Chrissy Henderson

November 27, 2020

IT FDN 110 A Au 20: Foundations Of Programming: Python

Assignment 07

https://github.com/chrissyhend1/IntroToProg-Python-Mod07

Demonstrating Pickling and Error Handling

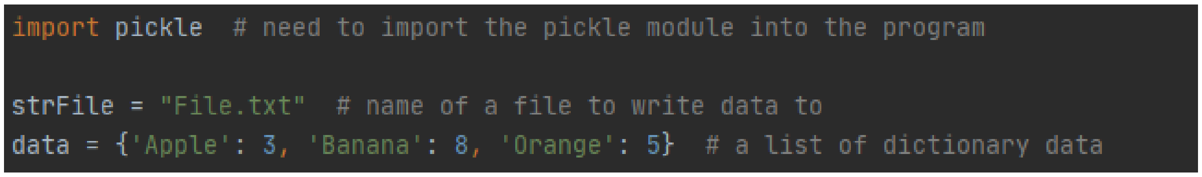
Introduction

Pickling is a way to convert data to and from binary objects to make the data "sterile". This data is not entirely secure, however, and should not be treated as such, but it does make the data more difficult to read using the naked eye. This process also makes files smaller therefor making them easier to transfer over a network or store in memory. This pickling process is unique to Python and cannot be used with other programming languages.

Errors are an expected issue when running any kind of code in Python. Error handling can be used to present custom messages when certain errors are encountered and give a user a more descriptive and more simplified reason why the code failed. Multiple "try-except" blocks can be used to catch multiple different errors and present different messages to the users.

Pickling in Python

There are four main ways to pickle and unpickle data including "dump", "dumps", "load", and "loads". In this example, we will first need to import the pickle program, set up a file name, and set up a new line of dictionary data to work with (Figure 1).

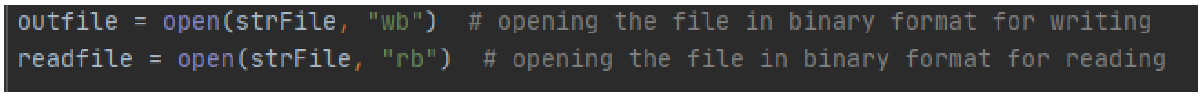


```python
import pickle  # need to import the pickle module into the program

strFile = "File.txt"  # name of a file to write data to
data = {'Apple': 3, 'Banana': 8, 'Orange': 5}  # a list of dictionary data
```

Figure 1. Importing pickle into Python and setting up the file name and list of dictionary data.

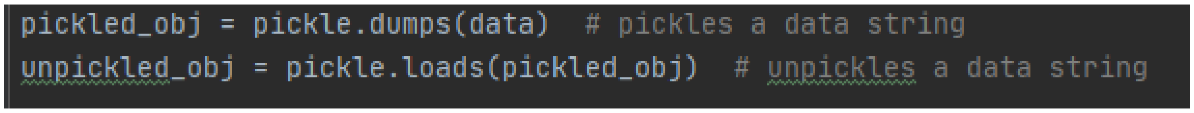
The second step is to create two lines that will open files in two different modes. We want to be able to open a file in "write" mode which usually involves adding a "w" at the end of the open string. However, because we will be working in binary, we will need to open the file in "wb" mode with the "b" standing for binary. The second line is similar but will involve putting "rb" at the end of the line to open and read the file in binary mode (Figure 2).



```
outfile = open(strFile, "wb")  # opening the file in binary format for writing
readfile = open(strFile, "rb")  # opening the file in binary format for reading
```

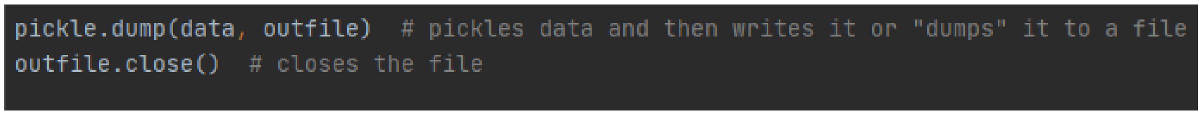
Figure 2. Opening the file in binary write mode and binary read mode respectively

