CHAPTER 5

Advance Function Topics

Lambda Functions

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know. We can also use the lambda keyword in Python to define an unnamed function.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Example

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

Output

When the above code is executed, it produces the following result −

```
Value of total : 30
Value of total : 40
```

Difference Between Lambda functions and def defined function are as follows:

Let’s look at this example and try to understand the difference between a normal def defined function and lambda function. This is a program that returns the cube of a given value:

```
# Python code to illustrate cube of a number
# showing difference between def() and lambda().
def cube(y):
    return y*y*y
lambda_cube = lambda y: y*y*y
# using the normally
# defined function
print(cube(5))
# using the lambda function
```

```
print(lambda_cube(5))
```

Output

```
125
125
```

As we can see in the above example both the cube() function and lambda_cube() function behave the same and as intended. Let's analyze the above example a bit more:

- **Without using Lambda:** Here, both of them return the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the *return* keyword.
- **Using Lambda:** Lambda definition does not include a "return" statement, it always contains an expression that is returned. We can also put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all. This is the simplicity of lambda functions.

Map() function

Map in Python is a function that works as an iterator to return a result after applying a function to every item of an iterable (tuple, lists, etc.). It is used when you want to apply a single transformation function to all the iterable elements. The iterable and function are passed as arguments to the map in Python.

Syntax of Map in Python

```
map(function, iterables)
```

In the above syntax:

- function: It is the transformation function through which all the items of the iterable will be passed.
- iterables: It is the iterable (sequence, collection like list or tuple) that you want to map.

Let's look at an example to understand the syntax of the map in Python better. The code below creates a list of numbers and then uses the Python map() function to multiply each item of the list with itself.

```
# Defining a function
def mul(i):
   return i * i
# Using the map function
x = map(mul, (3, 5, 7, 11, 13))
print (x)
print(list(x))
```

Output:

```
<map object at 0x7f7e02849ca0>
[9, 25, 49, 121, 169]
>
```

As you

Workings of the Python Map() Function

The map in Python takes a function and an iterable/iterables. It loops over each item of an iterable and applies the transformation function to it. Then, it returns a map object that stores the value of the transformed item. The input function can be any callable function, including built-in functions, lambda functions, user-defined functions, classes, and methods. Now, calculate the same multiplication as you did in the previous example, but this time with both the for loop and the map functions separately, to understand how the map() function works.

Example: Using For Loop

```
num = [3, 5, 7, 11, 13]
mul = []
for n in num:
    mul.append(n ** 2)
print (mul)
```

Output:

```
[9, 25, 49, 121, 169]
>
```

Example: Using the Python Map() Function

```
def mul(i):
    return i * i
num = (3, 5, 7, 11, 13)
resu = map(mul, num)
print(resu)
# making the map object readable
mul_output = list(resu)
print(mul_output)
```

Output:

```
<map object at 0x7fc5a85e1ca0>
[9, 25, 49, 121, 169]
>
```

As you can see, the map() function iterates through the iterable, just like the for loop. Once the iteration is complete, it returns the map object. You can then convert the map object to a list and print it. You would have also noticed that this time while using the map in Python, you defined the iterable separately and then passed it to the map() function. Thus, you can either define the iterable within the map() function or separately.

Using Map in Python with Lambda Functions

One of the most common use cases of the map function you might encounter is with the lambda functions. Lambda functions are anonymous functions, i.e., without any name. The lambda function

allows you to define the function right inside the map() function. Let's look at an example to see how the Python map() function is used with the lambda functions.

Example: Using Map() with Lambda Functions

In the code mentioned below, you will use a lambda function to double the numbers in the iterable using the map function.

```
num = (6, 9, 21, 44)
resu = map(lambda i: i + i, num)
print(list(resu))
```

Output:

```
[12, 18, 42, 88]
>
```

filter()

The filter() function in Python filters elements from an iterable based on a given condition or function and returns a new iterable with the filtered elements.

The syntax for the filter() is as follows: filter(function, iterable)

Here also, the first argument passed to the filter function is itself a function, and the second argument passed is an iterable (sequence of elements) such as a list, tuple, set, string, etc.

