

Name: Raj Nune

Date: 8-23-2023

Course Foundations of Programming Python

## Files & Exceptions

### Introduction:

In Module 7, we delved into several important concepts. We learned about the benefits of encapsulating built-in Python commands into functions, which enhances code readability and reusability. We also discussed the advantages of structured error handling, such as try/except blocks in Python, which improve the robustness of the code. We differentiated between text files, which store human-readable data, and binary files, which are intended for computer programs to read and process. We explored how the Exception class in Python is used to handle errors during program execution, and how new classes can be derived from the Exception class to create custom exceptions for specific error conditions. Finally, we introduced Markdown, a lightweight markup language for formatting text, and explained how it can be used on a GitHub webpage.

### Working with Text files:

In the initial code listing, a function bearing the name "save\_data" is defined, taking two essential arguments: "data" and "file\_name." Within this function, the process commences by initializing a file connection based on the provided "file\_name," after which it undertakes a writing operation facilitated by the invocation of the "open(file\_name, 'w')" command. Subsequently, the content encapsulated within the "data" parameter is inscribed onto the file through utilization of the "file.write(data + '\n')" directive. To ensure proper resource management, closure of the file is achieved through the "file.close()" instruction. During the subsequent phase of the code dedicated to presentation, the aforementioned function is invoked. This invocation is configured to relay the contents of the "data" variable, which is assigned the value of "strData," as well as the designated "file\_name," denoted as "strFileName." Ultimately, culminating the sequence, a declaration is outputted, succinctly confirming the successful preservation of data.

```
# ----- #
# Title: Listing 1
# Description: Writing to a file with write()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ----- #

# Data ----- #
strData = 'test data'
strFileName = 'AppData.txt'
```

```

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    file = open(file_name, "w")
    file.write(data + "\n")
    file.close()

# Presentation ----- #
save_data(strData, strFileName)
print('Data Saved!')

```

The provided script, titled "Listing 2," is oriented around reading and storing data using file operations. It commences by establishing specific data and file name parameters. The primary function, "save\_data," stores a given string, referred to as "data," into a file identified by "file\_name." This function ensures data integrity by opening the file in write mode, appending the data, and then closing the file. Subsequently, the "read\_data" function is introduced. This function reads and returns all string-based data encapsulated within a specified file. By opening the file in read mode, the function extracts the entire content using the "file.read()" method, providing a comprehensive dataset that is subsequently returned. The operational sequence concludes by demonstrating the utility of these functions. The "save\_data" function is initially invoked to store predefined data in a designated file. Subsequently, the "read\_data" function is employed to extract the stored data, which is then presented through print statements. Notably, an appended observation underscores the detection of an additional blank line during reading, possibly indicating a newline character at the data's conclusion.

## Listing 2:

```

# ----- #
# Title: Listing 2
# Description: Reading from a file with read()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

# Data ----- #
strData = 'test data\nmore test data'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with the name of file

```

```

        :return: nothing
        """
        file = open(file_name, "w")
        file.write(data + "\n")
        file.close()

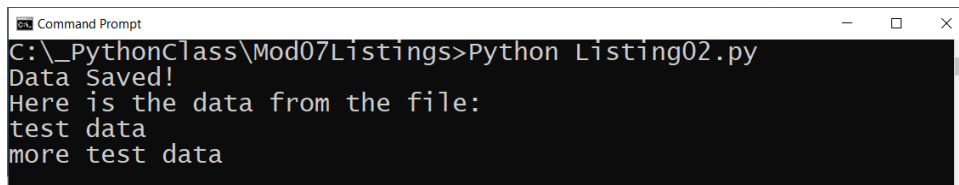
def read_data(file_name):
    """ Reads all string data from a file

    :param file_name: (string) with name of file
    :return: (string) of data read from the file
    """
    file = open(file_name, "r")
    data = file.read() # read all the data in the file at once
    file.close()
    return data

# Presentation ----- #
save_data(data=strData, file_name=strFileName)
print('Here is the data from the file:')
print(read_data(file_name=strFileName))
print('^ Note the extra blank line was read from the file too! ^')

```

Output:



```

C:\_PythonClass\Mod07Listings>Python Listing02.py
Data Saved!
Here is the data from the file:
test data
more test data

