Deborah Yenubari

02/25/2021

Fundamentals of Programming (Python)

Assignment 6

[DY202/IntroToProg-Python-Mod6 (github.com)](https://github.com/DY202/IntroToProg-Python-Mod6)

<https://dy202.github.io/Mod06/>

# Class and Functions

Function

A function groups together a set of statements so they can be run more than once, as when you plan on using a block of code multiple times. They can also let us specify parameters that can serve as inputs to the functions.

Functions will be one of most basic levels of reusing code in Python. You must define a function before you can use code to call the function. Calling the function executes the statements in the function.

A function

-begins with **def**

-then a space

-followed by the name of the function. Names must be relevant, as something\_len()for a length() function.

-Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them.

-after this a colon.

- here is the important step, you must indent to begin the code inside your function correctly as Python makes use of whitespace to organize code.

-Next is the docstring, where you write a basic description of the function. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

-after this you begin writing the code you wish to execute.

**def name\_of\_function(arg1,arg2):**

**'''**

***This is where the function's Document String (docstring) goes.***

**'''**

**# Do stuff here**

**# Return desired result**

PyCharm can display tool tips to show a developer's notes(docstring) (use ctrl + q to activate this option in PyCharm).

**def** **AddValues(value1=0.0, value2=0.0):**

*"""This function adds two values*

*:param value1: (float)the first number to add*

*:param value2: (float) the second number to add*

*:return: (float) sum of two numbers*

*"""*

**return value1 + value2**

**print(AddValues(5,10))** *# Use ctrl+ q in PyCharm to see docstrings*

Example of a function:

*# define the function*

**def ProcessSomething():**

**print("I'm")** *# first statement*

**print("processing data")** *# second statement*

*# Call the function*

**ProcessSomething()**

*# output*

**I’m**

**processing data**

Accepting parameters and arguments

Parameters in a function allow you to pass values into the function for processing.

*Values passed into parameters are called “arguments,”* Arguments are inputs for the function. You'll be able to use these inputs in your function and reference them.

*# define the function*

**def Process\_Something(parmMessage):**

**print(“The parameter was: “ + parmMessage)**

*# Call the function*

**Process Something(“argABC)**

There is no limit on how many parameters you can include.

For example, you can pass two arguments to a function that calculates a sum.

*# Define the function*

**def AddValues(value1,value2):**

**fltAnswer = value1 + value2**

**print("The Sum of the values is: " + str(fltAnswer))**

*# Call the function*

**AddValues(10, 5)**

It is standard to have parameter names without a prefix, but with the "snakecasing" or underscore between words.

Arguments are used to make a function perform different actions or return different results.

Using Variables as Arguments

Using argument variables is useful to access these values multiple times in a script.

*# --data code --#*

**fltV1= None # first argument**

**fltV2= None # second argument**

*# --processing code --#*

**def AddValues(value1, value2):**

**fltAnswer = value1 + value2**

**print(fltAnswer);**

*# --presentation (I/0) code --#*

**fltV1 = float(input("Enter value 1: "))**

**fltV2 = float(input("Enter value 2: "))**

**print("The Sum of %.2fand %.2f" % (fltV1, fltV2))**

**print("is: ", end='')**

**AddValues(fltV1, fltV2)**

Return Values

The *return* keyword allows you to actually save the result of the output of a function as a variable. The *print()* function simply displays the output to you, but doesn't save it for future use.

Functions can return one or more values. You capture returning values of a function in variables.

**def MyFunction():**

**return 'data'**

**v1= MyFunction()**

"Return values" make a function act as an expression. Evaluating a function as an expression means that you use the results of a function immediately without placing the result in a variable.

print(MyFunction())

Capturing the results in a variable allows you to use the variable of results multiple times without having to call the function again, *but using it as an expression does not*

Return values can be a single item of data or multiple items.

If you return multiple values, you need to bundle them into a collection and return that collection. In Python, simplify this process using the tuple packing and unpacking feature.

#--processing code--#

**def AddValues(value1, value2):**

**fltAnswer = value1 + value2**

**return value1, value2, fltAnswer** # pack tuple

# --presentation (I/0) code --#

**fltV1 = float(input("Enter value 1: "))**

**fltV2 = float(input("Enter value 2: "))**

**fltR1, fltR2, fltR3= AddValues(fltV1, fltV2)** # unpack tuple

**print("The Sum of %.2f and %.2f is %.2f" % (fltR1, fltR2, fltR3))**

Return values can also be retuned as a List object instead of a tuple

*# --data code --#*

**fltV1 = None** # *first argument*

**fltV2 = None** # *second argument*

**lstResults = None** *# list of results for processing*

*# --processing code --#*

**def AddValues(value1, value2):**

**fltAnswer = value1 + value2**

**return [value1, value2, fltAnswer]** *# createalist*

*# --presentation (I/0) code --#*

**fltV1 = float(input("Enter value 1: "))**

**fltV2 = float(input("Enter value 2: "))**

**lstResults= AddValues(fltV1, fltV2)** *# capture the list*

**print("The Sum of %.2f and %.2f is %.2f" %(lstResults[0], lstResults[1], lstResults[2]))**

Global vs. Local Variables

Variables in a script may be local or global.

