**Required argument, Keyword argument**

Arguments (**parameters**) are used to transfer data between the caller and the function. Generally, a function with list of **formal** **argument** (parameters) must have matching list of **actual** **arguments** (or **parameters**). List of actual arguments matching list of formal arguments is called **Required Arguments**. An example is given below:

List of **formal** **arguments**

def display(roll, name, marks):

print('Roll=',roll)

print('Name=',name)

print('Marks=',marks)

ro,na,ma=12,'RANJAN YADAV', 78.5

display(ro, na, ma)

List of **actual** **arguments** ro, na and ma are also called **required arguments**

display(na, ma, ro)

display(ma, ro, na)

**Outputs produced by the script**:

Roll= 12

Name= RANJAN YADAV

Marks= 78.5

Roll= RANJAN YADAV

Name= 78.5

Marks= 12

Roll= 78.5

Name= 12

Marks= RANJAN YADAV

In Python there is no strict data type for any variable. Value stored in a variable decides the data type of a variable. Therefore, value stored in an **actual** argument decides data type of the **formal** argument. Hence last two calls to function display() are syntactically but logically wrong. To eliminate this kind of logical error we need **Keyword** **argument**. Using **Keyword** **Argument**, formal argument's names are included with the actual arguments. Keyword arguments is nothing but the list of actual arguments (**formal** argument names are assigned values during the function call). Using **keyword** **arguments**, the list of **actual** arguments can be in any order but correct values will be assigned to the list of **formal** arguments. An example is given below:

List of **formal** arguments

def display(roll, name, marks):

print('Roll=',roll)

print('Name=',name)

print('Marks=',marks)

ro,na,ma=12,'RANJAN YADAV', 78.5

**Keyword Arguments**

Formal argument's names are assigned values during the function call

display(roll=ro, name=na, marks=ma)

display(name=na, marks=ma, roll=ro)

display(marks=ma, roll=ro, name=na)

**Outputs produced by the script**:

Roll= 12

Name= RANJAN YADAV

Marks= 78.5

Roll= RANJAN YADAV

Name= 78.5

Marks= 12

Roll= 78.5

Name= 12

Marks= RANJAN YADAV

It is very clear from the above example that using **keyword** **arguments**, order of the actual arguments does not necessarily match with order of formal arguments but still correct values are assigned to the formal arguments.

**Default argument**

As discussed above, every function with arguments must have same number of formal arguments and actual arguments. But we will see now that may not true every time. In Python, formal arguments can be assigned constant values and that function can be called without any actual arguments. Constant values assigned to the formal arguments are called **Default Arguments**. When actual argument is present during the function call, formal argument's default value is replaced by value present in the actual argument. Default arguments always starts from the **right**. An example is given below:

def foo(ch='\*', n=10):

**Explanation of outputs**

foo() calls the function without any actual arguments. Function foo() uses default values of the formal arguments to display \*\*\*\*\*\*\*\*\*\*. foo('#') calls the function with one actual argument. '#' replaces the default of ch but n's default value is still 10. So ########## is displayed. foo(7) calls the function with one actual argument. 7 replaces the default value of ch but n's default value is still 10. So, 7777777777 is displayed. foo('$',15) calls the function with two actual arguments. '$' replaces default value of ch and 12 replaces default value of n. So $$$$$$$$$$$$ is displayed. foo(n=20, ch='@') calls the function with two keyword arguments. '@' replaces default value of ch and 20 replaces default value of n. So @@@@@@@@@@@@@@@ is displayed.

for x in range(n):

print(ch,end='')

print()

foo()

foo('#')

foo(7)

foo('$',12)

foo(n=15, ch='@')

**Outputs produced by the script**:

\*\*\*\*\*\*\*\*\*\*

##########

7777777777

$$$$$$$$$$$$

@@@@@@@@@@@@@@@

As mentioned earlier, default arguments start from right. Examples of **incorrect use of default arguments** are given below:

