

Daniel Castro

August 22nd, 2023

IT FDN 110 A Su 23: Foundations Of Programming: Python

Assignment 07

<https://github.com/DanielCyar/IT-FDN-110-A-Su-23-Mod07>

Pickling and Exception Handling

Introduction

In Assignment 7, we explore the domains of exception handling and Python's pickling module, learning about their applications and intricacies. Building upon our prior knowledge, this assignment equips us to manage errors and utilize pickling techniques for data storage and retrieval. Through research and comprehensive documentation, we develop a deeper understanding of exception handling and data serialization. Additionally, we try to improve our GitHub webpage to showcase our insights and for knowledge dissemination. Through the assignment tasks, we reinforce our grasp on vital concepts, including structured error handling and the utilization of Python's pickling capabilities.

Pickling

In the programming context, pickling refers to the serialization of complex data structures into a compact binary format. This enables us to store intricate data efficiently for future use, as it reduces storage space while ensuring data integrity, as serialization allows for the preservation of the object's original state without losing any relevant information. Essentially, what it does is employ a serialization protocol that converts data into a byte stream, which can be transferred or transmitted across different systems. (Python Pickle Tutorial: Object Serialization, <https://www.datacamp.com/tutorial/pickle-python-tutorial>, 2022) (External Link). Using Python's pickling module allows us to do just that. Save complex data structures to be stored and/or retrieved later on. In my case, learning how it works or the practical uses of it gives me a better understanding of the concept and provides me some basis to explore ways to use it.

Table 1 shows some 'selected binary file access modes' that can be used while pickling, straight from the course textbook (M. Dawson, Python® Programming for the Absolute Beginner, 3rd Edition, 2010).

TABLE 7.3 SELECTED BINARY FILE ACCESS MODES	
Mode	Description
"rb"	Read from a binary file. If the file doesn't exist, Python will complain with an error.
"wb"	Write to a binary file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
"ab"	Append a binary file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.
"rb+"	Read from and write to a binary file. If the file doesn't exist, Python will complain with an error.
"wb+"	Write to and read from a binary file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
"ab+"	Append and read from a binary file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.

Table 1. Binary File Access Modes. (M. Dawson, 2010)

Meanwhile, table 2 us the selected pickle functions that we can use to either write to the binary file or read from it. (M. Dawson, Python® Programming for the Absolute Beginner, 3rd Edition, 2010).

TABLE 7.4 SELECTED PICKLE FUNCTIONS	
Function	Description
<code>dump(object, file, [,bin])</code>	Writes pickled version of object to file. If bin is True, object is written in binary format. If bin is False, object is written in less efficient, but more human-readable, text format. The default value of bin is equal to False.
<code>load(file)</code>	Unpickles and returns the next pickled object in file.

Table 2. Selected Pickle Functions. (M. Dawson, 2010)

Exception Handling

Exception handling enhances code's reliability, making it more robust and user-friendly. Its fundamental aspect is to ensure that the code can deal with unexpected situations effectively. Basically, the process consists of placing risky code within a **try** block, and if an exception arises, Python searches for a corresponding **except** block that can handle that specific exception. This way, one can also provide customized responses for different types of errors. Exception handling acts as a safety net, allowing the program to handle errors without crashing. (Python Exceptions: An Introduction, <https://realpython.com/python-exceptions>, 2018) (External Link).

I've already explored this concept on previous assignments, when learning about error handling and applying it to the script. For example, when creating the ToDo list, I used try/except to verify if there was a .txt file to read data. I did this while thinking, if a user wants to start writing to a file and there's no file to begin with, then the code will not run, as it encounters an error just at the start line. This time, I'm placing the referred table from the textbook to have some idea of the most common exception types.

