Sergey Timchenko

2/27/2021

Foundations Of Programming: Python

Assignment 07

Files and Exceptions

# Introduction

In this assignment the student learns how to read and write data from the text file, read and write data using pickling and binary files, intercept and handle errors during program execution. The student modifies the program created for Assignment 06 and added errors catching functionality and use of binary CDInventory.dat file.

Reading data from a text file

It is possible to load the data that will be processed in the program from the text file. To open file located in location different from location of the program file it is necessary to specify the file path[[1]](#footnote-1).

The *open()* function returns a file object, which has a *read()* method for reading the content of the file:

f = open("demofile.txt", "r")  
print(f.read())

The data from file could be extracted by line or by lines

f = open("demofile.txt", "r")  
print(f.readline())

f = open("demofile.txt", "r")  
print(f.readlines())

It is possible to loop directly thru the lines of a file using a *for* loop

f = open("demofile.txt", "r")  
for line in f:

print(line)

Writing to a text file

For writing the file we use ‘w’ option. Here is the example of writing the list to a file

f = open("demofile.txt", "w")  
lstTbl = [

{'ID': 1, 'Title': 'Bad', 'Artist': 'Michael Jackson'},

{'ID': 2, 'Title': 'The Big Wheel', 'Artist': 'Rurig'}

]

f.writelines(lstTbl)

f.close()

Here are the several options handling the file[[2]](#footnote-2)

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

# Working with the binary files

To work with binary files there is the *pickle* module in python. The *pickle*module takes the information associated with an object and serializes (or de-serializes) it in a way that it can easily be stored (or loaded) as binary information. It allows to dump data from the memory into the binary file and extracting the data to the program. Human is not able to read data in binary file but in doesn’t mean that the data is secured, the python program can extract data from the binary file.

Here is the example of pickling the data[[3]](#footnote-3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | >>>  >>> **import** **pickle**  >>>  >>> **from** **datetime** **import** datetime  >>>  >>>  >>> f = open("my\_pickle", "wb") *# remember to open the file in binary mode*  >>>  >>> pickle.dump(10, f)  >>> pickle.dump("a string", f)  >>> pickle.dump({'a': 1, 'b': 2}, f)  >>> pickle.dump(datetime.now(), f) *# serialize datetime.datetime object*  >>>  >>> f.close()  >>>  >>> |

And the example of loading various data from a binary file

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | >>>  >>> **class** **My\_class**:  ... **def** \_\_init\_\_(self, name):  ... self.name = name  >>>  >>>  >>> **def** func(): **return** "func() called"  ...  >>>  >>>  >>> f = open("other\_pickles", "wb")  >>>  >>> pickle.dump(My\_class, f) *# serialize class object*  >>>  >>> pickle.dump(2 + 3j, f) *# serialize complex number*  >>>  >>> pickle.dump(func, f) *# serialize function object*  >>>  >>> pickle.dump(bytes([1, 2, 3, 4, 5]), f) *# serialize bytes object*  >>>  >>> pickle.dump(My\_class("name"), f) *# serialize class instance*  >>>  >>> f.close()  >>> |

Similarly to the text files there several modes accessing the binary files

"rb" - Opens the file as read-only in binary format and starts reading from the beginning of the file. While binary format can be used for different purposes, it is usually used when dealing with things like images, videos, etc.

"wb" - Opens a write-only file in binary mode.

"wb" - Opens a file for writing and reading in binary mode.

"ab" - Opens a file for appending in binary mode.

# Structured Error Handling

When there might be the errors in the program run Python stops execution and exits the program

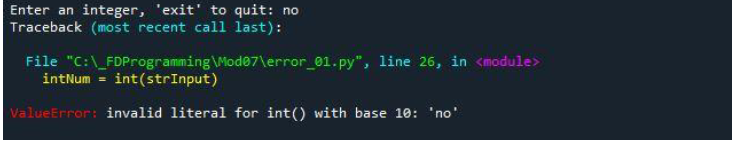


Figure 1 – Python execution error example

Using Python’s exceptions handling functionality is possible to intercept and handle exceptions so that program doesn’t end abruptly (even if a user enters “Hi!” when he is being asked for a number). At very last, we can make the program exit gracefully instead of crashing awkwardly.

One of the options to handle the errors is using *try except* combination[[4]](#footnote-4)

The *try* block will generate an exception, because *x* is not defined:

try:  
  print(x)  
except:  
  print("An exception occurred")

The exception handling can be organized in a *while* loop. The program will be asking for correct type of entry until user provides it.