"Dumps" and "loads" works with each other in that they pickle and unpickle strings of data as indicated by the "s" at the end of the words (Figure 3). This is a simple process, but these two functions rely on each other. A user will not be able to work with strings that have been pickled like they can with unpickled strings. This data needs to be unpickled first. "Dump" and "load" are very similar in that they involve pickling and unpickling data but the process it done to and from text file outside of python (Figure 4 & 5).



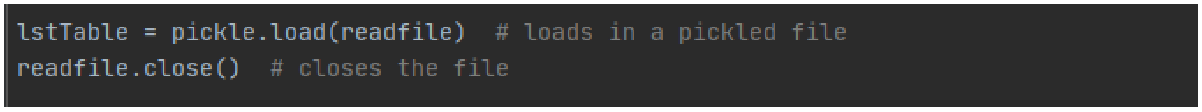
```
pickled_obj = pickle.dumps(data)  # pickles a data string
unpickled_obj = pickle.loads(pickled_obj)  # unpickles a data string
```

Figure 3. Pickling and unpickling a string of data



```
pickle.dump(data, outfile)  # pickles data and then writes it or "dumps" it to a file
outfile.close()  # closes the file
```

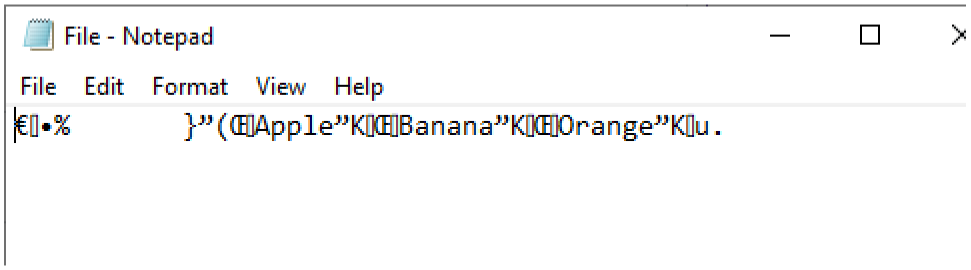
Figure 4. Pickling data and saving it to a file in "wb"mode



```
lstTable = pickle.load(readfile)  # loads in a pickled file
readfile.close()  # closes the file
```

Figure 5. Unpickling data from a file where the file is in "rb" mode

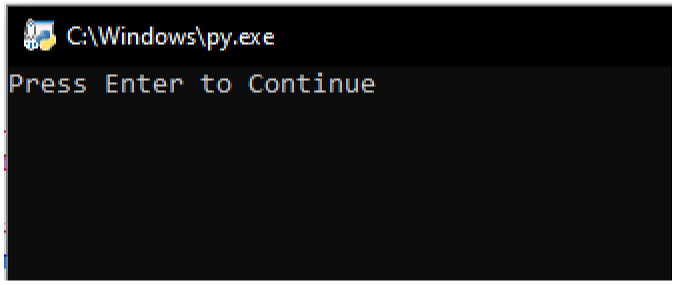
As you can see from the text file, the code successfully pickled the data into a text file (Figure 6).



```
€•%        }”(Œ Apple”K Œ Banana”K Œ Orange”K u.
```

Figure 6. The pickled text file

The code successfully ran in Windows command shell because it pickled and unpickled that data without any errors and stopped at the input step I programmed in (Figure 7).



