Unit-2(files, exceptions and modules)

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

The *open* Function(file() function)

Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are parameter details −

- **file_name** − The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Here is a list of the different modes of opening a file −

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.

The *file* Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all attributes related to file object −

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

```
#!/usr/bin/python

# 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
```

This produces the following result −

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

The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close()
```

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

This produces the following result −

```
Name of the file:  foo.txt
```

Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

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)
```

Here, passed parameter is the content to be written into the opened file.

Example

```
#!/usr/bin/python

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

# Close opend file
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

```
Python is a great language.
Yeah its great!!
```

The *read()* Method

The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

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

Let's take a file *foo.txt*, which we created above.

```
#!/usr/bin/python
```

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

Let us take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

# 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 file
fo.close()
```

This produces the following result −

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

Renaming and Deleting Files

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

To use this module you need to import it first and then you 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

Following is the example to rename an existing file *test1.txt* −

```
#!/usr/bin/python
import os

# Rename a file from test1.txt to test2.txt
```

```
os.rename( "test1.txt", "test2.txt" )
```

The *remove()* Method

You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

```
os.remove(file_name)
```

Example

Following is the example to delete an existing file *test2.txt* −

```
#!/usr/bin/python
import os

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

Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The *mkdir()* Method

You can use the *mkdir()* method of the **os** module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

Syntax

```
os.mkdir("newdir")
```

Example

Following is the example to create a directory *test* in the current directory −

```
#!/usr/bin/python
import os

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

The *chdir()* Method

You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

```
os.chdir("newdir")
```

Example

Following is the example to go into "/home/newdir" directory −

```
#!/usr/bin/python
import os

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

The *getcwd()* Method

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

Syntax

```
os.getcwd()
```

Example

Following is the example to give current directory −

```
#!/usr/bin/python
import os

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

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

Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

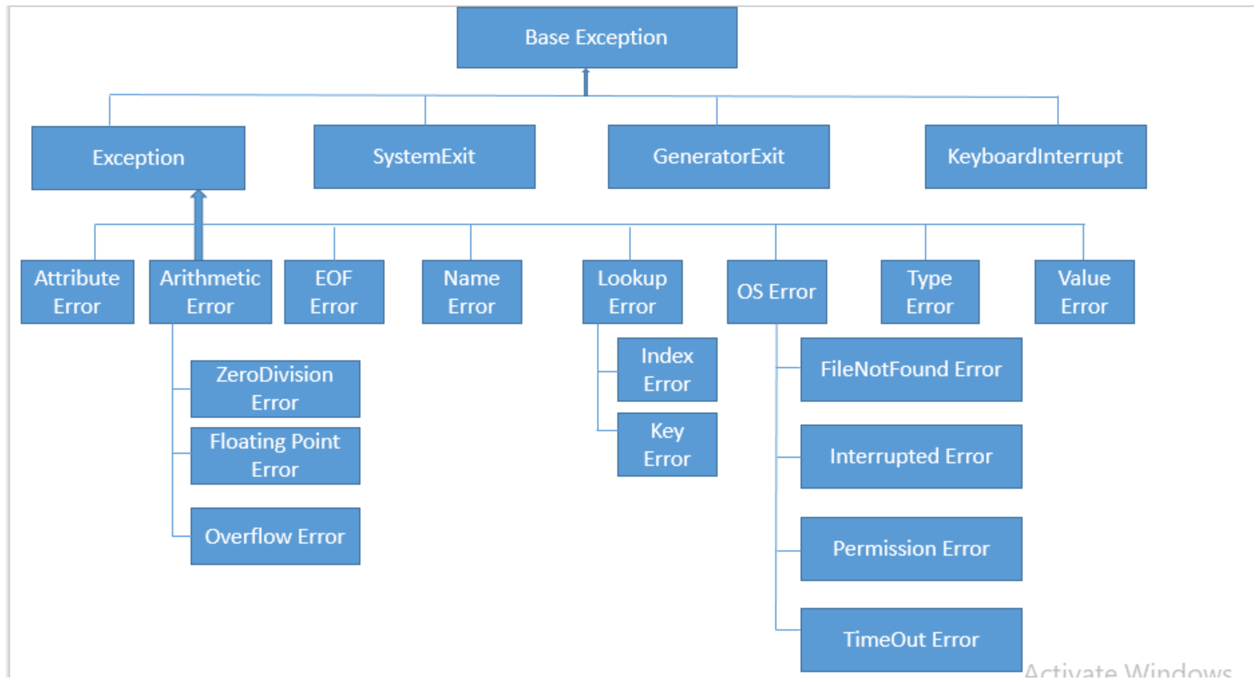
```
#!/usr/bin/python
import os

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

File & Directory Related Methods

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows −

- File Object Methods: The *file* object provides functions to manipulate files.
- OS Object Methods: This provides methods to process files as well as directories.

Python Exception



An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.
Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

The problem without handling exceptions

As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a **ZeroDivision** exception. Let's see the following example.

Example