Example 1 - usage of the filter():

You are given a list of integers and should filter the even numbers from the list.

```
# Using filter() to filter even numbers from a list
data = [1, 2, 3, 4, 5]
# The filter function filters the even numbers from the data
# and returns a filter object (an iterable)
evens = filter(lambda x: x % 2 == 0, data)
# Iterating the values of evens
for i in evens:
    print(i, end=" ")
# We can convert the filter object into a list as follows:
evens = list(filter(lambda x: x % 2 == 0, data))
# Printing the evens list
print(f"Evens = {evens}")
```

Output:

2 4

Evens = [2, 4]

reduce()

In Python, reduce() is a built-in function that applies a given function to the elements of an iterable, reducing them to a single value.

The syntax for reduce() is as follows: reduce(function, iterable[, initializer])

The function argument is a function that takes two arguments and returns a single value. The first argument is the accumulated value, and the second argument is the current value from the iterable.

The iterable argument is the sequence of values to be reduced.

The optional initializer argument is used to provide an initial value for the accumulated result. If no initializer is specified, the first element of the iterable is used as the initial value.

Here's an example that demonstrates how to use reduce() to find the sum of a list of numbers:

Example 1:

You are given a list containing some integers and you should find the sum of all elements in the list using reduce function. Below is the solution of the problem:

```
# Examples to understand the reduce() function

from functools import reduce

# Function that returns the sum of two numbers

def add(a, b):

   return a + b

# Our Iterable

num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# add function is passed as the first argument, and num_list is passed as the second argument

sum = reduce(add, num_list)

print(f"Sum of the integers of num_list : {sum}")

# Passing 10 as an initial value

sum = reduce(add, num_list, 10)

print(f"Sum of the integers of num_list with initial value 10 : {sum}")
```

Output

```
Sum of the integers of num_list : 55

Sum of the integers of num_list with initial value 10 : 65
```

Using Lambda Function with List Comprehension

We'll apply the lambda function combined with list comprehension and lambda keyword with a for loop in this instance. We'll attempt to print the square of numbers in the range 0 to 11.

Code

```
1. #Code to calculate square of each number of list using list comprehension
2. squares = [lambda num = num: num ** 2 for num in range(0, 11)]
3. for square in squares:
4.     print( square(), end = " ")
```

Output:

```
0 1 4 9 16 25 36 49 64 81 100
```

Using Lambda Function with if-else

We will use the lambda function with the if-else block.

Code

```
# Code to use lambda function with if-else
Minimum = lambda x, y : x if (x < y) else y
print(Minimum( 35, 74 ))
```

Output:

```
35
```

Using Lambda with Multiple Statements

Multiple expressions are not allowed in lambda functions, but we can construct 2 lambda functions or more and afterward call the second lambda expression as an argument to the first. Let's use lambda to discover the third maximum element.

Code

```
# Code to print the third-largest number of the given list using the lambda function
my_List = [ [3, 5, 8, 6], [23, 54, 12, 87], [1, 2, 4, 12, 5] ]
# sorting every sublist of the above list
sort_List = lambda num : ( sorted(n) for n in num )
# Getting the third largest number of the sublist
third_Largest = lambda num, func : [ l[ len(l) - 2] for l in func(num)]
result = third_Largest( my_List, sort_List)
print( result )
```

Output:

```
[6, 54, 5]
```

modules in Python?

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.

A module is a file containing definition of functions, classes, variables, constants or any other Python object

Creating Module.

Creating a module is nothing but saving a Python code with the help of any editor. Let us save the following code as mymodule.py

```
def sum(x,y):
  return x+y
def average(x,y):
  return (x+y)/2
def power(x,y):
  return x**y
```

The import mymodule statement loads all the functions in this module in the current namespace. Each function in the imported module is an attribute of this module object.

To call any function, use the module object's reference. For example, mymodule.sum().

```
import mymodule
print ("sum:",mymodule.sum(10,20))
print ("average:",mymodule.average(10,20))
print ("power:",mymodule.power(10, 2))
```

It will produce the following output −

```
sum:30
average:15.0
power:100
```

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 `mymodule`, we type the following in the Python prompt.

```
>>> import mymodule
```

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

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

```
>>> mymoudle.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. They are listed below..

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)
Run Code
```

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.