TABLE 7.6 SELECTED EXCEPTION TYPES

Exception Type	Description
<code>IOError</code>	Raised when an I/O operation fails, such as when an attempt is made to open a nonexistent file in read mode.
<code>IndexError</code>	Raised when a sequence is indexed with a number of a nonexistent element.
<code>KeyError</code>	Raised when a dictionary key is not found.
<code>NameError</code>	Raised when a name (of a variable or function, for example) is not found.
<code>SyntaxError</code>	Raised when a syntax error is encountered.
<code>TypeError</code>	Raised when a built-in operation or function is applied to an object of inappropriate type.
<code>ValueError</code>	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
<code>ZeroDivisionError</code>	Raised when the second argument of a division or modulo operation is zero.

Table 3. Selected Exception Types. (M. Dawson, 2010)

However, for a bigger list, the following webpage provides the exception hierarchy:

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

The Exception Hierarchy is a structured way of categorizing and organizing different types of errors that can occur during program execution. It is designed to help developers handle exceptions more effectively by providing a clear structure of exception types (ChatGPT, 2023) (External Link).

There's also a different way to handle errors in Python and it is **Structured Error Handling**. The main difference between them is that Exception Handling's syntax is based on using "try/except" blocks (and sometimes "finally" ones too), meanwhile structured error handling's syntax is based on if/elif/else statements, along with functions or methods to explicitly check for conditions that may lead to or avoid errors.

Getting into practice

In the practical part of Assignment 7, we will create a new script that demonstrates how Pickling and Structured Error Handling works.

Creating the script

Started with the initial template, I placed a small description of the script and added the changelog information.

```
# ----- #
# Title: Assignment 07
# Description: Creating a script that explains how pickling and exception handling works.
#              It reads and writes user data such as name and age of multiple users.
#
# ChangeLog (Who,When,What):
# DanielCastro,8.22.2023,Created starting script
# DanielCastro,8.23.2023,Added functions to write, read, get and display with error handling
# ----- #
```

I thought of integrating the 3 items in the same script: Pickling, Structured Error Handling and Exception Handling. First, I imported the pickling module.

```
import pickle
```

Then, following the separation of concerns, I created the data, process, IO and main sections of the script. In the data section, I initialized the variables that were going to be used.

```
# DATA SECTION
user_data = None           # Empty storage variable
data_list = []             # Empty List
file_name = 'UserData.dat' # .dat File Name
# DATA SECTION
```

Then, for the process section, I implemented this part to test the usage of Pickling in Python. I created a class called Processor and defined 2 functions inside. A save_data function to save the data to the file as binary by using the dump method of the pickle module (I placed an IOError exception in case it couldn't find the file we wanted to write to).

```
# PROCESS SECTION
class Process:

    @staticmethod
    def save_data(objFile, data_list):
        try:
            file = open(objFile, 'wb')
            pickle.dump(data_list, file)
            file.close()
            print("\nUser data saved successfully.")
        except IOError:
            print("ERROR: Could not save user data.")
```

Also, created a load_data function to read binary from the file and used the load method of the pickle module to read a row of data from the file (with a respective FileNotFoundError exception, in case the .dat file didn't exist in the first place).

```
@staticmethod
def load_data(objFile):
    try:
        file = open(objFile, 'rb')
        user_data = pickle.load(file)
        file.close()
        return user_data

    except FileNotFoundError:
        print("User data file not found.")
# PROCESS SECTION
```

Then, for the input/output section, I created a class named IO and inside of it, defined 2 functions as well. The first one gets input from the user and is where I tested both Structured Error Handling

and Exception Handling. First, I asked to **try** and retrieve the name and age of the user. Then, by using some if/elif statements I verified 2 things:

1. That the User's name was a string
2. That the User's age was an integer

If either of them didn't apply, it would raise an Exception depending on which one failed first. In here I also placed a condition that if the user enters "exit" on the name, it will stop running the program.

```
# INPUT/OUTPUT SECTION
class IO:

    @staticmethod
    def get_input():

        try:
            name = input("\nEnter user's name (or type 'exit' to finish): ")

            if name.lower() == 'exit':
                input("\n[Press enter to Exit...]")
                exit()
            age = input("Enter user's age: ")

            if name.isnumeric():
                raise Exception("\nERROR: Name should be a string.") # Structured Error Handling
            elif not age.isnumeric():
                raise ValueError("\nERROR: Age should be an integer.")
            else:
                return {"name": name, "age": age}

        except Exception as e:
            print(e) # Exception handling
        except ValueError as a:
            print(a)
```