Syntax error **Non-default argument follows default argument**. Since formal argument b is non-default argument. Possible solution, make formal argument b as default argument.

def foo(a=10,b,c=30):

avg=(a+b+c)/3

print(avg)

Syntax error **Non-default argument follows default argument**. Since formal argument c is non-default argument. Possible solution, make parameter c as default argument.

def foo(a=10,b=20,c):

avg=(a+b+c)/3

print(avg)

Syntax error **Non-default argument follows default argument**. Since formal arguments b and c are non-default arguments. Possible solution, make formal argument b and c as default argument.

def foo(a=10,b,c):

avg=(a+b+c)/3

print(avg)

Syntax error **Non-default argument follows default argument**. Since formal argument c is non-default argument. Possible solution, make formal argument c as default argument.

def foo(a,b=20,c):

avg=(a+b+c)/3

print(avg)

**Python module**

Module is a file which contains python functions, global variables, class definitions and etc. It is nothing but a Python script file which has python executable code/statement (.py file). It has single functionality.

**Python package**

Package is a collection of modules stored in a folder and must contains \_\_init\_\_.py file to let the Python interpreter know this is not just simple directory but a collection of modules.

**Python library**

Library is a collection of various packages (collection of related multiple functionalities). The defining characteristic of a library is that you are in control, you call the library.

**Python framework**

Framework is a collection of various libraries which design the code flow. The defining characteristic of a framework is Inversion of Control. The framework calls you, unlike module/package/library which has to be called. The flow of control and the flow of data is managed by the framework.

**Python pickle**

Pickle is used for serializing and de-serializing Python object structures. Serialization refers to the process of converting an object in memory to a byte stream that can be stored on disk. Later on, this byte stream can then be retrieved and de-serialized back to a Python object. In Python process of serialization and de-serialization is called is pickling. For pickling Python has **pickle** **module**. Important functions from pickle module:

pickle.dump(data, fileobj) – writes data into a binary using fileobj where fileobj is created using function open(). Since pickle.dump() writes into a binary file, so when opening a binary using open(), string 'b' is needed along with the file mode.

data=pickle.dump(fileobj) – reads from fileobj into a variable data where fileobj is created using function open(). While writing into a binary file or reading from a binary, variable data can either be a **list** or **tuple** or **dictionary** type data.

**Important point to remember** number of load() must match number dump() when reading from a binary file. Since it impossible to remember the count, try-except is used to read from a binary file without triggering **run-time** error. A Python script is given below showing how to write into a binary file and how to read from a binary file:

Variable bdf is a binary data file object created with open(). Generally binary data file is given the extension .DAT. File mode contains 2-characters string. 'a' for append mode and 'b' for binary file. If a binary file is to be opened in write mode then 'w' will replace 'a'.

Data to be written into the binary is a dictionary emp.

pickle.dump(emp,bdf) writes data into a binary file through a buffer. One record is written at time in the binary file.

import pickle

def addrec():

bdf=open('EMP.DAT', 'ab')

n=int(input('No. of records? '))

for k in range(n):

co:int(input('Code? '))

na:input('Name? ')

bs:float(input('BSal? '))

emp= {

'co':co,

'na':na,

'bs':bs

}

pickle.dump(emp,bdf)

bdf.close()

def showrec():

bdf=open('EMP.DAT', 'rb')

try:

while True:

emp=pickle.load(bdf)

print(emp['co'], emp['na'], emp['bs'], sep='\t')

except:

Variable bdf is a binary data file object created with open(). File mode contains 2-characters string. 'r' for read mode and 'b' for binary file.

emp=pickle.load(bdf) reads data from the binary file stores in a dictionary emp through a buffer. One record is read at time.

while True: will generate an infinite **while**-loop. When pickle.load() fails to read data from the file, a **run-time** error is triggered. The program control jumps to except part. File is closed and the function terminates.

bdf.close()

