**Module 6: Functions & Classes**

GitHub Repo: <https://github.com/einsteinboricua/IntroToProg-Python-Mod06>

# Introduction

The module presents students with functions and classes. Though students have already used functions (print() and input() being the most notable ones), they have not yet been through the process of creating their own. The different kinds of functions are explained (those that return a value vs those that don’t; those that take arguments vs those that don’t), and how to distinguish between using local variables and global variables (i.e. scope). Classes are also introduced as a way to group similar functions together. The week’s assignment will have students rework the assignment from the previous week by using classes and functions to carry out the logic.

# Course Content

The author is well versed with the contents for this week’s module as other high-level programming languages like Java and C# also make use of functions and classes. The author notes that, unlike those languages, Python allows arguments to be passed in any order (as long as they’re explicitly linked), or just one or none at all, with the function then assigning default values. To use the functions, coders must understand the differences in scope for the variables and be cautious to not edit global variables unless necessary. Python allows the protection by using the “global” keyword, whereas other languages may require “this” or even just allow the user to directly modify global variables within the function.

The author has also made use of function descriptions, particularly with Java, which allows coders to generate Javadoc, explaining what the functions do and which parameters they take, if any. It was encouraging to see that Python also makes use of the same functionality.

Finally, as part of the author’s daily work, debugging is known territory. When programming with Ada, debug statements printing the value of the variable are the best he can do, as Ada compilers require a separate debugger to work. However, when coding with Java or C/C#, Eclipse and Visual Studio have excellent debuggers. While the instructor continues to showcase PyCharm, the author decided to explore how VS Code works, especially since that’s the editor used at work and will be the editor of choice for the author’s subsequent courses after this one.

# Assignment

This week’s assignment had students use functions for the ToDo program that they’ve been using for the past two assignments. The instructor prepared a source file with some code filled in, but students were asked to fill in the code so that the functions worked as intended. The author will split this section into the two separate classes found within the source file.

## Class IO

The IO class within the source file is responsible for the prompts to the user and the display of the data. Students were tasked with filling out the code for the input\_new\_task\_and\_priority() and input\_task\_to\_remove() functions declared. The author made use of the print() and input() functions to achieve the tasks. As the input values are implicitly Strings, the author felt there was no need to cast the variables to String. Figure 1 shows the two functions filled out.

@staticmethod

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

""" Gets task and priority values to be added to the list

:return: (string, string) with task and priority

"""

strTask = input("What task do you wish to add? ").strip()

print()

strPriority = input("What is the task's priority? ").strip()

print()

return strTask, strPriority

@staticmethod

def input\_task\_to\_remove():

""" Gets the task name to be removed from the list

:return: (string) with task

"""

strTask = input("What task do you wish to remove? ").strip()

print()

return strTask

## Figure 1 I/O Functions filled out

## Class Processor

The Processor class is in charge of the main logic for the script. It is meant to do the same tasks as in the previous assignments, except students must fill out the declared functions so that the Processor object can do what it’s being asked to do. The instructor has three functions declared which student needed to complete.

The first function the author attempted was the add\_data\_to\_list() function, which takes in three arguments. Reusing code from the previous assignment, the author now pointed to the arguments being passed by the function to carry out the task of adding items to the list. When reviewing other assignments as part of the previous week’s assignment, he noticed a few had incorporated a while loop to allow the user to input as many tasks as desired before stopping altogether; in the interest to keep code simple enough, the author opted against this and instead will allow the user only one task at a time. He still made use of the Boolean flag to check if the task already existed; if it didn’t, then the task would be added; otherwise, the user would be warned that the task existed and would return them to the main menu. Figure 2 shows the code for this function.

