third"])

# outputs: [('first', None), ('second', None), ('third', None)]
print(list(contest_places.items()))

contest_places = dict.fromkeys(["first", "second", "third"], 'no winner')

# outputs: [('first', 'no winner'), ('second', 'no winner'), ('third', 'no winner')]
print(list(contest_places.items()))
```

Python: Create a Dictionary from Keys

Manage Dictionary Items

Add Dictionary Item

- Using the indexer operator and a new key, an item is added to a dictionary

```
html_colors = {
    "AliceBlue": "#F0F8FF",
    "AntiqueWhite": "#FAEBD7",
}

# add item
html_colors["Aqua"] = "#00FFFF"

# outputs: #00FFFF
print(html_colors["Aqua"])
```

Python: Add an Item to the Dictionary

Manage Dictionary Items

Get Dictionary Item, Add Item If Not Found

- The `setdefault` returns the value of an item if it present in the dictionary
- If the item is not in the dictionary it is added using the default value, and the value is returned

```
html_colors = {  
    "AliceBlue": "#F0F8FF",  
    "AntiqueWhite": "#FAEBD7",  
}  
  
# add item  
html_color = html_colors.setdefault("Aqua", "#00FFFF")  
  
# outputs: #00FFFF  
print(html_color)  
  
# outputs: True  
print("Aqua" in html_colors)
```

Python: Get Dictionary Item, Add Item If Not Found

Manage Dictionary Items

Remove Dictionary Item

- Using the `del` operator, an item is removed from a dictionary
- To check if a directory contains a key the `in` operator is used

```
html_colors = {
    "AliceBlue": "#F0F8FF",
    "AntiqueWhite": "#FAEBD7",
    "Aqua": "#00FFFF",
}

# remove item
del html_colors["Aqua"]

# outputs: False
print("Aqua" in html_colors)

# outputs: True
print("Aqua" not in html_colors)
```

Python: Remove an Item from the Dictionary

Manage Dictionary Items

Remove Dictionary Item By Key

- The `pop` removes an item by key value and returns it
- An optional second argument can be passed to `pop`, this will be the value returned if the key is not found in the dictionary

```
html_colors = {
    "AliceBlue": "#F0F8FF",
    "AntiqueWhite": "#FAEBD7",
    "Aqua": "#00FFFF",
}

# remove item by key
item_value = html_colors.pop("Aqua")

# output: #00FFFF
print(item_value)

# remove item by key
item_value = html_colors.pop("Some Key", "default value")

# output: default value
print(item_value)
```

Python: Remove an Item by Key from the Dictionary

Manage Dictionary Items

Remove Last Added Dictionary Item

- The `popitem` removes the last item added to the dictionary
- If dictionary is empty, a `KeyError` is raised

```
html_colors = {  
    "AliceBlue": "#F0F8FF",  
    "AntiqueWhite": "#FAEBD7",  
    "Aqua": "#00FFFF",  
}  
  
# remove last added item  
item_value = html_colors.popitem()  
  
# output: #00FFFF  
print(item_value)
```

Python: Remove Last Added Item from the Dictionary

Manage Dictionary Items

Remove All Items

- The `clear` function removes all items from the dictionary
- The `len` function returns the number of items in the dictionary

```
html_colors = {  
    "AliceBlue": "#F0F8FF",  
    "AntiqueWhite": "#FAEBD7",  
    "Aqua": "#00FFFF",  
}  
  
# output: 3  
print(len(html_colors))  
  
html_colors.clear()  
  
# output: 0  
print(len(html_colors))
```

Python: Remove All Items from the Dictionary

Dictionary Operations

Copy Dictionary

- The `copy` function makes a shallow copy of the dictionary

```
html_colors = {  
    "AliceBlue": "#F0F8FF",  
    "AntiqueWhite": "#FAEBD7",  
    "Aqua": "#00FFFF",  
}  
  
html_colors_copy = html_colors.copy()  
  
# outputs: {'AliceBlue': '#F0F8FF', 'AntiqueWhite': '#FAEBD7', 'Aqua': '#00FFFF'}  
print(html_colors_copy)
```

Python: Shallow Copy of the Dictionary

Dictionary Operations

Update Dictionary

- The `update` function will add new key-value pairs to an existing dictionary
- If the key already exists in the source dictionary, it is overwritten

```
html_colors = {  
    "AliceBlue": "#F0F8FF",  
    "AntiqueWhite": "#FAEBD7",  
    "Aqua": "#00FFFF",  
}  
  
html_colors.update({  
    "Aquamarine": "#7FFFAD",  
    "Azure": "#F0FFFF",  
})  
  
# outputs: {'AliceBlue': '#F0F8FF', 'AntiqueWhite': '#FAEBD7', 'Aqua': '#00FFFF',  
#           'Aquamarine': '#7FFFAD', 'Azure': '#F0FFFF'}  
print(html_colors)
```

Python: Shallow Copy of the Dictionary

Dictionary Operations

Merge Dictionaries

- The pipe "|" operator merges two dictionaries into a new dictionary object
- Keys are replaced right to left

```
html_colors_1 = {  
    "AliceBlue": "#F0F8FF",  
    "AntiqueWhite": "#FAEBD7",  
}  
  
html_colors_2 = {  
    "Aqua": "#00FFFF",  
    "Aquamarine": "#7FFFAD",  
}  
  
html_colors = html_colors_1 | html_colors_2  
  
# {'AliceBlue': '#F0F8FF', 'AntiqueWhite': '#FAEBD7',  
#  'Aqua': '#00FFFF', 'Aquamarine': '#7FFFAD'}  
print(html_colors)
```

Python: Merge Two Dictionaries into a New Dictionary

Access Dictionary Items

Retrieve a Dictionary Item

- The indexer operator "[key]" retrieves items from a dictionary
- If the key is not found, a Key Error is raised

```
html_colors = {  
    "AliceBlue": "#F0F8FF",  
    "AntiqueWhite": "#FAEBD7",  
    "Aqua": "#00FFFF",  
}  
  
# outputs #00FFFF  
print(html_colors["Aqua"])  
  
# outputs Key Error  
print(html_colors["Blue"])
```

Python: Retrieve a Value from the Dictionary

Access Dictionary Items

Retrieve an Item by Key, Return Default if Not Found

- The `get` function retrieves an item from the dictionary
- If a default value is specified and the key is not found, then the default value is returned, and no Key Error is raised

```
html_colors = {
    "AliceBlue": "#F0F8FF",
    "AntiqueWhite": "#FAEBD7",
    "Aqua": "#00FFFF",
}

# outputs Key Error
print(html_colors.get("Blue", "#0000FF"))
```

Python: Retrieve a Value If Present, Otherwise Return Default

Iterate over Dictionaries

Default Iterator

- The default iterator iterates over the keys of the dictionary
- The `iter` function iterates over the keys
- The `reversed` function reverses the keys

```
html_colors = {
    "AliceBlue": "#F0F8FF",
    "AntiqueWhite": "#FAEBD7",
    "Aqua": "#00FFFF",
}

for html_color_key in html_colors:
    # outputs each key
    print(html_color_key)

# ['AliceBlue', 'AntiqueWhite', 'Aqua', 'Aquamarine', 'Azure', 'Beige']
print(list(iter(html_colors)))

# ['Beige', 'Azure', 'Aquamarine', 'Aqua', 'AntiqueWhite', 'AliceBlue']
print(list(reversed(html_colors)))
```

Python: Iterating over a Dictionary

Iterate over Dictionaries

Iterate over Keys or Values

- Dictionary objects have a `keys` and `values` functions to return the keys and values of the dictionary, respectively
- The return value from the `keys` function is the `dict_keys` object, and the `values` function returns a `dict_values` object
- Both the `dict_keys` and `dict_values` objects are iterable

```
for html_color_key in html_colors.keys():
    # outputs each key
    print(html_color_key)

for html_color_value in html_colors.values():
    # outputs each value
    print(html_color_value)
```

Python: Explicitly Iterate over Keys or Values

Iterate over Dictionaries

Iterate over the Items

- Dictionary objects have an `items` function returns a `dict_items` object
- When the `dict_items` object is iterated over, it returns a list of tuples, one tuple for each item in the dictionary
- The first element of the tuple is the key, and the second element of the tuple is the value

```
for html_color_key, html_color_value in html_colors.items():
    print(f'{html_color_key}: {html_color_value}')
```

Python: Iterate over the Items of the Dictionary

Dictionary Views

View Objects

- The `dict_items`, `dict_keys`, `dict_values` objects returned from the `items`, `keys`, and `values` functions are view objects
- View objects provide a view of the original dictionary that is updated when the dictionary is updated