Variables declared in a function are considered local to the containing function and cannot be accessed outside of that function.

Variables declared in a "body" of the script are considered global to the containing script and can be used anywhere in the script.

Any code inside of the same function can "see" the local variable because that variable would be "inside of its scope!".

def MyFunction():

v1= 15 *# localprint*

(v1*) # This works!*

Any code outside of the function cannot "see" the local variable because that variable would be "outside of its scope!"

def MyFunction():

v1= 15 *# localprint*

(v1) *# This works!*

MyFunction()

print(v1) *# This causes a "NameError: name 'v1' is not defined"*

*# --data code --#*

*# Note: Variables declared in the body of the script are "Global"*

**v1 = 10** *# first argument*

**v2 = 5** *# second argument*

**gAnswer = None** *# result of processing*

*# --processing code --#*

**def AddValues(value1, value2):**

**global gAnswer** *# This refers to the "global" variable*

**gAnswer = value1 + value2**

**answer = value1 + value2** # This is a "local" variable!

**return answer**

*# --presentation (I/0)code --#*

**AddValues(v1, v2)**

**print('Global = ', gAnswer)**

**print('Local = ', AddValues(v1, v2))**

Classes

Classes are a way of grouping functions, variables, and constants.

By convention we give classes a name that starts with a capital letter, and every word begins with a capital- camelcasing.

*# --processing code --#*

class MathProcessor():

"""functions for processing simple math"""

@staticmethod

def AddValues(value1=0.0, value2=0.0):

"""This function adds two values

:paramvalue1: (float) the first number to add

:paramvalue2: (float) the second number to add

:return: (float) sum of two numbers

"""

return float(value1 +value2)

@staticmethod

def SubtractValues(value1=0, value2=0):

"""This function subtracts two values

:paramvalue1: (float) the first number to subtract

:paramvalue2: (float) the second number to subtract

:return: (float) sum of two numbers

"""

return float(value1 -value2)

*# --presentation (I/0) code*

print(MathProcessor.AddValues(5,10))

print(MathProcessor.SubtractValues(5,10))

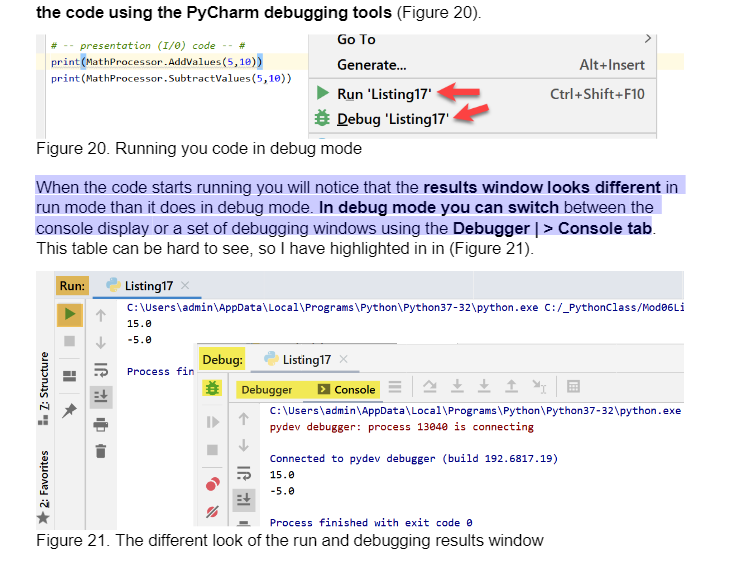
Separation of Concerns

Separating code into functions and classes make your code easier to read and to debug since smaller sets of statement are easier to test.

Functions with return values make code easier to divide into three layers of concern; data, processing, and presentation.

Debugging

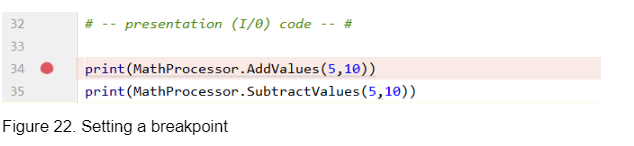
PyCharm’s debugging tool can make finding and fixing bugs much faster.

When the code starts running you will notice that the results window looks different in ‘run’ mode than it does in debug mode. In ‘debug’ mode you can switch between the console display or a set of debugging windows using the Debugger| >Console tab

Breakpoints

To use the set of Debugging windows you need to set a "breakpoint" in your code. A breakpoint tells the debugger when to pause running the code so that you can exam the information in the debugging windows.

