Study Guide

Intermediate Python

Created By: Alec Mather-Shapiro, Teaching Assistant

Module 1: Introduction

Lesson 1.1: Introduction

Skills Learned From This Lesson: Outline, Requirements, Objectives

- Joe has done everything and then some!
- Ubuntu 64 bit VM
    - Vim, pip, terminator
- Exercise: Labs and assignments
- Assessments to check our knowledge
- This is an intermediate course! There are prerequisites!
    - Functions, data types, script creation and use
- Objectives: Input/Output task, Understand and use classes, and create our own Python modules

Module 2: I/O

Lesson 2.1: Using Inputs Effectively Part 1

Skills Learned From This Lesson: Input, Delimiter, .split()

- Objectives
    - Learn to use and interpret from the use
    - Use command-line arguments
- Tokenization - uses discrete pieces of data
- Create a new project called interpreter.py
- Don’t forget the shebang line! (#!/usr/bin/Python3)
- Leave a comment explaining the purpose of your project
- Only going to interpreter string and number tokens
- Delimiter is |

- Create main and then verify that it can accept data
  - def main()
    - user_data = input(‘Insert Delimited Data: ‘)
    - print(user_data)
    - Return
  - main()
- It works!
- We need a way to break the strings apart on the delimiter
- .split() is a handy Python helper function for splitting data
- Add the delimiter
    - split_data = user_data.split(sep=’|’)
    - print(split_data)

Interpreter.py so far:

```
#!/usr/bin/Python3

#interpreter.py is a script which takes input of arbitrary length, separated by a delimiter and
#identifies the number and type of tokens

#delimiter: |

def main():
        user_data = input(‘Insert Delimited Data: ‘)
        split_data = user_data.split(sep=’|’)
        print(split_data)
        return

main()
```

Lesson 2.2: Using Inputs Effectively Part 2

Skills Learned From This Lesson: isnumeric(), Loops, .format()

- .isnumeric() is another handy Python helper functions
  - Returns true is all characters are numbers
- Add lists to count number of string and numeric tokens

Interpreter.py so far:

```
#!/usr/bin/Python3

#interpreter.py is a script which takes input of arbitrary length, separated by a delimiter and
#identifies the number and type of tokens

#delimiter: |

def main():
        num_token = []
        str_tokens = []
        user_data = input('Insert Delimited Data: ')
        split_data = user_data.split(sep='|')
        for i in split_tokens:
                If i.isnumeric():
                        num_tokens.append(i)
                else:
                        str_tokens.append(i)
        print('String tokens: {}\nNumeric Tokens: {}'.format(len(str_tokens),len(num_tokens)))

        return

main()
```

Lesson 2.3: Using Inputs Effectively Part 3

Skills Learned From This Lesson: .isnumeric(), .strip(), Complete

- .isnumeric() doesn't know what to do with white space
- .strip() will remove all the white space and allow the program to verify a number
- help(str.strip) - returns the string with the leading and trailing white spaces removed

Completed Interpreter.py:

```
#!/usr/bin/Python3

#interpreter.py is a script which takes input of arbitrary length, separated by a delimiter and
#identifies the number and type of tokens

#delimiter: |

def main():
        num_token = []
        str_tokens = []
        user_data = input('Insert Delimited Data: ')
        split_data = user_data.split(sep='|')
        for i in split_tokens:
                if i.strip().isnumeric():
                        num_tokens.append(i)
                else:
                        str_tokens.append(i)
        print('String tokens: {}\nNumeric Tokens: {}'.format(len(str_tokens),len(num_tokens)))

        return

main()
```

Lesson 2.4: Command Line Arguments

Skills Learned From This Lesson: Import Module, sys.arv, Arguments

- Create a new project called cmd_line.py
- Goal is to take input from the command line and print the arguments one at a time
- We need to import a module for this program to gain access to all of that modules functions
    - More on modules in the Module Module
- Need to import the sys module (import sys)
- len(sys.argv) will show us the number of arguments that were passed to sys.argv
- 1 prints with no arguments because the name of the executable counts as an argument
    - *GOTCHA* in Python is an 'off by 1' error