Then, the other function takes the data_list so far and displays the information as a list with name and age of each user inserted on the list by using a for loop.

```
@staticmethod
def display_data(data_list):
    print("\nLoaded user data:")
    for row in data_list:
        print(f"Name: {row['name']}, Age: {row['age']}")
# INPUT/OUTPUT SECTION
```

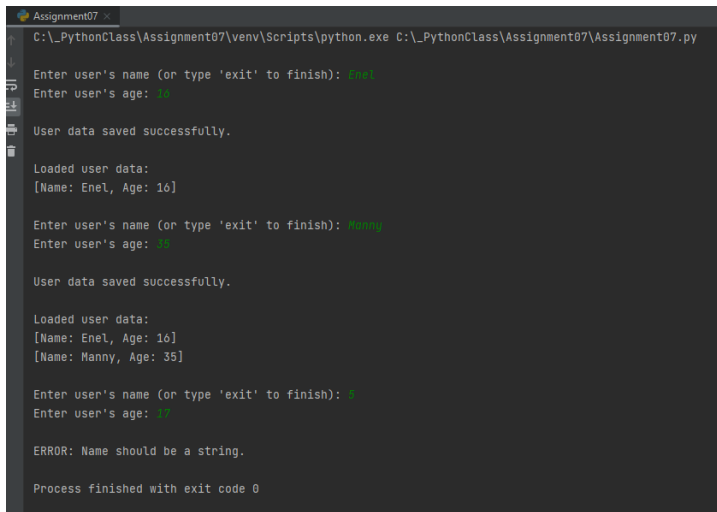
Finally, the main body of the script takes the input (where if no data is entered it will start asking again), then append the user data to the list, save the data to the .dat file. Then, it reads the data present in the list and displays it on the main screen before asking the user if he wants to enter any new users or exit.

```
# MAIN BODY OF THE SCRIPT
while True:

    user_data = IO.get_input()
    if user_data is None:
        break
    data_list.append(user_data)
    Process.save_data(file_name, data_list)
    loaded_users = Process.load_data(file_name)
    IO.display_data(loaded_users)
```

Running the script in Windows Command Line

Figure 1 shows the program running on PyCharm.

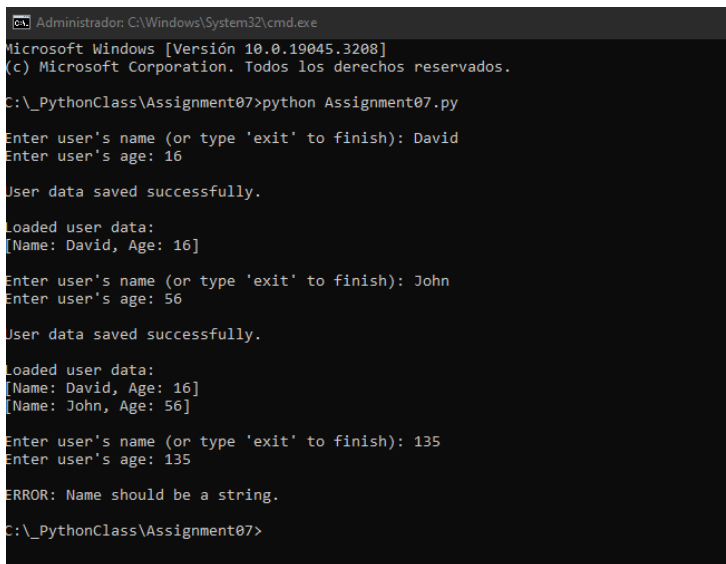
A screenshot of the PyCharm IDE showing the execution of a Python script. The terminal window displays the following text: "Enter user's name (or type 'exit' to finish): Enel", "Enter user's age: 16", "User data saved successfully.", "Loaded user data:", "[Name: Enel, Age: 16]", "Enter user's name (or type 'exit' to finish): Manny", "Enter user's age: 35", "User data saved successfully.", "Loaded user data:", "[Name: Enel, Age: 16]", "[Name: Manny, Age: 35]", "Enter user's name (or type 'exit' to finish):", "Enter user's age: 17", "ERROR: Name should be a string.", "Process finished with exit code 0". The script path shown at the top is "C:_PythonClass\Assignment07\venv\Scripts\python.exe C:_PythonClass\Assignment07\Assignment07.py".

```
Assignment07 > C:\_PythonClass\Assignment07\venv\Scripts\python.exe C:\_PythonClass\Assignment07\Assignment07.py
Enter user's name (or type 'exit' to finish): Enel
Enter user's age: 16
User data saved successfully.
Loaded user data:
[Name: Enel, Age: 16]
Enter user's name (or type 'exit' to finish): Manny
Enter user's age: 35
User data saved successfully.
Loaded user data:
[Name: Enel, Age: 16]
[Name: Manny, Age: 35]
Enter user's name (or type 'exit' to finish):
Enter user's age: 17
ERROR: Name should be a string.
Process finished with exit code 0
```