What is pip?

`pip` is the standard package manager for Python. We can use `pip` to install additional packages that are not available in the Python standard library. For example,

```
pip install numpy
```

If we had installed pip on our system, this command would have installed the numpy library.

How to install pip?

pip comes pre-installed on the Python versions 3.4 or older. We can check if pip is installed by using the following command in the console:

```
pip --version
```

If pip is already available in the system, the respective pip version is displayed, like:

```
pip 19.3.1 from C:\Python37\lib\site-packages\pip (python 3.7)
```

If we are using an older version of Python or do not have pip installed for some other reason, follow the steps as described in this link: pip installation

Using pip

pip is a command-line program. After its installation, a pip command is added which can be used with the command prompt.

The basic syntax of pip is:

```
pip <pip arguments>
```

Installing Packages with pip

Apart from the standard Python library, the Python community contributes to an extensive number of packages tailored for various development frameworks, tools, and libraries.

Most of these packages are officially hosted and published to the Python Package Index(PyPI). pip allows us to download and install these packages.

Basic Package Installation

The install command used to install packages using pip. Let's take an example:

Suppose we want to install requests, a popular HTTP library for Python. We can do it with the help of the following command.

```
pip install requests
```

Output

```
Collecting requests
```

```
  Using cached
https://files.pythonhosted.org/packages/51/bd/23c926cd341ea6b7dd0b2a00aba99ae0f828be89d72b2190f27c11d4b7fb/requests-2.22.0-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2
  Using cached
https://files.pythonhosted.org/packages/bc/a9/01ffebfb562e4274b6487b4bb1ddec7ca55ec7510b22e4c51f14098443b8/chardet-3.0.4-py2.py3-none-any.whl
Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1
  Using cached
https://files.pythonhosted.org/packages/b4/40/a9837291310ee1ccc242ceb6ebfd9eb21539649f193a7c8c86ba15b98539/urllib3-1.25.7-py2.py3-none-any.whl
Collecting idna<2.9,>=2.5
  Using cached
https://files.pythonhosted.org/packages/14/2c/cd551d81dbe15200be1cf41cd03869a46fe7226e7450af7a6545bfc474c9/idna-2.8-py2.py3-none-any.whl
Collecting certifi>=2017.4.17
  Downloading
https://files.pythonhosted.org/packages/b9/63/df50cac98ea0d5b006c55a399c3bf1db9da7b5a24de7890bc9cfd5dd9e99/certifi-2019.11.28-py2.py3-none-any.whl (156kB)
Installing collected packages: chardet, urllib3, idna, certifi, requests
Successfully installed certifi-2019.11.28 chardet-3.0.4 idna-2.8 requests-2.22.0 urllib3-1.25.7
```

Here, we can see that the `pip` has been used with the `install` command followed by the name of the package we want to install (`requests`).

All other dependencies like `chardet`, `urllib3` and `certifi` required for this package are also installed by `pip`.

Specifying Package Version

When `pip install` is used in its minimal form, `pip` downloads the most recent version of the package.

Sometimes, only a specific version is compatible with other programs. So, we can define the version of the package in the following way:

```
pip install requests==2.21.0
```

Here, we have installed the

Here, we have installed the `2.11.0` version of the `requests` library.

Listing Installed Packages with pip

The `pip list` command can be used to list all the available packages in the current Python environment.

```
pip list
```

Output

```
Package    Version
---------- ----------
certifi    2019.11.28
chardet    3.0.4
idna       2.8
pip        19.3.1
requests   2.22.0
setuptools 45.0.0
urllib3    1.25.7
wheel      0.33.6
```

Package Information with pip show

The `pip show` command displays information about one or more installed packages. Let's look at an example:

```
pip show requests
```

Output

```
Name: requests
Version: 2.22.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: c:\users\dell\desktop\venv\lib\site-packages
Requires: certifi, chardet, urllib3, idna
Required-by:
```

```
```

Here, the `show` command displays information about the `requests` library. Notice the **Requires** and **Required-by** column in the above output.