```
items = html_colors.items()  
  
# outputs: 6  
print(len(items))  
  
del html_colors["Aqua"]  
  
# outputs: 5  
print(len(items))
```

Python: Dictionary Items View Updates

Conclusion

What we've learned...

- Dictionaries are very useful data structure in Python
- Dictionaries map keys to values
- Keys can be any immutable value, while dictionary values can be any object
- Python provides a rich API for manipulating a dictionary object and performing operations between different dictionary objects
- Dictionary views enable the creation of views that update when the underlying dictionary object is changed



MASTERING PYTHON Comprehensions

Overview of Comprehensions

What is a Comprehension?

- Comprehensions are a concise syntax for creating lists, sets and dictionaries from any kind of sequence, set and mapping
- Generally comprehensions are referred to as list comprehensions, set comprehensions and dictionary comprehensions based upon the result type
- Conditions can be applied to comprehensions limiting the items from the original sequence, set, and mapping which are added to the new list, set or dictionary
- A lazy form of comprehension is a generator object and it can be passed to functions which expect to receive a generator
- Comprehensions enable concise yet easy to understand code to be written

Overview of Comprehensions

What is a Comprehension?

- Many comprehensions could be rewritten with `map`, `filter`, `reduce` and `lambdas` but such implementations can be slower and harder to understand¹
- Consider the code below, both examples generate a list of squared values:

```
squares = []
for x in range(10):
    squares.append(x**2)
```

Python: Generate with a Loop

```
squares = list(map(lambda x: x**2, range(10)))
```

Python: Generate with Functions

- Both example are verbose, and the second one is a little hard to read
- Consider the equivalent list comprehension:

```
squares = [x**2 for x in range(10)]
```

Python: Generate with a Comprehension

Working with List Comprehensions

Comprehension Syntax

- List comprehension syntax requires a `for-in` statement within square brackets []
- The order of operands is:
 - [<output expr> for <variable> in <iterable>]
- The result is a new list object which can be assigned to a variable or passed as an argument to a function

```
cart = [
    [ 'Coffee', 7.99, 2 ],
    [ 'Bread', 2.99, 1 ],
    [ 'Apple', 0.99, 2 ],
    [ 'Milk', 4.99, 1 ],
    [ 'Cola', 1.99, 4 ],
]

item_prices = [ item[1] for item in cart ]
# [7.99, 2.99, 0.99, 4.99, 1.99]
```

Python: Transform a List of Items to Prices with a List Comprehension

Working with List Comprehensions

Item Expression

- The expression for each item can simply return the item, an attribute of the item or run transformation which produces a new object using the item data
- Lists created by one list comprehension can be the iterable of another list comprehension

```
cart = [
    [ 'Coffee', 7.99, 2 ],
    [ 'Bread', 2.99, 1 ],
    [ 'Apple', 0.99, 2 ],
    [ 'Milk', 4.99, 1 ],
    [ 'Cola', 1.99, 4 ],
]

sales_tax = 0.07

item_prices = [ item[1] for item in cart ]

item_prices_plus_tax = [ round(price * (1 + sales_tax), 2)
                        for price in item_prices ]

# [8.55, 3.2, 1.06, 5.34, 2.13]
```

Python: Performing Calculations as part of a List Comprehension

Working with List Comprehensions

Item Variable

- Instead of an item variable a tuple can be used to create individual variables

```
cart_item_totals = [ item_name + ' ' + str(item_price * item_qty)
    for (item_name, item_price, item_qty) in cart]

cart_item_totals
```

Python: Using a Tuple to Create Independent Variables

- The iterable can be created from a range method such as generating a list of character codes
- When the item variable is not needed an underscore can fill its place

```
import random

letters = [ chr(code) for code in range(ord('a'), ord('z')) ]

random_letters = [ random.choice(letters) for _ in range(100) ]
```

Python: Using an Underscore to Ignore a Variable

Working with List Comprehensions

Practical Example

- List comprehensions can be paired with lambda functions to perform useful operations with map, reduce, filter, etc...
- The example below uses the reduce function with a lambda function to calculate the cart total using the item price and quantity

```
from functools import reduce

cart_subtotal = reduce(lambda x, y: x + y[0] * y[1], [item[1:] for item in cart], 0)

cart_subtotal
```

Python: Use a List Comprehension with the Reduce Function

Working with List Comprehensions

Nested Comprehensions

- One comprehension can be nested inside of another
- This can be useful, but it can impact readability
- Be careful about placing comprehensions inside the output expression, it will be re-evaluated on each iteration of the outer comprehension

```
cart = [
    [ 'Coffee', 7.99, 2 ],
    [ 'Bread', 2.99, 1 ],
    [ 'Apple', 0.99, 2 ],
    [ 'Milk', 4.99, 1 ],
    [ 'Cola', 1.99, 4 ],
]

sales_tax = 0.07

item_prices_plus_tax = [ round(price * (1 + sales_tax), 2)
    for price in [ item[1] for item in cart ] ]

item_prices_plus_tax

# [8.55, 3.2, 1.06, 5.34, 2.13]
```

Python: Nested List Comprehensions

Adding Conditions to Comprehensions

Optional Predicate

- When iterating over an iterable it is possible to apply a conditional expression to each item
- The conditional expression is optional and is called a predicate
- If the conditional expression returns **True** or truthy, the item expression is evaluated and the item is added to the new list
- If the conditional expression returns **False** or falsy, the item expression is not evaluated and the item is NOT added to the new list

```
cart_items_one_count = [ item for item in cart if item[2] == 1 ]  
# [['Bread', 2.99, 1], ['Milk', 4.99, 1]]
```

Python: Filter Items with a Quantity of 1

Exploring Dictionary Comprehensions

Creating a Dictionary

- In Python 3, dictionary comprehensions were added to the language
- A dictionary comprehension is the same as the list comprehension except for two syntax changes
- Dictionary comprehension syntax requires a `for-in` statement within curly braces `{ }`
- The order of operands is:
 - `{ <output key:value expression> for <variable> in <iterable> }`
- The result is a new dictionary object which can be assigned to a variable or passed as an argument to a function

```
food = {  
    "coffee": "beverage",  
    "pizza": "entree",  
    "cookie": "dessert",  
    "tea": "beverage",  
}  
  
beverages = {  
    k:v.upper()  
    for (k,v) in food.items()  
    if v == 'beverage'  
}  
  
# {'coffee': 'BEVERAGE', 'tea': 'BEVERAGE'}
```

Python: Dictionary Comprehension

Exploring Set Comprehensions

Creating a Set

- In Python 3, set comprehensions were added to the language
- A set comprehension is the same as the dictionary comprehension except for one syntax changes
- Set comprehension syntax requires a `for-in` statement within curly braces `{ }`
- The order of operands is:
 - `{ <output value expr> for <variable> in <iterable> }`
- The result is a new set object which can be assigned to a variable or passed as an argument to a function

```
food = {  
    "coffee": "beverage",  
    "pizza": "entree",  
    "cookie": "dessert",  
    "tea": "beverage",  
}  
  
food_groups = {  
    food_group  
    for food_group in food.values() }  
  
# {'beverage', 'dessert', 'entree'}
```

Python: Set Comprehension

Thinking of Comprehensions as Generators

"Generator" Comprehensions

- Instead of producing a list, set or dictionary directly, a comprehension be used a generator to lazily generate the values
- The generator can be passed to functions such as sum, min, and max which will invoke the generator to perform functions on the set of generated values

