Erica Peterson

August 22nd, 2022

Foundations of Programming: Python

Assignment 07

https://github.com/ericapet/ITFnd100-Mod07/blob/main/docs/index.md

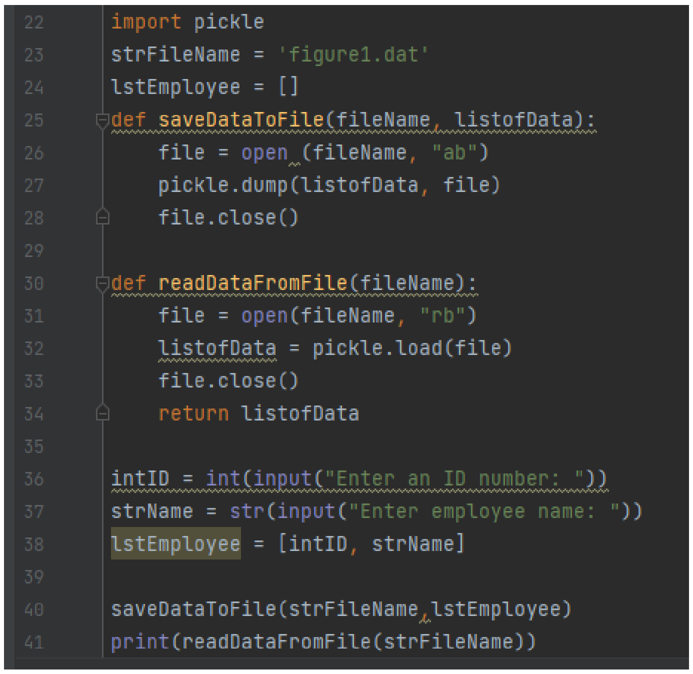
Pickling and Errors

Introduction

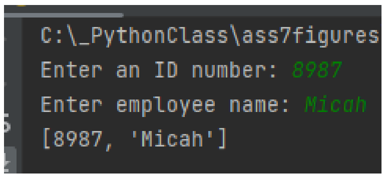
In this paper I will go over pickling and error handling, using the try-except function. Pickling is used to store data in a binary format, and this can be useful for making files smaller or sending files from one program to another. After discussing pickling and error-handling I will go over a program that uses both to do simple calculations and then save them to a binary file.

Pickling

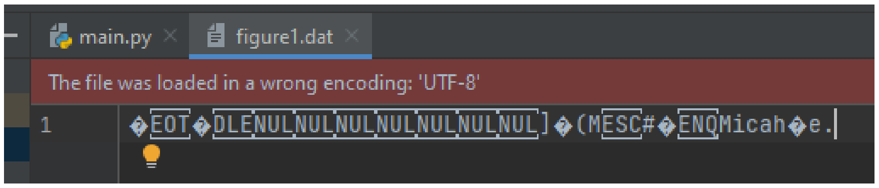
Data can be saved in a binary format, rather than plain text as we have seen in the previous assignments. In python, this is called pickling. Pickling is also used to serialize and deserialize a python object. Pickling is a way to convert python objects, such as lists or dictionaries, into a character stream. This character stream contains all the necessary information to reconstruct the object in another python script. Pickling is similar to files where you use the "w", "a" and "r" modes but instead of just "w", you would use "wb", "ab", or "rb" because everything is stored in bytes. "pickle.dump()" is used to pickle a file whereas "pickle.load()" is used to unpickle. An example of pickling and saving to a binary file can be found in Figure 1, on the next page.



```python
22  import pickle
23  strFileName = 'figure1.dat'
24  lstEmployee = []
25  def saveDataToFile(fileName, listofData):
26      file = open (fileName, "ab")
27      pickle.dump(listofData, file)
28      file.close()
29
30  def readDataFromFile(fileName):
31      file = open(fileName, "rb")
32      listofData = pickle.load(file)
33      file.close()
34      return listofData
35
36  intID = int(input("Enter an ID number: "))
37  strName = str(input("Enter employee name: "))
38  lstEmployee = [intID, strName]
39
40  saveDataToFile(strFileName,lstEmployee)
41  print(readDataFromFile(strFileName))
```



