Advanced Programming-Python (Files and Exception)

Presented by

Siddhanta Borah

DIFFERENT MODES OF OPENING A FILE

Modes	Description
r	It opens a file in reading mode. The file pointer is placed at the starting of the file. It is the default mode.
rb	It opens a file in reading only mode in binary format. The file pointer is placed at the starting of the file.
r+	It opens the file in both reading and writing mode. The file pointer is placed at the starting of the file.
rb+	It opens the file in both reading and writing mode in binary format. The file pointer is placed at the starting of the file.
W	It opens the file in writing only mode. If a file exists, it overwrites the existing file; otherwise, it creates a new file.
Wb	It opens the file in writing only mode in binary format. If a file exists, it overwrites the existing file; otherwise, it creates a new file.
w+	It opens the file in booth reading and writing mode. If a file exists, it overwrites the existing file; otherwise, it creates a new file.
wb+	It opens the file in booth reading and writing mode in binary format. If a file exists, it overwrites the existing file; otherwise, it creates a new file.
a	It opens a file for appending. The file pointer is placed at the end of the file. If the file does not exist in the directory, it creates a new file for writing.
ab	It opens a file for appending in binary format. The file pointer is placed at the end of the file. If the file does not exist in the directory, it creates a new file for writing.
a+	It opens a file for appending and reading. The file pointer is placed at the end of the file. If the file does not exist in the directory, it creates a new file for writing.
ab+	It opens a file for appending and reading in binary format. The file pointer is placed at the end of the file. If the file does not exist in the directory, it creates a new file for writing.

DIFFERENT MODES OF OPENING A FILE

Examples

```
>>>f = open(“test.txt”) # opens in r mode(reading only)
>>>f = open(“test.txt”,’w’)  # opens in w mode(writing only)
>>>f = open(“image.bmp”,’rb+’)  # read and write in binary mode
```

WRITING TO A FILE

After opening a file, we have to perform some operations on the file. Here, we will perform the write operation. In order to write into a file, we have to open it with `w` mode, or `a` mode, or any writing-enabling mode. We should be careful when using the `w` mode because in this mode overwriting persists in case the file already exists.

Syntax

```
fileobject.write(string)
```

Example

```
# open the file with w mode
>>>f = open("C:/Python27/test.txt","w")
# perform write operation
>>>f.write('writing to the file line 1\n')
>>>f.write('writing to the file line 2\n')
>>>f.write('writing to the file line 3\n')
>>>f.write('writing to the file line 4')
# close the file after writing
>>>f.close()
```

WRITING TO A FILE

After opening a file, we have to perform some operations on the file. Here, we will perform the write operation. In order to write into a file, we have to open it with `w` mode, or `a` mode, or any writing-enabling mode. We should be careful when using the `w` mode because in this mode overwriting persists in case the file already exists.

Syntax

```
fileobject.write(string)
```

Example

```
# open the file with w mode
>>>f = open("C:/Python27/test.txt","w")
# perform write operation
>>>f.write('writing to the file line 1\n')
>>>f.write('writing to the file line 2\n')
>>>f.write('writing to the file line 3\n')
>>>f.write('writing to the file line 4')
# close the file after writing
>>>f.close()
```

WRITING TO A FILE

The given example creates a file named `test.txt` if it does not exist, and overwrites into it if it exists. If you open the file, you will find the following content in it.

Output:

```
Writing to the file line 1
Writing to the file line 2
Writing to the file line 3
Writing to the file line 4
```

WRITING TO A FILE

The given example creates a file named `test.txt` if it does not exist, and overwrites into it if it exists. If you open the file, you will find the following content in it.

Output:

```
Writing to the file line 1
Writing to the file line 2
Writing to the file line 3
Writing to the file line 4
```

READING FROM A FILE

In order to read from a file, we must open the file in the reading mode (`r` mode). There are a number of methods available for reading. We can use `read(size)` method to read the data specified by `size`. If no `size` is provided, it will end up reading to the end of the file.

Reading from any file is done with the method `read()`. The `read()` method enables us to read the strings from an opened file.

Syntax

```
fileobject.read([size])
```

The count parameter gives the **number of bytes** to be read from an opened file. It starts reading from the beginning of the file until the `size` given. If no `size` is provided, it ends up reading until the end of the file.

READING FROM A FILE

Example

```
# open the file
>>>f = open("C:/Python27/test.txt", "r")
>>>f.read(7)        # read from starting 7 bytes of data
'writing'           # Output
>>>f.read(6)       # read next 6 bytes of data
'to the'            # Output
>>>f.read()         # read rest of the file
' file line 1\nwriting to the file line 2\nwriting to the file line 3\
nwriting to the file line 4\n'        # Output
>>>f.read()
''   # Output
```

Here we are using the file `'test.txt'` created earlier for reading. `f.read(7)` reads the first 7 bytes of data. After this, `f.read(6)` reads the next 6 bytes and then `f.read()` reads the rest of the file._When we try to read the file after fully reading it, we get an empty string. This is because we have ended up reading the whole file and no string is left to read in the file.

FILE POSITIONS

We have seen in the previous section that when we read a line or some data from a file, the pointer points to the next line or data, and that when we end up reading whole file, it returns the empty string. In this section, we will learn to check the current position of the pointer and to change the position of the pointer.

In Python, the `tell()` method tells us about the current position of the pointer. The current position tells us where reading will start from at present.

We can also change the position of the pointer with the help of `seek()` method. A number of bytes are passed to be moved by the pointer as arguments to the `seek()` method.