```
cart = [
    [ 'Coffee', 7.99, 2 ],
    [ 'Bread', 2.99, 1 ],
    [ 'Apple', 0.99, 2 ],
    [ 'Milk', 4.99, 1 ],
    [ 'Cola', 1.99, 4 ],
]

cart_item_count = sum(item[2] for item in cart)
# 10

max_item_count = max(item[2] for item in cart)
# 4
```

Python: Using List Comprehensions with Sum and Max

Conclusion

What we've learned...

- Comprehensions enable looping and conditional constructs to be evaluated as an expression and can be passed directly as arguments to methods or have their results assigned to variables
- There are three kinds of comprehensions: list, set and dictionary
- Each kind is determined by the kind of collection it produces
- In the comprehension, any kind of sequence, set or mapping can be used
- A lazy form of a comprehension produces a generator and may be referred to as a "generator" comprehension
- Comprehensions are a powerful, expressive syntax for writing Python code



MASTERING PYTHON

Operator Overloading

Overview of Operator Overloading

What does an operator do?

- Performing calculations such as adding numbers or concatenating strings is done all of the time in just about every programming language in existence
- The operations are so common and intuitive; rarely does a developer take a step back and ask what exactly does that "+" operator do?
- Why does the "+" operator in the context of two numbers perform an arithmetic operation and why does the "+" operator in the context of two string perform a concatenation?
- What if other types are used? Which operators only work with the same types, why do others work with different types?
- Can custom implementations of an operator be coded in Python?
- The ability to change the behavior of an operator in Python is known as operator overloading
- Python's built-in objects do lots of overloading and custom objects can do overloading too

Common Examples of Operator Overloading

Examples of Overloading in Python

- Consider the first example, two numbers are multiplied and an arithmetic operation is executed and the expected result is displayed
- Consider the second example, the same operator "*" is used but the first operand is a list not a number
- When a list is multiplied by a number it produces a new list where the original list is repeated the number of times by which it was multiplied
- In the final example, when a NumPy array (NumPy is a common Python library) is multiplied the values in the array are changed and not repeated

```
# 6  
3 * 2  
  
# [1, 2, 3, 1, 2, 3]  
[1,2,3] * 2  
  
import numpy as np  
  
# array([2, 4, 6])  
np.array([1,2,3]) * 2
```

Python: Examples of Overloading

Common Examples of Operator Overloading

Example of Pandas and Operators

- The Pandas library is another very popular library for Python for doing data analysis
- Pandas provided two data structures named Series and Data Frames
- Each of these complex structures can be used with operators to perform operations across all of the values
- Through the power of overloading operators these operations are done efficiently within the Pandas library and does not require the developer to write looping code in Python and manually perform each operation

```
import pandas as pd

pd.Series({ 'apples': 2, 'oranges': 4, 'grapes': 6 }) * 2

# apples      4
# oranges     8
# grapes     12
# dtype: int64

pd.DataFrame({
    'qty': pd.Series({
        'apples': 2, 'oranges': 4, 'grapes': 6 }),
    'amt': pd.Series({
        'apples': 1.49, 'oranges': 0.99, 'grapes': 2.49 })
}) * 2

#          qty  amt
# apples   4  2.98
# oranges  8  1.98
# grapes  12  4.98
```

Python: Pandas and Operator Overloading

Common Examples of Operator Overloading

Operators Do Not Work in All Contexts

- With all of those flexibility it would be tempting to think that operators have overloaded implementations for all operand contexts (all combinations of different types)
- In reality not all operations make sense for all object types
- Additionally, for custom objects the creator of the custom object must explicitly implement each desired overloaded operation

```
# '1,2,31,2,3'  
'1,2,3' * 2
```

Python: Using the Multiply Operator with a String and a Number

```
'2' * '2'  
-----  
TypeError Traceback (most rec  
<ipython-input-11-1fc9dc4d598a> in <module>  
----> 1 '2' * '2'  
  
TypeError: can't multiply sequence by non-int of type 'str'
```

Python: Multiply Operators does not work with Two Strings

Overloading Arithmetic Operators

Overloading Operators for Custom Classes

- In the class definition below, there are three special methods: `__init__`, `__add__`, `__str__`
- The method `__init__` is a class constructor (not covered here)
- The method `__str__` is the method called when an instance is coerced to a string value
- The method `__add__` is the method which overloads the "+" operator
- The operand on the left side of the "+" operator (`cart1`) is represented by `self` and the operand on the right side of the "+" operator (`cart2`) is represented by `another_cart`

```
class Cart:  
  
    def __init__(self, items): self._items = items  
  
    def __add__(self, another_cart):  
        return Cart(self._items + another_cart._items)  
  
    def __str__(self):  
        return str(self._items)  
  
cart1 = Cart([  
    'coffee creamer', 'butter', 'bottled water'  
])  
  
cart2 = Cart([  
    'bread', 'apples', 'apple cider vinegar'  
])  
  
str(cart1 + cart2)  
# "[['coffee creamer', 'butter', 'bottled water',  
#     'bread', 'apples', 'apple cider vinegar']]"
```

Python: Adding Two Cart Objects

Overloading Arithmetic Operators

Overloading Operators for Custom Classes

- In addition to `__add__`, there are many operators which can be overloaded
- To overload the "-" subtraction operator the special method `__sub__` is used
- To overload the "*" multiplication operator the special method `__mul__` is used
- To overload the "%" remainder operator the special method `__mod__` is used
- To overload the "/" division operator the special method `__truediv__` is used

```
class Cart:  
    # ...omitted...  
  
    def __mul__(self, qty_multiple):  
        return Cart(list(map(lambda item: {  
            'name': item['name'],  
            'qty': item['qty'] * qty_multiple,  
        }, self._items)))  
  
cart = Cart([  
    { 'name': 'coffee creamer', 'qty': 2 },  
    { 'name': 'butter', 'qty': 3 },  
    { 'name': 'bottled water', 'qty': 4 },  
])  
  
str(cart * 2)  
  
# "[{'name': 'coffee creamer', 'qty': 4},  
#      {'name': 'butter', 'qty': 6},  
#      {'name': 'bottled water', 'qty': 8}]"
```

Python: Override the Multiply Operator

Overloading Comparison Operators

Comparing Custom Classes

- Comparison operators can be overloaded: `__gt__ >`, `__lt__ <`, `__eq__ ==`, `__ne__ !=`, `__ge__ >=`, `__le__ <=`
- The name `self` is the operand on the left of the comparison operator, and the second parameter to the special method is the value on the right side of the comparison operator
- The operator should return `True`, `False` or a truthy/falsy value

```
class Cart:  
  
    # ...omitted...  
  
    def __gt__(self, another_cart):  
        return sum(item['qty'] for item in self._items) >  
            sum(item['qty'] for item in another_cart._items)  
  
    def __lt__(self, another_cart):  
        return sum(item['qty'] for item in self._items) <  
            sum(item['qty'] for item in another_cart._items)
```

Python: Overriding Comparison Operators

Overloading Comparison Operators

Comparing Custom Classes

- When comparing `cart1` and `cart2` the number of items in the carts is being used for the comparison

```
def __gt__(self, another_cart):  
    left = sum(item['qty']  
              for item in self._items)  
    right = sum(item['qty']  
               for item in another_cart._items)  
  
    return left > right
```

Python: Overloaded Greater-Than Operator

```
cart1 = Cart([  
    { 'name': 'coffee creamer', 'qty': 2 },  
    { 'name': 'butter', 'qty': 3 },  
    { 'name': 'bottled water', 'qty': 4 },  
])  
  
cart2 = Cart([  
    { 'name': 'orange', 'qty': 1 },  
    { 'name': 'apple', 'qty': 5 },  
    { 'name': 'bread', 'qty': 4 },  
])  
  
cart1 > cart2  
  
# False
```

Python: Comparing Two Carts

Overloading Index and Length

Accessing Items in an Object

- Using the special method `__getitem__` an index can be used on an object to access its data
- The data returned is determined by the custom implementation of the special method
- The second parameter can be used to access an item in a list by index (as shown here) or it could be used in numerous other ways to access other data structures and return many kinds of values
- The key is to make sure the indexing makes sense and is obvious to other developers using the class

```
class Cart:  
    # ...omitted...  
  
    def __getitem__(self, index):  
        return self._items[index]  
  
cart = Cart([  
    { 'name': 'coffee creamer', 'qty': 2 },  
    { 'name': 'butter', 'qty': 3 },  
    { 'name': 'bottled water', 'qty': 4 },  
])  
  
# {'name': 'butter', 'qty': 3}  
cart[1]
```

Python: Overriding the Get Item Operator

Overloading Index and Length

Calculating the Length

- Calculating the length of a list is self-explanatory, it is the number of items in the list
- Custom objects can have their length calculated too
- To support this the special method `__len__` must be implemented
- In the case of example, the length is calculated based upon the number of unique items in the cart
- The calculation could be any value which makes sense for the given class and its objects

```
class Cart:  
    # ...omitted...  
  
    def __len__(self):  
        return len(self._items)  
  
cart = Cart([  
    { 'name': 'coffee creamer', 'qty': 2 },  
    { 'name': 'butter', 'qty': 3 },  
    { 'name': 'bottled water', 'qty': 4 },  
])  
  