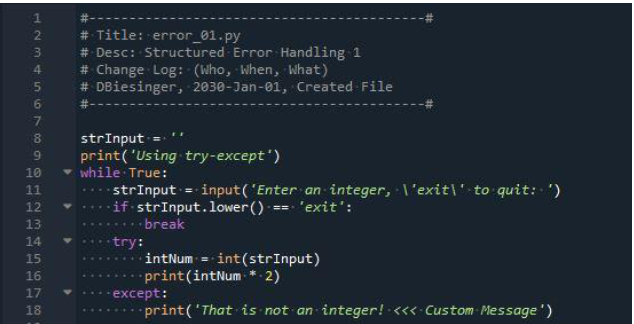


Figure 2 – While loop with try except error handling

The program example on Figure 2 doesn’t specify what kind of expiations it raises it will print “That not an integer! “ for any incorrect input. But it is possible to specify what kind of actions the program is supposed to do depending on different type of exception types.

Standard exceptions that can be raised are detailed at:

<http://docs.python.org/library/exceptions.html>

|  |  |
| --- | --- |
| Exception | Description |
| AssertionError | Raised when the assert statement fails. |
| AttributeError | Raised on the attribute assignment or reference fails. |
| EOFError | Raised when the input() function hits the end-of-file condition. |
| FloatingPointError | Raised when a floating point operation fails. |
| GeneratorExit | Raised when a generator's close() method is called. |
| ImportError | Raised when the imported module is not found. |
| IndexError | Raised when the index of a sequence is out of range. |
| KeyError | Raised when a key is not found in a dictionary. |
| KeyboardInterrupt | Raised when the user hits the interrupt key (Ctrl+c or delete). |
| MemoryError | Raised when an operation runs out of memory. |
| NameError | Raised when a variable is not found in the local or global scope. |
| NotImplementedError | Raised by abstract methods. |
| OSError | Raised when a system operation causes a system-related error. |
| OverflowError | Raised when the result of an arithmetic operation is too large to be represented. |
| ReferenceError | Raised when a weak reference proxy is used to access a garbage collected referent. |
| RuntimeError | Raised when an error does not fall under any other category. |
| StopIteration | Raised by the next() function to indicate that there is no further item to be returned by the iterator. |
| SyntaxError | Raised by the parser when a syntax error is encountered. |
| IndentationError | Raised when there is an incorrect indentation. |
| TabError | Raised when the indentation consists of inconsistent tabs and spaces. |
| SystemError | Raised when the interpreter detects internal error. |
| SystemExit | Raised by the sys.exit() function. |
| TypeError | Raised when a function or operation is applied to an object of an incorrect type. |
| UnboundLocalError | Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. |
| UnicodeError | Raised when a Unicode-related encoding or decoding error occurs. |
| UnicodeEncodeError | Raised when a Unicode-related error occurs during encoding. |
| UnicodeDecodeError | Raised when a Unicode-related error occurs during decoding. |
| UnicodeTranslateError | Raised when a Unicode-related error occurs during translation. |
| ValueError | Raised when a function gets an argument of correct type but improper value. |
| ZeroDivisionError | Raised when the second operand of a division or module operation is zero. |

Here is the example of handling multiple exceptions with a single try statement

print()

for value in (None, "Hi!"):

try:

print("Attempting to convert", value, "-->", end=" ")

print(float(value))

except TypeError:

print("I can only convert a string or a number!")

except ValueError:

print("I can only convert a string of digits!")

Using multiple except clauses allows to define unique reaction to different types of exceptions from the same *try* block.

Besides the automatically generated errors based on conditions defined by the Python, it is also possible to “raise” errors based on custom conditions.

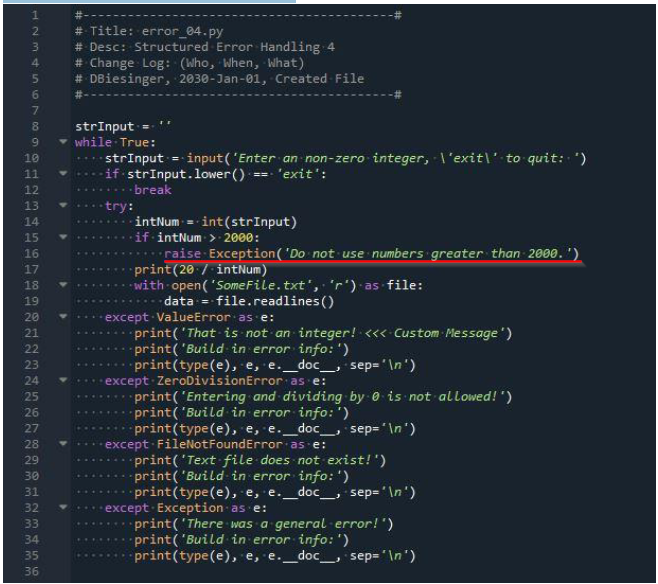


Figure 3 – Raised exceptions based on programmer conditions

# Creating a script

In this assignment I was asked to modify the Assignment 06 script by adding functionality to catch structured error in the area where the user interacts with program and use binary file to store the data instead of the text file.

