Module Seven

Topics

In this module we are looking at text and binary files and error handling functions. We also worked on the way to format Git Hub pages using markdown language.

Text Files

Once you close a program the results will cease to exist. In order to retain the results you need to save them to a file. The preferred way is to use a function in order to open a file, write data to the file and then close it. Using a function allows you to control the different steps of the processing. You can similarly read data from a file using the same steps.

In general there will be an extra break line at the end of text file that you will need to strip out.

Appending data will only work with a file that already exists. Other options for working with text data include readline() which will allow you to pick a particular row of data from a file. If you close the file after using readline and then perform the operation again you will get the same result. If you leave the file open then readline will act like a cursor and go to the next row.You have to repeat the readline function each time you want to advance a row, so it is commonly used with a while loop. In Python you can declare variables within a while loop and still call them outside of that function, but it is a best practice to declare variables before starting a processing statement.

Another function is called readlines() (as opposed to readline()). This function will create a list that will pull all the rows of text available. You can also call these functions using a for loop instead of a while loop. A for loop construction will automatically close the file after processing.

To combine functions together, such as reading and writing, you can use the with command.

Writing data will overwrite the contents of a file, while append will add data at the end of the existing content.

Binary data and Pickling

Import allows you to access other files that were previously created. Binary format allows you to both obscure the data and to make it smaller so it may be easier to use. Pickling is called by importing first, as the commands to pickle data are not inherent in Python. By pickling we take data and transform the format to binary and create it as a reference object, so that it will persist in the file and format we create. Pickling uses either the dump or read commands, dump to put data into binary format and read to transform it back into a human readable format.

Here is the lab for pickling:

```
# ------------------------------------------------ #
# Title: Lab7-1
```

```
# Description: A simple example of storing data in a binary
file
# ChangeLog: (Who, When, What)
# <YourName>,<1.1.2030>,Created Script
# ------------------------------------------------- #
import pickle  # This imports code from another code file!

# Data --------------------------------------------- #
strFileName = 'AppData.dat'
lstCustomer = []

# Processing --------------------------------------- #
def save_data_to_file(file_name, list_of_data):
    objFile = open(file_name, "ab")
    pickle.dump(list_of_data, objFile)
    objFile.close()
def read_data_from_file(file_name):
    objFile = open(file_name, "rb")
    list_of_data = pickle.load(objFile)  # load() only
loads one row of data.
    objFile.close()
    return list_of_data
# Presentation ------------------------------------- #
```

```
# TODO: Get ID and NAME From user, then store it in a list object
stridvalue = str(input("Enter an ID:"))
strnamevalue = str(input("Enter a Name: "))
lstCustomer= [stridvalue,strnamevalue]
# TODO: store the list object into a binary file
save_data_to_file(strFileName,lstCustomer)
# TODO: Read the data from the file into a new list object and display the contents
print(read_data_from_file(strFileName))
```

The data put into the file using the pickle.load function keeps it the same as it originally was created, regardless of the input data gathered from the input block.

I found this video interesting as it used a string variation for the load and dump functions, and it also created a class.

```
#from https://www.youtube.com/watch?v=XzkhtWYYojg

import pickle
#pickle.dump() #this works with a file type object
#pickle.dumps() #this works with a string type object
#pickle.load() #this works with a file type object
#pickle.loads() #this works with a string type object

class example_class: #create a class with a variety of data types
```

```
    a_number=25
    a_string="gotcha"
    a_list=[1,2,3]
    a_dictionary={"first":42, "second": "Sam I am","third":
[3,4,5]}
    a_tuple=(88,44)

my_object=example_class()
my_pickled_object=pickle.dumps(my_object)
print(f"This is my pickled object:\n{my_pickled_object}\n")
#shows the object in bianary format
#now we test to see if we can change the pickled object by
redefining one of the members of the class
my_object.a_dictionary=None
#unpickle and show the item we tried to change
my_unpickled_object=pickle.loads(my_pickled_object)
print(
    f"a_dictionary of unpickled object:
\n{my_unpickled_object.a_dictionary}\n")
```