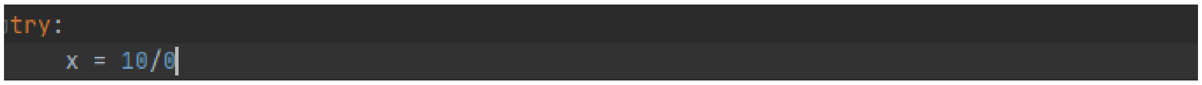
```
C:\Windows\py.exe
Press Enter to Continue
```

Figure 7. Code successfully ran in Windows command shell

Error Handling in Python

Error handling is a very useful technique in Python to help users better understand why the error is occuring and simplifying the messaging presented to the user. This is done in what is called a "try-except" block. The program will "try" the code, and if the program encounters an error, then the program will cycle through the "except" portion of the code. In this exercise, I explore several different errors that are built into Python as well as how to create a custom error.

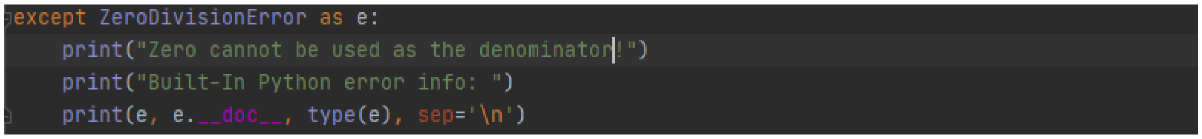
The first error we will be looking at is the ZeroDivisionError. This is a built in error that occurs when a number is attempting to be divided by 0. For example, we will "try" to set "x" to be equal to 10 divided by 0 (Figure 8).



```
try:
    x = 10/0
```

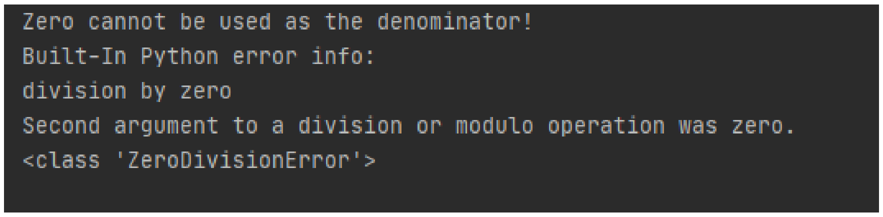
Figure 8. Setting up "x" to equal 10 divided by 0 in the try portion of the program

After this is tried by the program, the program will hit an error and therefor run through the except portion. When it hits the correct except portion (which in this case is for the ZeroDivisionError), the program will run through the code displayed in Figure 9. First, the program will save the error as the variable "e". Then the program will print the custom message that zero cannot be the denominator and then follow this up with the built-in Python error information that had been saved as "e". The third print line will print the Python error, the documentation for said error, the type of the error, and separate these each by a new line (Figure 10 & 11).



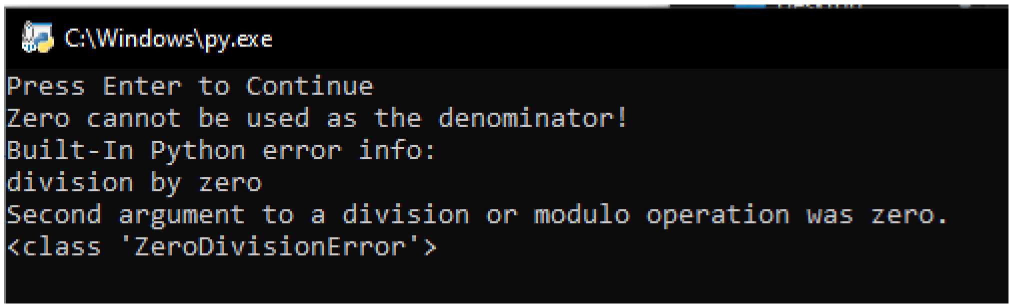
```
except ZeroDivisionError as e:
    print("Zero cannot be used as the denominator!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
```

Figure 9. The except code block for the error ZeroDivisionError



```
Zero cannot be used as the denominator!
Built-In Python error info:
division by zero
Second argument to a division or modulo operation was zero.
<class 'ZeroDivisionError'>
```

