CHAPTER 10

Files

10.1 Persistence

The programs are run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples

operating systems, which run whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

A file in a computer is a location for storing some related data.

It holds a specific name for itself.

The files are used to store data permanently on a non-volatile memory, such as hard disks.

RAM is a volatile memory type because the data it holds is lost, when we turn off the computer.

Hence, files are used for storing useful information or data for future reference.

Files are divided into two categories, **text files** and **binary files.**

Text files are simple texts in human readable format and binary files have binary data which is understood by the computer.

Programs maintain their data by simply reading and writing the text files.

When there is a need to read from or write to a file, we have to open it first.

Once reading or writing is done, we have to close the file in order to release the resources.

The order of the file operations is as follows:

1. Opening a file
2. Perform operations (Read or Write)
3. Close the file.

10.1.1. Opening a file

Python introduces the ***file*** object in order to perform some file operations.

Python has a built-in ***open()*** function to open files from the directory.

Two arguments that are mainly need they are : ***file name or file path*** and ***mode*** in which the file is opened.

Syntax

File_object=open(filename [,access_mode])

***file_name*:** File name contains a string type value containing the name of the file which we want to access.

access_mode: The value of access_mode specifies the mode in which we want to open the file.i.e, read, write, append, etc., the default access_mode is r (reading).

Sr.No.	Modes	Description
1	r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

10.1.2. Closing the file

Proper closing of a file frees up the resources held with the file.

The closing of file is done with a built-in function ***close()***

Syntax

Fileobject.close()

Example

```
>>>f= open("test.txt")                    #file open in read mode by default
>>>f=open("test1.txt","w")      #file opened in write mode
#perform file operation
>>>f.close()
```

'***with'*** keyword:

once the ***'with'*** block exits, file is automatically closed and ***file_object*** is destroyed.

Example:

```
>>> with open("test.txt" as f:
        print f.read()
```

10.1.3. The File Object Attributes

Once a file is opened and the user will have one *file* object and also can get various informations related to that file.

Sr.No.	Attribute	Description
1	**file.closed**	Returns true if file is closed, false otherwise.
2	**file.mode**	Returns access mode with which file was opened.
3	**file.name**	Returns name of the file.
4	**file.softspace**	Returns false if space explicitly required with print, true otherwise.

Example

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

Output

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

10.1.4. Reading and Writing Files

The *file* object have a set of access methods.

The write() Method

The ***write()*** method writes any string to an open file.

It is important to note that Python strings can have binary data and not just text.

The **write()** method does not add a newline **character ('\n')** to the end of the string.

Syntax

fileObject.write(string)

Example

```
# Open a filefo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n")
 # Close opend
filefo.close()
```

The read() Method

The ***read()*** method reads a string from an open file.

Syntax

fileObject.read([count])

Here, passed parameter is the number of bytes to be read from the opened file.

This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend
filefo.close()
```

This produces the following result −

```
Read String is :  Python is
```

File Positions

The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(offset[, from])* method changes the current file position.

The *offset* argument indicates the number of bytes to be moved.

The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example

```
# Open a file
 fo = open("foo.txt", "r+")
 str = fo.read(10)
print "Read String is : ", str
# Check current position
position = fo.tell()
print "Current file position : ", position
# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10)
print "Again read String is : ", str
# Close opend filef
o.close()
```

Output

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

10.1.5. Renaming and Deleting Files

Python **os** module provides methods that help user to perform file-processing operations, such as renaming and deleting files.

To use this module user need to import it first and then can call any related functions.

The rename() Method

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax

os.rename(current_file_name, new_file_name)

Example

```
import os
# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

Deleting the file (The remove() Method)

The user can use the *remove()* method to delete files.

The name of the file to be deleted as the argument.

Syntax

os.remove(file_name)

Example

```
import os
# Delete file test2.txt
os.remove("text2.txt")
```

10.1.6. Directories in Python

The mkdir() Method

The user can use the *mkdir()* method of the **os** module to create directories in the current directory.

an argument ➔the name of the directory to be created.