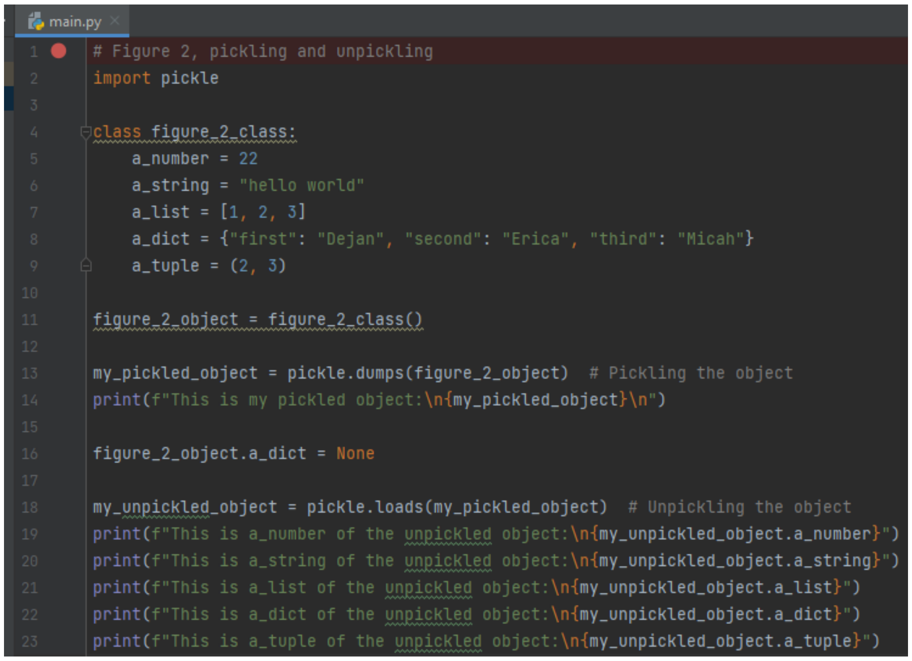
```
C:\_PythonClass\ass7figures
Enter an ID number: 8987
Enter employee name: Micah
[8987, 'Micah']
```



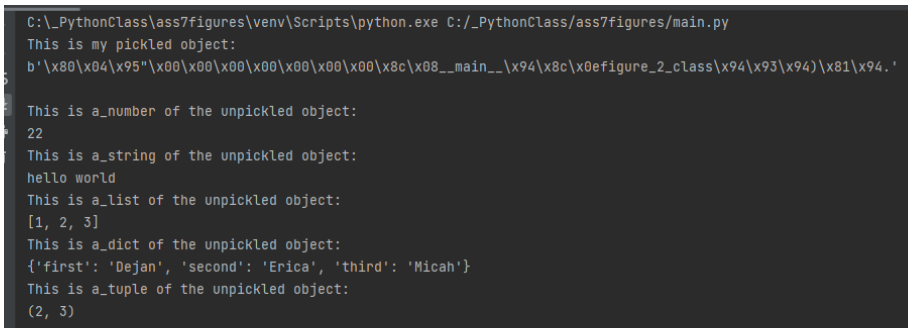
```
The file was loaded in a wrong encoding: 'UTF-8'
1   �EOT�DLENULNULNULNULNULNULNUL]�(MESC#�ENQMicah�e.
```

Figure 1. How to pickle data to save it in a binary file. The top image shows the script: In line 22, it is vital that you start with import pickle. Notice the use of "ab" and "rb" in lines 26 and 31. Line 32 is where the file is binarized using the pickle.load() function. The middle image shows the output while running the script, with user input in green. The bottom image shows the figure1.dat file, with binary characters.

Pickling is not only used to save data to a binary file- it can also be used to serialize and deserialize, as mentioned earlier. You can pickle any data type, such as dictionaries, tuples, lists, strings, etc. An example of pickling, using pickle.load(), and then unpickling, using "pickle.dump()", data can be found in Figure 2.



```python
# Figure 2, pickling and unpickling
import pickle

class figure_2_class:
    a_number = 22
    a_string = "hello world"
    a_list = [1, 2, 3]
    a_dict = {"first": "Dejan", "second": "Erica", "third": "Micah"}
    a_tuple = (2, 3)

figure_2_object = figure_2_class()

my_pickled_object = pickle.dumps(figure_2_object)  # Pickling the object
print(f"This is my pickled object:\n{my_pickled_object}\n")

figure_2_object.a_dict = None

my_unpickled_object = pickle.loads(my_pickled_object)  # Unpickling the object
print(f"This is a_number of the unpickled object:\n{my_unpickled_object.a_number}")
print(f"This is a_string of the unpickled object:\n{my_unpickled_object.a_string}")
print(f"This is a_list of the unpickled object:\n{my_unpickled_object.a_list}")
print(f"This is a_dict of the unpickled object:\n{my_unpickled_object.a_dict}")
print(f"This is a_tuple of the unpickled object:\n{my_unpickled_object.a_tuple}")
```