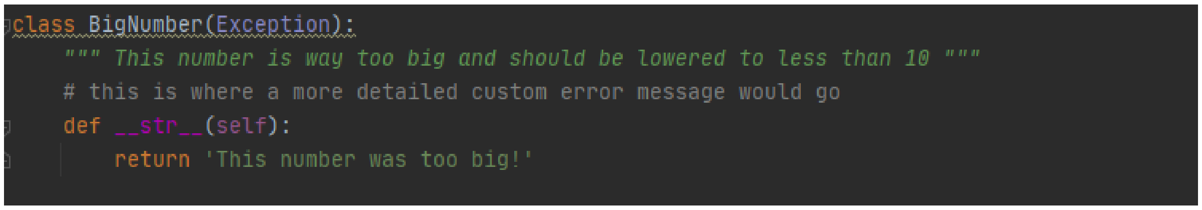
Figure 10. The output in PyCharm IDE from the program after a ZeroDivisionError has occured



```
C:\Windows\py.exe
Press Enter to Continue
Zero cannot be used as the denominator!
Built-In Python error info:
division by zero
Second argument to a division or modulo operation was zero.
<class 'ZeroDivisionError'>
```

Figure 11. The error message as seen in the Windows command shell

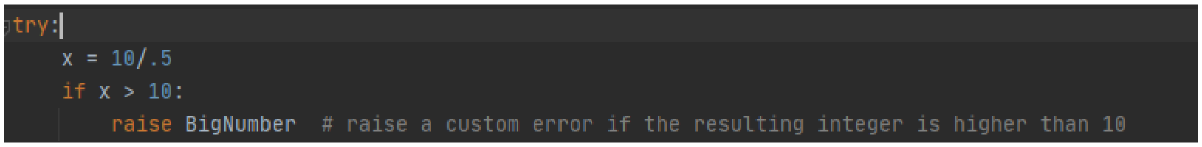
The second error I have set up is a custom error. As long as the error is set up as an exception before running the "try-except" block, the program will be able to run this custom error (Figure 12).



```
class BigNumber(Exception):
    """ This number is way too big and should be lowered to less than 10 """
    # this is where a more detailed custom error message would go
    def __str__(self):
        return 'This number was too big!'
```

Figure 12. The class code block used to define the custom "BigNumber" error

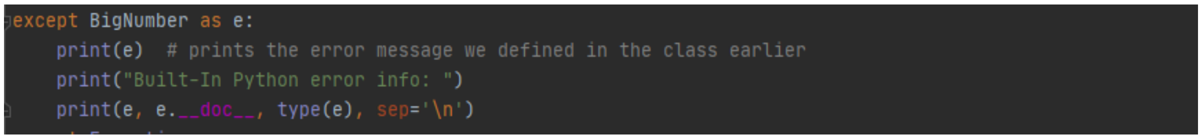
This custom error is set up to run if x is greater than 10. To do this, I made it so when x was created, it would equal 20 and therefor trigger the error. Python would not normally hit an error for this process, so I needed to add a line that would "raise" the exception if this certain requirement was met. Additionally, this piece needs to be in the "try" portion of the block or else it will not error out (Figure 13).



```
try:
    x = 10/.5
    if x > 10:
        raise BigNumber  # raise a custom error if the resulting integer is higher than 10
```

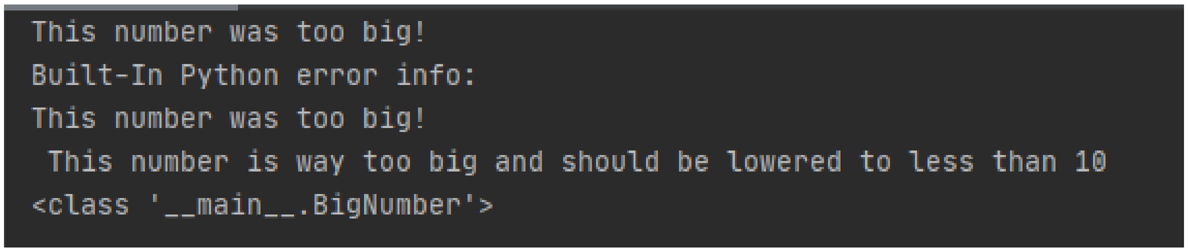
Figure 13. The try portion of the program that will raise the BigNumber error if x is greater than 10

