# **Nazneen Mohammed Ameen**

# **Mar 6, 2020**

# **Programming with Python**

# **Assignment07**

**Files and Exception**

**INTRODUCTION**

While a program is running, its data is in memory. When the program ends, or the computer shuts down, data in memory disappears. To store data permanently, you have to put it in a **file**. Files are usually stored on a hard drive, floppy drive, or CD-ROM.

In this module, you'll learn how to open files for read, write or append. You also learn about text files and binary files. you will know how to handle exceptions in your Python program using try, except and finally statements. This will motivate you to write clean, readable and efficient code in Python.

**Files**

By reading and writing files, programs can exchange information with each other and generate printable formats like PDF. Working with files is a lot like working with books. To use a book, you must open it. When you're done, you have to close it. While the book is open, you can either write in it or read from it. In either case, you know where you are in the book. Most of the time, you read the whole book in its natural order, but you can also skip around. Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language,0s and 1s).

* **Text files:**In this type of file, each line of text is terminated with a special character called EOL (End of Line), which is the new line character (‘\n’) in python by default.
* **Binary files:**In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

**Reading and Writing to text files**

we will be focusing on opening, closing, reading and writing data in a text file.

**File Access Modes**

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the **File Handle** in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. **Read Only (‘r’):**Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exist, raises I/O error. This is also the default mode in which file is opened.
2. **Read and Write (‘r+’):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
3. **Write Only (‘w’):** Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
4. **Write and Read (‘w+’):** Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. **Append Only (‘a’)**: Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read (‘a+’):**Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

**Opening a File**

It is done using the open () function. No module is required to be imported for this function.

File\_object = open("File\_Name","Access\_Mode")

|  |
| --- |
| The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.  # Open function to open the file "MyFile1.txt"  # (same directory) in append mode and  file1 = open("MyFile.txt","a")    # store its reference in the variable file1  # and "MyFile2.txt" in D:\Text in file2  file2 = open("D:\Text\MyFile2.txt","w+") |

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2.

**Closing a File**

close () function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

|  |
| --- |
| # Opening and Closing a file "MyFile.txt"  # for object name file1.  file1 = open("MyFile.txt","a")  file1.close() |
|  |

**Writing to a File**

There are two ways to write in a file.

1. **write ():** Inserts the string str1 in a single line in the text file.

File\_object.write(str1)

1. **writelines ():** For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

File\_object.writelines(L) for L = [str1, str2, str3]

**Reading from a File**

There are three ways to read data from a text file.

1. **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

File\_object.readgj([n])

1. **readline() :** Reads a line of the file and returns in form of a string. For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