Syntax

os.mkdir("newdir")

Example

```
import os
# Create a directory "test“
os.mkdir("test")
```

The chdir() Method

The user can use the *chdir()* method to change the current directory.

an argument ➔ name of the directory

Syntax

os.chdir("newdir")

Example

```
import os
# Changing a directory to "/home/newdir“
os.chdir("/home/newdir")
```

(iii)The getcwd() Method

The *getcwd()* method displays the current working directory.

Syntax

os.getcwd()

Example

```
import os
# This would give location of the current directory
os.getcwd()
```

(iv)The rmdir() Method

The *rmdir()* method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax

os.rmdir('dirname')

Example

```
import os
# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```

10.1.7. File & Directory Related Methods

File Object Methods: The *file* object provides functions to manipulate files.

OS Object Methods: This provides methods to process files as well as directories.

SNO	METHOD	DESCRIPTION
1	File.close()	It closes the file
2	File.flush()	It flushes the internal buffer momory
3	File.fileno()	It returns the file number
4	File.isatty()	It returns True if the file is connected with tty(-like) device, False otherwise.
5	File.next()	It returns the next line from the file
6	File.read([size])	Reads the size bytes from a file
7	File.readline([size])	It reads the entire one line from a file
8	File.readline([sizehint])	It reads until the end of the file using readline. It returns the list of lines reade.
9	File.seek([offset])	It changes the current position
10	File.tell()	It returns the current position
11	File.truncate([size])	It truncates the file
12	File.write(str)	It writes the str string to the file.
13	File.writelines(sequence)	It writes the sequence of stings into a file. If each string in the sequence should go into separate lines infile., the string should end with a new line character,'\n'

10.1.8. Format operator

Example:

```
>>> x = 52
>>> fout.write(str(x))
```

format operator(%)

When applied to integers, % is the modulus operator.

But when the first operand is a string, % is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted.

The result is a string.

Example

The format sequence '%d' means that the second operand should be formatted as an integer

`d stands for “decimal”

```
>>> camels = 42
>>> '%d' % camels
'42'              #output
```

The result is the string '42', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'              #output
```

10.1.9.Command Line Arguments

$ python test.py arg1 arg2 arg3

The Python **sys** module provides access to any command-line arguments via the **sys.argv**.

This serves two purposes −

sys.argv is the list of command-line arguments.

len(sys.argv) is the number of command-line arguments.

Here sys.argv[0] is the program

Example

```
 import sys
print 'Number of arguments:', len(sys.argv), 'arguments’.
print 'Argument List:', str(sys.argv)
```

Execution

```
$ python test.py arg1 arg2 arg3
```

Output

```
Number of arguments: 4arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

Parsing Command-Line Arguments

getopt.getopt method

This method parses command line options and parameter list.

Syntax

getopt.getopt(args, options, [long_options])

args − This is the argument list to be parsed.

options − This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).

long_options − This is optional parameter and if specified, must be a list of strings with the names of the long

options, which should be supported.

Long options, which require an argument should be followed by an equal sign ('=').

To accept only long options, options should be an empty string.

This method returns value consisting of two elements: the first is a list of **(option, value)** pairs.

The second is the list of program arguments left after the option list was stripped.

Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for

short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

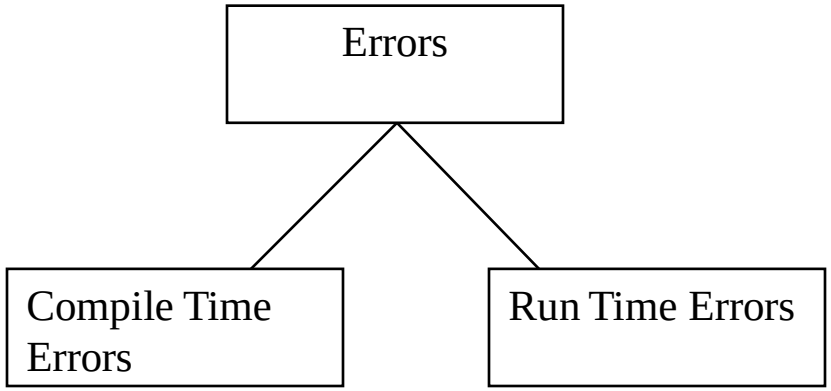
Exception getopt.GetoptError

The argument to the exception is a string indicating the cause of the error.

The attributes **msg** and **opt** give the error message and related option.

CHAPTER 11
Exceptions

11.1. Introduction



Compile time errors are also called syntax errors or parsing errors.
Runtime errors are called exceptions.
Python creates an exception object for every occurrence of these runtime errors.
The user must write a piece of code that can handle the error.

Example 1: compile time error