To set a breakpoint, locate a line of code where you would like the debugger to pause and left-click on the margin left of your code. Doing so will make a red dot appear, indicating the breakpoint has been added (Figure 22).



The next time you run your code in debug mode, the code pauses and lets you use the controls in debugging windows.

Important controls are noted in Figure 23:

1. The tab that lets you navigate between the debugging and console windows

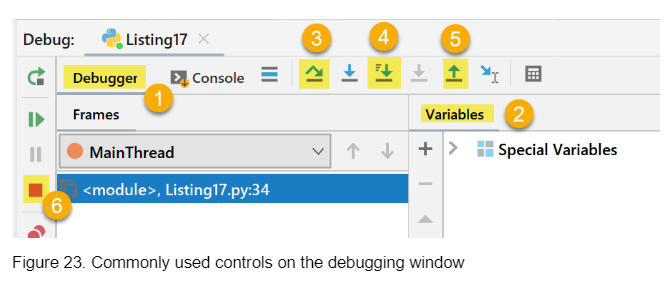
2. The "Variable" window that shows which variables are being used and their values

3. The "Step Over" button that allows you to skip over seeing the code in a function when it is called.

4. The "Step Into My Code" button that allows you to step into the next line of your code, without showing you lines that the Python runtime user to run you code (Which can be confusing!)

5. The "Step Out" button that allows you to stop showing the code in a called function and return to the line of code that called it.

6. The "Stop" button that allows you to stop debugging and running the script.

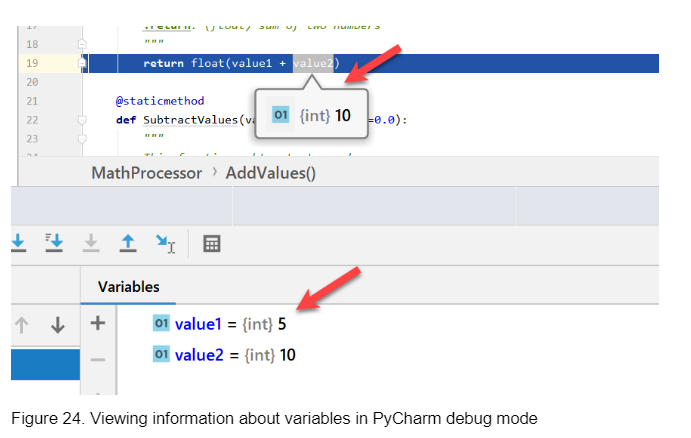


Walking Through Code

The control you use the most is the "Step Into My Code" button.

Using it, allows you to "walk" through each line of code that performs an actions. Code that does not perform actions, such as comments are skipped.

When a line of code is reached that contains variables, you can either hover over that variable to see what is current held in memory or look at the "Variables" window to see all the variable currently being used. Note that both the value and the data type are shown in PyCharm.



GitHub

It is good to make backups of code files and to make them available for others to access.

This has been accomplished using a network share on an organization’s server. While this is still a common practice, more and more organizations have embraced storing this code on the Internet via source control software.

One of the most popular source control software is Git, which can store your files on the Internet at GitHub.com.

“…storing the modifications in a central repository. This allows developers to easily collaborate, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute…”

GitHub Pages are public webpages hosted and easily published through GitHub.

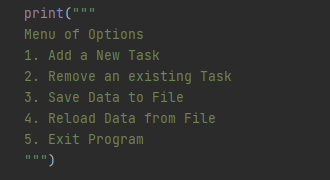
You can use GitHub Pages to host a website about yourself, your organization, or your project directly from a GitHub repository.

Most of the information discussed above is used in Assignment 06.

**Assignment Question 06**

"Assigment06\_Starter.py". Currently the code loads data from a file into a Python List of Dictionary objects. However, the code only uses a few functions, and your job is to add more functions to organize the code.

The Script must help execute a Menu of Options:



To create code that will perform the above operations for each choice of 1,2,3,4,5.

Begin by separating the code into three areas of:

# Data #

# Processing #

# Presentation #

**Declaring # Data #**

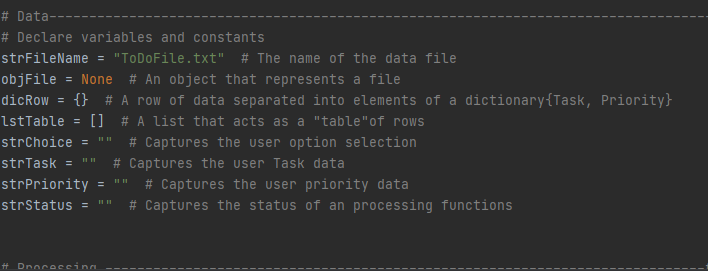


Fig 1: Declaring the variables and constants in the code

strFileName = name of the text file from which we read or write data

objFile = an object that represents a file

dicRow = a row with “Task “and “Priority” as keys

lstTable =[]

strChoice, strTask, strPriority are the input arguments for the functions used in the code. Parameters provided so the user can provide input/value ,and become arguments in the function.