Requires column shows which dependencies the `requests` library requires. And, **Required-by** column shows the packages that require `requests`.

Uninstalling a Package with pip

We can uninstall a package by using `pip` with the `pip uninstall` command.

Suppose we want to remove the `requests` library from our current Python environment. We can do it in the following way:

```
pip uninstall requests
```

Output

```
Uninstalling requests-2.22.0:
  Would remove:
    C:\Python37\lib\site-packages\requests-2.22.0.dist-info\*
    C:\Python37\lib\site-packages\requests\*
Proceed (y/n)? y
  Successfully uninstalled requests-2.22.0
```

As we can see, the `requests` package is removed after the final prompt.

Note: Even though the specified package is removed, the packages that were installed as dependencies are not removed. In this case, the dependencies (`chardet`, `urllib3`, and `certifi`) of the `requests` library aren't uninstalled.

If we need to remove the dependencies of a package as well, we can use the `pip show` command to view installed packages and remove them manually.

Using Requirement Files

A file containing all the package names can also be used to install Python packages in batches.

Let's take a look at an example:

Suppose we have a file **requirements.txt** which has the following entries:

```
numpy
```

```
Pillow
pygame
```

We can install all these packages and their dependencies by using a single command in pip.

```
pip install -r requirements.txt
```

Output

```
Collecting numpy
  Using cached
https://files.pythonhosted.org/packages/a9/38/f6d6d8635d496d6b4ed5d8ca4b9f193d0edc59999c3a6
3779cbc38aa650f/numpy-1.18.1-cp37-cp37m-win_amd64.whl
Collecting Pillow
  Using cached
https://files.pythonhosted.org/packages/88/6b/66f502b5ea615f69433ae1e23ec786b2cdadbe41a5cfb
1e1fabb4f9c6ce9/Pillow-7.0.0-cp37-cp37m-win_amd64.whl
Collecting pygame
  Using cached
https://files.pythonhosted.org/packages/ed/56/b63ab3724acff69f4080e54c4bc5f55d1fbdeeb19b92b7
0acf45e88a5908/pygame-1.9.6-cp37-cp37m-win_amd64.whl
Installing collected packages: numpy, Pillow, pygame
Successfully installed Pillow-7.0.0 numpy-1.18.1 pygame-1.9.6
```

Here, we have used the same install command with pip.

However, the additional argument -r specifies pip that we are passing a requirements file rather than a package name.

Creating Requirements File

As an alternative to manually creating the requirements file, pip offers the freeze command. Let's look at how to use this command.

Suppose our current Python environment has the following packages. It can be displayed using pip list.

```
Package    Version
---------- -------
numpy      1.17.0
Pillow     6.1.0
pip        19.3.1
```

```
pygame     1.9.6
setuptools 45.0.0
wheel      0.33.6
```

The packages that don't come preinstalled will Python are listed using the `freeze` command.

```
pip freeze
```

Output

```
numpy==1.17.0
Pillow==6.1.0
pygame==1.9.6
```

The `pip freeze` command displays the packages and their version in the format of the requirements file.

So this output can be redirected to create a requirements file using the following command:

```
pip freeze > requirements.txt
```

A new **requirements.txt** file is created in the working directory. It can later be used in other Python environments to install specific versions of packages.

Search packages in pip

The `search` command is used to search for packages in the command prompt. Let's look at an example:

```
pip search pygame
```

Output

```
pygame-anisprite (1.0.0)          - Animated sprites for PyGame!
pygame-ai (0.1.2)                 - Videogame AI package for PyGame
pygame-engine (0.0.6)             - Simple pygame game engine.
pygame-assets (0.1)               - Assets manager for Pygame apps
pygame-gui (0.4.2)                - A GUI module for pygame 2
pygame-spritesheet (0.2.0)        - Python pygame extension that provides SpriteSheet class.
pygame-minesweeper (1.0)          - Minesweeper game implemented in python using pygame
pygame-menu (2.1.0)               - A menu for pygame, simple, lightweight and easy to use
pygame-plot (0.1)                 - Quick visualization of data using pygame with a matplotlib style
pygame (1.9.6)                    - Python Game Development
```

...