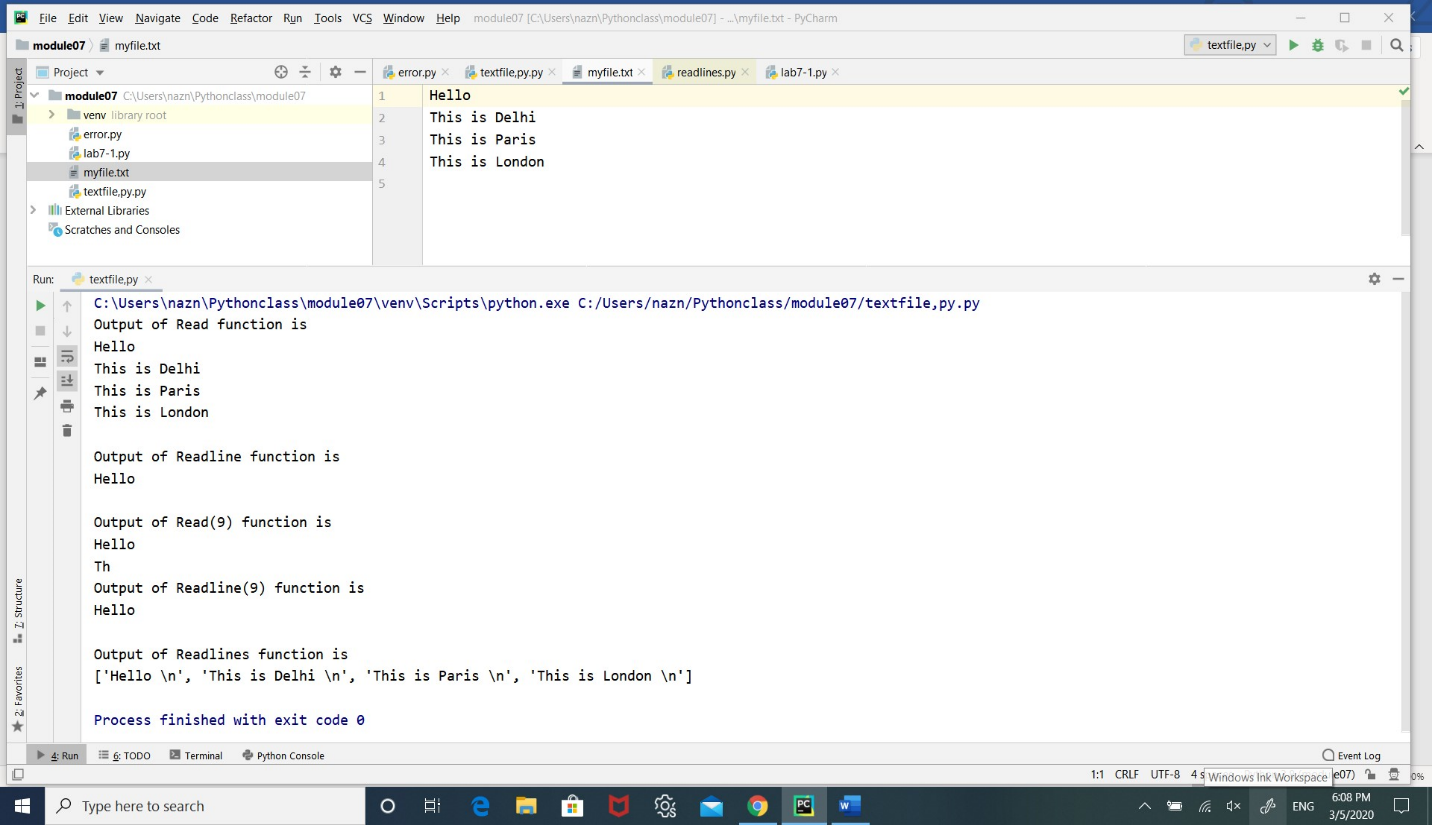
File\_object.readline([n])

1. **readlines() :** Reads all the lines and return them as each line a string element in a list.

File\_object.readlines()

