Unit-3: Python interaction with SQLite:

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

- 3.1.1 Setting PYTHONPATH, Concepts of Namespace and Scope
- 3.1.2 Concepts of Packages in python

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.

What is a function in Python?

In Python, a function is a group of related statements that performs a specific task.
Functions help break our program into smaller and modular chunks.
As our program grows larger and larger, functions make it more organized and manageable.
Furthermore, it avoids repetition and makes the code reusable.

Syntax of Function:

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

Above shown is a function definition that consists of the following components.

1. Keyword `def` that marks the start of the function header.
2. A function name to uniquely identify the function.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of the function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
7. An optional `return` statement to return a value from the function.

Example of a function:

```
def greet(name):

 """ comment: This function greets to the person passed in as a parameter """

        print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
Hello, Paul. Good morning!
```

Docstrings

The first string after the function header is called the docstring and is short for documentation string.

It is briefly used to explain what a function does.

The return statement

The `return` statement is used to exit a function and go back to the place from where it was called.

Syntax of return:

```
return [expression_list]
```

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the `return` statement itself is not present inside a function, then the function will return the `None` object.

For example:

```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Here, `None` is the returned value since `greet()` directly prints the name and no `return` statement is used.

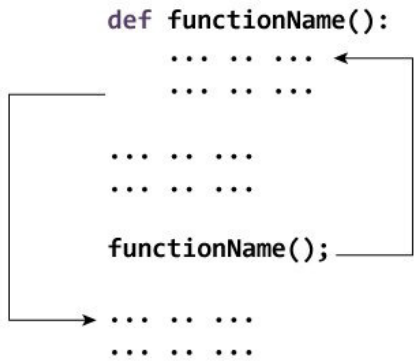
Example of return

```
def absolute_value(num):

"""comment :This function returns the absolute value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num
print(absolute_value(2))
print(absolute_value(-4))
```

Output

```
2
4
```

How Function works in Python?



Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

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

Python Modules

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for example: `example.py`, is called a module, and its module name would be `example.`
We use modules to break down large programs into small manageable and organized files.
Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as **example.py.**

```
# Python Module example

def add(a, b):
  """comment : This program adds two  numbers and return the result"""

  result = a + b
  return result

#addition.py and my.py
```

Here, we have defined a function `add()` inside a module named `example`. The function takes in two numbers and returns their sum.

How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python.
We use the **import** keyword to do this. To import our previously defined module `example`, we type the following in the Python prompt.

```
>>> import example
```

This does not import the names of the functions defined in `example` directly in the current symbol table. It only imports the module name `example` there.

Using the module name we can access the function using the dot . operator. For example:

```
>>> example.add(4,5.5)
9.5
```

Python has tons of standard modules.
You can check out the full list of Python standard modules and their use cases.
These files are in the Lib directory inside the location where you installed Python.
Standard modules can be imported the same way as we import our user-defined modules.
There are various ways to import modules.

Python import statement

We can import a module using the `import` statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example
# to import standard module math

import math
print("The value of pi is", math.pi)
```

When you run the program, the output will be:

```
The value of pi is 3.141592653589793
```

Import with renaming

We can import a module by renaming it as follows:

```
# import module by renaming it

import math as m
```

```
print("The value of pi is", m.pi)
```

We have renamed the `math` module as `m`.
This can save us typing time in some cases.
Note that the name `math` is not recognized in our scope.
Hence, `math.pi` is invalid, and `m.pi` is the correct implementation.

Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

```
# import only pi from math module

from math import pi
print("The value of pi is", pi)

#fromImport.py
```

Here, we imported only the `pi` attribute from the `math` module.
In such cases, we don't use the dot operator. We can also import multiple attributes as follows:

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
```

Import all names

We can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math

from math import *
print("The value of pi is", pi)
```

Here, we have imported all the definitions from the math module.
This includes all names visible in our scope except those beginning with an underscore(private definitions).
Importing everything with the asterisk (*) symbol is not a good programming practice.
This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in `sys.path`. The search is in this order.

