

Kyle O'Donnell

August 24, 2020

Foundations of Programming: Python

Assignment07

<https://github.com/kylenod/ITFDN-Mod07/blob/master/docs/index.md>

Pickling and Exception Handling in Python

Introduction

This paper documents the process of creating a Python script that demonstrates pickling and structured error-handling. The product of this week's assignment is a simple program, Assignment07.py, that lets dog-oriented businesses manage their database of dog clients. Topics covered in this assignment include binary files and exceptions, which were part of week 7's lessons. Additionally, the paper discusses online resources that were used for this assignment.

Writing the Script

Assignment07.py was created from scratch in PyCharm. Although it did not have a starting template, the structure of the script and several of its custom functions were based on previous assignments. As a first step, I created a new Python project in the **Assignment07** directory and updated the template header.

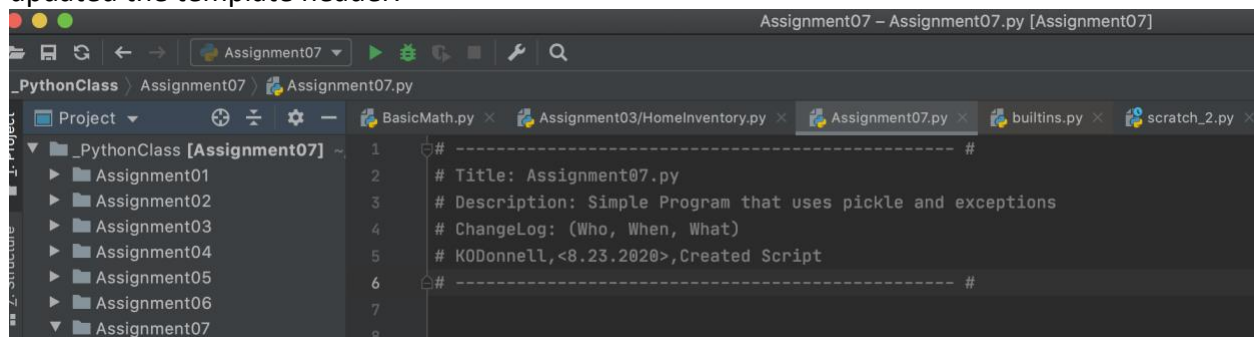


Figure 1. Screenshot of Script in PyCharm

Overview

This program is designed to be used by dog-oriented businesses, such as groomers, dog-walkers and boarders. When users launch the program, they can select one of the following options to help manage their dog database:

- 1) View a list of their clients by name
- 2) Add a new client
- 3) Search information for a specific client
- 4) Save new data to their database
- 5) Exit the program

Script Structure

The script has three main sections – data, processing and presentation.

The data layer declares objects that are used by the program.

```
# Data ----- #
# Declare variables and constants

strFileName = 'ClientData.dat' # Name of binary data file
objFile = None # Object that represents the file
strClientName = '' # Client name string
lstClients = [] # List of clients
strChoice = '' # Menu option string
intChoice = None # Menu option integer
dicNewClientInfo = {} # Dictionary with new client info
```

Figure 2. Data Section of Script

The processing section includes custom functions for processing, input/output functions and custom exception classes.

```
# Processing ----- #
class Processor:
    """ Performs Processing tasks """

    @staticmethod
    def read_data_from_file(file_name, list_of_clients): # Load data from binary file
        """ Reads data from a binary file into a list
        :param file_name: (string) with name of file:
        :param list_of_clients: (list) you want filled with file data:
        :return: (list) of data
        """
```

Figure 3. Processing Functions

```
# Presentation ----- #
class IO:
    @staticmethod
    def welcome_message(): # Prints welcome message
        """ Welcome users to program
        :return: nothing
        """
        print("\nWelcome to your DogForce - Your Business' Best Friend!!!!\n"
              "We'll help you manage your growing list of dog clients!")
        print()
```

Figure 4. Input/Output Functions

```
# Exceptions ----- #
class ValueTooLargeError(Exception): # Exception for numbers over 5
    """ Value too large for menu input """
    def __str__(self):
        return "Please select an option between 1-5!"

class StopError(Exception): # Exception for users that want to quit
    """ Close program """
    def __str__(self):
```

Figure 5. Custom Exceptions

The last section is the main body of the script, which provides logic for calling functions and interacting with users.

```
while True:
    I0.print_menu_tasks() # Prompt for menu choice
    try:
        strChoice = I0.input_menu_choice()
        if strChoice in ["quit", "close", "leave", "exit", "stop"]: # Raise exception for "quit" words
            raise StopError
        intChoice = int(strChoice) # Raise exception for non-integers
        if intChoice > 5: # Raise exception for values over 5
            raise ValueError
    except ValueError:
```

Figure 6. Main Body of Script

Pickling

In previous assignments, I have read, written and appended data to text files. In contrast, this week's program uses pickling to load and save data to a binary file. There are several reasons to pickle data instead of using a text file. First, pickling can preserve the structure of complex Python objects which might otherwise be lost when saved to a text file. Second, pickling is used with binary files which take up less memory than other types of files, making it easier for storing and transferring. Some of the limitations of pickling include the fact it can only work with certain types of objects, is limited to Python structures, and the output is not user-friendly (i.e. difficult to read).