**Note:**‘\n’ is treated as a special character of two bytes

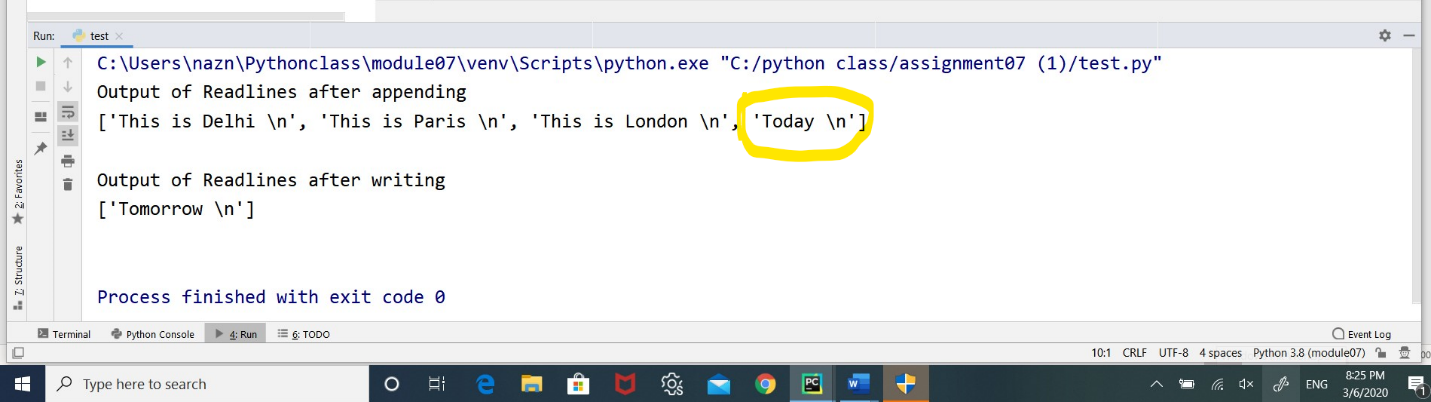
|  |
| --- |
| The Program below show various ways to read and write data in a file.  *#------------------------------------------------- # # Title: working with text file # Description: Program to show various ways to read and # ChangeLog: (Who, When, What) # NazneenM,3.4.2020, Created Script # -----------------------------------------------# # write data in a file.* file1 = open(**"myfile.txt"**, **"w"**) L = [**"This is Delhi \n"**, **"This is Paris \n"**, **"This is London \n"**]  *# \n is placed to indicate EOL (End of Line)* file1.write(**"Hello \n"**) file1.writelines(L) file1.close() *# to change file access modes* file1 = open(**"myfile.txt"**, **"r+"**)  print(**"Output of Read function is "**) print(file1.read()) print()  *# seek(n) takes the file handle to the nth # bite from the beginning.* file1.seek(0) print(**"Output of Readline function is "**) print(file1.readline()) print()  file1.seek(0) *# To show difference between read and readline* print(**"Output of Read(9) function is "**) print(file1.read(9)) print()  file1.seek(0) print (**"Output of Readline(9) function is "**) print(file1.readline(9)) file1.seek(0)  *# readlines function* print(**"Output of Readlines function is "**) print(file1.readlines()) print() file1.close()  the output for this program is shown below in **figure 7-1.** |

**figure 7-1 shows the result for above program**

**Appending to a File**

*# Python program to illustrate  
# Append vs write mode*file1 = open(**"myfile.txt"**, **"w"**)  
L = [**"This is Delhi \n"**, **"This is Paris \n"**, **"This is London \n"**]  
file1.writelines(row)  
file1.close()  
  
*# Append-adds at last*file1 = open(**"myfile.txt"**, **"a"**) *# append mode*file1.write(**"Today \n"**)  
file1.close()  
  
file1 = open(**"myfile.txt"**, **"r"**)  
print(**"Output of Readlines after appending"**)  
print(file1.readlines())  
print()  
file1.close()  
  
*# Write-Overwrites*file1 = open(**"myfile.txt"**, **"w"**) *# write mode*file1.write(**"Tomorrow \n"**)  
file1.close()  
  
file1 = open(**"myfile.txt"**, **"r"**)  
print(**"Output of Readlines after writing"**)  
print(file1.readlines())  
print()  
file1.close()

The output for this code shows the difference between append and write in figure 7-2.

**Figure 7-2 shows the result**

**Pickling**

In order to put values into a file, you have to convert them to strings. You have already seen how to do that with str:

>>> f.write (str(12.3))  
>>> f.write (str([1,2,3]))

The problem is that when you read the value back, you get a string. The original type information has been lost. In fact, you can't even tell where one value ends and the next begins:

>>>   f.readline()  
'12.3[1, 2, 3]'

The solution is pickling, Python pickle module is used for serializing and de-serializing a Python object structure. Any object in Python can be pickled so that it can be saved on disk. What pickle does is that it “serializes” the object first before writing it to file. Pickling is a way to convert a python object (list, dict, etc.) into a character stream. The idea is that this character stream contains all the information necessary to reconstruct the object in another python script.