- The current directory.
- `PYTHONPATH` **(an environment variable with a list of directories).**
- The installation-dependent default directory.

```
>>> import sys
>>> sys.path
['',
'C:\\Python33\\Lib\\idlelib',
'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs',
'C:\\Python33\\lib',
'C:\\Python33',
'C:\\Python33\\lib\\site-packages']
```

We can add and modify this list to add our own path.

```
>>> import sys
>>> sys.path
['', 'C:\\Python34\\Lib\\idlelib', 'C:\\windows\\system32\\python34.zip', 'C:\\Python34\\DLLs', 'C:\\P
on34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-packages']
>>>
```

The dir() built-in function

We can use the `dir()` function to find out names that are defined inside a module.
For example, we have defined a function `add()` in the module `example` that we had in the beginning.
We can use `dir` in `example` module in the following way:

```
>>> dir(example)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add']

#dir_math.py
```

Here, we can see a sorted list of names (along with `add`). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the `__name__` attribute contains the name of the module.

```
>>> import example
>>> example.__name__
'example'
```

All the names defined in our current namespace can be found out using the `dir()` function without any arguments.

```
>>> a = 1
>>> b = "hello"
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'b', 'math', 'pyscripter']
```

3.1.1 Setting PYTHONPATH, Concepts of Namespace and Scope

What is PYTHONPATH environment variable in Python?

PYTHONPATH is an environment variable which you can set to add additional directories where python will look for modules and packages.

For most installations, you should not set these variables since they are not needed for Python to run.

Python knows where to find its standard library.

The only reason to set PYTHONPATH is to maintain directories of custom Python libraries that you do not want to install in the global default location (i.e., the site-packages directory).

Concepts of Namespace and Scope

In python we deal with variables, functions, libraries and modules etc.

There is a chance the name of the variable you are going to use is already existing as name of another variable or as the name of another function or another method.

In such scenario, we need to learn about how all these names are managed by a python program.

A name in Python is just a way to access a variable like in any other languages.

However, Python is more flexible when it comes to the variable declaration. You can declare a variable by just assigning a name to it.

You can use names to reference values.

```
num = 5
str = 'Z'
seq = [0, 1, 1, 2, 3, 5]
```

You can even assign a name to a function.

```
def function():
 print('It is a function.')

foo = function
foo()
```

You can also assign a name and then reuse it. Check the below example; it is alright for a name to point to different values.

```
test = -1
print("type <test> :=", type(test))
test = "Pointing to a string now"
print("type <test> :=", type(test))
test = [0, 1, 1, 2, 3, 5, 8]
print("type <test> :=", type(test))
```

What are namespaces in Python?

A namespace is a simple system to control the names in a program. It ensures that names are unique and won't lead to any conflict.

Also, add to your knowledge that Python implements namespaces in the form of dictionaries. It maintains a name-to-object mapping where names act as keys and the objects as values. Multiple namespaces may have the same name but pointing to a different variable. Check out a few examples of namespaces for more clarity.

A namespace in Python is a collection of underlying keywords and objects that Python has within memory. It's a very common concept in Object-Oriented Programming.

dictionary : https://www.programiz.com/python-programming/dictionary

To simply put it, a namespace is a collection of names.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs.

This is the reason that built-in functions like `id()`, `print()` etc. are always available to us from any part of the program.

Each module creates its own global namespace.

These different namespaces are isolated. Hence, the same name that may exist in different modules does not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar is the case with class. The following diagram may help to clarify this concept.

In a Python program, there are four types of namespaces:

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

Local Namespace: All the names of the functions and variables declared by a program are held in this namespace. This namespace exists as long as the program runs.

As you learned in 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:

```
>>> def f():
...     print('Start f()')
...
...     def g():
...         print('Start g()')
...         print('End g()')
...         return
...
...     g()
...
...     print('End f()')
...     return
...

>>> f()
Start f()
Start g()
End g()
End 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.

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.

The interpreter also creates a global namespace for any **module** that your program loads with the `import` statement.

This namespace holds all the names of functions and other variables that are included in the modules being used in the python program. It encompasses all the names that are part of the Local namespace.

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.

This is the highest level of namespace . It encompasses Global Namespace which in turn encompasses the local namespace.

example: dir(__builtins__)

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeE
ror', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'Envi
onmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'Future
Warning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'I
nterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', '
NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'Ove
flowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'ReferenceError'
'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopIteration', 'SyntaxError', 'SyntaxWarning'
'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'Uni
codeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning
, 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__build_class__', '__de
bug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'ar
y', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyrigl
t', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'froze
nset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'l
n', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'p
rint', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
>>>
```

note :

[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**.]

Python Variable Scope

The namespace has a lifetime when it is available. That is also called the scope. Also the scope will depend on the coding region where the variable or object is located. You can see in the below program how the variables declared in an inner loop are available to the outer loop but not vice-versa.