# 3  
len(cart)
```

Python: Overriding the Length Operator

Overloading Contains and Iterables

Checking if a Custom Object has Something

- The `in` operator in Python checks to see if a Python object contains something
- In the case of a custom class, the special method `__contains__` can be implemented to provide custom logic determining what it means to contain something for that particular class
- In the case of the example, the `__contains__` method looks for an item in the list of items with a particular name

```
class Cart:  
    # ...omitted...  
  
    def __contains__(self, item_name):  
        return len([ item  
                    for item in self._items  
                    if item['name'] == item_name]) > 0  
  
cart = Cart([  
    { 'name': 'coffee creamer', 'qty': 2 },  
    { 'name': 'butter', 'qty': 3 },  
    { 'name': 'bottled water', 'qty': 4 },  
])  
  
# True  
'butter' in cart
```

Python: Overriding the Contains Operator

Overloading Contains and Iterables

Iterating Over a Custom Object

- Objects like lists are iterable
- Lists can be used with the `for-in` loop and iterated over
- Custom objects can be iterated over as well
- The class definition needs a special method named `__iter__` which returns an iterable
- In the case of the example, the `items` list is returned but any kind structure that returns an iterable type could be used

```
class Cart:  
    # ...omitted...  
  
    def __iter__(self):  
        return iter(self._items)  
  
cart = Cart([  
    { 'name': 'coffee creamer', 'qty': 2 },  
    { 'name': 'butter', 'qty': 3 },  
    { 'name': 'bottled water', 'qty': 4 },  
])  
  
for item in cart:  
    print(item['name'])  
  
# coffee creamer  
# butter  
# bottled water
```

Python: Overriding the Iteration Operator

Conclusion

What we've learned...

- Overloading operators is an important useful feature for making Python more expressive and easy to work with
- The key to using operators is to understand the how various type combinations of operands behave
- In addition to the Python types' overloaded operators, many popular Python libraries such as NumPy and Pandas gain much of their power from the use of overloaded operators
- Using special methods on custom classes, classes created by Python developers can leverage operator overloading to make the usage of those classes more expressive and convenient
- Operator overloading eliminates a lot of code when performing basic operations
- Not all types combinations work with all operators, when overloading be sure to overload when it makes sense and makes the program more readable and easier to code



MASTERING PYTHON

Object-oriented Programming - Part 1

Overview

Object-oriented Programming

- Object-oriented programming (OOP) is a programming paradigm centered on programming with objects¹
 - Typically, software programs divide data (and the structures which hold data) and code (the code which computes results using the data) into two separate parts
 - For example, the C programming language uses structs to store structured data such as a record of information, and it uses functions to run logic that uses the struct data to compute results
 - In a C program, there is a division between data and the code (or logic) that processes it
1. https://en.wikipedia.org/wiki/Object-oriented_programming

Overview

Object-oriented Programming

- As a program grows a natural connection forms between certain parts of the data and certain parts of the code
- For example, the data may be a list of records, and there is a function which adds new records to the list
- While the data and function are independent within the program structure, their usage is ultimately quite dependent and very related
- In fact, wherever the list is needed, it can be quite helpful to have the function or functions present to manage the list as needed
- In recognition of this relationship, the object-oriented programming paradigm emerged

Overview

Object-oriented Programming

- Objects are the foundational building block of object-oriented programming
- Objects are structures which contain both data and code
- The code of the object are the functions/methods that directly relate to the data stored within the object
- For example, a list of data and methods to add/remove/replace items in the list could be packaged together as a list object
- To define an object, most programming languages, including Python, provide a `class` definition syntax
- The `class` syntax provides the ability to define the type of the object, the data it contains and the functions/methods that manage the data and make it available to other objects or other parts of the program

Overview

Object-oriented Programming

- The term `class` refers to the specification from which the `object` is created
- To better explain, let's relate object-oriented programming to building a home
- To build a home, a blueprint is needed
- The blueprint contains all of the information need to build a certain home
- Using the blueprint, a home is built
- A person does not live in a blueprint, but they can use a blueprint to build a real home, that they can live in
- Additionally, many homes can be built off of the same blueprint
- A `class` is the blueprint, a home is the `object`

Overview

Object-oriented Programming

- When defining classes there are several important factors: encapsulation, inheritance, and polymorphism
- Encapsulation is the concept of data hiding, the data within an object belongs to the object, and the object may choose to limit access to it
- There are numerous ways to implement data hiding through language features and patterns such as closures and access modifiers
- Some languages have very well defined, strict encapsulation features allowing data to be defined as public, protected, and private
- Python supports those three ideas, but does not provide strict mechanisms for them; rather, community-accepted conventions and patterns are employed

Overview

Object-oriented Programming

- The second important factor of classes is inheritance
- Most object-oriented programming languages provide a scheme for classes to inherit from classes, including Python
- With inheritance, a Python class can inherit data structures and functions/methods from a parent class so it does not need to redefine them
- The parent class is called the superclass, and the child class is called the subclass
- Most object-oriented programming languages limit a class to inheriting from one parent class (single inheritance)
- Python, similar to C++, supports multiple inheritance where a class can inherit from many classes

Overview

Object-oriented Programming

- The final factor is polymorphism
- With polymorphism, two objects with the same method attributes can be used interchangeably regardless of the implementation of those methods
- In traditional object-oriented languages such as C++, Java, or C#, this means an object created from a subclass, can be assigned to a variable typed as the superclass because the subclass inherits from the superclass
- The classic example is a superclass named `Animal` and subclasses named `Dog` and `Cat`
- A `Dog` or `Cat` is an `Animal` because they inherit from it
- So wherever an `Animal` object is used within the code, a `Dog` object or a `Cat` object can be used

Overview

Object-oriented Programming

- Because of Python's dynamic typing nature, polymorphism is less well-defined and ultimately reveals itself as duck-typing
- Duck-typing is not limited to objects in Python, but when duck-typing is used with objects, it is called polymorphism (from the perspective of OOP)
- Essentially duck-typing focuses on an object having certain attributes when needed at run-time
- If those attributes are present, then the code will work even if not formally defined through a rigid class structure
- Python supports duck-typing even without classes, but when combined with classes, duck-typing is thought of as polymorphism

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

Classes

Class Definition

- Classes are defined with the `class` keyword
- Class constructors are defined with the `__init__` function
- Class attributes are both data properties and function properties such as `first_name` and `full_name`, respectively
- Functions always receive an instance of the object as their first parameter which is commonly named `self`

```
class Person:  
  
    def __init__(self, first_name, last_name):  
  
        self.first_name = first_name  
        self.last_name = last_name  
  
    def full_name(self):  
        return f"{self.first_name} {self.last_name}"
```

Python: Basic Class Definition

Classes

Create an Object Instance from a Class

- Objects are instances of classes, many objects can be created from the same class definition
- Unlike most languages, the `new` keyword is not used in Python to create an object from a class
- The class name is invoked with the parameters specified by the `__init__` function

```
person = Person("Bob", "Smith")
print(person.full_name())
```

Python: Creating an Instance of a Class

Classes

Add Attributes

- While attributes are generally set in the class' `__init__` function or defined on the class, attributes can be added outside the class definition
- New functions can be added to the class itself
- New data attributes can be added to the object instance

```
def age_info(self):
    return f"{self.last_name} {self.age}"

Person.age_info = age_info

person = Person("Bob", "Smith")

person.age = 32

print(person.age_info())
```

Python: Add New Data and Function Attributes

Classes

Remove Attributes

- Attributes are removed from object with the `del` keyword
- Also, the entire object can be deleted with the `del` keyword

```
del person.age  
del person
```

Python: Delete Attribute and Delete Object

Inheritance

Inheriting Attributes from a Parent Class

- Through inheritance, one class receives the attributes from another class
- The child class inherits the parent class using parentheses after the class name, then inside the parentheses, the parent class name is specified
- The `super` function is used to access the parent class' functions including the `__init__` constructor function

```
class Student(Person):  
  
    def __init__(self, student_id, first_name, last_name):  
        super().__init__(first_name, last_name)  
        self.student_id = student_id  
  
    def record_info(self):  
        return f"{self.student_id} {self.last_name}, {self.first_name}"
```

Python: Class Inheritance

Inheritance

Accessing Attributes from a Parent Class

- When an object is created from the child class (subclass), it has all of the attributes from the parent class (superclass)
- The object can call the functions defined on both its class and the superclass
- The `issubclass` function will return true if a class is a subclass of another class
- The `isinstance` function will return true if an object is an instance of a class or a superclass