In Assignment07.py, the first line of the script imports the pickle module so pickle functions can be used later in the script:

```
import pickle
```

Listing 1

Loading data with Pickle

The first custom function, **read_data_from_file()**, uses `pickle.load()` to read data from a binary file (objFile was assigned to the file ClientData.dat in the data section). More specifically, the function is “de-serializing” data into Python data structures from the binary stream it was previously saved to. The `pickle.load()` function will only load one object at a time (picking up where it left off within the open file). In this case, the program saves the database to a single list, otherwise that would have to be further accounted for in the program. The file is opened in “rb” or “read binary” mode, because it needs to read data from a binary file.

```
def read_data_from_file(file_name, list_of_clients):
    try:
        objFile = open(file_name, "rb")
        list_of_clients = pickle.load(objFile)
        objFile.close()
    except FileNotFoundError:
        print("I see you haven't started a database yet!\nThat's okay! We're here to help you get started.\n")
    return list_of_clients
```

Listing 2

Figure 7 shows how the binary data appears within a normal text editor. Although the format is not user-friendly, it is still somewhat interpretable.

```
1  [\x00H]\x00(\x00 Name\x00 Cecil\x00 Age\x00K\x00 Breed\x00shepherd\x00 Notes\x00 Good Boy!!!\x00ua.
```

Figure 7. Binary Data in Basic Text Editor

Once the data is loaded into a python list through pickling (Figure 8), it looks like a normal Python dictionary within a list (Figure 9)

```
1  import pickle
2  file_name = "ClientData.dat"
3
4  objFile = open(file_name, "rb")
5  list_of_clients = pickle.load(objFile)
6  objFile.close()
7
8  print(list_of_clients)
```

Figure 8. Loading Data with Pickle

```
[{'Name': 'Cecil', 'Age': 1, 'Breed': 'shepherd', 'Notes': 'Good Boy!!!'}]
```

Figure 9. Output of Printed Data Loaded From Binary File with Pickle

Saving Data with Pickle

Pickling can also be used to store data. In the function **save_data_to_file()**, I use `pickle.dump()` to “serialize” the current client data back to the binary format. In this case, the file is opened in “wb” or “write binary” mode to write data to a binary file.

```
def save_data_to_file(file_name, list_of_clients):
    object_file = open(file_name, "wb")
    pickle.dump(list_of_clients, object_file)
    object_file.close()
    return list_of_clients
```

Listing 3

Python Pickling Research

While working on this script, I read through several online resources about Python pickle. The one I found most useful was a Medium article appropriately titled “Pickling in Python” by Alison

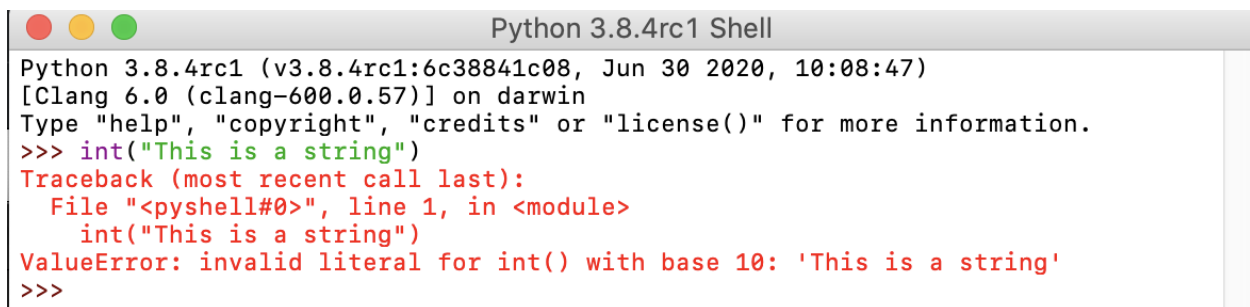
Salerno. I appreciated that it was a high-level overview that did not presuppose prior knowledge of other programming languages. Other articles either seemed to provide a comprehensive overview of pickling parameters or were meant to address specific use-cases for developers. This article concisely distilled high-level concepts, balancing prose with images and code examples. The article is available at the following link: <https://medium.com/swlh/pickling-in-python-ac3c7a045ae5>

Another page I found useful was the [www.geeksforgeeks.org](https://www.geeksforgeeks.org/understanding-python-pickling-example/) page “Understanding Python Pickling Example”. The detailed code example helped illustrate various use-cases for pickling: <https://www.geeksforgeeks.org/understanding-python-pickling-example/>

Structured Error Handling

Built-in Exceptions

Another concept explored thoroughly in week 7 was structured error handling. When an error occurs while running Python, the program provides information on the type of error that occurred. For example, if you try to cast a string of letters as an integer, Python will return a `ValueError`, because it cannot perform that function on that value (Figure 10).