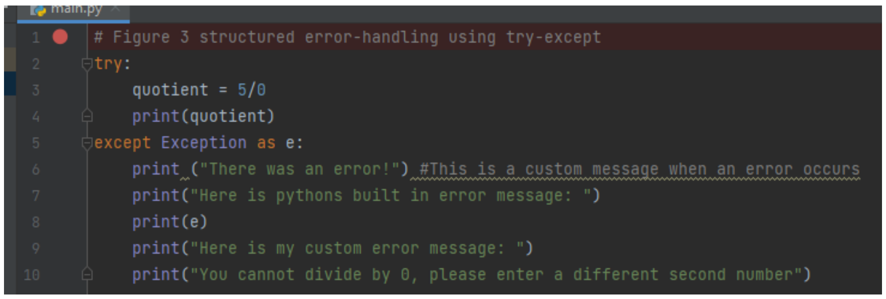


```
C:\_PythonClass\ass7figures\venv\Scripts\python.exe C:/_PythonClass/ass7figures/main.py
This is my pickled object:
b'\x80\x04\x95"\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\x94\x8c\x0efigure_2_class\x94\x93\x94)\x81\x94.'

This is a_number of the unpickled object:
22
This is a_string of the unpickled object:
hello world
This is a_list of the unpickled object:
[1, 2, 3]
This is a_dict of the unpickled object:
{'first': 'Dejan', 'second': 'Erica', 'third': 'Micah'}
This is a_tuple of the unpickled object:
(2, 3)
```

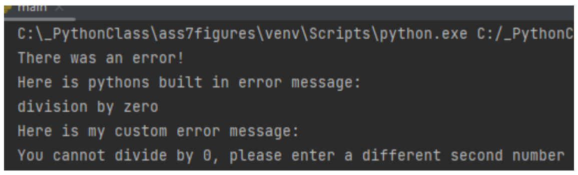
Figure 2. How to pickle and unpickle an object. The top image shows the script whereas the bottom image shows the output.

Structured Error-Handling

Structured error-handling is used to guess what errors a user might make and provide a clearer error message to the user. Basically, you want to look at your script, think of possible errors that could be caused by user input, and then tell the user what happened if they do cause an error. This is done using the try-except function, also known as try-catch in other languages. The reason why programmers use try-except rather than using the standard python error messages is because sometimes python error messages can be confusing for the average user who may not know programming. Using try-except makes your program more user friendly. Exception is a built-in python class that holds information about an error, python automatically creates an exception object when there is an error. The exception object will automatically fill in information about what caused the error. An example using try-except, and exception objects can be seen on the following page, in Figure 3.



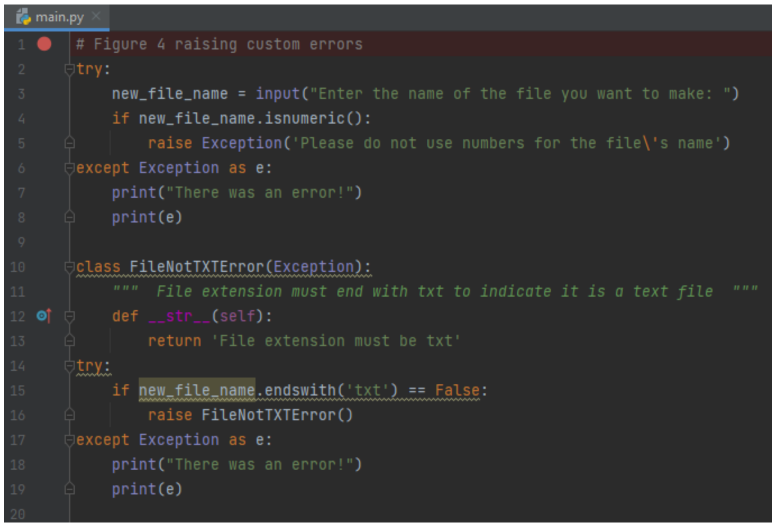
```python
1  # Figure 3 structured error-handling using try-except
2  try:
3      quotient = 5/0
4      print(quotient)
5  except Exception as e:
6      print ("There was an error!") #This is a custom message when an error occurs
7      print("Here is pythons built in error message: ")
8      print(e)
9      print("Here is my custom error message: ")
10     print("You cannot divide by 0, please enter a different second number")
```