Completed cmd_line.py

```
#!/usr/bin/Python3
Import sys

def main():
        #Objective: Print cmd line arguments
        #1 at a time
        #print(len(sys.argv))
        for i in sys.argv:
                print(i)
        return

main()
```

<u>Lesson 2.5</u>: File I/O - Opening, Writing, and Closing
Skills Learned From This Lesson: Modes, .write(), .read(), .seek()

- Objectives:
  - Create files in Python
  - Read from files in Python
  - Write to files in Python
  - Use non-text files in Python
- File name needs to be a string or Python will think it is a variable
- f = open('*filename.file*', *'mode'*)
  - We are using the file test.dat
  - Modes:
    - *w* will overwrite a file
      - *w+* - Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file
    - *a* will append a file
      - *a+* - Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file, irrespective of any intervening fseek(3) or similar.
    - *r* will read a file
      - *r+* - Open for reading and writing. The stream is positioned at the beginning of the file
- f.write('This is a test string')
  - .write() puts that string into the file at the current buffer location
- Add some lines to test.dat to check more functions
- A non-existent file will throw an error with r
  - Using a on a non-existent file will create the file
- .read() will return all the data that the function can store
- .readline() will read the file line by line
  - Because our cursor is at the end of the file, no data will be output to the screen yet
- .seek(0) indicates that we need to start at the beginning of the file

Lesson 2.6: File I/O - Opening, Writing, and Closing pt.2
Skills Learned From This Lesson: Read Binary, .read(), Modes

- Opening a file with w will blow away any existing file. There is a lab challenge on how to check for existing files before opening them
- We can call input in f.write to shortcut having to create another variable
- You can combine file functions with tokenization to mimic the functions of a database
- Using a jpg file
    - f = open('hero.jpg','rb')
        - rb is Read Binary. This indicates that this is not a text file
    - This will output all the hex bytes. Not easily readable but useful if you need to modify the bytes directly
- We can also open other types of files with rb
- Use f.read(#) to determine the number of bytes to read
- You can use Python to open binary data and search for strings

file_store.py

```
#!/usr/bin/Python3

#takes an arbitrary input and stores it in a file

def main():
        file_name = input('Filename: ')
        f.open(file_name,'w')
        f.write(input('Insert data to store: '))
        f.close()
        return

main()
```

Lesson 2.7: Interfacing with the OS

Skills Learned From This Lesson: Operating System, os.walk(), os.system()

- Objective:
  - Understand the OS Module
  - Examine files programmatically
  - Manipulate the system via Python
- import os - this module is by which the Python interpreter can directly interface with the OS
  - os.path is for finding a working with the actual paths
  - Lots of functions for interfacing with the OS
  - Read through the help manual for the module
- os.walk() - has various keyword arguments. Must have top argument in order to function and yields a 3 tuple
- A tuple is a data structure similar to a list.
  - tupl = (1,2,3)
  - Can be indexed
  - SImilar functions to lists
  - Immutable
- os.system executes commands in a sub shell
  - Pipes the commands back to the interpreter as you are working
  - Only executes for the total length of the command

Key Commands

```
for item_1, item_2, item_3 in os.walk(/home/user/Desktop):
        print('{}:{}:{}'.format(item_1,item_2,item_3))

os.system('pwd')
```