```python
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d" %c)
  #other code:
print("Hi I am other part of the program")
```

Output:

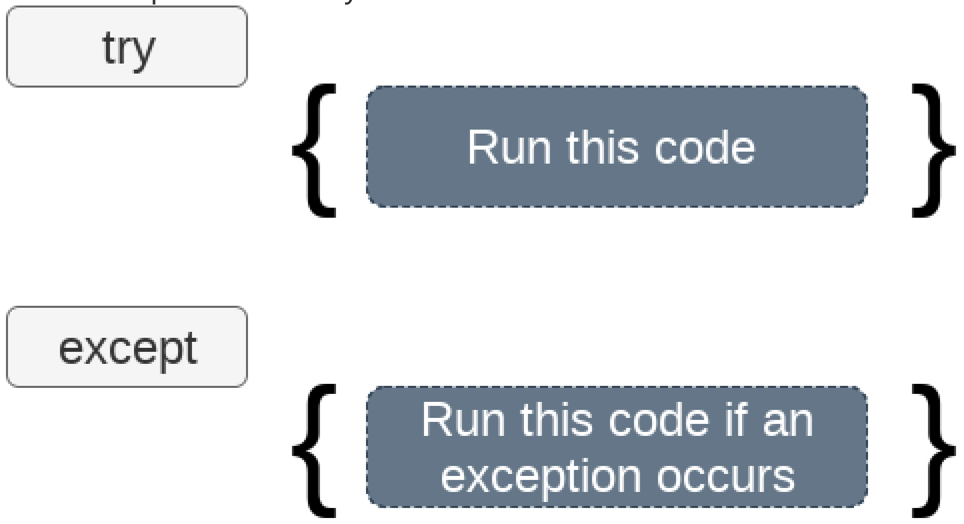
```
Enter a:10
Enter b:0
Traceback (most recent call last):
  File "exception-test.py", line 3, in <module>
    c = a/b;
ZeroDivisionError: division by zero
```

The above program is syntactically correct, but it through the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions. We can handle these exceptions in the following way.

Exception handling in python

The try-expect statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block. The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.



Syntax

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```

Consider the following example.

Example 1

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

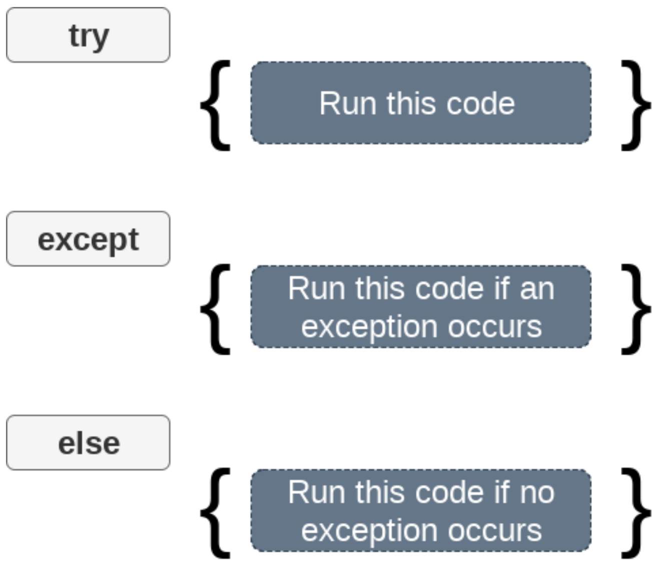
Output:

```
Enter a:10
Enter b:0
Can't divide with zero
```

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

The syntax to use the else statement with the try-except statement is given below.

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs

Consider the following program.

Example 2

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
# Using Exception with except statement. If we print(Exception) it will return exception class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
<class 'Exception'>
```

The except statement with no exception

Python provides the flexibility not to specify the name of exception with the exception statement.

Consider the following example.

Example

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except:
    print("can't divide by zero")
else:
    print("Hi I am else block")
```

The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
division by zero
```

Points to remember

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

Example

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

Output:

```
File not found
```

Declaring Multiple Exceptions

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

Syntax

```
try:
    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
    #block of code
```

Consider the following example.

```python
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

Output:

```
Arithmetic Exception
```

The try...finally block

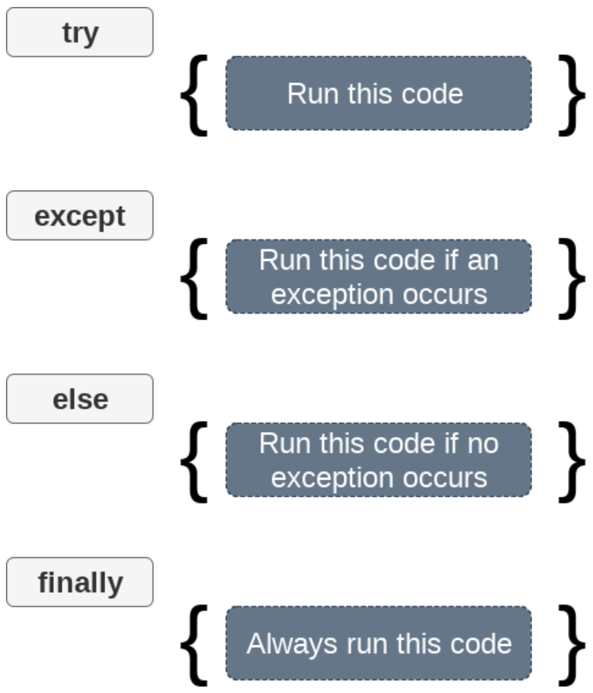
Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

Syntax

```python
try:
    # block of code
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code

Example

```python
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

Output:

```
file closed
Error
```

Raising exceptions

An exception can be raised forcefully by using the **raise** clause in Python. It is useful in that scenario where we need to raise an exception to stop the execution of the program.

For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

The syntax to use the raise statement is given below.

Syntax

```python
raise Exception_class,<value>
```

Points to remember

1. To raise an exception, the raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "**as**" keyword is used. "**e**" is used as a reference variable which stores the value of the exception.
4. We can pass the value to an exception to specify the exception type.

Example

```python
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Output:

```
Enter the age:17
The age is not valid
```

Example 2 Raise the exception with message

```python
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
# we can pass the message in the raise statement
        raise ValueError("That is  a negative number!")
except ValueError as e:
    print(e)
```

Output:

```
Enter a positive integer: -5
That is a negative number!
```

Example 3

```python
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   if b is 0:
      raise ArithmeticError
   else:
      print("a/b = ",a/b)
except ArithmeticError:
   print("The value of b can't be 0")
```