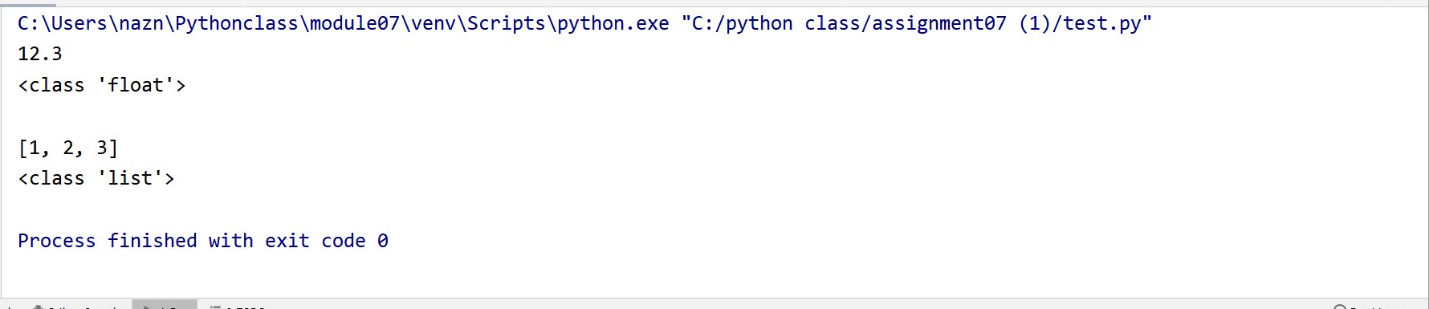
To store a data structure, use the **dump**method and then close the file in the usual way:as shown in the program below.

Pickled objects must be stored in a binary file, they cant be stored in a text file. So I open a new binary file named test.dat for writing by passing “wb” as the file access mode. Next I pickile and store to file by using dump function.

*#demonstrate pickling***import** pickle  
f = open(**"test.dat"**,**"wb"**)  
pickle.dump(12.3, f)  
pickle.dump([1,2,3], f)  
f.close()

Then we can open the file for reading and load the data structures we dumped:

f = open(**"test.dat"**,**"rb"**)  
x = pickle.load(f)  
print(x)  
print(type(x),**'\n'**)  
y = pickle.load(f)  
print(y)  
print(type(y))

the output for this program is in figure 7-3 below

**figure 7-3 result of pickling**

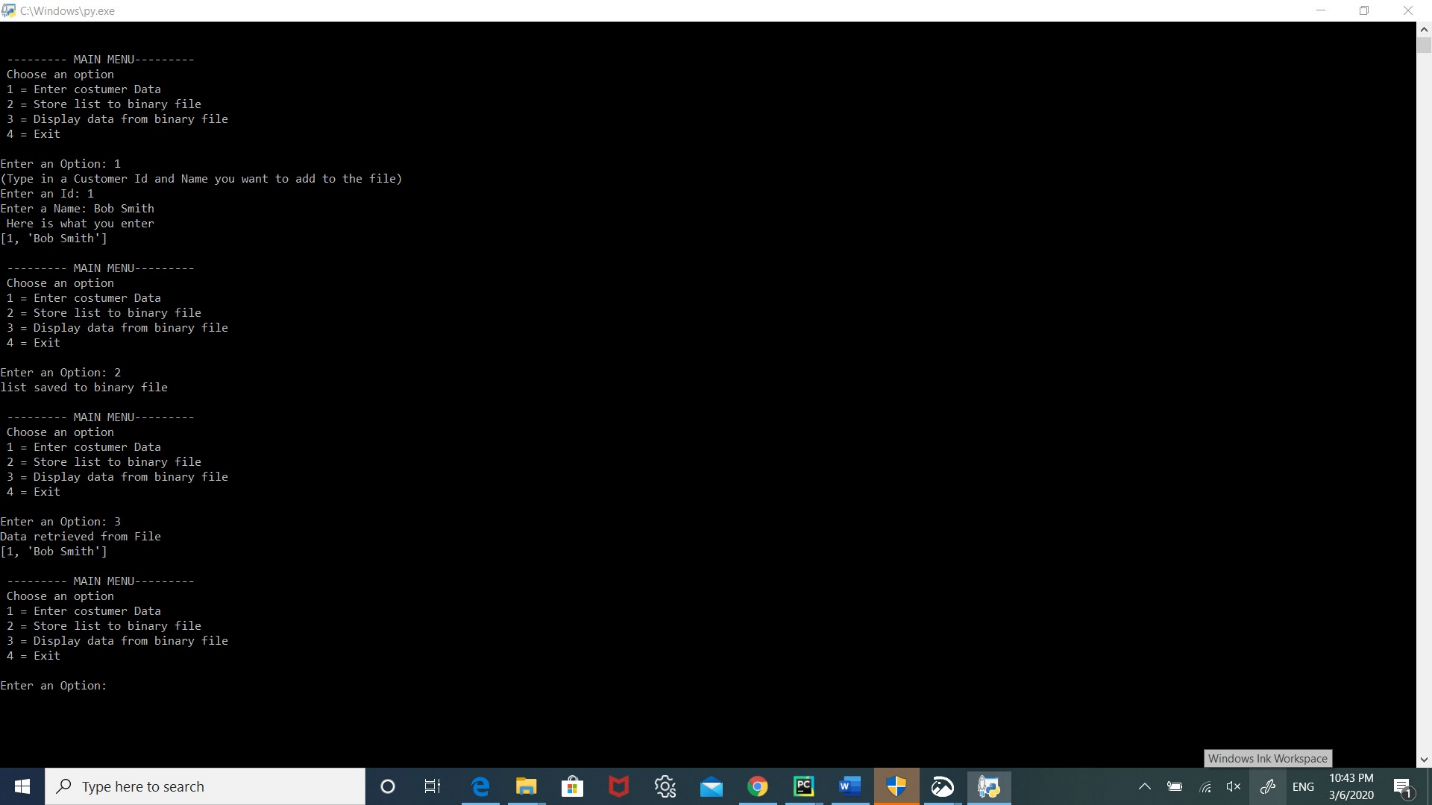
Each time we invoke load, we get a single value from the file, complete with its original type. <https://www.geeksforgeeks.org/reading-writing-text-files-python/>.