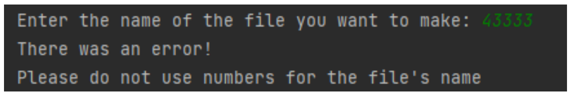
```
C:\_PythonClass\ass7figures\venv\Scripts\python.exe C:/_PythonC
There was an error!
Here is pythons built in error message:
division by zero
Here is my custom error message:
You cannot divide by 0, please enter a different second number
```

Figure 3. How to use try-except for structured error-handling. The top image shows the script, with the error occurring in line 3. Line 5 shows how to use an exception object, e, to show pythons built in error message. The bottom image shows the output when the script is run.

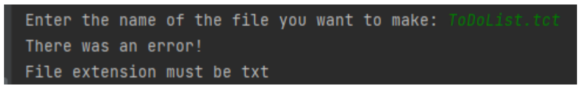
Python has many built in exception classes, such as a file not existing, or the zero-division error mentioned above. You can also raise your own custom errors if there is not a built-in one or if you do not like the way the built-in one works. An example of raising custom errors can be seen in Figure 4, below.



```python
# Figure 4 raising custom errors
try:
    new_file_name = input("Enter the name of the file you want to make: ")
    if new_file_name.isnumeric():
        raise Exception('Please do not use numbers for the file\'s name')
except Exception as e:
    print("There was an error!")
    print(e)

class FileNotTXTError(Exception):
    """  File extension must end with txt to indicate it is a text file   """
    def __str__(self):
        return 'File extension must be txt'
try:
    if new_file_name.endswith('txt') == False:
        raise FileNotTXTError()
except Exception as e:
    print("There was an error!")
    print(e)
```



```
Enter the name of the file you want to make: 43333
There was an error!
Please do not use numbers for the file's name
```

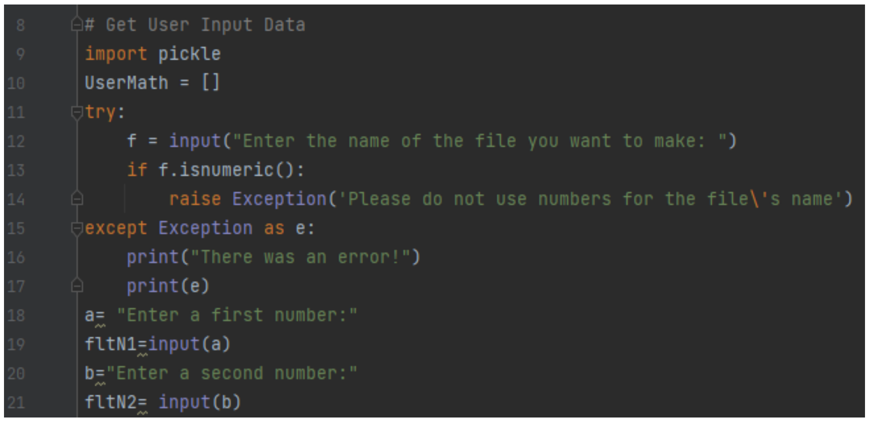


```
Enter the name of the file you want to make: ToDoList.tct
There was an error!
File extension must be txt
```

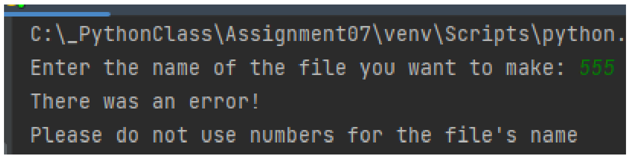
Figure 4. How to raise custom error messages. The top image shows the script for raising custom error messages. The middle image shows the error message when a file name is numeric. The bottom image shows the error message when the file does not end in .txt.

Using Pickling and Structured Error-Handling in a Script

Now that we know more about pickling and structured error-handling, I have designed a script to demonstrate how they work in a complete program. My program will do basic math calculations, based of numbers the user inputs, and then save them to a binary file that is named by the user. The first code block in this script focuses on gaining user input for the file name and numbers to do math with - this code block can be seen in Figure 5.