```

## Readline() Function:

In the context of the fourth listing, an observant analysis reveals the utilization of the "Readline()" function. This particular function exhibits the behavior of exclusively accessing the initial line contained within the designated text file, namely "AppData.txt." It is pertinent to note that with each invocation of the "read\_data\_row" function — a process comprising both opening and subsequently closing the file — the cursor is systematically repositioned to the inception of the file's content. This distinct dynamic is more apparent when we contemplate the series of events transpiring during the execution. In the initial occurrence of invoking "print(read\_data\_row(file\_name=strFileName))," the foremost line of data encapsulated within the "AppData.txt" file is retrieved. Post this operation, the file is promptly closed. However, during the subsequent invocation, characterized by the statement "print(read\_data\_row(file\_name=strFileName))," the earlier mentioned behavior becomes evident. Given the prior closure of the file within the "read\_data\_row" function, the data retrieval process recommences from the outset of the "AppData.txt" file. A viable resolution to this quandary involves maintaining the "read\_data\_row" function in an open state. By doing so, the "Readline()" function is empowered to access successive lines of data within the text file. This operational continuity is pivotal, enabling the second "print" statement to acquire the desired data seamlessly.

```

# ----- #
# Title: Listing 4
# Description: Reading a single line from a file with readline()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ----- #

# Data ----- #
strData = 'test data\nmore test data'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    file = open(file_name, "w")
    file.write(data + "\n")
    file.close()

def read_data_row(file_name):
    """ Reads a row of string data from a file

    :param file_name: (string) with name of file
    :return: (string) with one row of data from the file
    """
    file = open(file_name, "r")
    # readline() acts like a "cursor"
    data = file.readline() # read one row of data in the file
    file.close()
    return data

# Presentation ----- #
save_data(data=strData, file_name=strFileName)

print('Here is the first row of data from the file:')
print(read_data_row(file_name=strFileName))
print('^ Note the extra blank line was read from the file too! ^')

print('Here is the SAME row of data from the file:')
print(read_data_row(file_name=strFileName))

```

**Output:**

```
Run Main x
C:\Python\Python3.x\python.exe C:\Python\_PythonClass\Assignment07\Main.py
Here is the first row of data from the file:
test data

^ Note the extra blank line was read from the file too! ^
Here is the SAME row of data from the file:
test data

Process finished with exit code 0
```

### Lab 7-1:

The `save_data_to_file` function is pickling the data. It opens the specified file in binary write mode ("ab"), then it uses `pickle.dump()` to serialize `list_of_data` and write it into the file.

The `read_data_from_file` function is unpickling the data. It opens the file in binary read mode ("rb"), then it uses `pickle.load()` to read the file and deserialize the data.

### Input:

```
# ----- #
# 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):
    pass # TODO: Add code here
    file = open(file_name, "ab")
    pickle.dump(list_of_data, file)
    file.close()

def read_data_from_file(file_name):
    pass # TODO: Add code here
    file = open(file_name, "rb")
    list_of_data = pickle.load(file) # load() only loads one row of data.
    file.close()
```

```

    return list_of_data

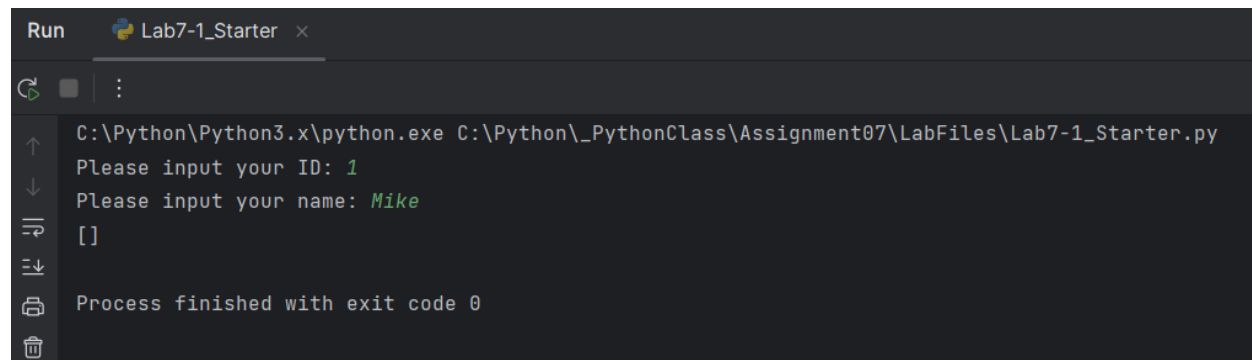
# Presentation ----- #
# TODO: Get ID and NAME From user, then store it in a list object

Idinput = input("Please input your ID: ")
nameinput = input("Please input your name: ")
customerinfo = [Idinput, nameinput]
# TODO: store the list object into a binary file
save_data_to_file(strFileName, customerinfo)

# TODO: Read the data from the file into a new list object and display the
contents
print(read_data_from_file(strFileName))