Below is an example code for take customer data from user then pickled in to list then write to a binary file “appData.dat” and after that we retrieve (unpickle) it to produce the list again “ load”. Finally, I print the unpickled lists to prove that the process worked.

*# ------------------------------------------------- #  
# Title: pickling to store data in Binary format  
# Description: A simple example of storing and retrieve  
# data in a binary file  
# ChangeLog: (Who, When, What)  
# NazneenM, 3.5.2020,Created Script  
# ------------------------------------------------- #***import** pickle *# This imports code from another codee!  
# Data -------------------------------------------- #*strFileName = **'AppData.dat'** *# The name of binary file*lstCustomer = [] *# A list for costumer inforamtion*objFile= **None** *# An object that represents a file  
  
# Processing -------------------------------------- #***def** save\_data\_to\_file(file\_name, list\_of\_data):  
 *"""* **:param** *file\_name: (string) with name of file* **:param** *list\_of\_data:(string) with data to save* **:return***:nothing  
 """  
 # now we store the data with the pickle.dump method* objFile = open(file\_name, **"wb"**) *# open binary file to overwrite* pickle.dump(list\_of\_data, objFile)  
 objFile.close() *# close the file***def** read\_data\_from\_file(file\_name, list\_of\_data):  
 *"""* **:param** *file\_name: (string) with name of file* **:param** *list\_of\_data:(string) with data to save* **:return***: List\_of\_data  
 """  
 # read the data back with the pickle.load method* objFile = open(file\_name, **"rb"**) *# open binary file to read from* list\_of\_data = pickle.load(objFile) *#load() only loads one row of data* **return** (list\_of\_data) *#return list\_0f\_ data* objFile.close() *# close file for security  
  