Figure 1. Assignment07.py running with PyCharm.

Running the script in Windows Command Line

To run the script from the Windows Command Line, it's just accessing the directory where the 'Assignment07.py' file is saved and type the name of the Python file. The script will start running (Figure 2).

A screenshot of the Windows Command Line running the Python script. The prompt is "Administrator: C:\Windows\System32\cmd.exe". The output shows: "Microsoft Windows [Versi3n 10.0.19045.3208]", "(c) Microsoft Corporation. Todos los derechos reservados.", "C:_PythonClass\Assignment07>python Assignment07.py", "Enter user's name (or type 'exit' to finish): David", "Enter user's age: 16", "User data saved successfully.", "Loaded user data:", "[Name: David, Age: 16]", "Enter user's name (or type 'exit' to finish): John", "Enter user's age: 56", "User data saved successfully.", "Loaded user data:", "[Name: David, Age: 16]", "[Name: John, Age: 56]", "Enter user's name (or type 'exit' to finish): 135", "Enter user's age: 135", "ERROR: Name should be a string.", "C:_PythonClass\Assignment07>".

```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Versi3n 10.0.19045.3208]
(c) Microsoft Corporation. Todos los derechos reservados.
C:\_PythonClass\Assignment07>python Assignment07.py
Enter user's name (or type 'exit' to finish): David
Enter user's age: 16
User data saved successfully.
Loaded user data:
[Name: David, Age: 16]
Enter user's name (or type 'exit' to finish): John
Enter user's age: 56
User data saved successfully.
Loaded user data:
[Name: David, Age: 16]
[Name: John, Age: 56]
Enter user's name (or type 'exit' to finish): 135
Enter user's age: 135
ERROR: Name should be a string.
C:\_PythonClass\Assignment07>
```

Figure 2. Assignment07.py running in Windows Command Line.

Once we press Option 5 and then press Enter, the code stops running. Figure 3 displays how the data was saved inside the UserData.dat file.

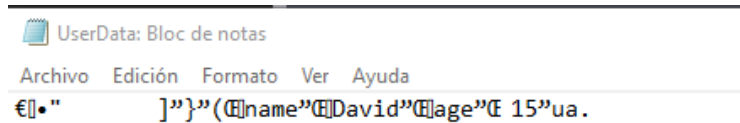


Figure 3. UserData.dat file with the saved data.

Summary

In Assignment 7, we explored Python's Pickling concepts and deepened our understanding on Error Handling while applying them to practical scenarios. We developed a script to test those concepts out and created a clear separation of concerns in our code, promoting modularity and clarity. By employing the Pickle module, we learned how to serialize and deserialize data, a crucial skill for preserving complex data structures. The implementation of structured error handling taught us the importance of managing unexpected situations, enhancing the robustness of our programs. This assignment not only expanded our knowledge of Python's capabilities but also improved our ability to write more reliable and efficient code.