**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

[Live Demo](http://tpcg.io/mIm72J)

#!/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

[Live Demo](http://tpcg.io/pMu0CZ)

#!/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.

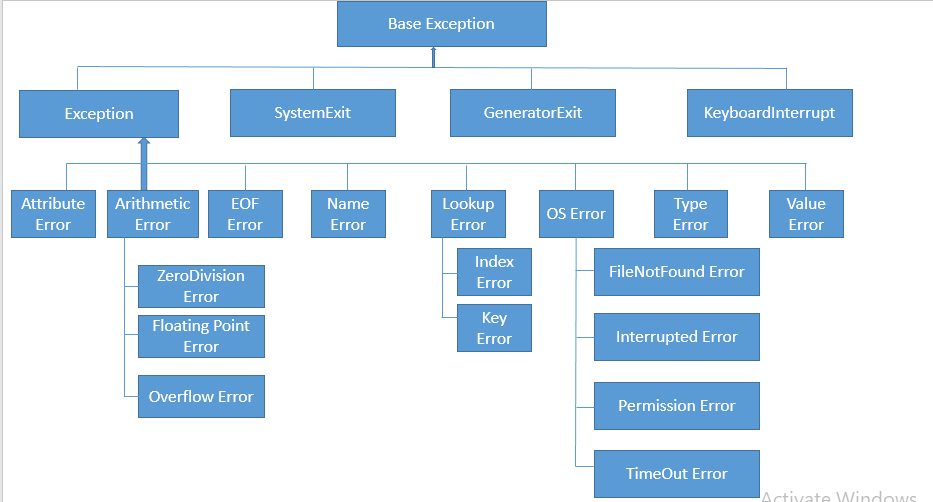
os.rmdir( "/tmp/test" )

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](https://www.tutorialspoint.com/python/file_methods.htm): The *file* object provides functions to manipulate files.
* [OS Object Methods](https://www.tutorialspoint.com/python/os_file_methods.htm): 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

1. a = int(input("Enter a:"))
2. b = int(input("Enter b:"))
3. c = a/b
4. **print**("a/b = %d" %c)
5. #other code:
6. **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;

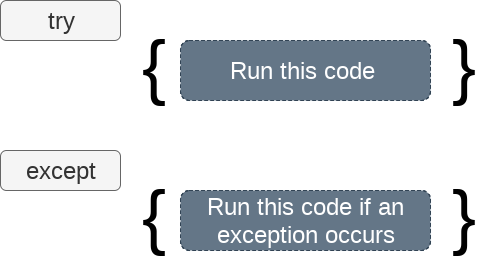
ZeroDivisionError: division by zero

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**

1. **try**:
2. #block of code
4. **except** Exception1:
5. #block of code
7. **except** Exception2:
8. #block of code
10. #other code

Consider the following example.

**Example 1**

1. **try**:
2. a = int(input("Enter a:"))
3. b = int(input("Enter b:"))
4. c = a/b
5. **except**:
6. **print**("Can't divide with zero")

**Output:**

Enter a:10

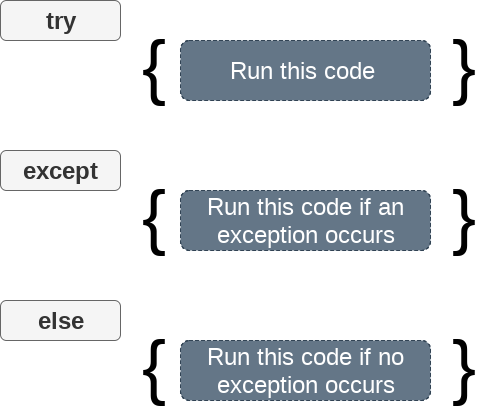
Enter b:0

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.

1. **try**:
2. #block of code
4. **except** Exception1:
5. #block of code
7. **else**:
8. #this code executes if no except block is executed



Consider the following program.

**Example 2**

1. **try**:
2. a = int(input("Enter a:"))
3. b = int(input("Enter b:"))
4. c = a/b
5. **print**("a/b = %d"%c)
6. # Using Exception with except statement. If we print(Exception) it will return exception class
7. **except** Exception:
8. **print**("can't divide by zero")
9. **print**(Exception)
10. **else**:
11. **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**

1. **try**:
2. a = int(input("Enter a:"))
3. b = int(input("Enter b:"))
4. c = a/b;
5. **print**("a/b = %d"%c)
6. **except**:
7. **print**("can't divide by zero")
8. **else**:
9. **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:

1. **try**:
2. a = int(input("Enter a:"))
3. b = int(input("Enter b:"))
4. c = a/b
5. **print**("a/b = %d"%c)
6. # Using exception object with the except statement
7. **except** Exception as e:
8. **print**("can't divide by zero")
9. **print**(e)
10. **else**:
11. **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**

1. **try**:
2. #this will throw an exception if the file doesn't exist.
3. fileptr = open("file.txt","r")
4. **except** IOError:
5. **print**("File not found")
6. **else**:
7. **print**("The file opened successfully")
8. 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**

1. **try**:
2. #block of code
4. **except** (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
5. #block of code
7. **else**:
8. #block of code

Consider the following example.