When the exception "BigNumber" is triggered, the "try-except" block will stop at the "except BigNumber" line, store the error as "e", and print out the error that was defined in the custom exception creation earlier (Figure 9). When printing out the "doc" amd "type" as done for the previous error, Python will only print what I defined in the description of the error myself (Figure 14 & 15).



```
except BigNumber as e:
    print(e)  # prints the error message we defined in the class earlier
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
```

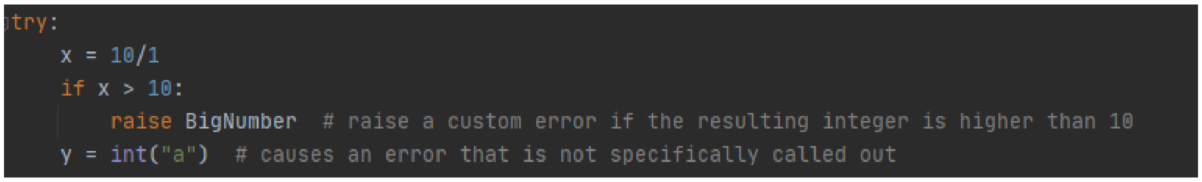
Figure 14. The except code block for the BigNumber error



```
This number was too big!
Built-In Python error info:
This number was too big!
 This number is way too big and should be lowered to less than 10
<class '__main__.BigNumber'>
```

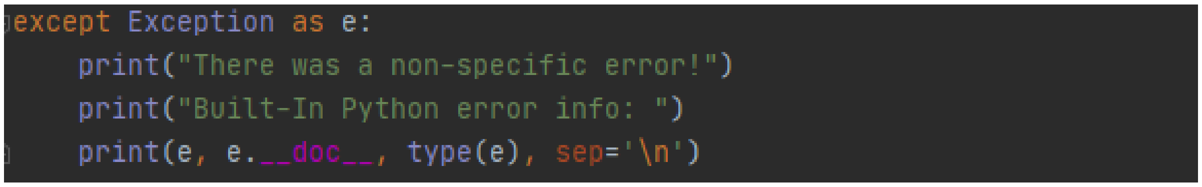
Figure 15. The output in PyCharm IDE from the program after the custom error BigNumber has occured

The next error we will explore is an all encompassing error. If an error occurs in the "try" portion and does not fall into any of the other exceptions, this error will be thrown. I have fixed the previous statements to not cause errors before reaching this line as well. In this case, I told the program to try to convert the letter "a" to an integer which is not possible (Figure 16). Since this error does not fit the other three errors that I defined, it called the last one on the list. The except block will print that the error is non-specific and then proceed to print out the built-in error information (Figure 17 & 18).



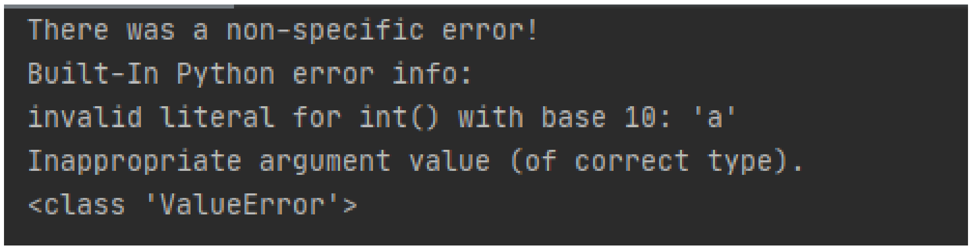
```
try:
    x = 10/1
    if x > 10:
        raise BigNumber  # raise a custom error if the resulting integer is higher than 10
    y = int("a")  # causes an error that is not specifically called out
```