Module 3: Classes

Lesson 3.1: Implementing Classes Part 1

Skills Learned From This Lesson: Objectives, Classes,

- Objectives
    - Understand the purpose of Classes in Python
    - Learn about Methods and Attributes
    - Learn to construct a Class
- Classes are one of the core pieces of object oriented programming
- There are 3 kinds of data types: integers, strings and data structures
- All data in Python is a class
    - >>> type(str)
    - <class ‘type’>
- Classes can have information functions and other operations in their namespaces
- Classes have Attributes (variables) and Methods (behaviours)
- When invoking a class we get a usable object built by that class
- You can create multiple instances of a class
    - Modification to a class on modifies that specific instance of the class
- Methods work different than normal functions
    - All methods take at least 1 positional argument because there is a silent “secret” argument that is given inside of a class
    - self is the built in attribute. All internal functions need self as their first argument
    - This is automatically given to any class. You don’t need to call it
- We want to create the __init__ function in every class
    - __*xxxx*__ indicate that this is a private function. Not for external use
- Create arguments to add to the class
- Use self.*argument* to address arguments in a class
- Don’t forget the shebang line
- Remember to invoke your functions!
- Don’t forget about the gotcha with *self*

classes_demo.py

```
#!/usr/bin/Python3
class myCustomClassOne:
    name = ‘’
    age = 0
    def __init__(self,name,age):
        self.name = name
        self.age = age
        return
    def print_vals()
        print(‘Name: {}\nAge: {}’.format(self.name,self.age))

cInstance = myCustomClassOne(‘Leif’, 40):
cInstance.print_vals(self)
```

Lesson 3.2: Implementing Classes Part 2

Skills Learned From This Lesson: Classes, Inheritance, Methods, Attributes

- Attributes are variables that exist within a class
- Methods are functions that exist within a class
- Inheritance allows us to build classes based on an older class
- Adding a class name in parentheses () allows a new class to inherit all the Methods and Attributes of another class
- We want to call the Method (in this case init) using the Person.__init__(self, name, age)
    - Name and age are the relevant attributes that we want for this class
- print_vals doesn't know to address the new variable
    - We can create a new print vals
    - Or we can create an extended print vals
- You can call the parent functions but that get very complex very quickly

classes_demo.py

```
#!/usr/bin/Python3
class Person:
        name = ‘’
        age = 0
        def __init__(self,name,age):
                self.name = name
                self.age = age
                return
        def print_vals()
                print(‘Name: {}\nAge: {}’.format(self.name,self.age))

class Employee(Person):
        def __init__(self, name, age, department):
                Person.__init__(self, name, age)
                Self.department = department
        def extended_print_vals(self)
                print(‘Name: {}\nAge: {}’.format(self.name,self.age, self.department))


emp = Employee(‘Joe’, 25, ‘Research’)
emp.extended_print_vals()
```

Lesson 3.3: Iterators and Generators Part 1

Skills Learned From This Lesson: Objectives, Iterators, Equals Conditions

- Objectives:
    - Learn what Iterators and Generators are
    - Implement Iterators and Generators
- Create and initialize the class
- A class needs two methods to be an Iterator
    - __iter__ returns the self object. This is done for FOR loops
    - __next__ provides the return when the iterators is being implemented
- We need to add boundary conditions in order for an iterator to work
    - if i >= 25:
        - raise StopIteration
- == vs = is a big gotcha in Python
- Rolling your own iterator allows you to really optimize and change the way iterators normally behave

iter_demo.py

```
#!/usr/bin/Python3

class IterDemo:
    def __init__(self):
        self.alphabet = ‘abcdefghijklmnopqrstuvwxyz’
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.i >= 25:
            raise StopIteration
        letter = self.alphabet[self.i]
        self.i = self.i +1
        return letter

ID = IterDemo()
for i in ID:
    print(i)
```

Lesson 3.4: Iterators and Generators Part 2

Skills Learned From This Lesson: Generators, Iterators, Yield

- Generators are iterators
- Generators use functions instead of classes
- Generators are a special type of Python function that can provide its information or return value without losing its place in execution
- We can better understand generators by using our own range function
- The main distinguisher for generators in Python is function that uses the ‘yield’ keyword instead of the ‘return’ keyword
- The generator should return all the numbers as they are served up, one at a time
- Should work the same as ‘for i in range(10)

generator_demo.py

```
#!/usr/bin/Python3

def custom_range(last):
        i = 0
        while i < last:
                yield i
                i += 1

for i in custom_range(10):
        print(i)
```