The structured error handling is shown in this part of the program

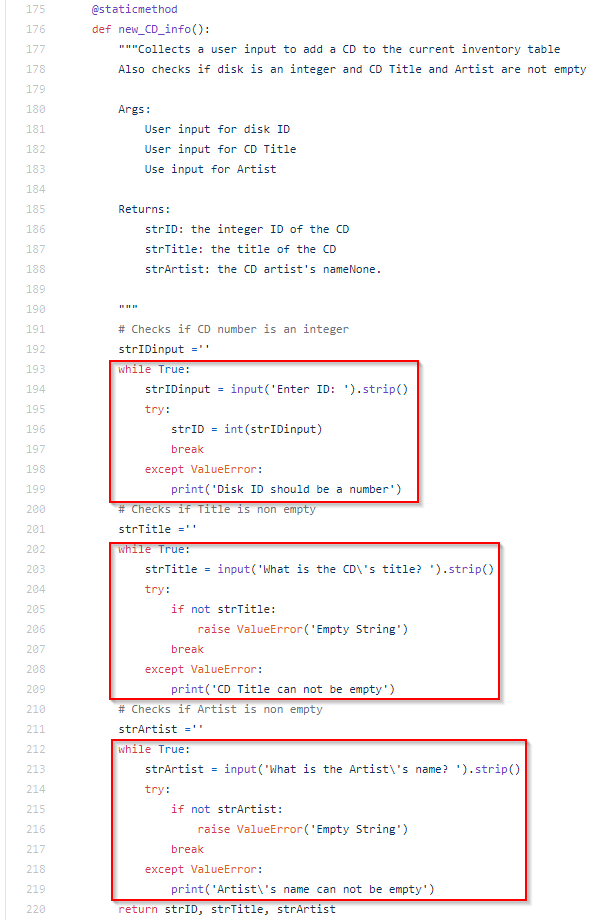


Figure 4 – Handling possible error of incorrect input

For disk ID the user should enter an integer. The program doesn’t continue execution until the integer is entered for disk ID. It the second and third part of the function I entered the code that checks if the user entered a string for CD Title and Artist.

In order to use a binary file for storing the data I renamed CDInventory.txt file into CDInventory.dat file. Also I deleted all of the data from the file.

After updating the code the program will operate with the binary file but because the file is empty the program is not able to operate with it. To fix this problem I created simple program that loads the disk data into binary file.



Figure 5 – Dumping initial data into CDInventory.dat file

The data in the file looks like this

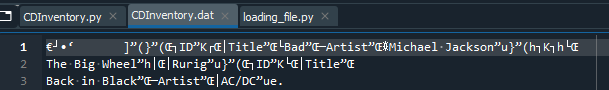


Figure 6 – The data in binary file

Using the main program we can add or delete data from CDInventory.dat file.

The block that reads the data from binary file is combined into load\_file function