```
print(issubclass(Student, Person)) # True

student = Student(1, "Bob", "Smith")

print(student.full_name()) # Outputs: Bob Smith
print(student.record_info()) # Outputs: Smith, Bob 1

print(isinstance(student, (Student, Person))) # True
```

Python: Objects Created from Class Inheritance

Inheritance

Calling Superclass Functions

- Generally, a function defined on the superclass is used on the subclass instance unchanged
- But, it is possible, to override the superclass' function and implement a new one on the subclass
- Also, the subclass' new function can call the original superclass function as well

```
class Student(Person):  
  
    def __init__(self, student_id, first_name, last_name):  
        super().__init__(first_name, last_name)  
        self.student_id = student_id  
  
    def full_name(self):  
        return "Override: " + super().full_name()  
  
    def record_info(self):  
        return f"{self.student_id} {self.last_name}, {self.first_name}"
```

Python: Override Superclass Function in the Subclass

Access Modifier Patterns

Encapsulation

- One important aspect of class-based object-oriented development is the ability to configure data hiding known as encapsulation
- With class-based object-oriented development it is common define the access level of class members (class members are known as attributes in Python classes)
- Python does not provide explicit access modifier keywords like other languages; instead, common patterns (ex. prepending underscores to attributes) are used to indicate the kind of access each member has

Access Modifier Patterns

Public, Protected, and Private

- In classical object-oriented programming, there are three levels of access modifiers: public, protected, and private
- Public means a member (or attribute) is fully accessible inside and outside of the class, as well as within subclasses
- Protected means a member is not accessible outside of the class, but is accessible within the class and subclasses
- Private means a member is only accessible in the class where it is defined
 - Private members are not accessible outside of the class nor within subclasses

Access Modifier Patterns

Public and Private

- All of the class attributes defined to this point have been public
- To mark something as private, the attribute name is prefixed with `_` (double underscores)

```
class Person:  
  
    def __init__(self, first_name, last_name):  
  
        self.__first_name = first_name  
        self.__last_name = last_name  
  
    def full_name(self):  
        return f"{self.__first_name} {self.__last_name}"
```

Python: Class with Private Attributes

Access Modifier Patterns

Private Name Mangling

- To enforce the private access, the name of the private attribute is mangled so that when code outside of the class attempts to access the original attribute name it cannot
- Nevertheless, the mangled name itself is publicly accessible (if the mangled name is called directly), but it should never be used

```
person = Person("Bob", "Smith")

# public
print(person.full_name())

# error:
# AttributeError: 'Person' object has no attribute '__first_name'
# print(person.__first_name)
# attributes prefixed with double underscores mangles the public attribute name
# like this do not directly access the name mangled attributes
print(person._Person__first_name)
```

Python: Accessing Private Attributes

Access Modifier Patterns

Public Properties

- While classes encapsulate data with private attributes, there are times when such data should be made accessible to code outside of the class
- To enable public access of private data attributes, the `@property` decorator can be used
- The `@property` enables the defining of special accessor functions related to getting, setting, and deleting attributes
- A public property combined with a private attribute allows for the creation of private data with certain well defined ways to access, modify, or delete the data

Access Modifier Patterns

Public Properties

- In the code below, the `@property` decorator defines a function for retrieving the data of the `__first_name` private attribute
- There is a corresponding `@first_name.setter` decorator which is pair with the first property function to support changing the `__first_name` private attribute

```
class Person:

    def __init__(self, first_name, last_name):
        self.__first_name = first_name
        self.__last_name = last_name

    @property
    def first_name(self):
        return self.__first_name

    @first_name.setter
    def first_name(self, value):
        self.__first_name = value
```

Python: Using Properties with Private Attributes

Access Modifier Patterns

Public Properties

- Properties are accessed like normal data properties, except they run a function depending upon whether the property is used in assignment, retrieval or deletion
- Properties should never be called like functions from code outside the class

```
person = Person("Bob", "Smith")

# runs this function
# @first_name.setter
# def first_name(self, value):

    person.first_name = "Tim"

# runs this function
# @property
# def first_name(self):

    print(person.first_name)
```

Python: Set Private Attribute through a Public Property

Access Modifier Patterns

Protected Attributes

- Protected attributes that can be accessed within a class and by a subclass
- Attributes are marked protected by prefixing the name with one underscore
- Python does not enforce protected, but code external to the class should never reference an underscore attribute

```
class Person:  
  
    def __init__(self, first_name, last_name):  
  
        self._first_name = first_name  
        self._last_name = last_name  
  
class Student(Person):  
  
    def __init__(self, student_id, first_name, last_name):  
        super().__init__(first_name, last_name)  
        self.student_id = student_id  
  
    def record_info(self):  
        return " ".join([ self.student_id,  
                        f"{self._last_name},",  
                        self._first_name ])
```

Python: Protected Attributes and Inheritance

Class Methods

Methods Available to the Class or the Instance

- Class methods are functions that are accessible on the class object itself, without having to create an instance
- Additionally, class methods are accessible on the instance
- Often, class methods are used to define helper methods for creating new instances of a class with common sets of arguments

```
class Person:

    @classmethod
    def create(cls, full_name):
        first_name, last_name = full_name.split(' ')
        return Person(first_name, last_name)

    # other class functions omitted

person = Person.create("Bob Smith")
```

Python: Class Method

Iterable Classes

Iterating Over an Object

- Python provides many class "dunder" functions that tie the class and the objects it creates into Python language features
- Two such functions are `__iter__` and `__next__` and they enable an object to be used directly within a `for-in` loop

```
class Student:  
    # other class functions omitted  
  
    def __iter__(self):  
        self._course_index = 0  
        return self  
  
    def __next__(self):  
        if self._course_index < len(self._courses):  
            course = self._courses[self._course_index]  
            self._course_index += 1  
            return course  
        else:  
            raise StopIteration
```

Python: Defining Dunder Methods to make Class Instances Iterable

Iterable Classes

Iterate Over the Object

- The iterable object is then used within a `for-in` loop as normal
- The return value from the iteration can be anything
- In the code below, when the student is iterated over, the courses they are registered for are returned one at a time

```
historyCourse = Course("History")
scienceCourse = Course("Science")

student = Student(1, "Bob", "Smith")

student.appendCourse(historyCourse)
student.appendCourse(scienceCourse)

for course in student:
    print(course)
```

Python: Iterate Over a Student's Courses

Conclusion

What we've learned...

- In Python, all data types are objects
- Classes define new data types that include both data and code
- Classes are defined with the `class` keyword and are a blueprint for new objects
- Objects are instantiated from classes, but Python does not use the `new` keyword
- In Python, the class constructor is the `__init__` function
- Functions defined on the class are bound to the class instance when they are invoked, the class instance is implicitly passed as the first parameter to the class functions
- Python classes support inheritance of attributes (data and code)

Conclusion

What we've learned...

- Access modifiers such as public, protected, and private are implemented through community-accepted patterns and Python attribute name mangling
- Class methods are functions that are called from the `class` object
- Class can implement special dunder functions that connect into Python language features such as the `for-in` loop



MASTERING PYTHON
Object-oriented Programming - Part 2

Overview

Object-oriented Programming

- In Part 1, the essentials of object-oriented programming with Python classes and objects were explored
- In Part 2, some additional patterns and language features related to classes will be explored
- The first half of the course will explore reusable class patterns including multiple inheritance, the mixin pattern, and class composition
- The second half of the course will explore alternative class patterns including topics such as abstract classes and the dependency injection pattern

Overview

Reusable Patterns

- Reusing code is an essential aspect of software development
- When working with classes, we learned in part 1 that one class can inherit attributes (especially function attributes) from a parent class
- In this course, Python class-based inheritance will be expanded upon
- First, Python supports multiple inheritance, we will discuss the how this works, and more importantly why (and why not) this language feature is used
- Expanding upon multiple inheritance, the mixin pattern will be explored demonstrating one way in which multiple inheritance can be beneficial
- Finally, we will compare/contrast another reusable pattern called composition and how it relates to inheritance

Overview

Alternative Patterns

- A common technique used in object-oriented programming and duck-typing in Python is the ability to have alternative implementations of the same class structure (polymorphism)
- One way to implement this in Python is through abstract classes and dependency injection
- With abstract classes, a class is defined (some function attributes are abstract and some fully implemented), but the class itself is never instantiated
- Then, other concrete classes inherit from the abstract class, using the functions defined by the class and/or providing implementations of those functions
- Selecting which concrete class is instantiated at run-time is accomplished through dependency injection

Development Environment

Getting Started

- For this course, you will need Python 3 to be installed and a text editor to write your Python code
- While there are many ways to install Python 3, and many text editors available, the following recommendations are made:
- Install Python 3
 - Windows - install Python 3 from the Python web site <https://www.python.org/>
 - macOS - install Python 3 with Homebrew <https://brew.sh/>

Development Environment

Getting Started

- For the text editor, Visual Studio Code, with Microsoft's Python extension, is recommended (<https://code.visualstudio.com/>)
 - Microsoft's Python extension provides a rich Python programming experience
 - Also, the Python extension includes Jupyter Notebooks and even helps to debug them

Multiple Inheritance

Inheriting from Multiple Classes

- Simply speaking, multiple inheritance is nothing more than a class inheriting from multiple parent classes
- For example, in the code below the `Child` class will inherit class definitions from both `ParentA` and `ParentB`
- In the simplest of scenarios, multiple inheritance works great, but when the multiple inheritance pattern becomes complex, multiple inheritance can lead to confusion