Lesson 3.5: Exceptions

Skills Learned From This Lesson: Exceptions, Try, Except

- Objectives:
    - Learn what Exceptions are used to do in Python
    - Learn to use built-in Exceptions
    - Write Custom Exceptions
- Fundamentally and exception is an error message
- Easy to use in Python
- Traceback tells you what function you are in when the mistake happened
- Try - Except block
- Except can be blank or with a specific exception type
- You want to write a specific exception type so you don't silently kill another exception
- Errors should never be silent unless they are explicitly silent
- Building try/except statements into a program help prevent the entire program from terminating even if an error comes up
- You can do anything you could do with a normal class with an Exception Class
- When creating exceptions, you still have to handle them the same way as built-in ones

except_demo.py

```
#!/usr/bin/Python3

class CustomException(Exception):
	print('Exception found')

def main():
	for i in range(10):
		if i % 3 == 0:
			raise CustomException:
'''
	try:
		list_one = [1,2,3,4,5]
			for i in range(6):
			print(list_one[i])
	except IndexError:
		print('Index out of range')
	print('Got past our error')

'''
main()
```

Lesson 3.6: Summary and Review

Skills Learned From This Lesson: Objectives, Summary, Review

- Lesson 1:
    - Understand the purpose of classes
    - Learn about Methods and Attributes
    - Construct a Class
    - **Everything in Python is a class**
- Lesson 2:
    - Learn what Iterators and Generators are
    - Implement Iterators and Generators
- Lesson 3:
    - Understand the concept of Exceptions
    - Learn to use built-in Exceptions
    - Write custom Exceptions

Module 4: Modules

Lesson 4.1: Module Deep-Dive

Skills Learned From This Lesson: Objectives, Modules, Importing, Packages

- Objectives:
    - Define Module
    - Discuss the import process
    - Examine a Module
- Any Python file can be a module
- Contain: variables, functions, running code
- Modules are secretly classes
- Packages are a directory of Python files
- Importing modules
    - Search through the import path for the given module
        - Uses finders for built-in modules
    - Give the module a name in the local space
    - Importlib governs the import process
- Import modules by adding ‘import *module*’ to your programs
- Can use dir() and help() to better understand the module
- Use ‘from *module* import *functions*’ when you only want to import specific functions from a module.
    - This also allows your to simply call the function with the *module.*
- Can use ‘from *module* import *.’The risk is that if there are name collisions it can break your program

Lesson 4.2: Creating a Module

Skills Learned From This Lesson: Modules, Import, Import As

- Objective
  - Using what we learned, create a Python Module and Package
- Create a directory for your module
- You have to have a file __init__.py to create a Python package
- We use a special invocatin because this is a Python Module
  - if __name__ == '__main__':
    - main()
  - This examines the module to determine if the module has been imported
  - Modules that are directly invoked the name of the function is main
- Go to the parent directory to prove that the module works
  - import myMod.my_math as my_math
  - When you have a nested, long or complicated name, import the module as something easier to use

my_math.py

```
#!/usr/bin/python3

def myAdd(a,b):
        return a + b

def mySub(a,b):
        return a-b

def myDiv(a,b):
        return a/b

def myMul(a,b):
        return a * b

def main():
        print(myAdd(1,2))

if __name__ == "__main__":
        main()
```

Lesson 4.3: Course Summary and Review

Skills Learned From This Lesson: Summary

- Module 1: I/O
  - Lesson 1: User Input
  - Lesson 2: File I/O
  - Lesson 3: Interfacing with the OS
- Module 2: Classes
  - Lesson 1: Implementing Classes
  - Lesson 2: Iterators and Generators
  - Lesson 3: Exceptions
- Module 3: Module Module
  - Lesson 1: Module Deep-Dive
  - Lesson 2: Creating Modules
  - Lesson 3: Course Summary and Review