  
 # Presentation ------------------------------------ #*print() *# this to look nice***while**(**True**): *# while loop repeat the loop until choose exit* print(**'''\n --------- MAIN MENU---------   
 Choose an option  
 1 = Enter costumer Data  
 2 = Store list to binary file  
 3 = Display data from binary file   
 4 = Exit  
 '''**)  
 choice = input(**'Enter an Option: '**) *# catching choice from user* **if** choice == **'1'**: *# checks if choice =1 then execute the following steps  
 # if not go to next elif* print(**"(Type in a Customer Id and Name you want to add to the file)"**)  
 *# Get ID and NAME From user, then store it in a list object* intId = int(input(**"Enter an Id: "**))  
 strName = str(input(**"Enter a Name: "**))  
 lstData = [intId, strName] *# put enter data to list* lstCustomer += lstData *# append more data to list* print(**" Here is what you enter "**) *# for more declaration* print(lstCustomer) *# print the list for user* **elif** choice == **'2'**: *# if choice =2 execute the following  
 # store the list object into a binary file* save\_data\_to\_file(strFileName, list\_of\_data=lstCustomer)  
 print(**'list saved to binary file'**) *# displayed for user that data saved* **elif** choice == **"3"**:  
 print (**"Data retrieved from File"**)  
 *# Read the data from the file into a new list object  
 # and display it* print(read\_data\_from\_file(strFileName, lstCustomer)) *# print the returning data from read* **elif** choice == **'4'**: *# if the choice =4 then exit the program* print(**"Good bye!"**) *# nice word for user* **break** *# exit the while loop* **else**:  
 print(**'Please Enter Choice 1,2,3,4 or 5!'**) *# if the choice not included go back to main menue .  
 # hint for user to choose the correct choice*

The program has menu which included three 4 choices , first choice for enter the data by the user , the second choice save it to binary file , 3rd choice is retrieve the data from binary file and print it for the user and finally the last choice is number 4 which is exit the program . In the below figure I show the execution for each choice and I added the binary file picture to the top after opening it in notepad. The output for this program is shown below in figure 7-4.

Figure 7-4 the result of the pickling and unpickling



**Exceptions in Python**

Python has many [built-in exceptions](https://www.programiz.com/python-programming/exceptions) which forces your program to output an error when something in it goes wrong. When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

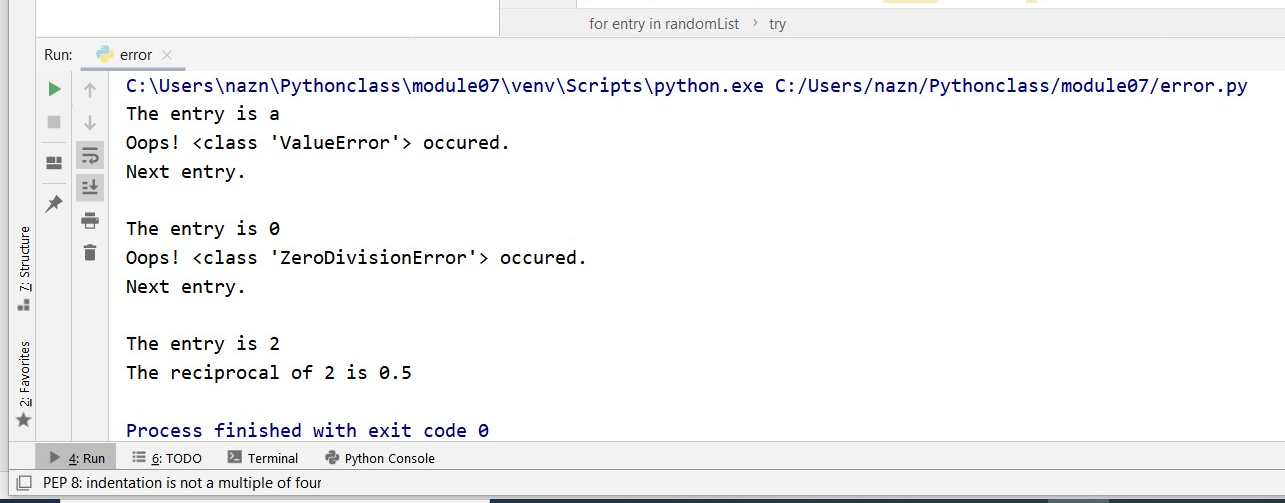
For example, if [function](https://www.programiz.com/python-programming/function) A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.

If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

**Catching Exceptions in Python**

In Python, exceptions can be handled using a try statement. A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause. It is up to us, what operations we perform once we have caught the exception. Here is a simple example.