```
class ParentA:  
    ...  
  
class ParentB:  
    ...  
  
class Child(ParentA, ParentB):  
    ...
```

Python: Multiple Inheritance

Multiple Inheritance

What is the Super in the Child Class?

- Which class does the `super` function return? `ParentA` or `ParentB`?
- To see the method resolution order, the Child class has a special dunder property named `__mro__`
- Using the value of the `__mro__` property, the path of `__init__` method resolution can be seen

```
class ParentA:  
    ...  
  
class ParentB:  
    ...  
  
class Child(ParentA, ParentB):  
  
    def __init__(self):  
        # Is super() ParentA or ParentB?  
        super().__init__()
```

Python: What is Super?

```
# (<class '__main__.Child'>, <class '__main__.ParentA'>, <class '__main__.ParentB'>, <class 'object'>  
print(Child.__mro__)
```

Python: Dunder MRO

Multiple Inheritance

What is the Super in the Child Class?

- The order `ParentA` and `ParentB` in the class `Child` definition does impact the `__mro__`
- For complex class hierarchies, the method resolution order can be complex
- Python computes this order using the C3 Linearization algorithm
 - https://en.wikipedia.org/wiki/C3_linearization

```
class ParentA:  
    ...  
  
class ParentB:  
    ...  
  
class Child(ParentA, ParentB):  
    def __init__(self):  
        # Is super() ParentA or ParentB?  
        super().__init__()
```

Python: What is Super?

- The method resolution order is determined by the class structure in the Python code

Mixin Pattern

Useful Multiple Inheritance

- Very few programming languages support multiple inheritance
- Multiple inheritance can be confusing to follow and can lead to problems such as the diamond problem
- Nevertheless, when used properly, multiple inheritance can be useful
- One such useful pattern in Python is the mixin pattern
- With the mixin pattern, generic/common functionality on one class can be inherited by another class
- To use this pattern, the mixin class cannot inherit from any other classes

Mixin Pattern

Useful Multiple Inheritance

- The `ColorList` class inherits from multiple classes
- The `DictionaryList` class provides the primary functionality for `ColorList`
- The mixin classes add extra functionality to further enhance `ColorList`
- Observe the `JsonOutputMixin` class does not inherit from another class and it does not have an `__init__` function

```
class JsonOutputMixin:  
  
    def to_json(self):  
        return json.dumps(self._items)  
  
class ColorList(DictionaryList, JsonOutputMixin,  
                 CsvOutputMixin, YamlOutputMixin):  
  
    def append(self, color_name, color_hexcode):  
        super().append(name=color_name,  
                      hexcode=color_hexcode)
```

Python: Color List inherits from Multiple Mixins

```
colors = ColorList()  
colors.append("red", "ff0000")  
colors.append("green", "00ff00")  
colors.append("blue", "00ffff")  
  
print(colors.to_json())
```

Python: Using the Json Mixin Function

Composition

Relationship Between Two Classes

- A popular alternative to inheritance is composition
- Inheritance and composition model two different kinds of class relationship
- Consider three classes: Vehicle, Car, Engine
 - A Car "**is a**" Vehicle, so the relationship is one of inheritance
 - A Car "**has an**" Engine, so the relationship is one of composition
- A Car needs the functionality of the Engine just like it needs the functionality of the Vehicle, but incorporating that functionality is different based upon whether it's an "**is a**" or "**has a**" relationship
- Commonly, composition is used to avoid multiple inheritance

Composition

Relationship Between Two Classes

- Composition is implemented by adding a new attribute to a class of the type being composed in
- While the Car can access the protected members of the Vehicle class, it cannot access the protected members of the Engine class
- But, the Car can utilize the public members of the Engine object within the methods of the Car class

```
class Vehicle:  
    ...  
  
class Engine:  
    ...  
  
class Car(Vehicle):  
  
    def __init__(self, engine):  
        self.engine = engine # Engine Class Object
```

Python: Class Inheritance and Composition

Abstract Classes

Metaclasses

- Abstract classes are classes with a definition of attributes, but the class itself cannot be instantiated
- The attributes can be fully implemented or they can be contract-only, leaving the subclass of the abstract class to implement it
- Classes that inherit from an abstract class and can be instantiated are called "concrete" classes
- Abstract classes are created using the `ABCMeta` metaclass from Standard Library's `abc` module
 - There is also an `ABC` helper class that can be used as well

Abstract Classes

Metaclasses

- The `ABCMeta` class is a metaclass used to define classes as abstract classes
- Abstract classes can inherit from the `ABC` helper class or the `metaclass` option can be specified on the abstract class

```
from abc import ABC

class MyAbstractClass(ABC):
    pass
```

Python: Inherit from ABC Helper Class

```
from abc import ABCMeta

class MyAbstractClass(metaclass=ABCMeta):
    pass
```

Python: Metaclass Option

Abstract Classes

Abstract Methods

- Generally, abstract classes are used to provide shared functionality to concrete classes (classes which can be instantiated) or to define methods that must be implemented on classes that inherit them
- Abstract methods are methods which need to be defined on the concrete class, but are not implemented in on the abstract class
- The `@abstractmethod` decorator can be combined with class methods, static methods, and properties, but it must be the last decorator applied to the attribute

```
class Logger(metaclass=ABCMeta):  
    @abstractmethod  
    def log(self, entry):  
        ...
```

Python: Abstract Class with Abstract Method

Dependency Injection

Injecting Dependencies at Run-Time

- Traditionally, programs determined their specific dependencies at compile-time
- Whatever class was specified when the program was compiled would be the class used when the code executed
- As object-oriented programming matured and common patterns of usage emerged, the dependency injection pattern came into use
- The idea was that a dependency class contract could be specified at compile-time, then when the program executed one of any number of classes which implemented the contract could be used to fulfill the request for the dependency

Dependency Injection

Injecting Dependencies at Run-Time

- The key part is that the selection of the specific class to fulfill the dependency did not occur until run-time
- At run-time when a particular dependency was needed, any class which satisfied the contract could be used to instantiate the object and satisfy the dependency
- To determine which class would be used was settled by programming code and data

Dependency Injection

Getting Started

- Unfortunately, Python's Standard Library does not provide an official implementation of the dependency injection pattern
 - In other languages, dependency injection is usually provided as part of a framework such as a web application framework, it is not directly provided in the language itself
- To use dependency injection, you could write your own implementation that would include a container to configure the injected dependencies and other utility functions such as an inject decorator to execute the injections
- Another option (and probably a better one) is to use one of the many packages available for dependency injection

Dependency Injection

Getting Started

- One such package is the Dependency Injector Package available on PyPi:
<https://pypi.org/project/dependency-injector/>



Logo: Dependency Injector (<https://github.com/ets-labs/python-dependency-injector>)

- It is installed within your project using the following command:

```
python -m pip install dependency-injector
```

Command-Line: Install the Dependency Injection Python Package

Dependency Injection

Container

- For dependency injection to work, it needs to know what class to instantiate for each dependency
- A single container is created to configure a class for each dependency

```
class Container(containers.DeclarativeContainer):  
  
    logger_service = providers.Factory(  
        CSVLogger,  
        file_name="log.txt",  
    )  
  
    colors_list_service = providers.Factory(  
        ColorsList,  
        logger=logger_service,  
    )
```

Python: Dependency Injection Container

- When the code runs, the container retrieves the object instance (creating it, if necessary) of the dependency
- While the focus here is resolving a dependency to a class instance, the resolved dependencies are not limited to class instances but can be any kind of data

Dependency Injection

Wiring Up a Container

- An instance of the container is created and wired to the module to support the `@inject` decorator