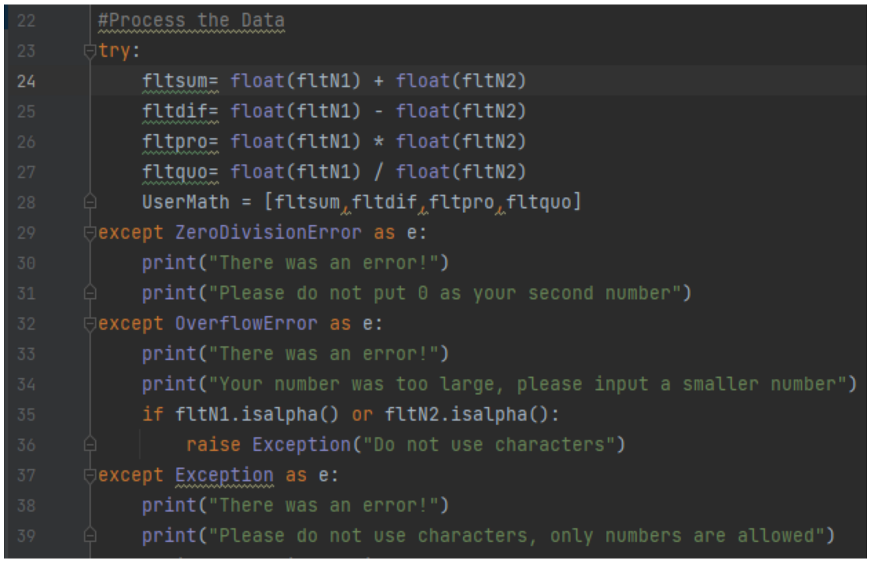
```
8   # Get User Input Data
9   import pickle
10  UserMath = []
11  try:
12      f = input("Enter the name of the file you want to make: ")
13      if f.isnumeric():
14          raise Exception('Please do not use numbers for the file\'s name')
15  except Exception as e:
16      print("There was an error!")
17      print(e)
18  a= "Enter a first number:"
19  fltN1=input(a)
20  b="Enter a second number:"
21  fltN2= input(b)
```



```
C:\_PythonClass\Assignment07\venv\Scripts\python.
Enter the name of the file you want to make: 555
There was an error!
Please do not use numbers for the file's name
```

Figure 5. The top image shows the code block to obtain user input data. Notice line 9 has the vital import pickle code. Line 11-17 deals with gaining the file name and using structured error handling to make sure that the file has an alphabet character name, not numeric. Lines 18-21 get the user input numbers to perform mathematical equations on. The bottom image shows what happens if the user inputs a file name with numbers, causing the try except to run and outputting our custom error message, "Please do not use numbers for the files name".

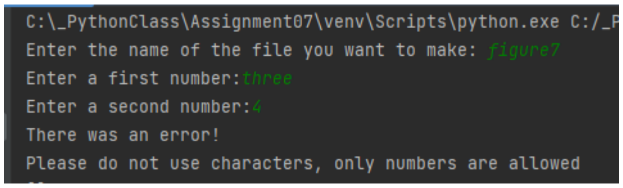
The next code block, processes the mathematical data to perform simple calculations, including 3 try-excepts. Two of the try-excepts are built-in: ZeroDivisionError and OverflowError. ZeroDivisionError happens when you try to divide a number by 0. OverflowError happens when the number entered is too large and the computer cannot perform calculations with it. The third custom try-except is that you cannot use alphabetical characters, only numbers. An example of the code block can be found in Figure 6, on the next page.



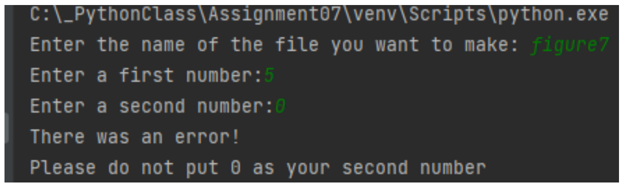
```
22      #Process the Data
23      try:
24          fltsum= float(fltN1) + float(fltN2)
25          fltdif= float(fltN1) - float(fltN2)
26          fltpro= float(fltN1) * float(fltN2)
27          fltquo= float(fltN1) / float(fltN2)
28          UserMath = [fltsum,fltdif,fltpro,fltquo]
29      except ZeroDivisionError as e:
30          print("There was an error!")
31          print("Please do not put 0 as your second number")
32      except OverflowError as e:
33          print("There was an error!")
34          print("Your number was too large, please input a smaller number")
35          if fltN1.isalpha() or fltN2.isalpha():
36              raise Exception("Do not use characters")
37      except Exception as e:
38          print("There was an error!")
39          print("Please do not use characters, only numbers are allowed")
```

Figure 6. The code block to do basic calculations, including three try-except statements. The first try-except, ZeroDivisionError, can be seen in lines 29-31. The second try-except, OverflowError, can be seen in lines 31-34. The custom try-except, for when the input is alphabetical, can be seen in lines 35-39.