Output:

```
Enter a:10
Enter b:0
The value of b can't be 0
```

Python Modules

What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Example

Save this code in a file named `mymodule.py`

```python
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the `import` statement:

Example

Import the module named mymodule, and call the greeting function:

```python
import mymodule

mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax: *module_name.function_name*.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file `mymodule.py`

```python
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example

Import the module named mymodule, and access the person1 dictionary:

```python
import mymodule

a = mymodule.person1["age"]
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

Example

Create an alias for `mymodule` called `mx`:

```python
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

Import and use the `platform` module:

```python
import platform

x = platform.system()
print(x)
```

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function:

Example

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Note: The dir() function can be used on *all* modules, also the ones you create yourself.

Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

Example

The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module.

Example: `person1["age"]`, **not** ~~`mymodule.person1["age"]`~~

Python Modules

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provides us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

Example

In this example, we will create a module named as file.py which contains a function func that contains a code to print some message on the console.

Let's create the module named as **file.py.**

```
#displayMsg prints a message to the name being passed.
def displayMsg(name)
  print("Hi "+name);
```

Here, we need to include this module into our main module to call the method displayMsg() defined in the module named file.

Loading the module in our python code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

The import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The syntax to use the import statement is given below.

```
import module1,module2,........ module n
```

Hence, if we need to call the function displayMsg() defined in the file file.py, we have to import that file as a module into our module as shown in the example below.

Example:

```
import file;
name = input("Enter the name?")
file.displayMsg(name)
```

Output:

```
Enter the name?John
Hi John
```

The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

```
from < module-name> import <name 1>, <name 2>..,<name n>
```

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

calculation.py:

```
#place the code in the calculation.py
def summation(a,b):
   return a+b
def multiplication(a,b):
   return a*b;
def divide(a,b):
   return a/b;
```

Main.py:

```
from calculation import summation
#it will import only the summation() from calculation.py
a = int(input("Enter the first number"))
```

```
b = int(input("Enter the second number"))
print("Sum = ",summation(a,b)) #we do not need to specify the module name while accessing summation()
```

Output:

```
Enter the first number10
Enter the second number20
Sum =  30
```

The from...import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using *.

Consider the following syntax.

```
from <module> import *
```

Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

```
import <module-name> as <specific-name>
```

Example

```
#the module calculation of previous example is imported in this example as cal.
import calculation as cal;
a = int(input("Enter a?"));
b = int(input("Enter b?"));
print("Sum = ",cal.summation(a,b))
```

Output:

```
Enter a?10
Enter b?20
Sum =  30
```

Using dir() function

The dir() function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Consider the following example.

Example

```
import json

List = dir(json)

print(List)
```

Output:

```
['JSONDecoder',    'JSONEncoder',    '__all__',    '__author__',
'__builtins__', '__cached__', '__doc__',
'__file__',  '__loader__',  '__name__',  '__package__',  '__path__',
'__spec__', '__version__',
'_default_decoder',  '_default_encoder',  'decoder',  'dump',  'dumps',
'encoder', 'load', 'loads', 'scanner']
```

The reload() function

As we have already stated that, a module is loaded once regardless of the number of times it is imported into the python source file. However, if you want to reload the already imported module to re-execute the top-level code, python provides us the reload() function. The syntax to use the reload() function is given below.

```
1. reload(<module-name>)
```

for example, to reload the module calculation defined in the previous example, we must use the following line of code.