```
if __name__ == '__main__':
    container = Container()
    container.wire(modules=[sys.modules['__name__']])

    main()
```

Python: Wire Up a Container

Dependency Injection

Injecting a Dependency into a Function

- The `@inject` decorator is applied to the main function and a marker `Provide[Container.colors_list_service]` is assigned to the parameter
- When the function is invoked, the `colors_list_service` will be passed into the main function through the `colors_list` parameter, automatically

```
@inject
def main(colors_list=Provide[Container.colors_list_service]):
    ...
```

Python: Inject Dependencies into a Function with the Inject Decorator

Dependency Injection

Injecting a Dependency into a Service

- The `Logger` instance will be passed into the constructor of the `ColorsList` service when the injector creates an instance of the service
- The specific `Logger` class is defined inside of the container class explored earlier

```
class ColorsList:  
    """Colors list class"""\n\n    def __init__(self, logger):  
        self.colors = []  
        self.logger = logger
```

Python: Injecting Dependencies through the Contructor

Conclusion

What we've learned...

- Python is one of the few languages that supports multiple inheritance
- While multiple inheritance can lead to some problems, it also can be used effectively through patterns such as the mixin pattern
- Inheritance models an "is a" pattern, and composition models a "has a" pattern
- Both inheritance and composition are important parts of building object-oriented programs with Python
- Python provides built-in support for abstract classes through metadata programming
- Also, it's possible to utilize the dependency injection pattern in Python using Python packages that implement the dependency injection pattern



MASTERING PYTHON
Concurrent and Parallel Processing

Overview

Processing Models

- The computing power of servers, desktops and even mobile devices is incredible
- With a variety of technologies such as hyper-threading, multiple cores, and simply fast CPU processing, computers can perform many operations very fast
- To leverage these hardware features there are many concurrent processing models that can be used
 - Multi-threading
 - Mutil-processing
 - Async/Await
 - Subprocesses
- The appropriateness of each of these processing models is dependent upon the problem to be solved

Overview

Concurrent and Parallel Processing

- Concurrent and Parallel processing is a broad term used to describe doing multiple things at the same time
- The idea of "at the same time" can be literally "at the same time" or "nearly at the same time"
- The "nearly at the same time" model, known as Concurrent Processing, means that two blocks of code share a single processing unit (CPU or Core) and are interleaved, they are not executed in sequence
- The "literally at the same time" model, known as Parallel Processing, means that each block of code is executing on its own processing unit (CPU or Core) at the exact same time
- From the user perspective Concurrent and Parallel Processing can appear to the same
- The big difference between Concurrent and Parallel processing that Concurrent Processes share a processing unit and Parallel Processes each have their own processing unit

Overview

Concurrent and Parallel Processing

- Concurrent Processing
 - On a hardware level is implemented with hyper-threading and fast CPU
 - On software level is implemented with threading and asynchronous programming
- Parallel Processing
 - On a hardware level is implemented with multiple CPUs or multiple CPU Cores
 - On a software level is implemented with multiple processes and subprocesses

Overview

Concurrent and Parallel Processing with Python

- Python's Standard Library provides numerous APIs for concurrent and parallel processing
- Concurrent Processing
 - `threading` module provides APIs to create and manage multiple threads within a Python program
 - `asyncio` module provide APIs for asynchronous programming
 - `concurrent.futures` module provides APIs for launching multiple thread (concurrent) tasks
- Parallel Processing
 - `multiprocessing` module provides APIs to launch and manage multiple processes within a Python programm
 - `subprocess` module provides APIs for running external programs including capturing their output
 - `concurrent.futures` module provides APIs for launching multiple process (parallel) tasks

Threading

Create a Thread

- The following code uses the `Thread` class and a function to start a new thread

```
def do_it() -> None:
    my_thread = threading.current_thread()
    print(f"{my_thread.ident} - starting thread")
    print(f"{my_thread.ident} - thread name: {my_thread.name}")
    print(f"{my_thread.ident} - thread native id: {my_thread.native_id}")
    print(f"{my_thread.ident} - exiting thread")

def main() -> None:
    thread_1 = threading.Thread(target=do_it, name="MyFirstThread")
    thread_1.start()
```

Python: Create a New Thread

Threading

Pass Arguments to a Thread

- Arguments are passed to the new thread object using the `args` parameter

```
import threading

def do_it(some_data: str) -> None:
    my_thread = threading.current_thread()
    print(f"{my_thread.ident} - starting thread")
    print(f"{my_thread.ident} - thread some data: {some_data}")
    print(f"{my_thread.ident} - exiting thread")

def main() -> None:
    thread = threading.Thread(target=do_it, args=('some message',))
    thread.start()
```

Python: Pass Arguments to a Thread

Threading

Thread Local Data

- The threading API provides a `local` function to create thread local data
- The local data object is created outside the thread, then referenced inside the thread
- The local data in the thread will be the data for the thread that is executing

```
local_data = threading.local()

def other_stuff() -> None:
    my_thread = threading.current_thread()
    print(f"{my_thread.ident} - other stuff: {my_thread.name} == {local_data.thread_name}")

def do_it() -> None:
    my_thread = threading.current_thread()
    print(f"{my_thread.ident} - starting thread")
    local_data.thread_name = my_thread.name
    other_stuff()
    print(f"{my_thread.ident} - exiting thread")
```

Python: Thread Local Data

Threading

Thread Class

- In addition to creating a thread with the `Thread` class and a function, a new thread can be created from a class that inherits from the `Thread` class

```
import threading

class DoItThread(threading.Thread):

    def __init__(self, some_data: str) -> None:
        threading.Thread.__init__(self)
        self.some_data = some_data

    def run(self) -> None:
        my_thread = threading.current_thread()
        print(f"{my_thread.ident} - starting thread")
        print(f"{my_thread.ident} - thread some data: {self.some_data}")
        print(f"{my_thread.ident} - exiting thread")

    def main() -> None:
        thread = DoItThread("some message")
        thread.start()
```

Python: Thread Class

Async IO

Perform an Asynchronous Operation

- Another approach to multi-threaded coding asynchronous programming with `async/await`

```
async def main_async() -> None:  
  
    start_date = date(2019, 1, 1)  
    end_date = date(2019, 2, 28)  
  
    async with aiohttp.ClientSession() as session:  
  
        for single_date in business_days(start_date, end_date):  
            single_date_str = single_date.strftime("%Y-%m-%d")  
            url = f'https://api.ratesapi.io/api/{single_date_str}?base=USD&symbols=EUR,CAD'  
            async with session.get(url) as resp:  
                rates.append(await resp.json())  
  
    rates = []  
  
    loop = asyncio.get_event_loop()  
    loop.run_until_complete(main_async())
```

Python: Async/Await Http Requests

Launching Multiple Threads

Multiple Thread Objects

- When writing multi-threaded code it is common to create multiple threads
- Multiple threads make sense when code is IO bound, CPU bound does not typically benefit from multiple threads

```
def main() -> None:  
  
    thread_1 = threading.Thread(target=do_it, name="MyFirstThread")  
    thread_1.start()  
  
    thread_2 = threading.Thread(target=do_it, name="MySecondThread")  
    thread_2.start()
```

Python: Multiple Thread Objects

Launching Multiple Threads

Concurrent Futures - Threads

- One way of launching multiple threads and collecting their results is to use the Thread Pool Executor

```
from concurrent.futures import ThreadPoolExecutor, wait
from random import randint
import time, threading

def task() -> str:
    my_thread = threading.current_thread()
    print(f"starting task {my_thread.ident}")
    time.sleep(randint(1,5))
    print(f"stopping task {my_thread.ident}")
    return f"thread {my_thread.ident} done"

with ThreadPoolExecutor() as executor:
    for future in wait([executor.submit(task) for _ in range(3)]).done:
        print(future.result())
```

Python: Launching Multiple Threads with the Thread Pool Executor

Processes

Create a Process

- Similar to threads, new processes can be created using the `Process` class

```
import multiprocessing as mp

def do_it() -> str:
    my_process = mp.current_process()
    print(f"{my_process.ident} - starting process")
    print(f"{my_process.ident} - process name {my_process.name}")
    print(f"{my_process.ident} - stopping process")
    return f"{my_process.ident} - process done"

def main() -> None:
    process_1 = mp.Process(target=do_it, name="ProcessOne")
    process_1.start()
```

Python: Create and Start a Process

Processes

Process Class

- Processes can be defined with a class that inherits from the `Process` class