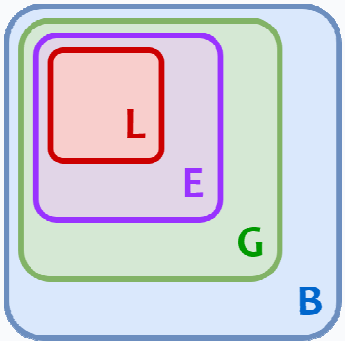
Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.

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.



Python searches for 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.

Example of Scope and Namespace in Python

```
5. def outer_function():
6.     b = 20
7.     def inner_func():
8.         c = 30
9. a = 10
```

Here, the variable a is in the global namespace.
Variable b is in the local namespace of `outer_function()` and c is in the nested local namespace of `inner_function()`.
When we are in `inner_function()`, c is local to us, b is nonlocal and a is global.
We can read as well as assign new values to c but can only
read b and a from `inner_function()`.
If we try to assign as a value to b, a new variable b is created in the local namespace which is different than the nonlocal b.
The same thing happens when we assign a value to a.
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()
    print('a =', a)

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

As you can see, the output of this program is

```
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)
```

The output of the program is.

```
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:

- Create a new folder named D:\MyApp.
- Inside MyApp, create a subfolder with the name 'mypackage'.
- 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

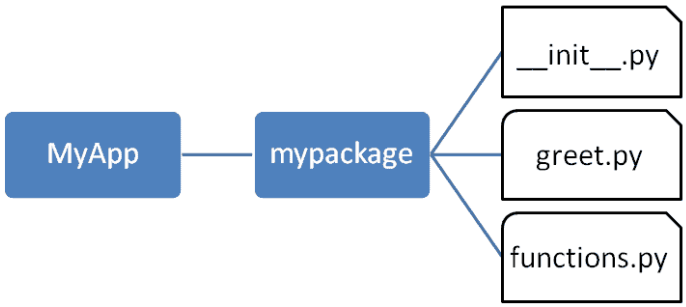
```
def SayHello(name):
    print("Hello ", name)
```

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:



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
```

note : https://realpython.com/python-modules-packages/

video : https://youtu.be/urE5MuYd-YM

__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.

3.2 Importing sqlite3 module

SQLite in general is a server-less database that you can use within almost all programming languages including Python. Server-less means there is no need to install a separate server to work with SQLite so you can connect directly with the database.

SQLite is a lightweight database that can provide a relational database management system with zero-configuration because there is no need to configure or set up anything to use it.

To use sqlite3 module, you must first create a connection object that represents the database and then optionally you can create a cursor object, which will help you in executing all the SQL statements.

Python SQLite Database Connection

Use the following steps to connect to SQLite

1. **Import sqlite3 module**
   `import sqlite3` statement imports the sqlite3 module in the program.
   Using the classes and methods defined in the sqlite3 module we can communicate with the SQLite database.

2. **Use the connect() method**
   Use the `connect()` method of the `connector` class with the database name.
   To establish a connection to SQLite, you need to pass the database name you want to connect.
   If you specify the database file name that already presents on the disk, it will connect to it.
   But if your specified SQLite database file doesn't exist, SQLite creates a new database for you.
   This method returns the SQLite Connection Object if the connection is successful.

3. **Use the cursor() method**
   Use the `cursor()` method of a connection class to create a cursor object to execute SQLite command/queries from Python.

Cursor Object

It is an object that is used to make the connection for executing SQL queries. It acts as middleware between SQLite database connection and SQL query. It is created after giving connection to SQLite database.