```
1. reload(calculation)
```

Namespaces in Python

A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.
In a Python program, there are four types of namespaces:

1. Built-In
2. Global
3. Enclosing
4. Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they're no longer needed. Typically, many namespaces will exist at any given time.

The Built-In Namespace

The **built-in namespace** contains the names of all of Python's built-in objects. These are available at all times when Python is running. You can list the objects in the built-in namespace with the following command:

>>>

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'BaseException','BlockingIOError', 'BrokenPipeError', 'BufferError',
 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
 'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
 '__doc__', '__import__', '__loader__', '__name__', '__package__',
 '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate',
 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```
You'll see some objects here that you may recognize from previous tutorials—for example, the StopIteration exception, built-in functions like max() and len(), and object types like int and str.
The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

The Global Namespace

The **global namespace** contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

Strictly speaking, this may not be the only global namespace that exists. The interpreter also creates a global namespace for any **module** that your program loads with the import statement.

The Local and Enclosing Namespaces

As you learned in the previous tutorial on functions, the interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.
Functions don’t exist independently from one another only at the level of the main program. You can also define one function inside another:

>>>

```
 1>>> def f():
 2...     print('Start f()')
 3...
 4...     def g():
 5...         print('Start g()')
 6...         print('End g()')
 7...         return
 8...
 9...     g()
10...
11...     print('End f()')
12...     return
13...
14
15>>> f()
16Start f()
17Start g()
18End g()
19End f()
```

In this example, function g() is defined within the body of f(). Here’s what’s happening in this code:

- **Lines 1 to 12** define f(), the **enclosing** function.
- **Lines 4 to 7** define g(), the **enclosed** function.
- On **line 15**, the main program calls f().
- On **line 9**, f() calls g().

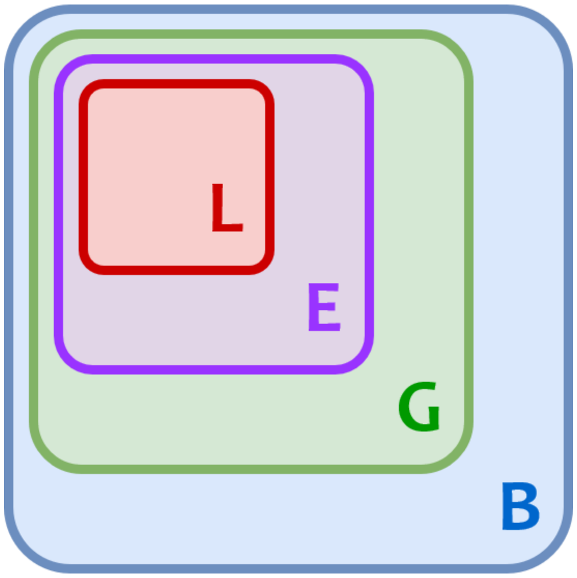
When the main program calls f(), Python creates a new namespace for f(). Similarly, when f() calls g(), g() gets its own separate namespace. The namespace created for g() is the **local namespace**, and the namespace created for f() is the **enclosing namespace**. Each of these namespaces remains in existence until its respective function terminates. Python might not immediately reclaim the memory allocated for those namespaces when their functions terminate, but all references to the objects they contain cease to be valid.

Variable Scope

The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they're all maintained separately and won't interfere with one another.
But that raises a question: Suppose you refer to the name x in your code, and x exists in several namespaces. How does Python know which one you mean?
The answer lies in the concept of **scope**. The scope of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.
To return to the above question, if your code refers to the name x, then Python searches for x in the following namespaces in the order shown:

1. **Local**: If you refer to x inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
2. **Enclosing**: If x isn't in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
3. **Global**: If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
4. **Built-in**: If it can't find x anywhere else, then the interpreter tries the built-in scope.

This is the **LEGB rule** as it's commonly called in Python literature (although the term doesn't actually appear in the Python documentation). The interpreter searches for a name from the inside out, looking in the **l**ocal, **e**nclosing, **g**lobal, and finally the **b**uilt-in scope:



If the interpreter doesn't find the name in any of these locations, then Python raises a NameError exception.

Examples

Several examples of the LEGB rule appear below. In each case, the innermost enclosed function g() attempts to display the value of a variable named x to the console. Notice how each example prints a different value for x depending on its scope.

Example 1: Single Definition

In the first example, x is defined in only one location. It's outside both f() and g(), so it resides in the global scope:

>>>

```
>>> x = 'global'

>>> def f():
...
...     def g():
...         print(x)
...
...     g()
...