*# ------------------------------------------------- #  
# Title: import module sys  
# Description: A try-catch with multiple error messages  
# ChangeLog: (Who, When, What)  
# NazneenM,3.4.2020, Created Script  
# -----------------------------------------------#  
# import module sys to get the type of exception***import** sys  
randomList = [**'a'**, 0, 2]  
**for** entry **in** randomList:  
 **try**:  
 print(**"The entry is"**, entry)  
 r = 1 / int(entry)  
 **break  
 except**:  
 print(**"Oops!"**, sys.exc\_info()[0], **"occured."**)  
 print(**"Next entry."**)  
 print()  
print(**"The reciprocal of"**, entry, **"is"**, r)

**The Output in the figure 7-5**

**Figure 7-5 result of above code**

In this program, we loop until the user enters an integer that has a valid reciprocal. The portion that can cause exception is placed inside try block. If no exception occurs, except block is skipped and normal flow continues. But if any exception occurs, it is caught by the except block. Here, we print the name of the exception using ex\_info() function inside sys module and ask the user to try again. We can see that the values 'a' and '1.3' causes **ValueError** and '0' causes **ZeroDivisionError.**

**Catching Specific Exceptions in Python**

In the above example, we did not mention any exception in the except clause. This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch. A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

try:

# do something

pass

except ValueError:

# handle ValueError exception

pass

except (TypeError, ZeroDivisionError):

# handle multiple exceptions

# TypeError and ZeroDivisionError

pass

except:

# handle all other exceptions

pass

**Raising Exceptions**

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise. We can also optionally pass in value to the exception to clarify why that exception was raised.

>>> raise KeyboardInterrupt

Traceback (most recent call last):

...

KeyboardInterrupt

>>> raise MemoryError("This is an argument")

Traceback (most recent call last):

...

MemoryError: This is an argument

>>> try:

... a = int(input("Enter a positive integer: "))

... if a <= 0:

... raise ValueError("That is not a positive number!")

... except ValueError as ve:

... print(ve)

**... Output ...**

Enter a positive integer: -2

That is not a positive number!

**try...finally**

The try statement in Python can have an optional **finally clause**. This clause is executed no matter what, and is generally used to release external resources. For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI). In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the **finally clause** to guarantee execution.

Here is an example of [file operations](https://www.programiz.com/python-programming/file-operation) to illustrate this.

try:

f = open("test.txt",encoding = 'utf-8')

# perform file operations

finally:

f.close()

This type of construct makes sure the file is closed even if an exception occurs.

**Python Custom Exceptions**

Python has many [built-in exceptions](https://www.programiz.com/python-programming/exceptions) which forces your program to output an error when something in it goes wrong. However, sometimes you may need to create custom exceptions that serves your purpose.

In Python, users can define such exceptions by creating a new class. This exception class must be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived form this class.

>>> class CustomError(Exception):

... pass

...

>>> raise CustomError

Traceback (most recent call last):

...

\_\_main\_\_.CustomError

>>> raise CustomError("An error occurred")

Traceback (most recent call last):

...

\_\_main\_\_.CustomError: An error occurred

Here, we have created a user-defined exception called CustomError which is derived from the Exception class. This new exception can be raised, like other exceptions, using the raise statement with an optional error message. When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file. Many standard modules do this. They define their exceptions separately as exceptions.py or errors.py (generally but not always).

User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise. Most implementations declare a custom base class and derive others exception classes from this base class. This concept is made clearer in the following example.

**Example: User-Defined Exception in Python**

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors. This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

# define Python user-defined exceptions

class Error(Exception):

"""Base class for other exceptions"""

pass

class ValueTooSmallError(Error):

"""Raised when the input value is too small"""

pass

class ValueTooLargeError(Error):

"""Raised when the input value is too large"""

pass

# our main program

# user guesses a number until he/she gets it right

# you need to guess this number

number = 10

while True:

try:

i\_num = int(input("Enter a number: "))

if i\_num < number:

raise ValueTooSmallError

elif i\_num > number:

raise ValueTooLargeError

break

except ValueTooSmallError:

print("This value is too small, try again!")

