Unit-3: Python interaction with SQLite:

3.1 Module: Concepts of module and Using modules in python.

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.
- Python Modules 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 **message.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

1. 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 message.py, we have to import that file as a module into our module as shown in the example(**main.py**) below.

```
import message
name = input("Enter the name : ")
message.displayMsg(name)
```

Output:

```
Enter the name : Bhumika Patel
Hi Bhumika Patel
```

2. 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 subtraction(a,b):
       return a-b
def multiplication(a,b):
        return a*b
def divide(a,b):
        return a/b
```

Main_calc.py:

```
#it will import only the summation() from Calculation.py
from Calculation import summation
a = int(input("Enter the first number : "))
b = int(input("Enter the second number : "))
#we do not need to specify the module name while accessing summation()
print("Sum = ",summation(a,b))
```

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 *
```

Main_calc.py:

```
#it will import only the summation() from Calculation.py
from Calculation import *
a = int(input("Enter the first number : "))
b = int(input("Enter the second number : "))

print("Sum of Two Numbers = ", summation(a,b))
print("Subtraction of two Numbers = ", subtraction(a,b))
print("Multiplication of two Numbers = ", multiplication(a,b))
print("Division of two Numbers = ", divide(a,b))
```

Output:

```
Enter the first number : 10
Enter the second number : 5
Sum of Two Numbers =  15
Subtraction of two Numbers =  5
Multiplication of two Numbers =  50
Division of two Numbers =  2.0
```

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:(RenameMod.py)

```
#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
```

Reloading a Module

Suppose you have already imported a module and using it. However, the owner of the module added or modified some functionalities after you imported it. So, you can reload the module to get the latest module using the reload() function, as shown below.

```
>>> import Calculator

>>> reload(Calculator)
```

Python - Built-in Modules

- The Python interactive shell has a number of built-in functions. They are loaded automatically as a shell starts and are always available, such as print() and input() for I/O, number conversion functions int(), float(), complex(), data type conversions list(), tuple(), set(), etc.
- In addition to built-in functions, a large number of pre-defined functions are also available as a part of libraries bundled with Python distributions. These functions are defined in modules are called **built-in modules.**
- Built-in modules are written in C and integrated with the Python shell. Each built-in module contains resources for certain system-specific functionalities such as OS management, disk IO, etc. The standard library also contains many Python scripts (with the .py extension) containing useful utilities.

To display a list of all available modules, use the following command in the Python console:
>>> help('modules')

Example:

```
>>> import math
>>> print(math.pi)
3.141592653589793
```

You can also use the **dir()** function to know the names and attributes of a module.

```
>>> dir("math")
```

3.1.1 Setting PYTHONPATH, Concepts of Namespace and Scope

Module Search Path

When the import statement is encountered either in an interactive session or in a script:

- First, the Python interpreter tries to locate the module in the current working directory.
- If not found, directories in the PYTHONPATH environment variable are searched.
- If still not found, it searches the installation default directory.

As the Python interpreter starts, it put all the above locations in a list returned by the sys.path attribute.

Python - sys Module

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. You will learn some of the important features of this module here.

sys.path

This is an environment variable that is a search path for all Python modules.

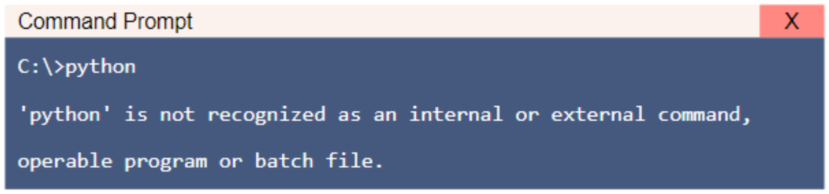
```
>>> import sys

>>>sys.path

['', 'C:\\python36\\Lib\\idlelib', 'C:\\python36\\python36.zip',
'C:\\python36\\DLLs', 'C:\\python36\\lib', 'C:\\python36',
'C:\\Users\\acer\\AppData\\Roaming\\Python\\Python36\\site-packages',
'C:\\python36\\lib\\site-packages']
```

If the required module is not present in any of the directories above, the message ModuleNotFoundError is thrown.

```
>>> import MyModule
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
    ModuleNotFoundError: No module named 'MyModule'
```