>>> f()
```

```
12global
```

The print() statement on **line 6** can refer to only one possible x. It displays the x object defined in the global namespace, which is the string 'global'.

Example 2: Double Definition

In the next example, the definition of x appears in two places, one outside f() and one inside f() but outside g():

>>>

```
 1>>> x = 'global'
 2
 3>>> def f():
 4...     x = 'enclosing'
 5...
 6...     def g():
 7...         print(x)
 8...
 9...     g()
10...
11
12>>> f()
13enclosing
```

As in the previous example, g() refers to x. But this time, it has two definitions to choose from:

- **Line 1** defines x in the global scope.
- **Line 4** defines x again in the enclosing scope.

According to the LEGB rule, the interpreter finds the value from the enclosing scope before looking in the global scope. So the print() statement on **line 7** displays 'enclosing' instead of 'global'.

Example 3: Triple Definition

Next is a situation in which x is defined here, there, and everywhere. One definition is outside f(), another one is inside f() but outside g(), and a third is inside g():

>>>

```
 1>>> x = 'global'
 2
 3>>> def f():
 4...     x = 'enclosing'
 5...
 6...     def g():
 7...         x = 'local'
 8...         print(x)
 9...
10...     g()
11...
12
13>>> f()
14local
```

Now the print() statement on **line 8** has to distinguish between three different possibilities:

- **Line 1** defines x in the global scope.
- **Line 4** defines x again in the enclosing scope.
- **Line 7** defines x a third time in the scope that’s local to g().

Here, the LEGB rule dictates that g() sees its own locally defined value of x first. So the print() statement displays 'local'.

Example 4: No Definition

Last, we have a case in which g() tries to print the value of x, but x isn’t defined anywhere. That won’t work at all:

>>>

```
 1>>> def f():
 2...
 3...     def g():
 4...         print(x)
 5...
 6...     g()
 7...
 8
 9>>> f()
10Traceback (most recent call last):
11  File "<stdin>", line 1, in <module>
```

```
12  File "<stdin>", line 6, in f
13  File "<stdin>", line 4, in g
14NameError: name 'x' is not defined
```
This time, Python doesn’t find x in any of the namespaces, so the print() statement on **line 4** generates a NameError exception.

Python Namespace Dictionaries

Earlier in this tutorial, when namespaces were first introduced, you were encouraged to think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. In fact, for global and local namespaces, that’s precisely what they are! Python really does implement these namespaces as dictionaries.

Note: The built-in namespace doesn’t behave like a dictionary. Python implements it as a module.

Python provides built-in functions called globals() and locals() that allow you to access global and local namespace dictionaries.

The globals() function

The built-in function globals() returns a reference to the current global namespace dictionary. You can use it to access the objects in the global namespace. Here’s an example of what it looks like when the main program starts:

>>>

```
>>> type(globals())
<class 'dict'>

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
```
As you can see, the interpreter has put several entries in globals() already. Depending on your Python version and operating system, it may look a little different in your environment. But it should be similar.

Now watch what happens when you define a variable in the global scope:

>>>

```
>>> x = 'foo'

>>> globals()
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'x': 'foo'}
```

After the assignment statement x = 'foo', a new item appears in the global namespace dictionary. The dictionary key is the object’s name, x, and the dictionary value is the object’s value, 'foo'.

You would typically access this object in the usual way, by referring to its symbolic name, x. But you can also access it indirectly through the global namespace dictionary:

>>>

```
1>>> x
2'foo'
3>>> globals()['x']
4'foo'
5
6>>> x is globals()['x']
7True
```

The is comparison on **line 6** confirms that these are in fact the same object.

You can create and modify entries in the global namespace using the globals() function as well:

>>>

```
>>> globals()['y'] = 100

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'x': 'foo', 'y': 100}

>>> y
100

>>> globals()['y'] = 3.14159

>>> y
3.14159
```

The statement on **line 1** has the same effect as the assignment statement y = 100. The statement on **line 12** is equivalent to y = 3.14159.
It's a little off the beaten path to create and modify objects in the global scope this way when simple assignment statements will do. But it works, and it illustrates the concept nicely.

The locals() function

Python also provides a corresponding built-in function called locals(). It's similar to globals() but accesses objects in the local namespace instead:

>>>

```
>>> def f(x, y):
...     s = 'foo'
...     print(locals())
...