As an example, I will also show how this code block runs and what happens when the user tries to enter an alphabetical input or a 0 as the second input. This can be seen below in Figure 7.



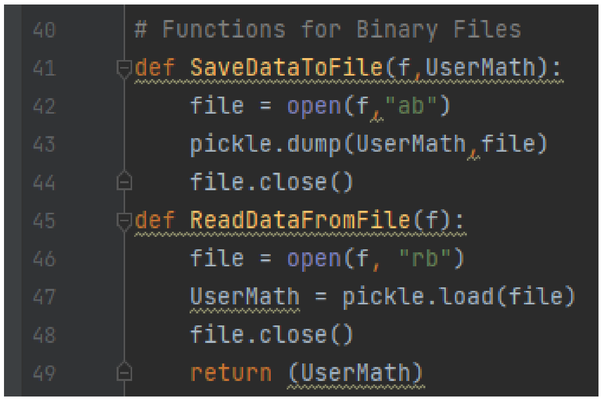
```
C:\_PythonClass\Assignment07\venv\Scripts\python.exe C:/_P
Enter the name of the file you want to make: figure7
Enter a first number:three
Enter a second number:4
There was an error!
Please do not use characters, only numbers are allowed
```



```
C:\_PythonClass\Assignment07\venv\Scripts\python.exe
Enter the name of the file you want to make: figure7
Enter a first number:5
Enter a second number:0
There was an error!
Please do not put 0 as your second number
```

Figure 7. The top image shows the error message that is presented to the user when they use alphabetical characters instead of numerical. The bottom image shows the error message when the user tries to divide by 0.

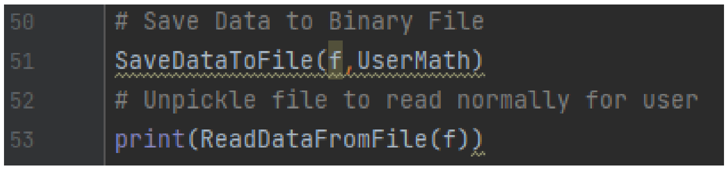
The next code block creates the functions to save data to a binary file and to read data from that binary file. This is done using the pickle.dump() and pickle.load() functions. The code block can be found in Figure 8, on the following page.



```
40  # Functions for Binary Files
41  def SaveDataToFile(f,UserMath):
42      file = open(f,"ab")
43      pickle.dump(UserMath,file)
44      file.close()
45  def ReadDataFromFile(f):
46      file = open(f, "rb")
47      UserMath = pickle.load(file)
48      file.close()
49      return (UserMath)
```

Figure 8. Functions for creating and saving to a binary file (lines 41-44) and reading data from that file and displaying it to the user (line 45-49). Notice the "ab" in line 42 and the "rb" in line 46, which tell us this is binary and stored in bytes.