A screenshot of a terminal window titled "Python 3.8.4rc1 Shell". The window shows the Python interpreter's startup information, including the version (v3.8.4rc1) and the platform (darwin). It then displays a prompt where the user enters the command `int("This is a string")`. The interpreter responds with a `ValueError: invalid literal for int() with base 10: 'This is a string'`. The error message is preceded by a `Traceback (most recent call last):` and the file `<pyshell#0>`, line 1, in `<module>`.

```
Python 3.8.4rc1 (v3.8.4rc1:6c38841c08, Jun 30 2020, 10:08:47)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> int("This is a string")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    int("This is a string")
ValueError: invalid literal for int() with base 10: 'This is a string'
>>>
```

Figure 10. Built-in `ValueError` in Python

Although we ideally prefer to avoid errors, there are many instances in which it's helpful to identify different types of errors so that they can be handled appropriately within a program. Python has a large number of built-in errors known as exceptions that can be leveraged for this purpose. In the `read_data_from_file()` function used at the beginning of the script, I employed **try-except** clauses and the built-in `FileNotFoundError` to handle instances in which the file being read does not yet exist.

```
def read_data_from_file(file_name, list_of_clients):
    try:
        objFile = open(file_name, "rb")
        list_of_clients = pickle.load(objFile)
        objFile.close()
    except FileNotFoundError:
        print("I see you haven't started a database yet!\nThat's okay! We're here to help you get started.\n")
        return list_of_clients
```

Listing 4

First, the function will try to load data from the object file. If the file is not found, it will skip to the except clause for **FileNotFoundError** and print the string below it. Because the error value already exists in Python, I was able to create the except clause without having to add much additional logic. When the except clause is reached, it will simply print the string from the statement and continue running (Figure 11). By contrast, if I did not include the clause, the error would still occur, but the displayed message would not be user-friendly (Figures 12).

```
Press enter to continue
Traceback (most recent call last):
  File "/Users/kyleodonnell/Documents/_PythonClass/Assignment07/Assignment07.py", line 156, in <module>
    lstClient = Processor.read_data_from_file(strFileName, lstClient) # read file data
  File "/Users/kyleodonnell/Documents/_PythonClass/Assignment07/Assignment07.py", line 25, in read_data_from_file
    objFile = open(file_name, "rb")
FileNotFoundError: [Errno 2] No such file or directory: 'ClientData.dat'
```

Figure 11. FileNotFoundError without Exception Clause in Script

```
Press enter to continue
I see you haven't started a database yet!
That's okay! We're here to help you get started.
```

Figure 12. FileNoteFoundError with Exception Clause in Script

In the **get_client_data()** function, I create an except clause for the built-in **ValueError** to catch instances in which the “age” input cannot be cast as an integer.

```
def get_client_data():
    client = str(input("Enter name: \n ").capitalize())
    while True:
        try:
            age = int(input("Enter age: \n "))
            break
        except ValueError:
            print("Please enter age as the nearest round number!")
```

Listing 5

In this case, the **try-except** clause occurs within a **while loop**. If the user enters a non-integer value that goes to this exception, they will be prompted to reenter a menu option. Once they input a valid age, the loop breaks and the program moves on to the next task.

Creating Custom Exceptions

In addition to leveraging existing exceptions, Python makes it easy to create custom exception classes derived from the same class. The advantage of the custom classes is that they can handle

use-cases specific to your program design. In Assignment07.py, I created two custom classes shown in Listing 6.

```
class ValueTooLargeError(Exception):
    """ Value too large for menu input """
    def __str__(self):
        return "Please select an option between 1-5!"

class StopError(Exception):
    """ Close program """
    def __str__(self):
        return 'Closing Program...Goodbye!'
```

Listing 6

The **ValueTooLargeError** catches instances in which the option input is an integer above 5 (the menu options are 1-5).

The **StopError** handles instances in which the user has implemented a non-integer string that clearly indicates they want to leave the program. This error is included to distinguish these inputs from more generic **ValueErrors** in which the user has simply entered a non-integer value.