FILE POSITIONS

Example

```
# open the file
>>>f=open("C:/Python27/test.txt", "r")
# read 28 bytes of data
>>>f.read(28)
 'writing to the file line 1'                 # Output
# check the current position
>>>f.tell()
28L          # Output
# change the position to beginning
>>>f.seek(0)
# again read 28 bytes
>>>f.read(28)
 'writing to the file line 1'                 # Output
```

RENAMING A FILE

Renaming a file in Python is done with the help of the `rename()` method. The `rename()` method is passed with two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_filename, new_filename)
```

Example

```
# import os
>>>import os
# renaming the file
>>>os.rename(“C:/Python27/test.txt”,”C:/Python27/test1.txt”)
```

Here, the `test.txt` file in `C:/Python27` directory is renamed `test1.txt`.

DELETING A FILE

Deleting a file in Python is done with the help of the `remove()` method. It takes the filename as an argument to be deleted.

Syntax

```
os.remove(filename)
```

Example

```
# import os
>>>import os
# deleting the file
>>>os.remove("C:/Python27/test1.txt")
```

EXCEPTIONS

While writing a program, we often end up making some errors. There are many types of errors that can occur in a program. The error caused by writing an improper syntax is termed *syntax error* or *parsing error*; these are also called *compile time errors*.

Errors can also occur at *runtime* and these runtime errors are known as *exceptions*. There are various types of runtime errors in Python. Let us look at a few examples. When a file we try to open does not exist, we get a *FileNotFoundError*. When a division by zero happens, we get a *ZeroDivisionError*. When the module we are trying to import does not exist, we get an *ImportError*. Python creates an exception object for every occurrence of these run-time errors. The user must write a piece of code that can handle the error. If it is not capable of handling the error, the program prints a trace back to that error along with the details of why the error has occurred.

Example

Compile time error (Syntax error)

```
>>>a = 3
>>>if (a < 4)                    # Semicolon is not included
SyntaxError: invalid syntax            # Output
```

EXCEPTIONS

ZeroDivisionError

```
>>>5/0
```

Output:

```
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    5/0
ZeroDivisionError: integer division or modulo by zero
```

Here, we tried to divide 5 by 0. As a result, the interpreter prints `ZeroDivisionError`.

HANDLING EXCEPTIONS

Whenever an exception occurs in Python, it stops the current process and passes it to the calling process until it is handled. If there is no piece of code in your program that can handle the exception, then the program will crash.

For example, assume that a function X calls the function Y, which in turn calls the function Z, and an exception occurs in Z. If this exception is not handled in Z itself, then the exception is passed to Y and then to X. If this exception is not handled, then an error message will be displayed and our program will suddenly halt.

1. **`try`...`except`** Python provides a `try` statement for handling exceptions. An operation in the program that can cause the exception is placed in the `try` clause while the block of code that handles the exception is placed in the `except` clause. The block of code for handling the exception is written by the user and it is for him to decide which operation he wants to perform after the exception has been identified.

Syntax

```
try:
the operation which can cause exception here,
.......................
```

HANDLING EXCEPTIONS

```
 except Exception1:
if there is exception1, execute this.
except Exception2:
if there is exception2, execute this.
.........................
else:
if no exception occurs, execute this.
```

HANDLING EXCEPTIONS

2. `except` with No Exception We can also write our `try-except` clause with no exception. All types of exceptions that occur are caught by the `try-except` statement. However, because it catches all exceptions, the programmer cannot identify the root cause of a problem that may occur. Hence, this type of programming approach is not considered good.

Syntax

```
try:
    The statements that can cause exceptions
    ...................................
except:
    If Exception occurs, execute this
    ...................................
else:
    If no exception occurs, execute this
```

HANDLING EXCEPTIONS

Example

```
>>>while True:
...         try:
...                 a = int(raw_input("Enter an integer: "))
...                 div = 10/a
...                 break
...         except:
...                 print "Error Occured"
...                 print "Please Enter Valid Value"
...                 print()
... print "Division is",div
```

HANDLING EXCEPTIONS

The output of the program in different scenarios will be:

```
Enter an integer: c
Error Occurred
Please Enter Valid Value

Enter an integer: 0
Error Occurred
Please Enter Valid Value

Enter an integer: 10.2
Error Occurred
Please Enter Valid Value

Enter an integer: 5
Division is 2
```

In the above example, `break` statement is used instead of `else` statement because the `else` statement only executes when there is no exception.

HANDLING EXCEPTIONS

3.**try....finally** The `try` statement in Python has an optional `finally` clause that can be associated with it. The statements written in `finally` clause will always be executed by the interpreter, whether the `try` statement raises an exception or not.

Syntax

```
try:
the operation which can cause exception here,
    .........................
    This may be skipped due to exception
finally ():
    This will always execute, no matter what
    ..................................
```

HANDLING EXCEPTIONS

Example

```
>>> try:
...         file = open("testfile","w")
...         try:
...                 file.write("Write this to the file")
...         finally:
...                 print "Closing file"
...                 file.close()
... exceptIOError:
...         print "Error Occurred"
```

In the given example, when an exception is raised by the statements of `try` block, the execution is immediately passed to the `finally` block. After all the statements inside the `finally` block are executed, the exception is raised again and is handled by the `except` block that is associated with the next higher layer `try` block.

In the above example, a nested try block is used, which means a try block inside another try block. The nested try blocks are allowed in python programming language. Although it is not considered as good programming practice but it may be useful sometimes.

PROBLEM FROM FILE EXCEPTION

1) Write a python program to create a file and write all your information such as roll number, semester, branch and name.
2) Write a python program to read complete information from a text file.
3) Write a python program to rename a text file.
4) Write a python program to close a file in case of there is any error using try-finally clause.
5) Write a python program to demonstrate exception handling.
6) Write a python program to demonstrate the file and file I/O operations.
7) Write a program to catch on Divide by zero Exception. Add a finally block too.

Thank You