```
a=3
if (a<4)
File "<string>", line 2  #output
 if (a<4)
    ^
SyntaxError: invalid syntax
```

Example 1: run time error

```
a=3
b=a/0
Traceback (most recent call last):     #output
File "<string>", line 2, in <module>
ZeroDivisionError: division by zero
```

11.2. Built-in Exceptions

S.No.	Exception Name	Description
1	**Exception**	Base class for all exceptions
2	**StopIteration**	Raised when the next() method of an iterator does not point to any object.
3	**SystemExit**	Raised by the sys.exit() function.
4	**StandardError**	Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError**	Base class for all errors that occur for numeric calculation.
6	**OverflowError**	Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError**	Raised when a floating point calculation fails.
8	**ZeroDivisionError**	Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError**	Raised in case of failure of the Assert statement.
10	**AttributeError**	Raised in case of failure of attribute reference or assignment.
11	**EOFError**	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError**	Raised when an import statement fails.

S.No.	Exception Name	Description
1	**Exception**	Base class for all exceptions
2	**StopIteration**	Raised when the next() method of an iterator does not point to any object.
3	**SystemExit**	Raised by the sys.exit() function.
4	**StandardError**	Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError**	Base class for all errors that occur for numeric calculation.
6	**OverflowError**	Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError**	Raised when a floating point calculation fails.
8	**ZeroDivisionError**	Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError**	Raised in case of failure of the Assert statement.
10	**AttributeError**	Raised in case of failure of attribute reference or assignment.
11	**EOFError**	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError**	Raised when an import statement fails.

S.No.	Exception Name	Description
25	**SystemExit**	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError**	Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError**	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError**	Raised when a generated error does not fall into any category.
29	**NotImplementedError**	Raised when an abstract method that needs to be implemented in

11.3. 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 cod in our program that can handle the exception, then the program will crash.

To handle any unexpected error in Python programs

1. **Exception Handling**
2. **Assertions**

Try …except

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.

Syntax

try:

The operation which can cause exception here,

……..……..

except Exception1:

If there is exception1, execute this.

except Exception2:

If there is exception2, execute this.

…….

else:

if no exception occurs, execute this.

1. A try block can have multiple except clauses associated with it,that can cause different types of exceptions.

2. The statement in the else block will execute only when the statements in try block do not raise any exception.

Example 1:

```
a=10
b=0
try:
    c=a/0
except ZeroDivisionError:
    print("divide by zero error")
b=4
c=a/b
print("a/b = ",a/b)
```

output

```
divide by zero error
a/b =  2.5
```

Example 2:

```
try:
    f=open('e:\test.txt','r')
  f.readline()
except IOError:
    print('error: cannot open the file in read mode')
else:
    print('file read successfully')
    f.close()
```

Output

```
error: cannot open the file in read mode
```

except with no Exception

All types of exceptions that occur are caught by the try…except statement.

because it catches all exceptions, the programmer cannot identify the root cause of a problem that may occur.

Syntax

try:

The operation which can cause exception here,

……..……..

except:

If there is exception1, execute this.

else:

if no exception occurs, execute this.

Example:

```
while True:
  try:
        a= int(input('Enter an integer value:'))
        div=10/a
        break
 except:
        print('Error occured')
        print('pl. enter valid values')
        print()
print('Division :',div)
```

Output