PATH variable

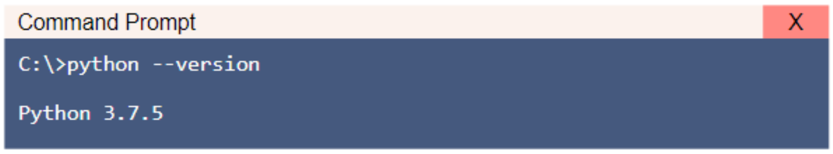
- The PATH variable is a list of directories where each directory contains the executable file for a command.
- When a command is entered into the Windows command prompt, the prompt searches in the PATH variable for an executable file with the same name as the command; in the case that the **required file is not found**, it responds with an error message that states that the specified command was not recognized.



```
C:\>python

'python' is not recognized as an internal or external command,

operable program or batch file.
```

To avoid this error is to add the executable file's directory to the PATH variable. Oftentimes, this needs to be done when installing Python.
The complete path of python.exe can be added by:

1. Right-clicking This PC and going to Properties.
2. Clicking on the **Advanced system** settings in the menu on the left.
3. Clicking on the **Environment Variables** button on the bottom right.
4. In the System variables section, selecting the Path variable and clicking on Edit. The next screen will show all the directories that are currently a part of the PATH variable.
5. Clicking on New and entering Python's install directory.
6. Usually you can find the python installed binary in this path location: C:\Users\Bhumika\AppData\Local\Programs\Python\Python39



```
C:\>python --version

Python 3.7.5
```

The PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.
Here is a typical PYTHONPATH from a Windows system –

```
set PYTHONPATH = c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system –

```
set PYTHONPATH = /usr/local/lib/python
```

❖ What is a Namespace in Python?

What is Name?

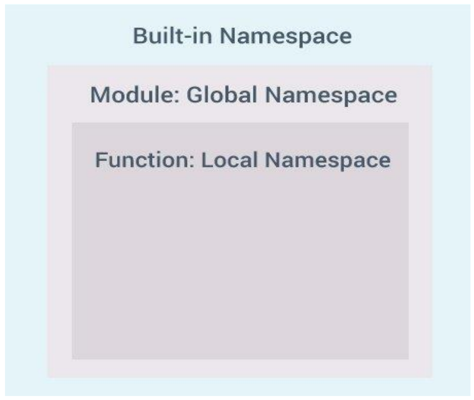
Name (also called identifier) is simply a name given to objects. Everything in Python is an object. Name is a way to access the underlying object.

What is NameSpace?

- A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method.
- Python itself maintains a namespace in the form of a Python dictionary. Let's go through an example, a directory-file system structure in computers.
- Needless to say, that one can have multiple directories having a file with the same name inside every directory. But one can get directed to the file, one wishes, just by specifying the absolute path to the file.
- Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace. So, the division of the word itself gives a little more information. Its Name (which means name, a unique identifier) + Space(which talks something related to scope). Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.
- Different namespaces can co-exist at a given time but are completely isolated.
- A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs.

Types of namespaces :

When Python interpreter runs solely without any user-defined modules, methods, classes, etc. Some functions like print(), id() are always present, these are built-in namespaces. When a user creates a module, a global namespace gets created, later the creation of local functions creates the local namespace. The built-in namespace encompasses the global namespace and the global namespace encompasses the local namespace.



The lifetime of a namespace :

A lifetime of a namespace depends upon the scope of objects, if the scope of an object ends, the lifetime of that namespace comes to an end. Hence, it is not possible to access the inner namespace's objects from an outer namespace.

Python Variable Scope

- Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.
- A scope is the portion of a program from where a namespace can be accessed directly without any prefix.
- At any given moment, there are at least three nested scopes.
    1. Scope of the current function which has local names
    2. Scope of the module which has global names
    3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.
If there is a function inside another function, a new scope is nested inside the local scope.

Example of Scope and Namespace in Python:

```
def outer_function():
    b = 20     # local namespace
    def inner_func():
        c = 30     # nested local namespace

a = 10     # global namespace
```

However, if we declare a as global, all the reference and assignment go to the global a. Similarly, if we want to rebind the variable b, it must be declared as nonlocal. **The following example will further clarify this.**

```
def outer_function():
    a = 20

    def inner_function():
        a = 30
        print('a =', a)

    inner_function()          #function call
    print('a =', a)

a = 10
outer_function()    # function call
print('a =', a)
```

OUTPUT:

```
a = 30
a = 20
a = 10
```

In this program, three different variables a are defined in separate namespaces and accessed accordingly. While in the following program,

```
def outer_function():
    global a
    a = 20

    def inner_function():
        global a
        a = 30
        print('a =', a)

    inner_function()
    print('a =', a)

a = 10
outer_function()
print('a =', a)
```

OUTPUT:

```
a = 30
a = 30
a = 30
```

Here, all references and assignments are to the global a due to the use of keyword global.

3.1.2 Concepts of Packages in python

We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily. In the same way, a package in Python takes the concept of the modular approach to next logical level. As you know, a module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files. Let's create a package named mypackage, using the following steps:

1. Create a new folder named D:\MyApp.
2. Inside MyApp, create a subfolder with the name 'mypackage'.
3. Create an empty __init__.py file in the mypackage folder.

Using a Python-aware editor like IDLE, create modules greet.py and functions.py with the following code:

```
#greet.py
def SayHello(name):
    print("Hello  ", name)

#functions.py
def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```

That's it. We have created our package called mypackage. The following is a folder structure:

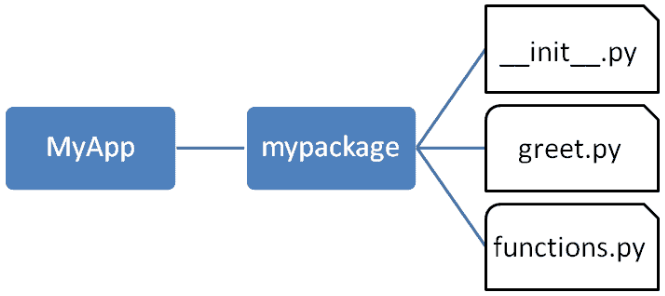


Figure: Package Folder Structure

Importing a Module from a Package

Now, to test our package, navigate the **command prompt** to the MyApp folder and invoke the Python prompt from there

```
D:\MyApp>python
```

Import the functions module from the mypackage package and call its power() function.

```
>>> from mypackage import functions
>>> functions.power(3,2)
9
```

It is also possible to import specific functions from a module in the package.

```
>>> from mypackage.functions import sum
>>> sum(10,20)
30
>>> average(10,12)
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
NameError: name 'average' is not defined
```

__init__.py

The package folder contains a special file called __init__.py, which stores the package's content. It serves two purposes:

1. The Python interpreter recognizes a folder as the package if it contains __init__.py file.
2. __init__.py exposes specified resources from its modules to be imported.

An empty __init__.py file makes all functions from the above modules available when this package is imported. Note that __init__.py is essential for the folder to be recognized by Python as a package. You can optionally define functions from individual modules to be made available.

The __init__.py file is normally kept empty. However, it can also be used to choose specific functions from modules in the package folder and make them available for import.

Modify **__init__.py** as below:

```
from .functions import average, power
from .greet import SayHello
```

The specified functions can now be imported in the interpreter session or another executable script.

Create “test.py” in the MyApp folder to test mypackage.

```
from mypackage import power, average, SayHello
SayHello('Bhumika')
x=power(3,2)
print("power(3,2) : ", x)
```

Note that functions power() and SayHello() are imported from the package and not from their respective modules, as done earlier. The output of the above script is:

```
D:\MyApp>python test.py

Hello world

power(3,2) : 9
```

3.2 Importing sqlite3 module
3.2.1 connect () and execute() methods
3.2.2 Single row and multi-row fetch (fetchone(), fetchall())
3.2.3 Select, Insert, update, delete using execute () method.
3.2.4 commit () method.

- We will learn how to perform CRUD operations in Python with the SQLite database.
- Python has built-in support for SQLite in the form of the sqlite3 module.
- This module contains functions for performing persistent CRUD operations on SQLite database.
- Standard Python distribution has in-built support for SQLite database connectivity. It contains sqlite3 module which adheres to DB-API 2.0 and is written by Gerhard Haring.
- As per the prescribed standards, the first step in the process is to obtain the connection to the object representing the database.
- In order to establish a connection with a SQLite database, sqlite3 module needs to be imported and the connect() function needs to be executed.

Following is the python code to connect the database if it exists otherwise it will create a new database and then connect to it.

```
import sqlite3
con = sqlite3.connect('DBUSingPython.db')
print "Database opened!!!"
```

The following methods are defined in the connection class:

Method	Description
cursor()	Returns a Cursor object which uses this Connection.
commit()	Explicitly commits any pending transactions to the database. The method should be a no-op if the underlying db does not support transactions.
rollback()	This optional method causes a transaction to be rolled back to the starting point. It may not be implemented everywhere.
close()	Closes the connection to the database permanently. Attempts to use the connection after calling this method will raise a DB-API Error.