**# Processing Data #**

Create a Class as they help group functions, variables, and constants. A class named ‘Processor’ to group four different functions for processing

- to read data from file

- add data to list

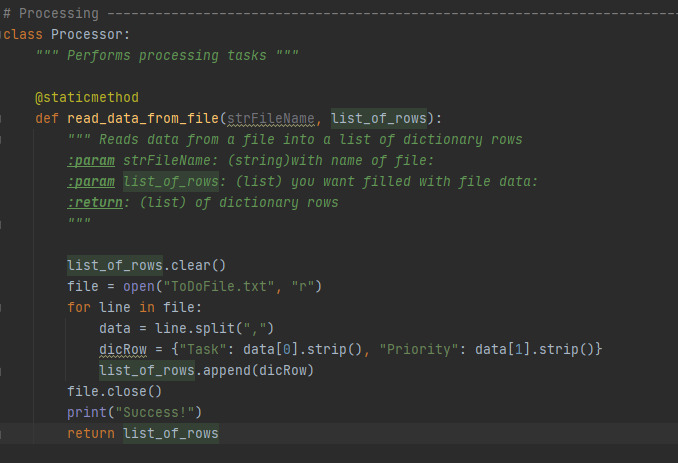
- remove data from list

- save data to file

These four processing functions will be used along with input/output functions from a different class IO, to run the code in Main Script. Will discuss each function below before we put together the Main Script at the end.

The first function under @staticmethod under class Processor,

**def read\_data\_from\_file** with two arguments, **strFileName**, and **list\_of\_rows**.



clear data output while previously run.

**list\_of\_rows.clear()**

then open a text file with the name “ToDoFile.txt” to read, “r”.

**file = open(“ToDoFile.txt”, “r”)**

Using a ‘for’ loop, for each line in text file, split data,

Then arrange data as in a dictionary Row with ‘Task’ as key, and data at index[0], as value and ‘Priority’ as key and data at index[1] as value.

Add/append all the rows as in a table format, list\_of\_rows

**for line in file:**

**data = line.split(“,”)**

**dicRow = {“Task”: data[0].strip(), “Priority”: data[1].strip()}**

**list\_of\_rows.append(dicRow)**

**file.close()** # *after retrieving data close file*

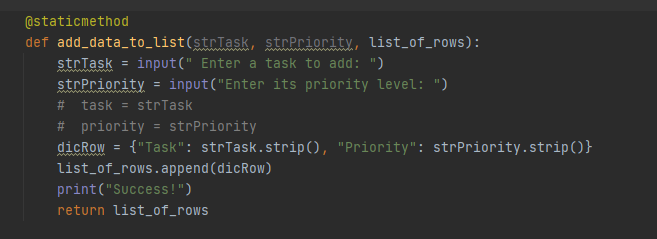
**print(“Success”)**

**return list\_of\_rows,”Success”** # *returns argument list\_of\_rows*

strip() returns a string where all the white space(tabs, spaces, and newlines) at the beginning and end is removed.

The second function under @staticmethod under class Processor,

**def add\_data\_to\_list(strTask, strPriority, list\_of\_rows):**



strTask, strPriority, list\_of\_rows as arguments with input values

**def add\_data\_to\_list(task,priority,list\_of\_rows)**

strTask = input(“Enter a task to add: “)

strPriority = input(“Enter its priority value: “)

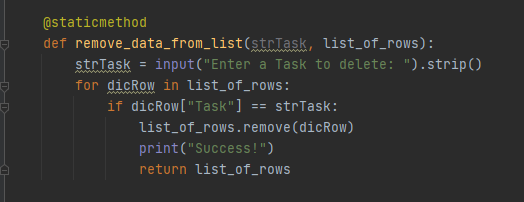
and form a dicRow as key: value inside table, list\_of\_rows

**dicRow = {“Task”: task.strip(), “Priority”: priority.strip()}**

**list\_of\_rows.append(dicRow)**  # update list\_of\_rows

**def remove\_data\_from\_list(task, list\_of\_rows):**

When the user add input to the parameter ’strTask’, data is removed from list of rows once. To remove multiple rows, need to delete each one.

 @staticmethod,

**def remove\_data\_from\_list(task, list\_of\_rows):**

Indent, then statement **strTask = input(“Enter a Task to delete: “).strip()**

If strTask is inside ‘for’ loop, you will have to enter ‘task to delete’ multiple times, until the ‘task to delete’ is found, if in the 7th row, then 7 times to catch it.

Hence outside ‘for’ loop.