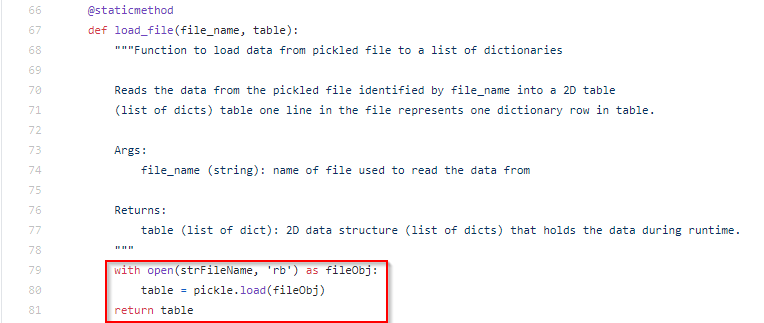


Figure 6 – loading the data from CDInventory.dat file

The updated data dumps into CDInventory.dat file by using write\_file function

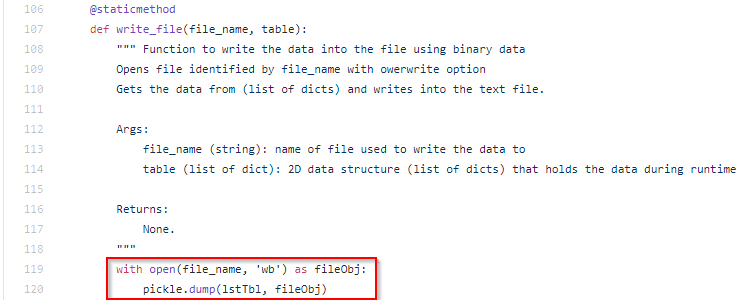


Figure 7 – loading the data from CDInventory.dat file

# The result of the program run

After the code is modified we are testing the program in Spider sub window

1. First we are loading the info from the file and displaying the inventory

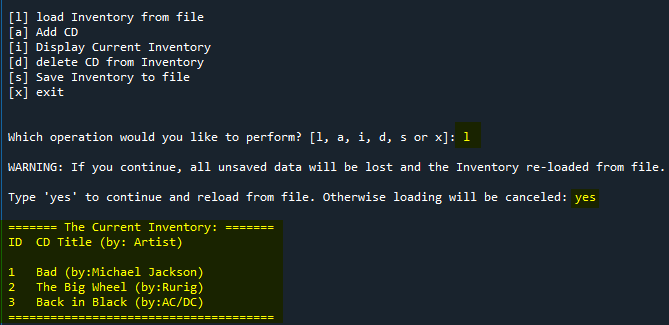


Figure 8 –Choice [l] and [i] demonstration

1. Then loading the new disk info with incorrect info

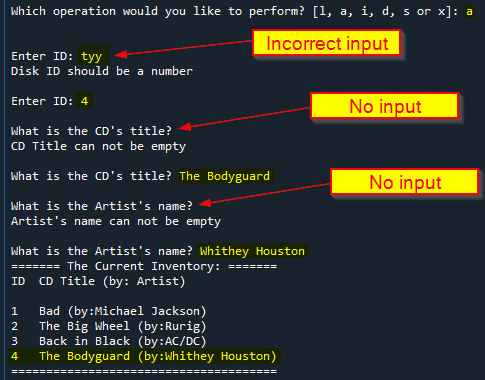


Figure 9 –Choice [a] demonstration

1. Deleting the disk from the list

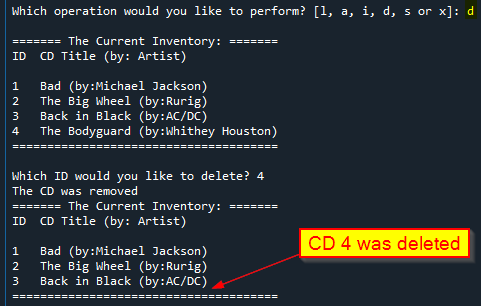


Figure 10 –Choice [d] demonstration

1. Saving the data into the binary file

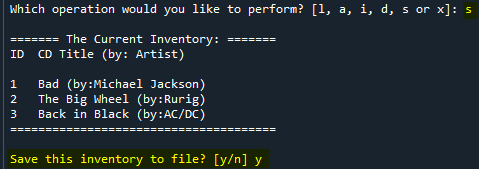


Figure 11 –Choice [s] demonstration

Now we want to check if the program runs correctly in Anaconda command line window.

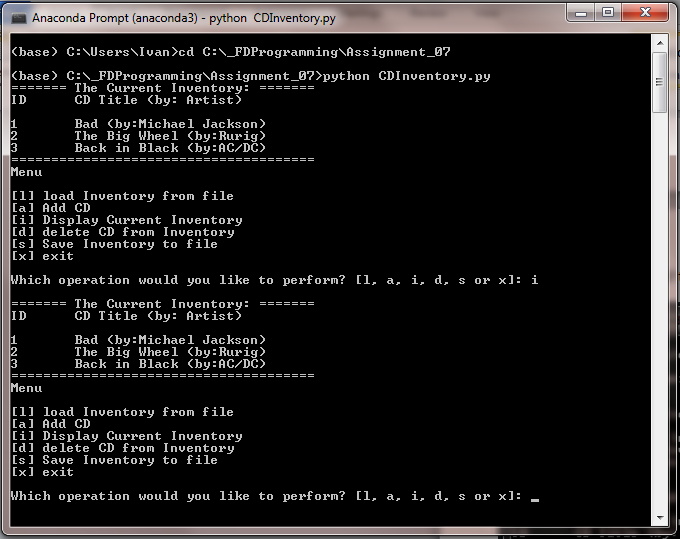


Figure 12 – Running the program in Anaconda Prompt window

The files for this Assignment were uploaded to GitHub [here](https://github.com/snt2177/Assignment_07.git).

# Summary

In this Assignment I learned about file read write operations and error handling.

To save the storage space the data from computer memory can be directly dumped into a binary file and it can be loaded back to the program for processing.

I learned how to handle exceptions raised during the execution of the program. I saw how to trap for specific exceptions and how to write code to deal with them.

1. <https://www.w3schools.com/python/python_file_open.asp> , retrieved 2021-Feb-28 [↑](#footnote-ref-1)
2. <https://www.w3schools.com/python/python_file_handling.asp>, retrieved 2021-Feb-28 [↑](#footnote-ref-2)
3. <https://thepythonguru.com/pickling-objects-in-python/>, retrieved 2021-Feb-28 [↑](#footnote-ref-3)
4. <https://www.w3schools.com/python/python_try_except.asp>, retrieved 2021-Feb-28 [↑](#footnote-ref-4)