I also found the example in the book interesting as it used a shelve function to act like a dictionary for the pickled object:

```
# Pickle It
# Demonstrates pickling and shelving data
```

```
# from Python Programming for the Absolute Beginner by
Michael Dawson, p.200

import pickle, shelve


print("Pickling lists.")

variety = ["sweet", "hot", "dill"]

shape = ["whole", "spear", "chip"]

brand = ["Claussen", "Heinz", "Vlassic"]

f = open("pickles1.dat", "wb") #wb=write bianary

pickle.dump(variety, f) #put list  variety into file f

pickle.dump(shape, f)

pickle.dump(brand, f)

f.close()


print("\nUnpickling lists.")

f = open("pickles1.dat", "rb") #rb= read binary

variety = pickle.load(f)

shape = pickle.load(f)

brand = pickle.load(f)

print(variety)

print(shape)

print(brand)

f.close()
```

```
print("\nShelving lists.")
s = shelve.open("pickles2.dat") #shelve creates a
dictionary like object but is more memory efficient
s["variety"] = ["sweet", "hot", "dill"] #create a key and
assigns values to it, here variety is the key
s["shape"] = ["whole", "spear", "chip"]
s["brand"] = ["Claussen", "Heinz", "Vlassic"]
s.sync()    # make sure data is written

print("\nRetrieving lists from a shelved file:")
print("brand -", s["brand"])
print("shape -", s["shape"])
print("variety -", s["variety"])
s.close()

input("\n\nPress the enter key to exit.")
```

I also found this site useful: https://www.geeksforgeeks.org/understanding-python-pickling-example/

Try Except

Using the try except commands allows you to put in tests as a statement is running to determine if expected results are being found. You can also create your own error statements to assist with understanding those that are automatically created by Python. Here is the lab where we are testing the division by zero rule.

```
# Using try
try:
```

```
    quotient = 5/0
    print(quotient)
except:
    print("There was an error! <<< Custom Message!\n")
#
# Now, without try
quotient = 5/0
print('never gets to this line')
```

We can also use the except block like a function by calling in the variable you are capturing. You will need to put in the class that the variable is, and then you can use commands document and string. Document and string are called with double underscores around them like this:

```
# ------------------------------------------------ #
# Title: Listing 12
# Description: A try-catch with better error messages
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# Anya Pryor, 11/29/21, Added comments
# ------------------------------------------------ #

try:
    quotient = 5/0
    print(quotient)
```

```
except Exception as e:
    print("There was an error! << Custom Message")

    print("Built-In Pythons error info: ")
    print(e)
    print(type(e)) #the class of the exception
    print(e.__doc__) #references the documentation on the
exception
    print(e.__str__()) #references the subcategory in the
documentation
```

There are subcategories of error messages within the class of exceptions, such as file not found. If you chain together exception blocks the order in which you put them will determine your results. For this reason, the most general exception should be at the end as a final error check. You can use the generic catch in order to see the class of the error using **type** and then you can create a new except block to catch that class of exception.

The documentation in Python has a list of base classes, that is further divided into exception classes, which are in a hierarchy of parent and class exceptions. Here is a link to general info on the exceptions: https://docs.python.org/3/library/exceptions.html#bltin-exceptions.

You can customize the error messages that are returned by simply using the exception command, it doesn't need to be within the try except block. Your custom messages can be embedded within a class you create, and they can use the string command as well.

Python also allows for conditional logic to be embedded in the try except blocks, allowing you to refine the type of testing you are doing beyond the order of exceptions that you test for. Using the else-if syntax will allow for this, as in listing 15:

```
# ------------------------------------------------ #
# Title: Listing 15
# Description: A try-catch with manually raised errors
#              using custom error classes
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------ #

class CustomError(Exception):
    """  Some custom error info in the DocString  """
    def __str__(self):
        return 'Some custom error message'

class FileNotTXTError(Exception):
    """  File extension must end with txt to indicate it is
a text file  """
    def __str__(self):
        return 'File extension not txt'

try:
    new_file_name = input("Enter the name of the file you
want to make: ")
```

```
    if new_file_name.isnumeric():
        raise Exception('Do not use numbers for the file\'s
name')
    elif new_file_name.endswith('txt') == False:
        raise FileNotTXTError()
    else:
        raise CustomError()

except FileNotTXTError as e:
    print("There was a non-specific error!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
except Exception as e:
    print("There was a non-specific error!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
```

The code above is similar to the multiple error handling in Dawson (p.206):

```
# Handle It
# Demonstrates handling exceptions
# From Python for Absolute Beginners by Michael Dawson
p.206
```

```
# try/except
try:
    num = float(input("Enter a number: "))
except:
    print("Something went wrong!")

# specifying exception type
try:
    num = float(input("\nEnter a number: "))
except ValueError:
    print("That was not a number!")

# handle multiple exception types
print()
for value in (None, "Hi!"):
    try:
        print("Attempting to convert", value, "-->", end="
")
        print(float(value))
    except (TypeError, ValueError):
        print("Something went wrong!")

print()
for value in (None, "Hi!"):
```

```
    try:
        print("Attempting to convert", value, "-->", end="
")
        print(float(value))
    except TypeError:
        print("I can only convert a string or a number!")
    except ValueError:
        print("I can only convert a string of digits!")
```

```
# get an exception's argument
try:
    num = float(input("\nEnter a number: "))
except ValueError as e:
    print("That was not a number! Or as Python would
say...")
    print(e)
```

```
# try/except/else
try:
    num = float(input("\nEnter a number: "))
except ValueError:
    print("That was not a number!")
else:
    print("You entered the number", num)
```

```
input("\n\nPress the enter key to exit.")
```

The code above shows us the different types of error messages you can create from the same input.