What is cursor?
A cursor is a Python object that enables you to work with the database. It acts as a handle for a given SQL query; it allows the retrieval of one or more rows of the result. Hence, a cursor object is obtained from the connection to execute SQL queries using the following statement:

```
>>> cur=db.cursor()
```

The following methods of the cursor object are useful.

Method	Description
execute()	Executes the SQL query in a string parameter
executemany()	Executes the SQL query using a set of parameters in the list of tuples
fetchone()	Fetches the next row from the query result set.
fetchall()	Fetches all remaining rows from the query result set.
callproc()	Calls a stored procedure.
close()	Closes the cursor object.

- The commit() and rollback() methods of the connection class ensure transaction control.
- The execute() method of the cursor receives a string containing the SQL query.
- A string having an incorrect SQL query raises an exception, which should be properly handled. That's why the execute() method is placed within the try block and the effect of the SQL query is persistently saved using the commit() method.
- If however, the SQL query fails, the resulting exception is processed by the except block and the pending transaction is undone using the rollback() method.

Create a New Table

A string enclosing the CREATE TABLE query is passed as parameter to the execute() method of the cursor object. The following code creates the student table in the test.db database.

```
#CreateDBTab.py
import sqlite3
db=sqlite3.connect('test.db')
try:
   cur =db.cursor()
   cur.execute('''CREATE TABLE student (
   StudentID INTEGER PRIMARY KEY,
   name TEXT (20) NOT NULL,
   age INTEGER,
   marks REAL);''')
   print('Table created successfully...')
except:
   print('Error in operation...')
   db.rollback()
db.close()
```

This can be verified using the .tables command in sqlite shell.

Insert a Record

We have to use it as a parameter to the execute() method. To account for possible exceptions, the execute() statement is placed in the try block as explained earlier. The complete code for the inset operation is as follows:

```
#InsertRec.py
import sqlite3
db=sqlite3.connect('test.db')
qry="insert into student(StudentID, name, age, marks) values(1,'Shivam', 20, 80);"
try:
   cur=db.cursor()
   cur.execute(qry)
   db.commit()
   print("1 record added successfully.")
except:
   print("Error in operation.")
   db.rollback()
db.close()
```

You can check the result by using the SELECT query in Sqlite shell.

Retrieve Records

When the query string holds a SELECT query, the execute() method forms a result set object containing the records returned. Python DB-API defines two methods to fetch the records:

1. **fetchone():** Fetches the next available record from the result set. It is a tuple consisting of values of each column of the fetched record.
2. **fetchall():** Fetches all remaining records in the form of a list of tuples. Each tuple corresponds to one record and contains values of each column in the table.

When using the fetchone() method, use a loop to iterate through the result set, as below:

```
#SelectRec.py
import sqlite3
db=sqlite3.connect('test.db')
i=0
sql="Select * from student;"
cur=db.cursor()
cur.execute(sql)
while True:
    record=cur.fetchone()
    if record==None:
        break
    i=i+1
    print(record)
print(i," records")
db.close()
```

When executed, the following output is displayed in the Python shell:

```
(1, 'Rajeev', 20, 50.0)
(2, 'Vijaya', 16, 75.0)
(3, 'Amar', 18, 70.0)
(4, 'Deepak', 25, 87.0)
4  records
```

The fetchall() method returns a list of tuples, each being one record.

```
students=cur.fetchall()
for rec in students:
        print (rec)
```

Update a Record

The query string in the execute() method should contain an UPDATE query syntax. To update the value of 'age' to 17 for 'Amar', define the string as below:

```
qry="update student set age=17 where name='Amar';"
```

You can also use the substitution technique to pass the parameter to the UPDATE query.

```
#UpdateRec.py
import sqlite3
db=sqlite3.connect('test.db')
qry="update student set age=? where name=?;"
try:
    cur=db.cursor()
    cur.execute(qry, (19,'Mihir'))
    db.commit()
    print("Record updated successfully...")
except:
    print("Error in operation...")
    db.rollback()
db.close()
```

Delete a Record

The query string should contain the DELETE query syntax. For example, the below code is used to delete 'Bill' from the student table.

```
qry="DELETE from student where name='Mihir';"
```

You can use the ? character for parameter substitution.

```
import sqlite3
db=sqlite3.connect('test.db')
qry="DELETE from student where name=?;"
try:
    cur=db.cursor()
    cur.execute(qry, ('Mihir',))
    db.commit()
    print("Record deleted successfully…")
except:
    print("Error in operation…")
    db.rollback()
db.close()
```