Here, we have searched for a library called pygame. All other packages that match the keyword are displayed. This command is helpful for finding related packages.

MYSQL with Python

To build the real world applications, connecting with the databases is the necessity for the programming languages. However, python allows us to connect our application to the databases like MySQL, SQLite, MongoDB, and many others.

In this section of the tutorial, we will discuss Python - MySQL connectivity, and we will perform the database operations in python. We will also cover the Python connectivity with the databases like MongoDB and SQLite later in this tutorial.

Install mysql.connector

To connect the python application with the MySQL database, we must import the mysql.connector module in the program.

The mysql.connector is not a built-in module that comes with the python installation. We need to install it to get it working.

Execute the following command to install it using pip installer.

1. > python -m pip install mysql-connector

Or follow the following steps.

1. Click the link:

https://files.pythonhosted.org/packages/8f/6d/fb8ebcbbaee68b172ce3dfd08c7b8660d09f91d8d5411298bcacbd309f96/mysql-connector-python-8.0.13.tar.gz to download the source code.

2. Extract the archived file.

3. Open the terminal (CMD for windows) and change the present working directory to the source code directory.

1. $ cd mysql-connector-python-8.0.13/

4. Run the file named setup.py with python (python3 in case you have also installed python 2) with the parameter build.

1. $ python setup.py build

5. Run the following command to install the mysql-connector.

1. $ python setup.py install

This will take a bit of time to install mysql-connector for python. We can verify the installation once the process gets over by importing mysql-connector on the python shell.

Database Connection

In this section of the tutorial, we will discuss the steps to connect the python application to the database.

There are the following steps to connect a python application to our database.

1. Import mysql.connector module
2. Create the connection object.
3. Create the cursor object
4. Execute the query

Creating the connection

To create a connection between the MySQL database and the python application, the connect() method of mysql.connector module is used.

Pass the database details like HostName, username, and the database password in the method call. The method returns the connection object.

The syntax to use the connect() is given below.

1. Connection-Object= mysql.connector.connect(host = <host-name> , user = <username> , passwd = <password>)

Consider the following example.

Example

```
1. import mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
4. #printing the connection object
5. print(myconn)
```

Output:

```
<mysql.connector.connection.MySQLConnection object at 0x7fb142edd780>
```

Here, we must notice that we can specify the database name in the connect() method if we want to connect to a specific database.

Example

```
1. import mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google", dat
   abase = "mydb")
4. #printing the connection object
```

5. **print**(myconn)

Output:

```
<mysql.connector.connection.MySQLConnection object at 0x7ff64aa3d7b8>
```

Creating a cursor object

The cursor object can be defined as an abstraction specified in the Python DB-API 2.0. It facilitates us to have multiple separate working environments through the same connection to the database. We can create the cursor object by calling the 'cursor' function of the connection object. The cursor object is an important aspect of executing queries to the databases.

The syntax to create the cursor object is given below.

1. <my_cur> = conn.cursor()

Example

```
1. import mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google", dat
   abase = "mydb")
4.
5. #printing the connection object
6. print(myconn)
7.
8. #creating the cursor object
9. cur = myconn.cursor()
10.
11. print(cur)
```

Creating new databases

In this section of the tutorial, we will create the new database PythonDB.

Getting the list of existing databases

We can get the list of all the databases by using the following MySQL query.