Figure 16. The try portion of the code that will cause an error if "a" is converted to an integer



```
except Exception as e:
    print("There was a non-specific error!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
```

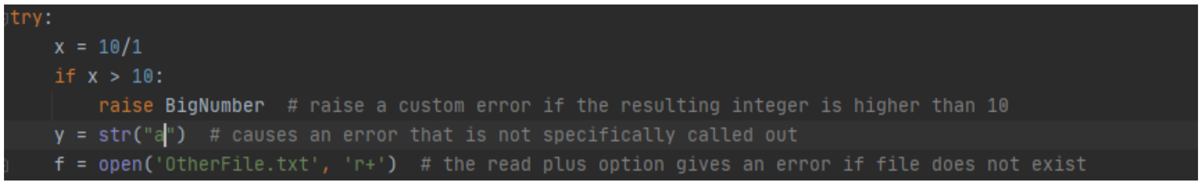
Figure 17. The except code block for the non-specific error



```
There was a non-specific error!
Built-In Python error info:
invalid literal for int() with base 10: 'a'
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

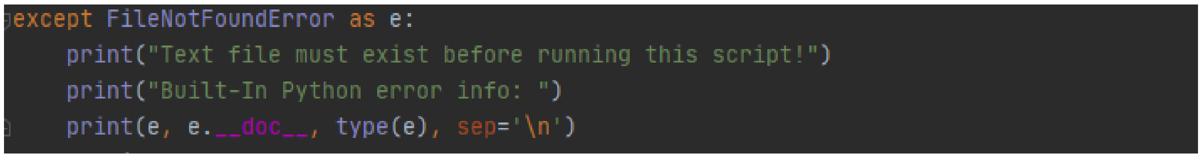
Figure 18. The output in PyCharm IDE from the program after a ValueError has occured

The final error that I will demonstrate is the "FileNotFoundError" error. This error will be thrown if a file is opened in the program, but it does not exist. In this example, I told the program to open a file that does not exist (Figure 19). If this error is raised, the program will print the message that the file must exist before running the script and then proceed to print out the built in Python error information (Figure 20 & 21).



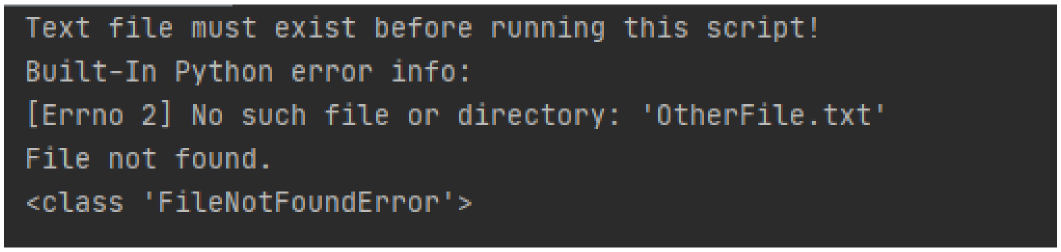
```
try:
    x = 10/1
    if x > 10:
        raise BigNumber  # raise a custom error if the resulting integer is higher than 10
    y = str("a")  # causes an error that is not specifically called out
    f = open('OtherFile.txt', 'r+')  # the read plus option gives an error if file does not exist
```

Figure 19. The try portion of the code that will raise a FileNotFoundError if there is an attempt to open a nonexistent file



```
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')
```

Figure 20. The except code block for the FileNotFoundError



```
Text file must exist before running this script!
Built-In Python error info:
[Errno 2] No such file or directory: 'OtherFile.txt'
File not found.
<class 'FileNotFoundError'>
```

Figure 21. The output in PyCharm IDE from the program after a FileNotFound has occured

Summary

Pickling can be a great method for adding a little data security to your data and also saving some space in either memory or in transfering data over a network. Additionally error handling can be an amazing tool for simplifying built-in Python errors and presenting them to users or creating your own error messages if certain specific guidelines need to be followed.