Pavel (Pasha) Beresnev

12/1/20

IT FDN 110A

Assignment 07

# How to use structured error handling and pickling in Python

## Introduction:

In this module we will discuss how to use structured error handling in Python, along with Pickling. I will demonstrate these concepts in a script and go over several error handling scenarios. Finally, I’ll show how this script runs in the command line as well as PyCharm.

## Structured Error Handling using Existing Exceptions:

Python has several ways to handle errors that can be controlled by the coder rather than using the default error messages. A way to do this is using a try/except block. The way this works is we write the statement that we think could draw an error using “try”, followed by the statement. Then “except” is used to write the code that runs if the “try” block generates an error. We can then write an “else” block that has code which will execute if there is no error, as shown in Fig 1:

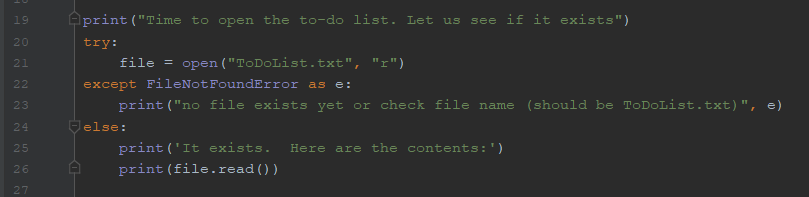


Figure : Try/Except/Else Block with FileNotFoundError

In this case I print to the user that we will try to open a to-do list, as shown in the “try” block. Since I am not sure if the file exists, I write the “except” block to catch that error using the variable “e”. Note, here I specifically call out the error type that will be caught, though I could just write “Exception” to catch any error. Then if the called-out error happens, I print out that no file exists along with the Python default error message stored in the variable. If there is no error, I simply print that it exists and the contents of the file in the “else” block using the “read” method.

## Raising Custom Errors:

Another option in Python is to raise custom error messages rather than using Python’s default messages. To demonstrate this, I prompt the user to enter a number using “input” in a “try” block. If that input contains a letter, I raise a custom exception to not enter letters. Because this is raised, the “except” block will run, where I catch the error as a variable, “e”, and print it out as shown in Fig 2:

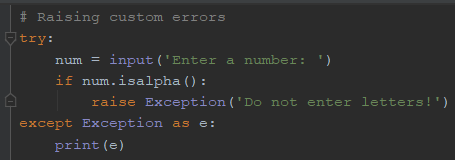


Figure : Raising custom error

## Raising custom Exception class:

Finally, Python allows the developer to also create a custom exception class rather than using the default ones. I demonstrate this again using a try/except block, but this time I also create a custom class. Once again, I use the example of asking the user for a number with “input” in the “try” block. I check if that entry has letters and if so, I raise an error. This time, however, I created a custom Exception class called “NumError”. When the error is raised the “except” block once again will run where I capture the Exception as “e”. In this case I also print “e” where the custom error is shown, as demonstrated in Fig 3-4:

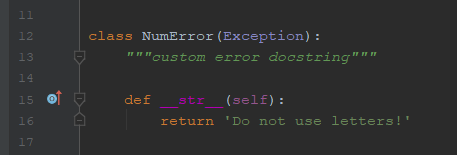


Figure : Custom Exception Class

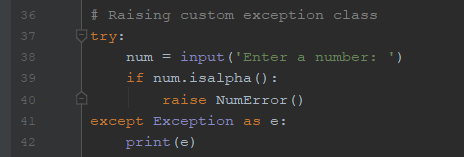


Figure : Raising Custom Exception Class

## Pickling

The next topic is pickling. This is a way to store data in Python using various objects, such as dictionaries, lists, or tuples, rather than first converting those to a string and then writing the string to a text file, like we’ve seen before. To do this, first we import the “pickle” module at the start of the script as shown in Fig 5:

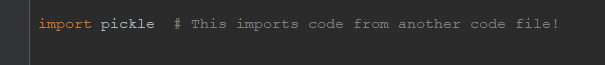


Figure : Import pickle

To demonstrate this, I’ll first use a file name, “AppData.dat” and create a blank list called “lstCustomer” in the Data section of the code. Then in the processing section I create two functions, one to save data to the file and another to read data from the file. The save data function takes a file name and a list of data. Then I create an “objFile” variable that stores the created/opened binary file in “append” mode, indicated by “ab” in the “open” method. Then I use the “dump” method within “pickle” to add the list of data to the file. Then I close the file.

To read data from the file my function takes a file name. Then I create a variable again to open the file, this time in “rb” mode, allowing reading of a binary file. Then I assign a variable to the data that is loaded from the file, using the pickle.load() method. Finally, I close the file and print the data, as shown in Fig 6:

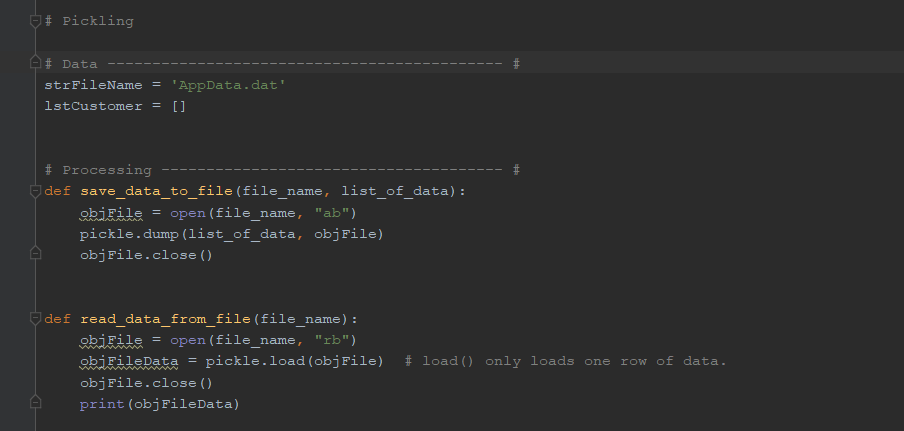


Figure : Pickling data/processing example

In the presentation code I create two variables to get an integer ID and Name string using “input. I then replace the blank list from before with these two inputs. Then I call the “save\_data\_to\_file” function to save the list to the binary file we made. After this I read data from the file using the second function created, as shown in Fig 7:

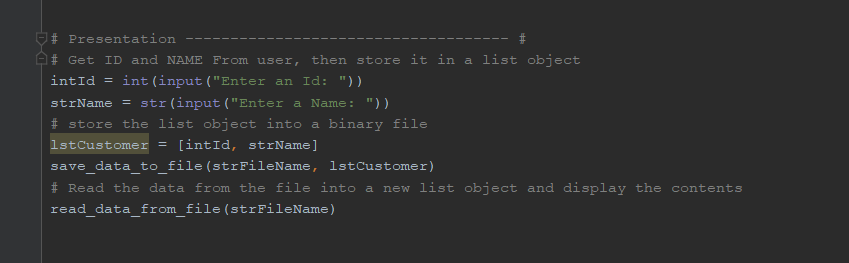


Figure : Presentation code for pickling example

## Output:

Fig 8 demonstrates the code running in PyCharm. First I run it where no text file exists in the folder, hence I get the error messages described. I then demonstrate that entering a letter on both prompts generates the custom errors described. Finally, I create the binary file as described, shown in Fig 9:

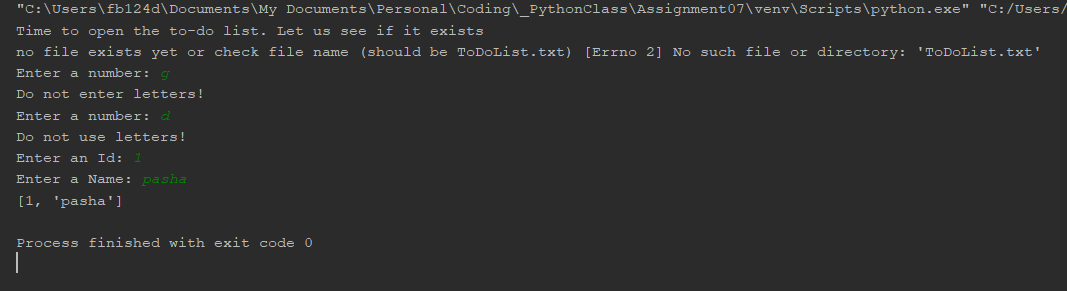


Figure : PyCharm Run with No Existing Text File

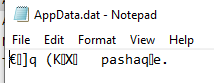


Figure : Binary File from PyCharm Script

Next I demonstrate what this looks like in PyCharm if there is an existing Text file, which runs as described. See Fig 10:

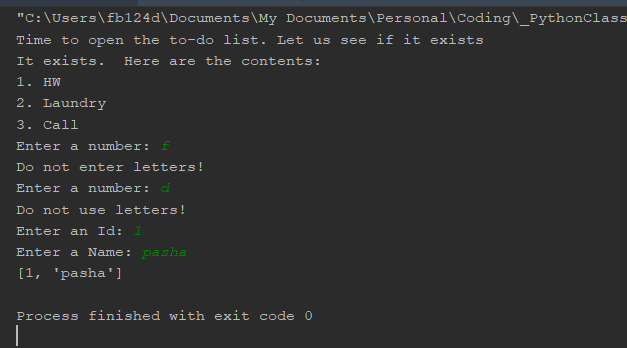


Figure : PyCharm Run with Existing Text File

Fig 11 shows a similar run to the first run but in the Command Prompt:

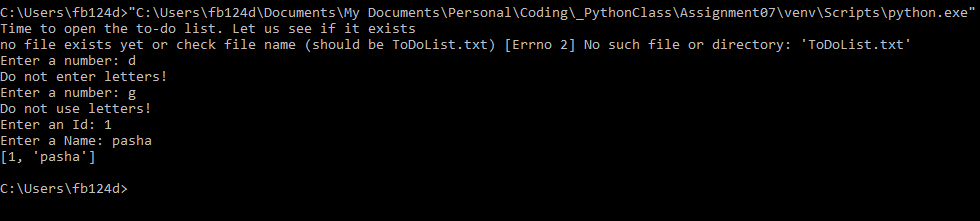


Figure : Command Prompt Run with no Existing Text File

Fig 12 shows what the resulting binary file looks like:

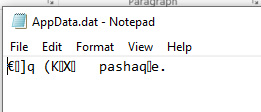


Figure : Resulting Binary File from Command Prompt Run

Finally, Fig 13 demonstrates the Command Prompt run but with an existing text file:

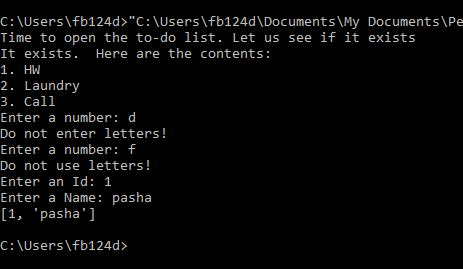


Figure : Command Line Run with Existing Text File

## Conclusion:

In this module we learned how to perform structured error handling using try/except and raising existing errors, custom errors or custom Exceptions. This allows a developer to handle different scenarios when running a script. Finally, we also looked at how to use pickling to save data from Python in the existing format, whether it’s a list, dictionary, tuple, etc. and then read it back in that format rather than having to convert the data to a string and write it as a text file.