1. > show databases;

Example

```
1. import mysql.connector
2.
3. #Create the connection object
```

```
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.     dbs = cur.execute("show databases")
11. except:
12.     myconn.rollback()
13. for x in cur:
14.     print(x)
15. myconn.close()
```

Output:

```
('EmployeeDB',)
('Test',)
('TestDB',)
('information_schema',)
('javatpoint',)
('javatpoint1',)
('mydb',)
('mysql',)
('performance_schema',)
('testDB',)
```

Creating the new database

The new database can be created by using the following SQL query.

```
1. >  create database <database-name>
```

Example

```
1. import mysql.connector
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
```

```
8.
9.  try:
10.     #creating a new database
11.     cur.execute("create database PythonDB2")
12.
13.     #getting the list of all the databases which will now include the new database PythonDB
14.     dbs = cur.execute("show databases")
15.
16. except:
17.     myconn.rollback()
18.
19. for x in cur:
20.         print(x)
21.
22. myconn.close()
```

Output:

```
('EmployeeDB',)
('PythonDB',)
('Test',)
('TestDB',)
('anshika',)
('information_schema',)
('javatpoint',)
('javatpoint1',)
('mydb',)
('mydb1',)
('mysql',)
('performance_schema',)
('testDB',)
```

Creating the table

In this section of the tutorial, we will create the new table Employee. We have to mention the database name while establishing the connection object.

We can create the new table by using the CREATE TABLE statement of SQL. In our database PythonDB, the table Employee will have the four columns, i.e., name, id, salary, and department_id initially.

The following query is used to create the new table Employee.

```
> create table Employee (name varchar(20) not null, id int primary key, salary float not null , Dept_Id int not null)
```

Example

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

#creating the cursor object
cur = myconn.cursor()

try:
    #Creating a table with name Employee having four columns i.e., name, id, salary, and department id
    dbs = cur.execute("create table Employee(name varchar(20) not null, id int(20) not null primary key, salary float not null, Dept_id int not null)")
except:
    myconn.rollback()

myconn.close()
```

Now, we may check that the table Employee is present in the database.

Alter Table

Sometimes, we may forget to create some columns, or we may need to update the table schema. The alter statement used to alter the table schema if required. Here, we will add the column branch_name to the table Employee. The following SQL query is used for this purpose.

```
alter table Employee add branch_name varchar(20) not null
```

Consider the following example.

Example

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

```

```
#creating the cursor object
cur = myconn.cursor()

try:
    #adding a column branch name to the table Employee
    cur.execute("alter table Employee add branch_name varchar(20) not null")
except:
    myconn.rollback()

myconn.close()
```

Insert Operation

Adding a record to the table

The **INSERT INTO** statement is used to add a record to the table. In python, we can mention the format specifier (%s) in place of values.

We provide the actual values in the form of tuple in the execute() method of the cursor.

Consider the following example.

Example

```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")
#creating the cursor object
cur = myconn.cursor()
sql = "insert into Employee(name, id, salary, dept_id, branch_name) values (%s, %s, %s, %s, %s)"

#The row values are provided in the form of tuple
val = ("John", 110, 25000.00, 201, "Newyork")

try:
    #inserting the values into the table
    cur.execute(sql,val)

    #commit the transaction
```

```
    myconn.commit()

except:
    myconn.rollback()

print(cur.rowcount,"record inserted!")
myconn.close()
```

Output:

```
1 record inserted!
```

Insert multiple rows

We can also insert multiple rows at once using the python script. The multiple rows are mentioned as the list of various tuples.

Each element of the list is treated as one particular row, whereas each element of the tuple is treated as one particular column value (attribute).

Consider the following example.

Example

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

#creating the cursor object
cur = myconn.cursor()
sql = "insert into Employee(name, id, salary, dept_id, branch_name) values (%s, %s, %s, %s, %s)"
val = [("John", 102, 25000.00, 201, "Newyork"),("David",103,25000.00,202,"Port of spain"),("Nick",104,90000.00,201,"Newyork")]

try:
    #inserting the values into the table
    cur.executemany(sql,val)

    #commit the transaction
    myconn.commit()
    print(cur.rowcount,"records inserted!")
```

```
18.
19. except:
20.     myconn.rollback()
21.
22. myconn.close()
```

Output:

```
3 records inserted!
```

Row ID

In SQL, a particular row is represented by an insertion id which is known as row id. We can get the last inserted row id by using the attribute lastrowid of the cursor object.

Consider the following example.

Example