‘for’ loop, **for dicRow in list\_of\_rows:**

**If dicRow[“Task”] == strTask:**

**list\_of\_rows.remove(dicRow)**

**print(“Success!”)**

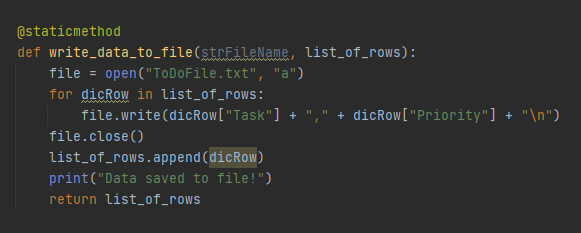
**return list\_of\_rows**

The last function under class Processor

@staticmethod

**def write\_data\_to\_file(strFileName, list\_of\_rows):**

All the functions help to execute the Menu of Options provided to the user, and this last function in processing helps execute choice 4 in the Menu.



Arguments used are strFileName and list\_of\_rows.

**strFileName = “ToDoFile.txt” in the append or ‘a’ form.**

Request text file “ToDoFile.txt” open,

‘for’ loop, scan data in each dicRow in list\_of \_rows table

Write it into text file,

concatenate data, value of key ’Task’ first, followed by a ‘,’ and add value of key ‘priority’ followed by a line.

Append the created row to the list\_of\_rows.

Close the file. Print, ’Success’ return list\_of\_rows

**file = open(“ToDoFile.txt”, “a”)**

**for dicRow in list\_of\_rows:**

**file.write(dicRow[“Task”] + “,” + dicRow[“Priority”] + “\n”)**

**file.close()**

**list\_of\_rows.append(dicRow)**

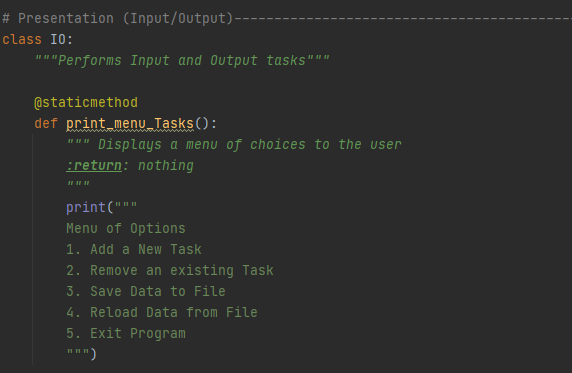
**print(“Data saved to file!”)**

**return list\_of\_rows**

With the processing functions defined, create input/output or presentation functions, created under a class IO or Input Output.

Functions under a new class IO will take input and return variable arguments to process or format data read or written into text file after processing.

**def print\_menu\_Tasks():**



The first function↑ under class IO, ( with a docstring “”” Performs Input and Output tasks”””) under @staticmethod, presents or displays a print statement with the Menu of Options to the user. It returns nothing.

**“””**Displays a menu of choices to the user : return: nothing

“””

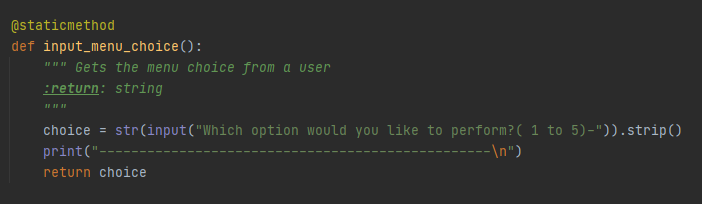
**print(“””**

**Menu of Options**

1. **Add a New Task**
2. **Remove an existing Task**
3. **Save Data to File**
4. **Reload Data from File**
5. **Exit Program**

**“””)**

**def input\_menu\_choice():**



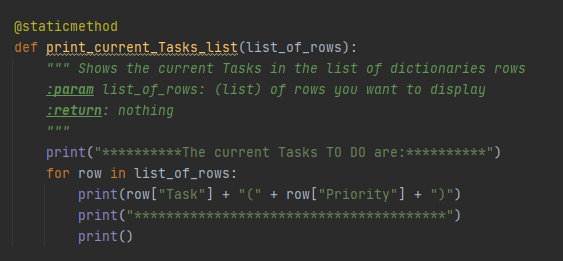
The second function↑ prints a statement that asks the user to enter a choice of operation or which statement is to be executed. Parameter returned as a str(input) variable argument.

“””Gets the menu choice from a user”””

**choice = str(input(“Which option would you like to perform?( 1 to 5) –“)).strip()**

return choice.

The variable argument ‘choice’ will be used for processing in the Main Script.

**def print\_current\_Tasks\_list(list\_of\_rows)**

The above function↑ formats data read from a file, obtained in function read\_data\_from\_file and presents as output to the user, with list\_of\_rows as parameter.

Print a statement on current tasks, then ‘for’ loop.

For each row in list\_of\_rows, concatenate a row with value of key ‘Task’ first, with value of key ‘ Priority’ next, ‘Priority’ enclosed within parentheses.