```
Enter an integer value:0
Error occured
pl. enter valid values

Enter an integer value:c
Error occured
pl. enter valid values

Enter an integer value:10.2
Error occured
pl. enter valid values

Enter an integer value:5
Division : 2.0
```

except with Multiple Exceptions

Syntax

try:

The operation which can cause exception here,

……

……..……..

except (Exception1 [, Eception 2 [,.., Exception n]]):

If any of the exception occurs from the above list, execute this.

else:

if no exception occurs, execute this.

Example:

```
try:
    x=int(input("Enter the value of x"))
    for i in range(x):
        print(i)
except (TypeError,ZeroDivisionError,ValueError):
    print(“Runtime Error Occurs” )
else:
    print("no error")
```

Output 1:

```
Enter the value of x5
0
1
2
3
4
No error
```

Output 2:

```
Enter the value of xa
 Runtime Error Ocurs
```

try…finally

The ***try*** statement in python has an optional ***finally*** clause. The statement written in ***finally*** clause will always be executed by the interpreter, whether the ***try*** statement raises an exception or not.

(a)With a ***try*** clause, we can use either ***except*** or ***finally***, but not both.

(b)We cannot use ***else*** clause along with a ***finally*** clause.

Syntax

try:

The operation which can cause exception here,

……..……..

finally:

if no exception occurs, execute this.

Example:

```
try:
    f=open('e:\test.txt','w+')
    try:
        file.write('Write this to file test.txt')
    finally:
        print('closing the file')
        f.close()
except IOError:
    print('Error Occurred')
```

Output

Error Occurred

11.4. Exception With Arguments

The ***except*** clause in Python can also have an argument.

Arguments give information about the problem due to which the exception has occurred.

We can accept the exception's argument by passing a variable in the ***except*** the clause.

The contents of arguments may vary from exception to exception.

Syntax

try:

The operation which can cause exception here,

……..…….

except Exceptiontype, Argument:

If exception occurs, execute this.

Example 1:

```
def integer(a):
    try:
        return int(a)
    except ValueError as Argument:
        print('The Value does not contain Numbers',Argument)
integer('helloworld')
```

Output

```
The Value does not contain Numbers invalid literal for int() with base 10:
'helloworld'
```

Example 2:

```
def integer(a):
    try:
        return int(a)
    except ValueError as Argument:
        print('The Value does not contain Numbers',Argument)
print('The argument value is ',integer(10))
```

Output

```
The argument value is  10
```

11.5. User Defined Exceptions

The exception class is to be derived, directly or indirectly from Exception clause.

Example 1:

```
class CustomError(Exception):
    pass
raise CustomError('Error Occurred')
```

Output

```
        Traceback (most recent call last):
         File "<string>", line 5, in <module>
__main__.CustomError: Error Occurred
```

```
class Error(Exception):    """Base class for other exceptions"""
      pass
class ValueTooSmallError(Error):    """Raised when value is too small"""
      pass
 class ValueTooLargeError(Error):    """Raised when value is too large"""
      pass
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!")
 else:
       print() print("Congratulations! You guessed it correctly.")
```

Output

Enter a number: 5

This value is too small, try again!

Enter a number: 25

This value is too large, try again!

Enter a number: 10

Congratulations! You guessed it correctly.

References:

1. Allen B Downey. “Think Python: How to Think like a Computer Scientist”, Green Tea Press,version 2.0.17.
2. E. Balagurusamy , “Introduction to Computing and Problem Solving Using Python”,Mc Graw hill education.
3. Guido van Rossum and Fred L. Drake Jr, “An Introduction to Python – Revised and updated for Python 3.2”, Network Theory Ltd., 2011.
4. John V Guttag, “Introduction to Computation and Programming Using Python”, Revised and expanded Edition, MIT Press , 2013
5. www.w3school.com
6. https://www.programiz.com/python-programming.
7. https://www.tutorialspoint.com