```
import multiprocessing as mp

class DoItProcess(mp.Process):

    def __init__(self, some_data: str) -> None:
        mp.Process.__init__(self)
        self.some_data = some_data

    def run(self) -> None:
        my_process = mp.current_process()
        print(f"{my_process.ident} - starting process")
        print(f"{my_process.ident} - process some data {self.some_data}")
        print(f"{my_process.ident} - stopping process")

    def main() -> None:
        process_1 = DoItProcess("some message")
        process_1.start()
```

Python: Pass Arguments to a Process

Launching Multiple Processes

Create Multiple Process Objects

- Multiple processes can be created as well
- The number of processes should generally not exceed the number of CPUs or Cores

```
import multiprocessing as mp

def do_it() -> str:
    my_process = mp.current_process()
    print(f"{my_process.ident} - starting process")
    print(f"{my_process.ident} - process name {my_process.name}")
    print(f"{my_process.ident} - stopping process")
    return f"{my_process.ident} - process done"

def main() -> None:
    process_1 = mp.Process(target=do_it, name="ProcessOne")
    process_1.start()

    process_2 = mp.Process(target=do_it, name="ProcessTwo")
    process_2.start()
```

Python: Create and Start a Process

Launching Multiple Processes

Process Pool

- Multiple processes can be created and managed using a process pool

```
from multiprocessing import Pool

def f(x: int) -> int:
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

Python: Process Pool

Launching Multiple Processes

Concurrent Futures - Processes

- Another approach to managing multiple processes is the Process Pool Executor

```
from concurrent.futures import ProcessPoolExecutor, wait
from random import randint
import time, os

def task() -> str:
    process_id = os.getpid()
    print(f"starting process {process_id}")
    time.sleep(randint(1,5))
    print(f"stopping process {process_id}")
    return f"process {process_id} done"

def main() -> None:
    with ProcessPoolExecutor() as executor:
        for future in wait([executor.submit(task) for _ in range(3)]).done:
            print(future.result())
```

Python: Multiple Processes with the Process Pool Executor

Subprocesses

Create a Subprocess

- Processes created with the `Process` class are new processes running code within the primary application
- Subprocesses are separate programs launched by the Python application, the separate program is not part of the code base for the Python application
- When using the `run` function the output of program cannot be captured, but the exit code is returned

```
import subprocess

result = subprocess.run("where chkdsk.exe")

print(f"result: {result}")
print(f"return code: {result.returncode}")

result = subprocess.run("where chkdsk12.exe")

print(f"result: {result}")
print(f"return code: {result.returncode}")
```

Python: Run a Subprocess

Subprocesses

Create a Subprocess

- Using the `Popen` constructor, a subprocess can be created which allows the Python program to capture the output of the program and its exit code

```
import subprocess

p = subprocess.Popen(
    "where chkdsk.exe",
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT)

retval = p.wait()
print(retval)

if p and p.stdout:
    for line in p.stdout.readlines():
        print(str(line, "UTF-8"))
```

Python: Run a Subprocess and Capture Results



MASTERING PYTHON Socket Programming

Overview

Sockets

- Python provides low-level networking through sockets
- The `socket` module is a low-level networking module provided as part of Python's Standard Library
- The sockets API is very similar to the C socket library
- The sockets module provides support for numerous hardware network devices and communication protocols

Using Sockets

Open a Server Socket Connection

- The `socket` function provides a context manager to manage the closing of the socket connection
- The constant `AF_INET` sets up an IP4 connection
- The constant `SOCK_STREAM` sets up a TCP connection

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as socket_server:
```

Python: Open a Socket with the Context Manager

Using Sockets

Server Socket Bind and Listen

- The socket object is bound to an IP address and a port
- The `listen` function connects the socket to the IP address and port to receive connections

```
socket_server.bind(('127.0.0.1', 5000))  
socket_server.listen()
```

Python: Bind to Host and Port, and Listen for Connections

Using Sockets

Connection and Send/Receive Data

- The `accept` function waits for a new connection from a client, when the connection is established it returns a tuple of a socket object and the client address information
- The `recv` function waits to receive data from the client
- The `sendall` function sends data to the client
- The data received from the client and sent to the client is stored as bytes
- To extract string data, the data must be encoded to a UTF-8 string
- To send string date, the string data must be encoded to a UTF-8 string

```
conn, addr = socket_server.accept()  
  
data = conn.recv(2048)  
  
conn.sendall(data)
```

Python: Manage a Client Connection

Using Sockets

Open a Client Socket

- Client sockets are opened the same way as server sockets
- The `socket` module is a low-level networking module provided as part of Python's Standard Library
- The `socket` function provides a context manager to manage the closing of the socket connection
- The constant `AF_INET` sets up an IP4 connection
- The constant `SOCK_STREAM` sets up a TCP connection

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as socket_client:
```

Python: Open a Client Socket

Using Sockets

Open a Client Socket

- The server's IP address and port are passed to the client socket's `connect` function to connect to the server
- Using a `b` string, the string is converted to bytes, and sent to the server with the `sendall` function
- The `recv` function receives data from the server

```
socket_client.connect(("127.0.0.1", 5000))  
  
socket_client.sendall(b"sample data")  
  
data = socket_client.recv(2048)
```

Python: Connect to the Server and Send/Receive Data



MASTERING PYTHON
Database Programming

Overview

Database Programming

- Python defines the DBI (Database Interface) so that different database drivers can implement similar APIs
- Python provides an API for the SQLite in the Standard Library
- For other databases, a third-party package is needed
- There are many packages for popular database such as Postgresql, MySQL, Oracle and SQL Server
- In this course, the `pyodbc` package will be used

Install Database Package

pyodbc Package

- The `pyodbc` package can be used to connect to multiple kinds of database including SQL Server (both the local version and the Azure version)
 - Package Info: <https://pypi.org/project/pyodbc/>
 - Documentation: <https://github.com/mkleehammer/pyodbc/wiki>
- To install the `pyodbc` package, run the following command

```
pip install pyodbc
```

Python: Install pyodbc Package

Using pyodbc

Connect to the Database

- A connection string is needed to connect to a database
- The `connect` function provides context manager so that the connection can be managed with a `with` statement
- The context manager will commit any SQL statements executed against the database

```
conn_options = [
    "DRIVER={ODBC Driver 17 for SQL Server}",
    r"SERVER=localhost\SQLExpress",
    "DATABASE=somedb",
    "Trusted_Connection=yes",
]

conn_string = ";" .join(conn_options)

with pyodbc.connect(conn_string) as con:
```

Python: Connect to the Database

Using pyodbc

Query the Database

- The `execute` function on the connection object can be used to query the database

```
rates = con.execute("select ClosingDate, CurrencySymbol from Rates")  
  
for rate in rates:  
    print(rate.ClosingData, rate.CurrencySymbol)
```

Python: Query the Database

Using pyodbc

Parameterized Query

- To parameterize queries, question marks are used as placeholders for query parameters
- Query parameters are passed in a tuple to the `execute` function

```
sql = " ".join([
    "select ClosingDate, CurrencySymbol, ExchangeRate",
    "from Rates",
    "where CurrencySymbol = ?",
])
rates = con.execute(sql, (symbol,))
```

Python: Parameterized Query

Using pyodbc

Insert Data

- To insert data into a table, an insert query is used
- To insert data, a parameterized query is used

```
sql = " ".join([
    "insert into rates (ClosingDate, CurrencySymbol, ExchangeRate)",
    "values (?, ?, ?)",
])

con.execute(sql, ('2021-01-07', 'EUR', 0.9))

rates = con.execute("select * from rates")
```

Python: Insert Data with a Parameterized Query

Using pyodbc

Execute Many

- It's possible to run a query multiple times with a different set of parameter values with the `executemany` function

```
lots_of_rates = [
    ('2021-01-07', 'EUR', 0.9),
    ('2021-01-08', 'EUR', 0.8),
    ('2021-01-09', 'EUR', 0.7),
    ('2021-01-10', 'EUR', 0.9),
    ('2021-01-11', 'EUR', 0.85),
]

sql = " ".join([
    "insert into rates (ClosingDate, CurrencySymbol, ExchangeRate)",
    "values (?, ?, ?)",
])

with con.cursor() as cur:
    cur.executemany(sql, lots_of_rates)
```

Python: Execute Query Many Times for a Collection of Data