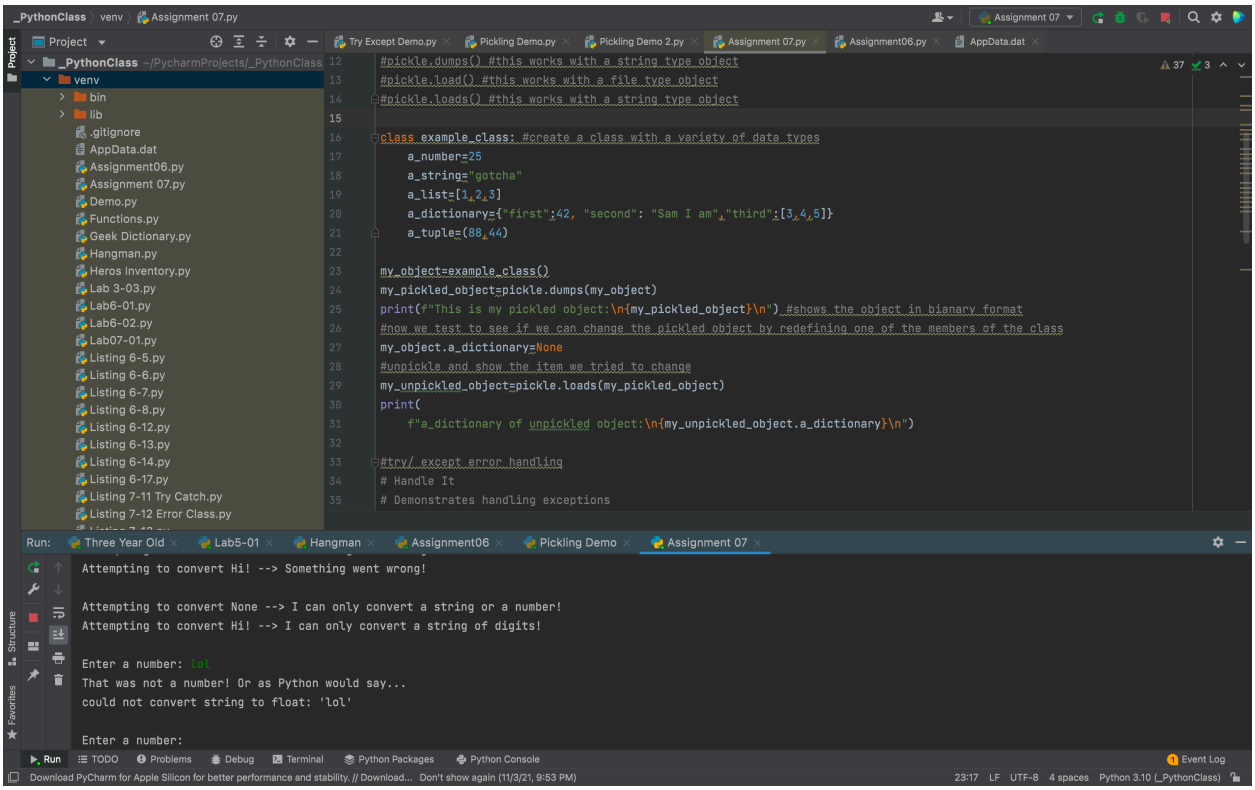
Git Hub Web Pages

To format information on a git hub page you can use the commands from Markdown language. The first mistake I made was to not use md as the file type (I thought there was an underscore there), but once I figured that out then the markdown commands worked. Similar to the preset formatting of headings and body text you can use # for a heading, ## for a subsection. If you also put in Python code you will get some strange looking pages! So it's best to put three back ticks (under the Tilda symbol) before and after the code block. Here is my page:https://github.com/anyapryor/ITFnd100-Mod07/blob/main/docs/index.md

---

Assignment

Here is a screen shot of the program in PyCharm:



Summary

In this module we used pickling to create a static object that is stored in binary format. To read the object we needed to load the object again, and tested that the object would persist. We also started to learn about classes with the exception class, which references a hierarchy of information. We can manipulate this hierarchy by using else if constructs within the try except construction. Error handling can be customized, but you need to be mindful of what type of error you are encountering and what message would be most useful. We also worked on presenting our findings in a more readable form on GitHub.