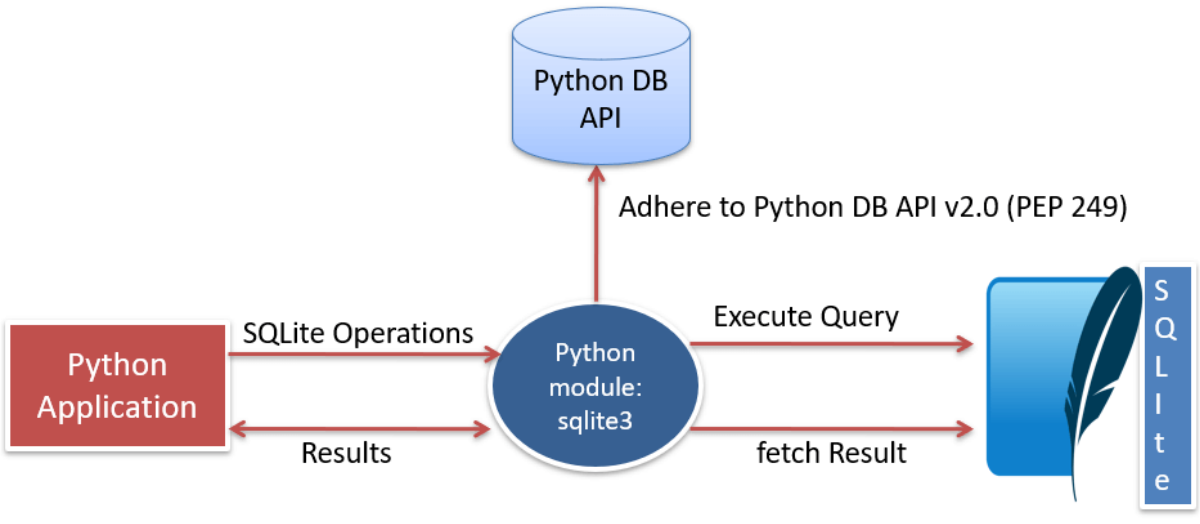
Syntax: *cursor_object=connection_object.execute("sql query");*

4. **Use the execute() method**
   The execute() methods run the SQL query and return the result.

5. **Extract result using fetchall()**
   Use `cursor.fetchall()` or `fetchone()` or `fetchmany()` to read query result.

6. **Close cursor and connection objects**
   use `cursor.clsoe()` and `connection.clsoe()` method to close the cursor and SQLite connections after your work completes

7. **Catch database exception if any that may occur during this connection process.**



3.2.1 connect () and execute() methods.

sqlite3.connect(database [,timeout ,other optional arguments])

This API opens a connection to the SQLite database file. You can use ":memory:" to open a database connection to a database that resides in RAM instead of on disk. If database is opened successfully, it returns a connection object.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The timeout parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

If the given database name does not exist then this call will create the database. You can specify filename with the required path as well if you want to create a database anywhere else except in the current directory.

program : connection.py

connection.execute(sql [, optional parameters])

This routine is a shortcut of the above execute method provided by the cursor object and it creates an intermediate cursor object by calling the cursor method, then calls the cursor's execute method with the parameters given.

- Execute the query using a `con.execute(query)`
    - **program : create_table.py**

3.2.2 Single row and multi-row fetch (fetchone(), fetchall())

The sqlite3.Cursor class is an instance using which you can invoke methods that execute SQLite statements, fetch data from the result sets of the queries. You can create **Cursor** object using the cursor() method of the Connection object/class.

Example

```
import sqlite3

#Connecting to sqlite
conn = sqlite3.connect('example.db')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()
```

Methods

Following are the various methods provided by the Cursor class/object.

Sr.No	Method & Description
1	execute() This routine executes an SQL statement. The SQL statement may be parameterized (i.e., placeholders instead of SQL literals). The psycopg2 module supports placeholder using %s sign For example:cursor.execute("insert into people values (%s, %s)", (who, age))
2	executemany() This routine executes an SQL command against all parameter sequences or mappings found in the sequence sql.
3	fetchone() cursor.fetchone() method returns a single record or None if no more rows are available. OR This method fetches the next row of a query result set, returning a single sequence, or None when no more data is available. Program : Fetch_one.py
4	fetchmany() This routine fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available. The method tries to fetch as many rows as indicated by the size parameter.

5	fetchall() This routine fetches all (remaining) rows of a query result, returning a list. An empty list is returned when no rows are available. Program : Fetch_all.py

Fetch all rows from database table using cursor's fetchall()

Now, let see how to use `fetchall` to fetch all the records. To fetch all rows from a database table, you need to follow these simple steps: –

- Create a database Connection from Python. Refer Python SQLite connection, Python MySQL connection, Python PostgreSQL connection.
- Define the SELECT query. Here you need to know the table and its column details.
- Execute the SELECT query using the `cursor.execute()` method.
- Get resultSet (all rows) from the cursor object using a `cursor.fetchall()`.
- Iterate over the ResultSet using `for` loop and get column values of each row.
- Close the Python database connection.
- Catch any SQL exceptions that may come up during the process.
-

3.2.3 Select, Insert, update, delete using execute () method.

- **insert_record.py**
- **select.py**
- **update.py**

3.2.4 commit () method.