These exceptions are used in the main body of the script.

```
while True:
    IO.print_menu_tasks()
    try:
        strChoice = IO.input_menu_choice()
        if strChoice in ["quit", "close", "leave", "exit", "stop"]:
            raise StopError
        intChoice = int(strChoice)
        if intChoice > 5:
            raise ValueTooLargeError
    except ValueError:
        print("Please provide a number input!")
    except ValueTooLargeError as e:
        print(e)
    except StopError as e:
        print(e)
        break
    else:
        if intChoice == 1:
```

Listing 7

When users provide an input, I first check to see if the string matches any “quit” words. I then raise the custom **StopError**, which prints the error string from the class and breaks the loop.

Next, I cast the input as an integer. If the input string cannot be cast as an integer, it will automatically raise a **ValueError** and go to the exception clauses for **ValueError**. If the input is an integer above 5, it will reach the **ValueTooLargeError**. Note that for the custom classes, the errors must be manually raised with logic built into the program.

If none of the exceptions are raised, the program goes to the else clause and executes one of the menu functions. Note that because the custom exceptions are derived from the existing class of built-ins, it is easy to incorporate them into except clauses using similar syntax to the built-ins.

Exception Handling Research

I found two sites that were particularly helpful for learning about exception handling:

- TechBeamers (<https://www.techbeamers.com/use-try-except-python/>, 2020) [External Site]
- Programiz (<https://www.programiz.com/python-programming/user-defined-exception>, 2020) [External Site]

Both pages provide high-level information about exceptions without including too much technical detail or requiring background knowledge of programming. The visual layouts make them easy to navigate and both use abundant images to help illustrate their points. Programiz has the additional feature of allowing visitors to run sample code directly on their site.

Testing in PyCharm

Once the script was completed, I tested it in Pycharm. Figures 13-17 show each of the exceptions being handled by the program as expected.

```
Welcome to your DogForce - Your Business' Best Friend!!!!
We'll help you manage your growing list of dog clients!

Press enter to continue
I see you haven't started a database yet!
That's okay! We're here to help you get started.

Menu of Options
1) See List of Clients
2) Add a New Client
3) Search for Client Information
4) Save Data to Computer
5) Exit Program
```

Figure 13. Exception Handling for FileNotFoundError in Script


```
Which option would you like to perform? [1 to 5] - 3

Enter name:
Kelby
Enter age:
eight
Please enter age as the nearest round number!
Enter age:
```

Figure 14. Exception Handling for ValueError in Script - 1

```
Which option would you like to perform? [1 to 5] - add client

Please provide a number input!
```

Figure 15. Exception Handling for ValueError in Script - 2

```
Which option would you like to perform? [1 to 5] - 7

"Please select an option between 1-5!"
```

Figure 16. Exception Handling for Custom ValueError

```
Which option would you like to perform? [1 to 5] - stop

Closing Program...Goodbye!
```

Figure 17. Exception Handling for Custom StopError

Running in Terminal

Next, I ran the program in Terminal as a Console application. Figures 18-22 show the programs tasks and exceptions executing successfully.

```
Which option would you like to perform? [1 to 5] - 1

***** Current Client List: *****
1. Kelby
2. Cecil
3. Seashell
*****
```

Figure 18. View Client List

```
Which option would you like to perform? [1 to 5] - 2

Enter name:
Comet
Enter age:
10
Enter breed:
Terrier
Enter any additional notes on Comet.
Woof!!
Comet has been saved to your database!
Press enter to continue 3
```

Figure 19. Add New Client

```
Which option would you like to perform? [1 to 5] - 3
What client would you like to look up?
Comet

Client Name: Comet
Age: 10
Breed: Terrier
Notes: Woof!!
Press enter to continue
```

Figure 20. Search Client Data

```
4) Save Data to Computer
5) Exit Program

Which option would you like to perform? [1 to 5] - 9
"Please select an option between 1-5!"
```

Figure 21. Custom Exception ValueError

```
Which option would you like to perform? [1 to 5] - exit
Closing Program...Goodbye!
(base) KYLEs-MacBook-Pro:Assignment07 kyleodonnell$
```

Figure 22. Custom Exception StopError

Checking the Binary File

As a final step, I checked the binary file "ClientData.dat" that the pickled data was saved to. Figure 23 shows the data from the previous step had been saved to the file.

```
ClientData.dat
Ãlq({q(XNameqXKelbyqXAgeqXBreedaXShepherdqXNotesqXGood Girl!!!!qu}q (XNameq
XCecila
XAgeq
KXBreeda
XShepherdqXNotesqXGood Boy!!!!qu}q(h
XSeashellq
Kh
XBeagleqhXAwwwwooooooo!!!!qu}q(h
XCometq
K
h
XTerrierqghXWoof!!que.
```

Figure 23. Viewing Binary File with Saved Data

Summary

This paper documents the process of creating and testing a basic program that uses pickling and structured error handling in Python. It includes an overview of both topics, as well as some recommended resources for learning about them.