```

## Output:



```

Run Lab7-1_Starter x
C:\Python\Python3.x\python.exe C:\Python\_PythonClass\Assignment07\LabFiles\Lab7-1_Starter.py
Please input your ID: 1
Please input your name: Mike
[]
Process finished with exit code 0

```

## Structured Error handling:

Structured error handling employs a mechanism known as a 'try-except' block to systematically intercept and manage errors that may arise during program execution. This approach provides an enhanced user experience by substituting standard Python error messages with custom error messages generated by the application. This not only improves user comprehension of encountered issues but also contributes to a more polished and user-friendly software environment.

In Listing thirteen, we encounter the following code. Within the 'try' block, a segment of code is designated for execution, wherein Python endeavors to perform a mathematical operation, specifically, 'quotient = 5/0'. This calculation is mathematically undefined and results in a ZeroDivisionError. Subsequently, we encounter an 'except ZeroDivisionError as e:' statement. This construct functions as an error handler designed to catch ZeroDivisionErrors. Should such an error arise within the 'try' block, the code beneath this line will be executed. Continuing within the 'try' block, we find the statement 'f = open('SomeFile.txt', 'r+')', which endeavors to open a file named 'SomeFile.txt' in both read and write modes ('r+'). This operation introduces the possibility of encountering a FileNotFoundError, especially if the file does not exist. To address this, an 'except FileNotFoundError as e:' clause follows. This construct captures FileNotFoundError instances, and if such an error emerges from the preceding 'try' block, the code below this line will be executed. Resuming the 'try' block, we encounter the line 'f.write(quotient)', representing an attempt to write the calculated quotient into the file. However, this operation is contingent upon the absence of any prior errors. To address unforeseen exceptions not covered by the

preceding 'except' clauses, a more general 'except Exception as e:' statement is present. This clause acts as a catch-all for unspecified exceptions. In the event that an exception not handled by the preceding clauses arises, the code beneath this line will be executed.

#### Listing 13:

```
try:
    quotient = 5/0
    f = open('SomeFile.txt', 'r+') # the read plus option gives an error if
    # file does not exist
    f.write(quotient) # causes an error if the file does not exist
except ZeroDivisionError as e:
    print("Please do no use Zero for the second number!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
except FileNotFoundError as e:
    print("Text file must exist before running this script!")
    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: ")
```

#### Final project for module 7:

The script begins by importing the 'pickle' module and designating a file name as 'data.pkl'. Two functions are defined: 'save\_file' for storing data in binary format and 'read\_the\_file' for reading and retrieving the stored data. The script then captures an employee's name through user input and presents a cleaning menu, awaiting numeric input representing a cleaning task selection. Upon validation, the employee's name and chosen cleaning task are stored as a list, which is then saved using the 'save\_file' function. Finally, the stored data is read and displayed using the 'read\_the\_file' function, showcasing the employee's name and their assigned cleaning task.