1. **try**:
2. a=10/0;
3. **except**(ArithmeticError, IOError):
4. **print**("Arithmetic Exception")
5. **else**:
6. **print**("Successfully Done")

**Output:**

Arithmetic Exception

The try...finally block

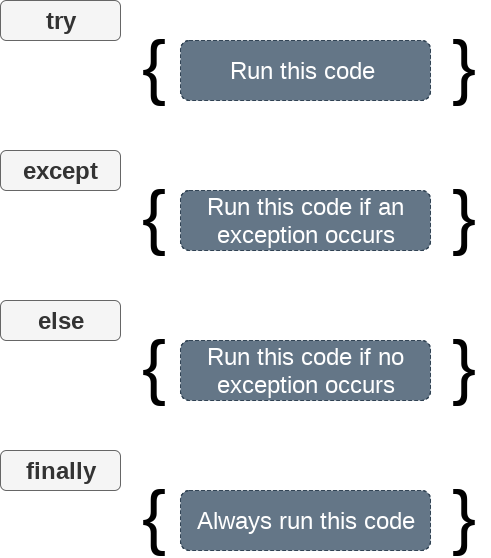
Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

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**

1. **try**:
2. # block of code
3. # this may throw an exception
4. **finally**:
5. # block of code
6. # this will always be executed



**Example**

1. **try**:
2. fileptr = open("file2.txt","r")
3. **try**:
4. fileptr.write("Hi I am good")
5. **finally**:
6. fileptr.close()
7. **print**("file closed")
8. **except**:
9. **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**

1. **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**

1. **try**:
2. age = int(input("Enter the age:"))
3. **if**(age<18):
4. **raise** ValueError
5. **else**:
6. **print**("the age is valid")
7. **except** ValueError:
8. **print**("The age is not valid")

**Output:**

Enter the age:17

The age is not valid

**Example 2 Raise the exception with message**

1. **try**:
2. num = int(input("Enter a positive integer: "))
3. **if**(num <= 0):
4. # we can pass the message in the raise statement
5. **raise** ValueError("That is  a negative number!")
6. **except** ValueError as e:
7. **print**(e)

**Output:**

Enter a positive integer: -5

That is a negative number!

**Example 3**

1. **try**:
2. a = int(input("Enter a:"))
3. b = int(input("Enter b:"))
4. **if** b **is** 0:
5. **raise** ArithmeticError
6. **else**:
7. **print**("a/b = ",a/b)
8. **except** ArithmeticError:
9. **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

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:

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

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

### Example

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

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:

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:

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"]~~

Top of Form

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.**

1. #displayMsg prints a message to the name being passed.
2. **def** displayMsg(name)
3. **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.

1. **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:

1. **import** file;
2. name = input("Enter the name?")
3. 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.

1. **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:**

1. #place the code in the calculation.py
2. **def** summation(a,b):
3. **return** a+b
4. **def** multiplication(a,b):
5. **return** a\*b;
6. **def** divide(a,b):
7. **return** a/b;

**Main.py:**

1. **from** calculation **import** summation
2. #it will import only the summation() from calculation.py
3. a = int(input("Enter the first number"))
4. b = int(input("Enter the second number"))
5. **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.

1. **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.

Learn more

volume is gedempt

The syntax to rename a module is given below.

1. **import** <module-name> as <specific-name>

Example

1. #the module calculation of previous example is imported in this example as cal.
2. **import** calculation as cal;
3. a = int(input("Enter a?"));
4. b = int(input("Enter b?"));
5. **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

1. **import** json
3. List = dir(json)
5. **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)

Scope of variables

In Python, variables are associated with two types of scopes. All the variables defined in a module contain the global scope unless or until it is defined within a function.

All the variables defined inside a function contain a local scope that is limited to this function itself. We can not access a local variable globally.

If two variables are defined with the same name with the two different scopes, i.e., local and global, then the priority will always be given to the local variable.

Consider the following example.

Example

1. name = "john"
2. **def** print\_name(name):
3. **print**("Hi",name) #prints the name that is local to this function only.
4. name = input("Enter the name?")
5. print\_name(name)

**Output:**

Hi David

Python packages

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules, and sub-modules. The packages are used to categorize the application level code efficiently.

Let's create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name Employees on path /**home**.

2. Create a python source file with name ITEmployees.py on the path /**home**/**Employees**.

**ITEmployees.py**

1. **def** getITNames():
2. List = ["John", "David", "Nick",    "Martin"]
3. **return** List;

3. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().

4. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is \_\_init\_\_.py which contains the import statements of the modules defined in this directory.

**\_\_init\_\_.py**

1. **from** ITEmployees **import** getITNames
2. **from** BPOEmployees **import** getBPONames

5. Now, the directory **Employees** has become the package containing two python modules. Here we must notice that we must have to create \_\_init\_\_.py inside a directory to convert this directory to a package.

6. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

**Test.py**

1. **import** Employees
2. **print**(Employees.getNames())

**Output:**

['John', 'David', 'Nick', 'Martin']

We can have sub-packages inside the packages. We can nest the packages up to any level depending upon the application requirements.

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian, and Student. The sub-packages contain the python modules.

Python packages