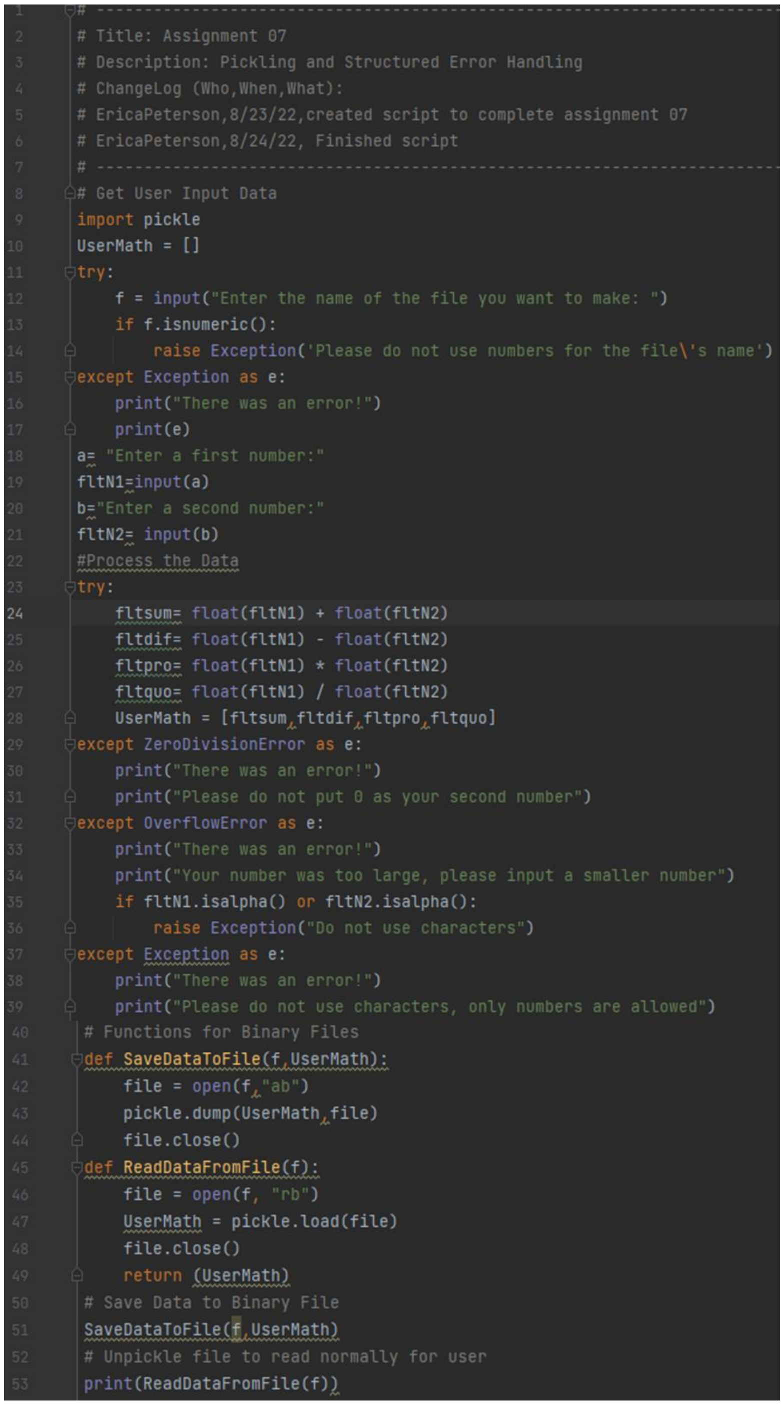
The final part of this code is to run the above-mentioned functions and present the saved data to the user. These two code blocks can be seen in Figure 9.



```
50  # Save Data to Binary File
51  SaveDataToFile(f,UserMath)
52  # Unpickle file to read normally for user
53  print(ReadDataFromFile(f))
```

Figure 9. How to save the data to a binary file and then read it out for the user to see.

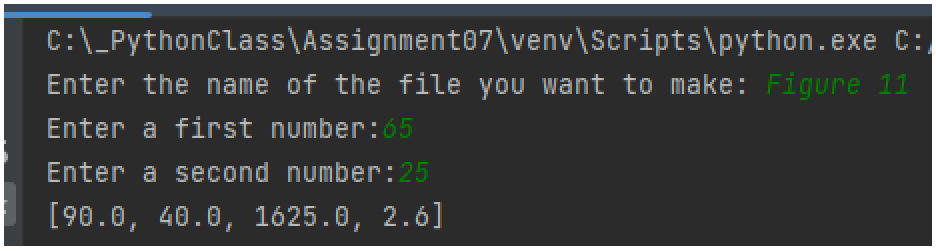
Now that we have seen every individual part of the code block, we can put it all together and see how it runs in both PyCharm and Command Prompt. First, lets look at the entire script in Figure 10, on the following page. in Figure 11, we can look at how the code runs in PyCharm, with no errors, including an image of the file that was created.