#### Input:

```
import pickle

filename = 'data.pkl'

def save_file(data, filename):
    file = open(filename, "wb")
    pickle.dump(data, file)
    file.close()

def read_the_file(filename):
    file = open(filename, "rb")
    data = pickle.load(file)
```

```

        file.close()
        return data

employee_name = input("Please enter customer name: ")

print("Menu for cleaning \n 1. Mop \n 2. Dishes \n 3. Dusting")
while True:
    employee_cleaning = input("Please input number: ")
    if employee_cleaning.isnumeric() and 1 <= int(employee_cleaning) <= 3:
        break
    else:
        print("Invalid input. Please enter a numeric value between 1 and 3.")

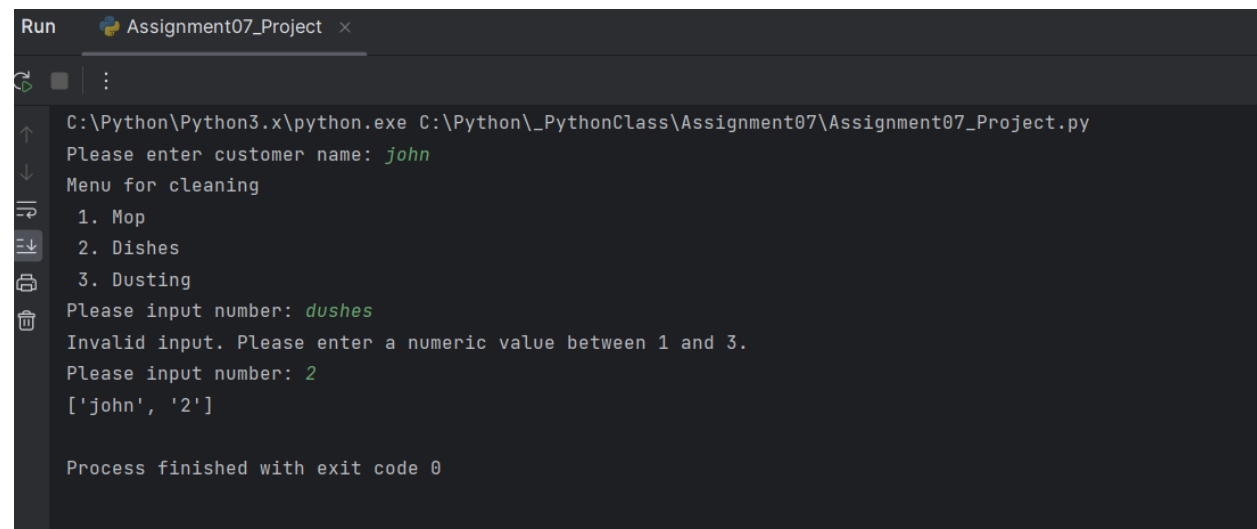
employee_cleans = [employee_name, employee_cleaning]

save_file(employee_cleans, filename)

print(read_the_file(filename))

```

### Output:



The screenshot shows a Python IDE window titled "Assignment07\_Project". The output console displays the following text:

```

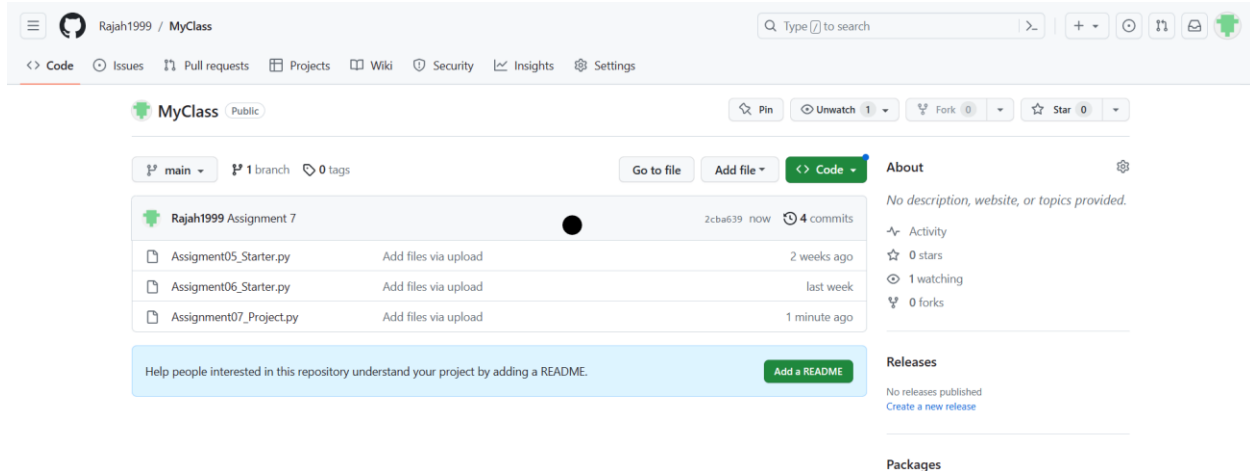
C:\Python\Python3.x\python.exe C:\Python\PythonClass\Assignment07\Assignment07_Project.py
Please enter customer name: john
Menu for cleaning
1. Mop
2. Dishes
3. Dusting
Please input number: dushes
Invalid input. Please enter a numeric value between 1 and 3.
Please input number: 2
['john', '2']

Process finished with exit code 0

```

### Github Upload:





## Conclusion:

Completing this assignment was a bit tougher, not so much because of the code itself, but mainly because I struggled to come up with a clear plan. It took me a while to figure out what kind of code project to work on. Once I got that sorted, I just made sure my code matched the project requirements. It's been a lesson in choosing the right project and sticking to the rules, and it helped me understand