**print(“\*\*\*\*\*\*\*\*\*\*\*\*The current Tasks TO DO are: \*\*\*\*\*\*\*\*\*\*\*\*”)**

**for row in list\_of\_rows:**

**print(row[“Task”] + “(“ + row[“Priority”] + “)”)**

**print(“\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*”)**

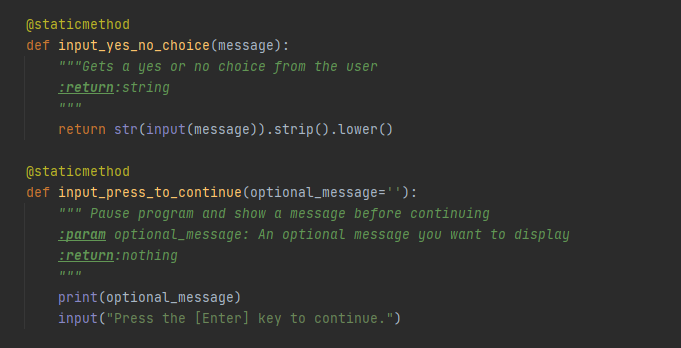
**print() # an extra line**

The next two functions display a message and an optional message requesting user input.

**def input\_yes\_no\_choice(message):**

message = “Save this Data to file (y/n)- ?” is stated in the Main Script.

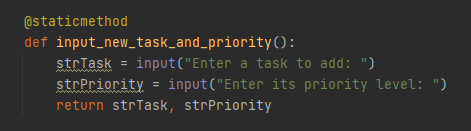
function returns a string variable argument

**def input\_press\_to\_continue(optional\_message =’’)**

Function prints optional message = “Are you sure you want to reload data from file(y/n)- ?” is stated in the Main Script.

asks for user input (“Press the [Enter] key to continue.”)

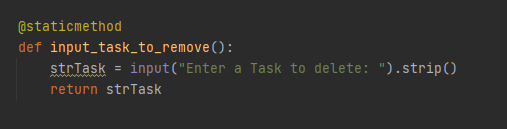
**def input\_new\_task\_and\_priority():**



Function returns two variable arguments used in processing add\_data\_to\_list.

strTask takes input value on the new task, and strPriority takes value on the new Task’s priority.

**def input\_task\_to\_remove():**

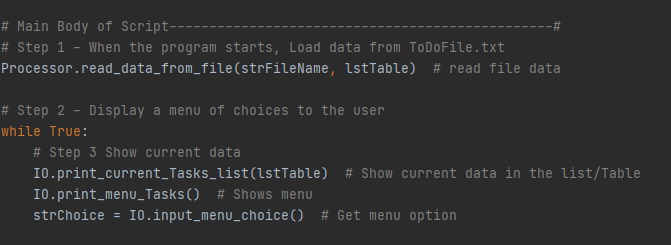


To delete a task, asks user for input value for key ‘Task’ to delete .

The returned value is used as an argument in processing remove\_data\_from\_file

NOTE: However as I could not use the returned values for processing in Processor function def add\_data\_to\_list and def remove\_data\_from\_list, I have not used these IO function in the main script.

Now that Processing and IO class functions have been defined, the Main Script may call for the different functions inside each class.

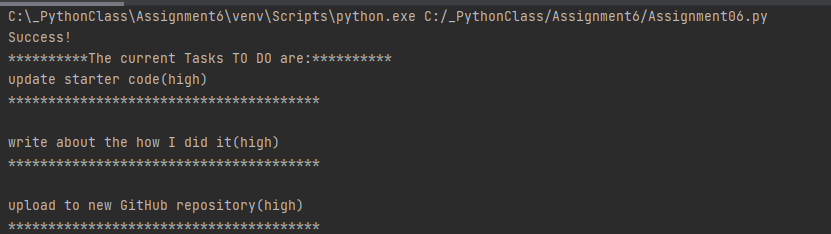


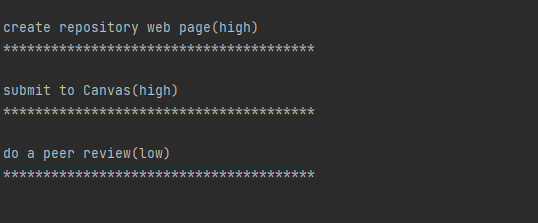
Step 1 is to read data from text file, ‘ToDoFile.txt’

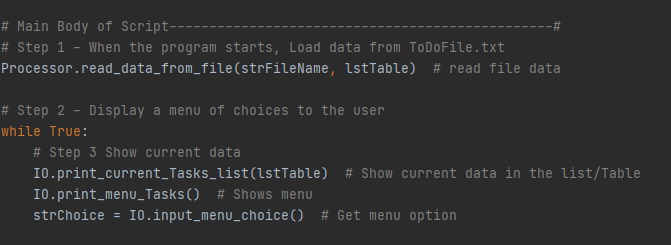
Under class Processor, call function read\_data\_from\_file with arguments strFileName, and lstTable