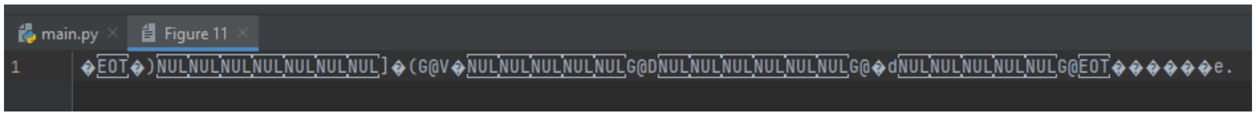
```python
# ---------------------------------------------------------------------------
# Title: Assignment 07
# Description: Pickling and Structured Error Handling
# ChangeLog (Who,When,What):
# EricaPeterson,8/23/22,created script to complete assignment 07
# EricaPeterson,8/24/22, Finished script
# ---------------------------------------------------------------------------
# Get User Input Data
import pickle
UserMath = []
try:
    f = input("Enter the name of the file you want to make: ")
    if f.isnumeric():
        raise Exception('Please do not use numbers for the file\'s name')
except Exception as e:
    print("There was an error!")
    print(e)
a= "Enter a first number:"
fltN1=input(a)
b="Enter a second number:"
fltN2= input(b)
#Process the Data
try:
    fltsum= float(fltN1) + float(fltN2)
    fltdif= float(fltN1) - float(fltN2)
    fltpro= float(fltN1) * float(fltN2)
    fltquo= float(fltN1) / float(fltN2)
    UserMath = [fltsum,fltdif,fltpro,fltquo]
except ZeroDivisionError as e:
    print("There was an error!")
    print("Please do not put 0 as your second number")
except OverflowError as e:
    print("There was an error!")
    print("Your number was too large, please input a smaller number")
    if fltN1.isalpha() or fltN2.isalpha():
        raise Exception("Do not use characters")
except Exception as e:
    print("There was an error!")
    print("Please do not use characters, only numbers are allowed")
# Functions for Binary Files
def SaveDataToFile(f,UserMath):
    file = open(f,"ab")
    pickle.dump(UserMath,file)
    file.close()
def ReadDataFromFile(f):
    file = open(f, "rb")
    UserMath = pickle.load(file)
    file.close()
    return (UserMath)
# Save Data to Binary File
SaveDataToFile(f,UserMath)
# Unpickle file to read normally for user
print(ReadDataFromFile(f))
```

Figure 10. The completed script to do basic math and save it in a binary file, including structured error-handling.

In Figure 11, on the following page, we can look at how the code runs in PyCharm, with no errors, including an image of the binary file that was created.



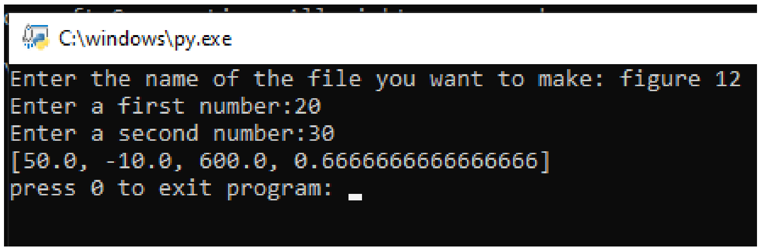
```
C:\_PythonClass\Assignment07\venv\Scripts\python.exe C:/
Enter the name of the file you want to make: Figure 11
Enter a first number:65
Enter a second number:25
[90.0, 40.0, 1625.0, 2.6]
```



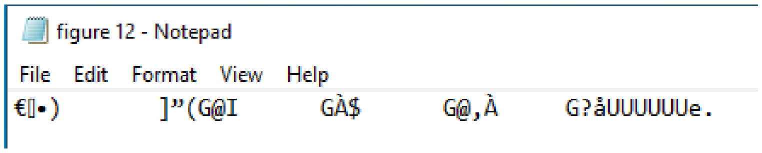
```
�EOT�)NULNULNULNULNULNULNUL]�(G@V�NULNULNULNULNULG@DNULNULNULNULNULNULG@�dNULNULNULNULNULG@EOT�������e.
```

Figure 11. The script running in PyCharm is seen in the top image, including user input in green. The bottom image shows the binary file, named Figure 11, as inputted.

Next, we can confirm that this script also runs in Command Prompt, with no user errors. This can be seen in Figure 12. Also in Figure 12 is the file created when the script is run in Command Prompt.



```
Enter the name of the file you want to make: figure 12
Enter a first number:20
Enter a second number:30
[50.0, -10.0, 600.0, 0.6666666666666666]
press 0 to exit program: _
```



```
€•)        ]”(G@I       GÀ$        G@,À       G?åUUUUUUe.
```

Figure 12. The completed script running in command prompt is seen in the top image. The bottom image shows the binary file created by the script, including the correct name for the file, figure 12.

Summary

In this paper I went over pickling and structured error-handling and then combined these ideas into a program that showcased what they do. Structured error-handling will be very important in future projects to make them more user friendly, especially as the script gets more complex. Pickling is also useful for sharing files across programs.