```
1. import mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
   base = "PythonDB")
4. #creating the cursor object
5. cur = myconn.cursor()
6.
7. sql = "insert into Employee(name, id, salary, dept_id, branch_name) values (%s, %s, %s, %s
   , %s)"
8.
9. val = ("Mike",105,28000,202,"Guyana")
10.
11. try:
12.     #inserting the values into the table
13.     cur.execute(sql,val)
14.
15.     #commit the transaction
16.     myconn.commit()
17.
18.     #getting rowid
19.     print(cur.rowcount,"record inserted! id:",cur.lastrowid)
20.
21. except:
22.     myconn.rollback()
```

```
23.
24. myconn.close()
```

Output:

```
1 record inserted! Id: 0
```

Read Operation

The SELECT statement is used to read the values from the databases. We can restrict the output of a select query by using various clause in SQL like where, limit, etc.

Python provides the fetchall() method returns the data stored inside the table in the form of rows. We can iterate the result to get the individual rows.

In this section of the tutorial, we will extract the data from the database by using the python script. We will also format the output to print it on the console.

Example

```
1. import mysql.connector
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
   base = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Reading the Employee data
11.    cur.execute("select * from Employee")
12.
13.    #fetching the rows from the cursor object
14.    result = cur.fetchall()
15.    #printing the result
16.
17.    for x in result:
18.        print(x);
19. except:
20.    myconn.rollback()
21.
22. myconn.close()
```

Output:

1.9M

58

FOUR almighty pillars of OOP that can improve your PHP skills (from procedural to basics of OOP)

```
('John', 101, 25000.0, 201, 'Newyork')
('John', 102, 25000.0, 201, 'Newyork')
('David', 103, 25000.0, 202, 'Port of spain')
('Nick', 104, 90000.0, 201, 'Newyork')
('Mike', 105, 28000.0, 202, 'Guyana')
```

Reading specific columns

We can read the specific columns by mentioning their names instead of using star (*).

In the following example, we will read the name, id, and salary from the Employee table and print it on the console.

Example

```
1. import mysql.connector
2. #Create the connection object
3. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",databas
   e = "PythonDB")
4. #creating the cursor object
5. cur = myconn.cursor()
6. try:
7.     #Reading the Employee data
8.     cur.execute("select name, id, salary from Employee")
9.
10.    #fetching the rows from the cursor object
11.    result = cur.fetchall()
12.    #printing the result
13.    for x in result:
14.        print(x);
15. except:
16.    myconn.rollback()
17. myconn.close()
```

Output:

```
('John', 101, 25000.0)
```

```
('John', 102, 25000.0)
('David', 103, 25000.0)
('Nick', 104, 90000.0)
('Mike', 105, 28000.0)
```

The fetchone() method

The fetchone() method is used to fetch only one row from the table. The fetchone() method returns the next row of the result-set.

Consider the following example.

Example

```
1. import mysql.connector
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
   base = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.     #Reading the Employee data
11.     cur.execute("select name, id, salary from Employee")
12.
13.     #fetching the first row from the cursor object
14.     result = cur.fetchone()
15.
16.     #printing the result
17.     print(result)
18.
19. except:
20.     myconn.rollback()
21.
22. myconn.close()
```

Output:

```
('John', 101, 25000.0)
```

Formatting the result

We can format the result by iterating over the result produced by the fetchall() or fetchone() method of cursor object since the result exists as the tuple object which is not readable.

Consider the following example.

Example

```
1. import mysql.connector
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
   base = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.
11.    #Reading the Employee data
12.    cur.execute("select name, id, salary from Employee")
13.
14.    #fetching the rows from the cursor object
15.    result = cur.fetchall()
16.
17.    print("Name    id    Salary");
18.    for row in result:
19.        print("%s    %d    %d"%(row[0],row[1],row[2]))
20. except:
21.    myconn.rollback()
22.
23. myconn.close()
```

Output:

```
Name    id    Salary
John    101    25000
John    102    25000
David    103    25000
```

```
        Nick    104    90000
Mike    105    28000
```

Using where clause

We can restrict the result produced by the select statement by using the where clause. This will extract only those columns which satisfy the where condition.

Consider the following example.

Example: printing the names that start with j