Function reads lines from file, splits each line and concatenates into rows as keys and values using the function IO.print\_current\_Tasks\_list(lstTable) in a ‘while’ loop

Output for Processor.read\_data\_from\_file(strFileName, lstTable)

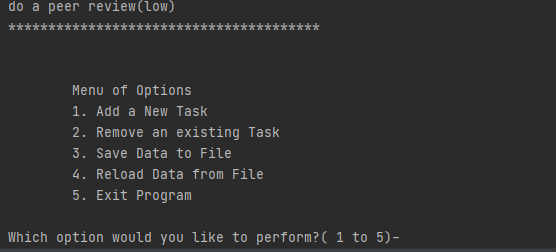






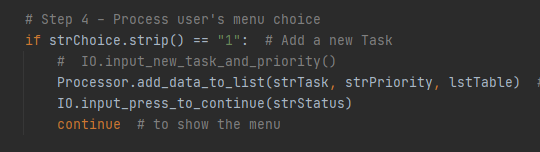
For Step 2 and Step 3, under class IO the function IO. print\_menu\_Tasks(), displays a menu of user options

Inside a ‘while’ loop, function IO. input\_menu\_choice() asks user input for the variable strChoice, value returned as variable argument

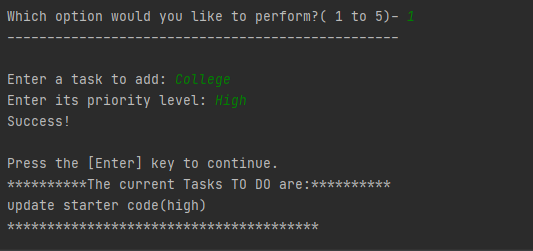


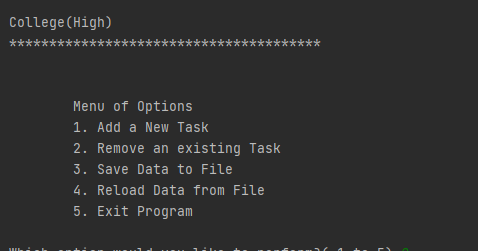
Step 4 begins executes the if…statement

**If strChoice.strip() == “1”**: # add data to list

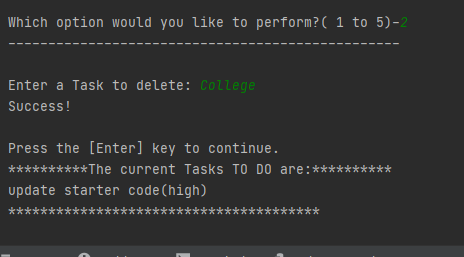


Class Processor.add\_data\_to\_list functions asks user input for strTask, strPriority and appends them to the current data table, list\_of\_rows.



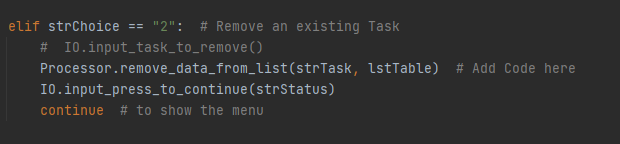


Inside a ‘while’ loop, function IO. input\_menu\_choice() as long as ‘True’ asks user input for the variable strChoice, value returned as variable argument

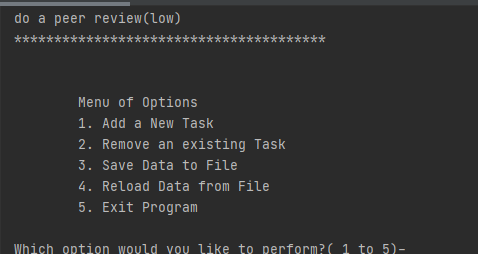


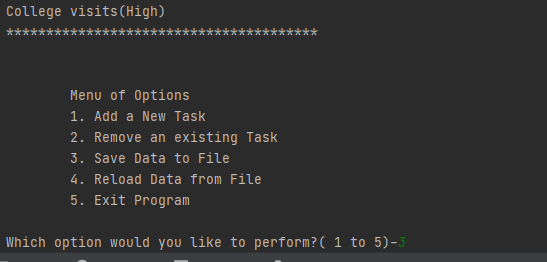
For Step 4, elif… clause

**elif strChoice == “2”:** # remove existing data



remove\_data\_from\_list(strTask, lstTable) under class Processor, asks for input value returned as argument and used. IO.input\_press\_to\_continue allows the user to press ‘enter’ and displays menu of options. In the output below, value entered under key’Task’ ==College was removed form lstTable