@staticmethod

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

""" Adds data to a list of dictionary rows

:param task: (string) with name of task:

:param priority: (string) with name of priority:

:param list\_of\_rows: (list) you want filled with file data:

:return: (list) of dictionary rows

"""

row = {"Task": str(task).strip(), "Priority": str(priority).strip()}

#Boolean flag to check if task exists

exists = False

#Loop through all items on list

for hTier in list\_of\_rows:

if row["Task"].lower() in hTier["Task"].lower():

#Key (Task) exists; return to main menu

print("Task already exists; returning to main menu.")

print()

exists = True

break

#If the key does not exist, add row to the list

if not exists:

list\_of\_rows.append(row)

print("Added \""+task+"\" with priority "+priority)

print()

# TODO: Add Code Here!

return list\_of\_rows

## Figure 2 Task Adding Function

The next one the author attempted was the remove\_data\_from\_list() function, which took in two arguments. Once again, the author reused code from the previous assignment, adapted to the arguments being passed rather than global variables, to complete the function. In a similar manner as the add\_task function, the remove\_data function also used a Boolean flag to check if the task to be removed exists; if it does, then the task is removed; if not, then the user is warned and returned to the main menu. Figure 3 shows the function.

@staticmethod

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

""" Removes data from a list of dictionary rows

:param task: (string) with name of task:

:param list\_of\_rows: (list) you want filled with file data:

:return: (list) of dictionary rows

"""

#Boolean flag to check if task exists

exists = False

#Get the dictionary item

for row in list\_of\_rows:

#Check if the key (Task) exists and if it does, remove it

if task.lower() in row["Task"].lower():

exists = True

list\_of\_rows.remove(row)

print("Removed \""+task+"\" from the list.")

print()

break

#Key does not exist; return to main menu

if not exists:

print("Task does not exist; returning to main menu.")

print()

# TODO: Add Code Here!

return list\_of\_rows

## Figure 3 Remove Task Function

The final task to fill out was the write\_data\_to\_file() function which would save the global list to a file. Yet again, code reuse was employed here, tailored to the arguments passed. The author found out that the list of rows was actually not modified at all and, once written to the file, would simply be returned. Figure 4 shows the function code.

@staticmethod

def write\_data\_to\_file(file\_name, list\_of\_rows):

""" Writes data from a list of dictionary rows to a File

:param file\_name: (string) with name of file:

:param list\_of\_rows: (list) you want filled with file data:

:return: (list) of dictionary rows

"""

# TODO: Add Code Here!

file\_obj = open(file\_name,"w")

#If the list is empty, no data to save so file contents are deleted

if len(list\_of\_rows) == 0:

print("List is empty; all items on file are deleted")

file\_obj.close()

else:

#There are items to save so let's write them

print("Saving current data to "+file\_name)

print()

counter = 1

#Get each dictionary row from list

for row in list\_of\_rows:

#If the task is not the last one on the table, add to file with newline at the end

if counter != len(list\_of\_rows):

file\_obj.write(row["Task"]+","+row["Priority"]+"\n")

counter+=1

#If item is the last, no new line at the end

else:

file\_obj.write(row["Task"]+","+row["Priority"])

file\_obj.close()

#list\_of\_rows is not modified so returning the same list

return list\_of\_rows

## Figure 4 Write to File Function

## Execution

The author made use of VS Code’s debug function a couple of times; most notably, when troubleshooting the remove function. When adding data, the list would accept the arguments and did its checks successfully, meaning that it rejected adding tasks already on the list. However, for the remove part, the author was always getting that the task did not exist. When using the debugger, he realized that, in the for loop, the string being compared was not modified, versus the one on the list which was modified to be lowercase. After setting it to lowercase, the function worked as intended.

Below are full runs from VS Code and the Terminal output. They also contain a screenshot of their respective run being executed.



## File 1: VS Code Output



## File 2: Terminal Output

# Summary

For this week, continued using dictionaries and lists, but this time within the confines of functions, which are in turn declared within classes. The author is familiar with both concepts as Java requires classes for any program (even main). The author was also able to see the flexibility Python offers when it comes to using functions with arguments and how Python protects global variables from local scope. Finally, though the author has had plenty of experience debugging code, he used VS Code’s Python debugger to troubleshoot, allowing him the opportunity to explore another debugger.