>>> f(10, 0.5)
{'s': 'foo', 'y': 0.5, 'x': 10}
```

When called within f(), locals() returns a dictionary representing the function's local namespace. Notice that, in addition to the locally defined variable s, the local namespace includes the function parameters x and y since these are local to f() as well.
If you call locals() outside a function in the main program, then it behaves the same as globals().
There's one small difference between globals() and locals() that's useful to know about. globals() returns an actual reference to the dictionary that contains the global namespace. That means if you call globals(), save the return value, and subsequently define additional variables, then those new variables will show up in the dictionary that the saved return value points to.

Packages in Python

Introduction

We learned that modules are files containing Python statements and definitions, like function and class definitions. We will learn in this chapter how to bundle multiple modules together to form a package.

A package is basically a directory with Python files and a file with the name __init__.py. This means that every directory inside of the Python path, which contains a file named __init__.py, will be treated as a package by Python. It's possible to put several modules into a Package.

Packages are a way of structuring Python's module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name, let's say A, for example. The submodule A of the package P1 and the submodule A of the package P2 can be totally different. A package is imported like a "normal" module. We will start this chapter with a simple example.

A Simple Example

a.py
b.py
__init__.py

We will demonstrate with a very simple example how to create a package with some Python modules. First of all, we need a directory. The name of this directory will be the name of the package, which we want to create. We will call our package "simple_package". This directory needs to contain a file with the name __init__.py. This file can be empty, or it can contain valid Python code. This code will be executed when a package is imported, so it can be used to initialize a package, e.g. to make sure that some other modules are imported or some values set. Now we can put all of the Python files which will be the submodules of our module into this directory. We create two simple files a.py and b.py just for the sake of filling the package with modules.

The content of a.py:

```python
def bar():
    print("Hello, function 'bar' from module 'a' calling")
```

The content of b.py:

```python
def foo():
    print("Hello, function 'foo' from module 'b' calling")
```

We will also add an empty file with the name __init__.py inside of simple_package directory.

Let's see what happens, when we import simple_package from the interactive Python shell, assuming that the directory simple_package is either in the directory from which you call the shell or that it is contained in the search path or environment variable "PYTHONPATH" (from your operating system):

```python
import simple_package
```

```
simple_package/a

---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)
<ipython-input-3-347df8a711cc> in <module>
```

```
----> 1 simple_package/a

NameError: name 'a' is not defined
simple_package/b
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-e71d2904d2bd> in <module>
----> 1 simple_package/b

NameError: name 'b' is not defined
```

We can see that the package simple_package has been loaded but neither the module "a" nor the module "b"! We can import the modules a and b in the following way:

```
from simple_package import a, b
a.bar()
b.foo()
```

```
Hello, function 'bar' from module 'a' calling
Hello, function 'foo' from module 'b' calling
```

As we have seen at the beginning of the chapter, we can't access neither "a" nor "b" by solely importing simple_package.

Yet, there is a way to automatically load these modules. We can use the file `__init__`.py for this purpose. All we have to do is add the following lines to the so far empty file `__init__`.py:

```
    import simple_package.a

    import simple_package.b
```

It will work now:

```
import simple_package
simple_package.a.bar()
simple_package.b.foo()
```

```
Hello, function 'bar' from module 'a' calling
Hello, function 'foo' from module 'b' calling
```