```
1.  import mysql.connector
2.
3.  #Create the connection object
4.  myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
    base = "PythonDB")
5.
6.  #creating the cursor object
7.  cur = myconn.cursor()
8.
9.  try:
10.     #Reading the Employee data
11.     cur.execute("select name, id, salary from Employee where name like 'J%'")
12.
13.     #fetching the rows from the cursor object
14.     result = cur.fetchall()
15.
16.     print("Name    id    Salary");
17.
18.     for row in result:
19.         print("%s    %d    %d"%(row[0],row[1],row[2]))
20. except:
21.     myconn.rollback()
22.
23. myconn.close()
```

Output:

```
Name    id    Salary
John    101    25000
```

```
John    102    25000
```

Example: printing the names with id = 101, 102, and 103

```
1.  import mysql.connector
2.
3.  #Create the connection object
4.  myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
    base = "PythonDB")
5.
6.  #creating the cursor object
7.  cur = myconn.cursor()
8.
9.  try:
10.     #Reading the Employee data
11.     cur.execute("select name, id, salary from Employee where id in (101,102,103)")
12.
13.     #fetching the rows from the cursor object
14.     result = cur.fetchall()
15.
16.     print("Name    id    Salary");
17.
18.     for row in result:
19.         print("%s    %d    %d"%(row[0],row[1],row[2]))
20. except:
21.     myconn.rollback()
22.
23. myconn.close()
```

Output:

```
Name    id    Salary
John    101    25000
John    102    25000
David    103    2500
```

Ordering the result

The ORDER BY clause is used to order the result. Consider the following example.

Example

```
1. import mysql.connector
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
   base = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Reading the Employee data
11.    cur.execute("select name, id, salary from Employee order by name")
12.
13.    #fetching the rows from the cursor object
14.    result = cur.fetchall()
15.
16.    print("Name    id    Salary");
17.
18.    for row in result:
19.        print("%s    %d    %d"%(row[0],row[1],row[2]))
20. except:
21.    myconn.rollback()
22.
23. myconn.close()
```

Output:

```
Name    id    Salary
David   103    25000
John    101    25000
John    102    25000
Mike    105    28000
Nick    104    90000
```

Order by DESC

This orders the result in the decreasing order of a particular column.

Example

```
1. import mysql.connector
2.
3. #Create the connection object
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
   base = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Reading the Employee data
11.    cur.execute("select name, id, salary from Employee order by name desc")
12.
13.    #fetching the rows from the cursor object
14.    result = cur.fetchall()
15.
16.    #printing the result
17.    print("Name    id    Salary");
18.    for row in result:
19.        print("%s    %d    %d"%(row[0],row[1],row[2]))
20.
21. except:
22.    myconn.rollback()
23.
24. myconn.close()
```

Output:

```
Name    id    Salary
Nick    104    90000
Mike    105    28000
John    101    25000
John    102    25000
David    103    25000
```

Update Operation

The UPDATE-SET statement is used to update any column inside the table. The following SQL query is used to update a column.

```
> update Employee set name = 'alex' where id = 110
```

Consider the following example.

Example

```
import mysql.connector

#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",database = "PythonDB")

#creating the cursor object
cur = myconn.cursor()

try:
    #updating the name of the employee whose id is 110
    cur.execute("update Employee set name = 'alex' where id = 110")
    myconn.commit()
except:

    myconn.rollback()

myconn.close()
```

Delete Operation

The DELETE FROM statement is used to delete a specific record from the table. Here, we must impose a condition using WHERE clause otherwise all the records from the table will be removed.

The following SQL query is used to delete the employee detail whose id is 110 from the table.

x

```
> delete from Employee where id = 110
```

Consider the following example.

Example

```
import mysql.connector

#Create the connection object
```

```
4. myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "google",data
   base = "PythonDB")
5.
6. #creating the cursor object
7. cur = myconn.cursor()
8.
9. try:
10.    #Deleting the employee details whose id is 110
11.    cur.execute("delete from Employee where id = 110")
12.    myconn.commit()
13. except:
14.
15.    myconn.rollback()
16.
17. myconn.close()
```

******