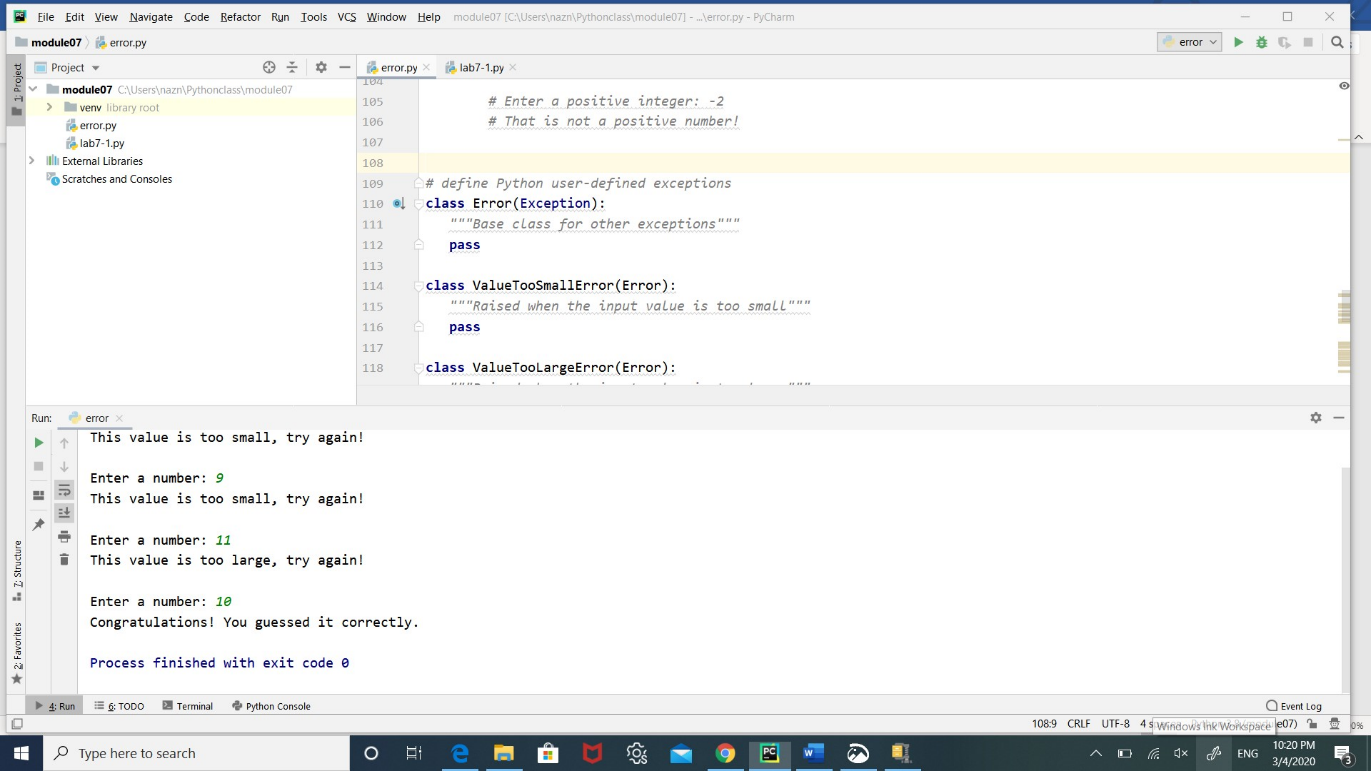
print()

except ValueTooLargeError:

print("This value is too large, try again!")

print()

print("Congratulations! You guessed it correctly.")

Here is a sample run of this program in figure 7-6.

**Figure 7-6 shows the result**

Here,I have defined a base class called Error. The other two exceptions (ValueTooSmallError and ValueTooLargeError) that are actually raised by our program are derived from this class. This is the standard way to define user-defined exceptions in Python programming, but you are not limited to this way only.

<https://www.programiz.com/python-programming/exception-handling>

**Summary**

In this module you learned about files and exception. you learned how to read from text files. You saw how to read a single character or an entire file at once. You learned several different ways to read one full line at a time, probably the most common way to read a text file. You also learned how to write to text files—everything from a single character to a list of strings. Next, you learned how to save more complex data to files through pickling and how to manage a group of pickled objects in a single binary file. Then, you saw how to handle exceptions raised during the execution of a program. You saw how to trap for specific exceptions and how to write code to deal with them.