File Handling and Exception Handling

File Handling in Python

A file is a container in computer storage devices used for storing data.

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)
3. Close the file

Python File Open

The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:

"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

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Open a File

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())
```

If the file is located in a different location, you will have to specify the file path, like this:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the **read()** method returns the whole text, but you can also specify how many characters you want to

```
return:f = open("demofile.txt", "r")

print(f.read(5))
```

Return the 5 first characters of the file

Read Lines

You can return one line by using the **readline()** method:

Read one line of the file:

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

By calling **readline()** two times, you can read the two first lines:

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

By looping through the lines of the file, you can read the whole file, line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Close the file when you are finish with it:

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

Closing a file will free up the resources that were tied with the file. It is done using the **close()** method in Python.

Write to an Existing File

To write to an existing file, you must add a parameter to the **open()** function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Open the file "demofile3.txt" and overwrite the content:

```python
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the overwriting:
f = open("demofile3.txt", "r")
print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the **open()** method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

The open command will open the file in the read mode and the for loop will print each line present in the file.

```
# a file named "geek", will be opened with the reading mode.
file = open('geek.txt', 'r')

# This will print every line one by one in the file
for each in file:
    print (each)
```

In this example, we will see how we can read a file using the with statement.

with statement in Python

In Python, with statement is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Observe the following code example on how the use of with statement makes code cleaner.

```
# file handling

# 1) without using with statement
file = open('file_path', 'w')
file.write('hello world !')
file.close()

# 2) without using with statement
file = open('file_path', 'w')
try:
    file.write('hello world')
finally:
    file.close()
```

```python
# using with statement
with open('file_path', 'w') as file:
    file.write('hello world !’)
```

```python
# Python code to illustrate with()
with open("geeks.txt") as file:
    data = file.read()

print(data)
```

We can also split lines while reading files in Python. The split() function splits the variable when space is encountered. You can also split using any characters as you wish.

```python
# Python code to illustrate split() function
with open("geeks.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

```python
# Python code to illustrate with() alongwith write()
with open("file.txt", "w") as f:
    f.write("Hello World!!!")
```

```python
# Python code to create a file
file = open('test.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

Create a file called “myfile.txt":

```python
f = open("myfile.txt", “x")
```

Result: a new empty file is created!

Create a new file if it does not exist:

```python
f = open("myfile.txt", "w")
```

Delete a File

To delete a file, you must import the OS module, and run its **os.remove()** function:

```python
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Check if file exists, then delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the **os.rmdir()** method:

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

Exception Handling

Error in Python can be of two types i.e. Syntax errors and Exceptions.

Errors are problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which change the normal flow of the program.

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.

- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.

- **NameError**: This exception is raised when a variable or function name is not found in the current scope.

- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.

- **KeyError**: This exception is raised when a key is not found in a dictionary.

- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.

- **ImportError**: This exception is raised when an import statement fails to find or load a module.

Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur.

We can view all the built-in exceptions using the built-in `local()` function as follows:

```
print(dir(locals()['__builtins__']))
```

Exception Handling

The `try...except` block is used to handle exceptions in Python. Here's the syntax of `try...except` block:

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

Here, we have placed the code that might generate an exception inside the `try` block. Every `try` block is followed by an `except` block.

When an exception occurs, it is caught by the `except` block. The `except` block cannot be used without the try block.

The **try** block lets you test a block of code for errors.

The **except** block lets you handle the error.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

Example: Exception Handling Using try...except

```python
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

Catching Specific Exceptions in Python

For each `try` block, there can be zero or more `except` blocks. Multiple `except` blocks allow us to handle each exception differently.

The argument type of each `except` block indicates the type of exception that can be handled by it. For example,

```python
try:

    even_numbers = [2,4,6,8]
    print(even_numbers[5])

except ZeroDivisionError:
    print("Denominator cannot be 0.")

except IndexError:
    print("Index Out of Bound.")

# Output: Index Out of Bound
```

assert Keyword

The **assert** keyword is used when debugging code.

The **assert** keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError.

```
x = "hello"

#if condition returns True, then nothing happens:
assert x == "hello"

#if condition returns False, AssertionError is raised:
assert x == “goodbye”
```

Write a message if the condition is False:

```
x = "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye", "x should be 'hello'"
```

Python try with else clause

In some situations, we might want to run a certain block of code if the code block inside `try` runs without any errors.

For these cases, you can use the optional `else` keyword with the `try` statement.

Let's look at an example:

```python
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Note: Exceptions in the `else` clause are not handled by the preceding except clauses.

Python try...finally

In Python, the `finally` block is always executed no matter whether there is an exception or not.

The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

Let's see an example,

```python
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")
```

Custom Exceptions

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the **raise** keyword.

Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The **raise** keyword is used to raise an exception.You can define what kind of error to raise, and the text to print to the user.

Raise a TypeError if x is not an integer:

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```