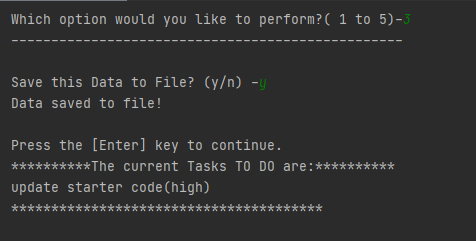


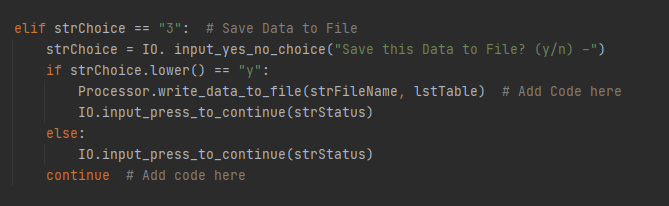
Step 4 in the Main Script

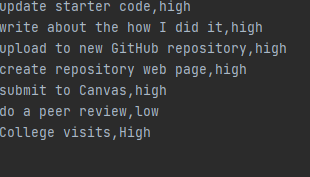
**elif strChoice == “3”:** # save data to file

strChoice = IO. Input\_yes\_no\_choice function gets user input “y” or “n”

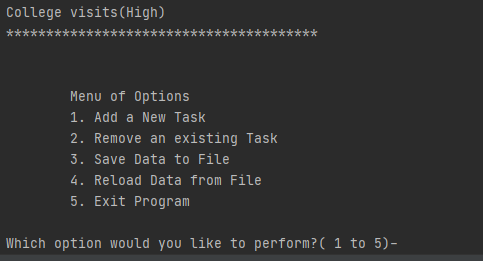
if “y”, function write\_data\_to\_file(strFileName,lstTable) is called and value

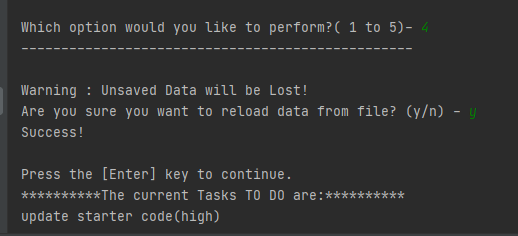
entered for ‘Task’:‘College visits” added in the output and saved to the text file,”ToDoFile.txt”, while displaying a menu of options on pressing enter.

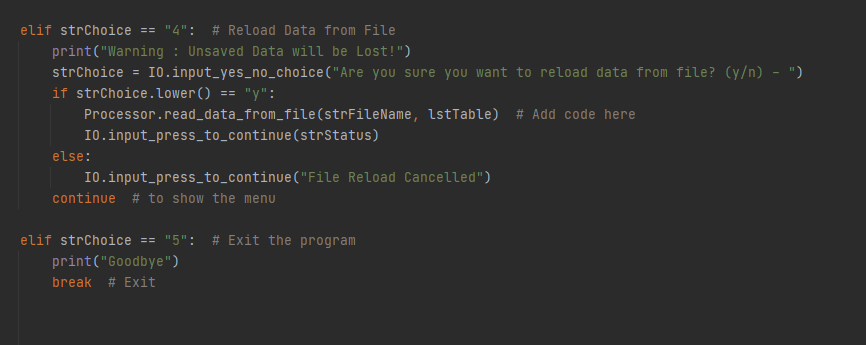




Data saved in “ToDoFile.txt”







Continuing with Step 4, else if **strChoice ==4** # reload data from file.

Print statement( **“warning:Unsaved Data will be lost!”)**

User input y/n

**strChoice = IO.input\_yes\_no\_choice(“Are you sure…….(y/n)-“)**

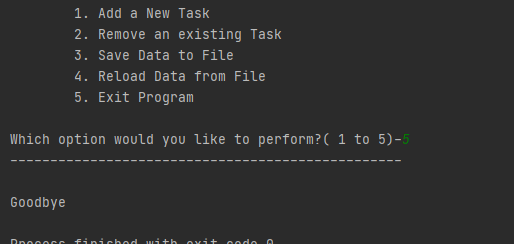
a nested if …statement

**if strChoice.lower() == “y”:** under class Processor, function read\_data\_from\_file(strFileName, lstTable) is called and IO. Input\_press\_to\_continue # press enter to continue # strStatus

**else: break** out of the nested ‘if’ statement and return to menu of options by pressing enter.

Continuing, **if elif strChoice == “5”:**

break # exit from the program.



Conclusion

The functions defined under class Processor and IO worked as desired to carry out the menu of options for the user, but was unable to have two IO functions def input\_new\_task\_and\_priority(): and def input\_task\_to\_remove perform as required. The values entered under the above mentioned IO functions would not serve as arguments in the processing functions.

Note to Mr Randall: I have not been able to make the script work on command prompt. My system has hung and shut down a few times this week, I would be grateful for help in Environment variables/ user variables in system variables, to add a new Path so Python works in the command prompt. Thank you so much!