# ----- main -----

while True:

print('1. Add Records')

print('2. Show Records')

print('0. Exit')

ch=input('Choice[0-2]? ')

if ch=='1': addrec()

elif ch=='2': showrec()

elif ch=='0': break

**CSV file**

A CSV file (Comma Separated Values file) is a type of plain text file that stores tabular data in plain test format. Generally, CSV files use a comma to separate each specific data value. Here’s how a CSV file look like:

CODE,NAME,BASIC,YEARS

1001,ALOK NIGAM,72000,14

1002,BIDISHA JAIN,82000,19

1003,CHANDAN DUA,69000,10

1004,DEEPAK CHOPRA,94000,23

1005,ERIKA TIWARI,74000,15

Notice each piece of data is separated by a comma (default delimiter/separator). Normally, the first line represents a header row. It is used as name for every column (field). Every subsequent line after header is actual data. CSV files are normally created by programs that handle large amounts of data. They are a convenient way to export data from spreadsheets and databases as well as import or use it. The Python **csv** **module** provides functions to read from CSV file and write into CSV file. Important functions from csv module:

csvobj=csv.writer(fileobj, lineterminator='\n') – will create a CSV writer object using fileobj where fileobj is created using function open().

csvboj.writerow(rec) will write data stored in a **list/tuple** type of data rec into a CSV file through a buffer. If keyword argument lineterminator='\n' is missing then after every record (line) there will be a blank line.

csvboj.writerow(recs) will write data stored in a **nested list/tuple** type of data recs into a CSV file through a buffer. If keyword argument lineterminator='\n' is missing then after every record (line) there will be a blank line.

recs=csv.reader(fileobj) will create a CSV reader object and will read all the records (all the lines or the entire file) file into a variable recs through a buffer.

A Python script is given below showing how to write into a CSV file and how to read from a CSV file:

import csv

def csvappend():

cdf=open('result.csv', 'a')

csvobj=csv.writer(cdf, lineterminator='\n')

n=int(input('No. of records? '))

for x in range(n):

ro=int(input('Roll? '))

na=input('Name? ')

ma=float(input('Marks? '))

rec=[ro,na,ma]

csvobj.writerow(rec)

cdf.close()

def csvdisplay():

cdf=open('result.csv', 'r')

recs=csv.reader(cdf)

for rec in recs:

print('%i %-10s %5.1f'%(int(rec[0]),rec[1],float(rec[2])))

cdf.close()

Variable cdf is a CSV data file object created with open().

csv.writer(cdf,lineterminator='\n') creates a CSV writer object csvobj.

rec is a list containing ro, na and ma.

csvobj.writerow(rec) write data into a CSV file.

recs=csv.reader(cdf) creates a CVS reader object and reads the entire file into a variable recs.

**for**-loop iterates over recs and print()displays the CSV file in a tabular manner using format specifier.

# ----- Menu -----

while True:

print('1. Append')

print('2. Display')

print('0. Exit')

ch=input('Choice[0-2]? ')

if ch=='1': csvappend()

elif ch=='2': csvdisplay()

elif ch=='0': break

**Python pyplot**

Matplotlib is a plotting library for the Python programming language. Python pyplot is a Matplotlib module which is used for data visualization.

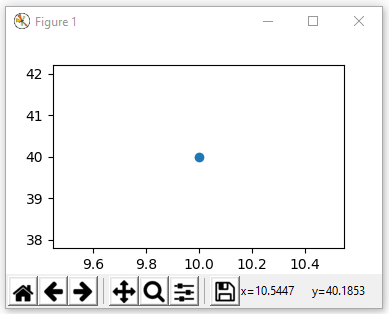
plot(x,y,'o') - will plot a **blue solid circle** at (x,y) coordinate. But the point will not be displayed on the screen. Function show() will display the plotted point on the screen. Python scripts are given below to plot a point:

import matplotlib.pyplot as pp

pp.plot(10, 40, 'o') #plots a red solid square

pp.show()

**Output produced by the script**

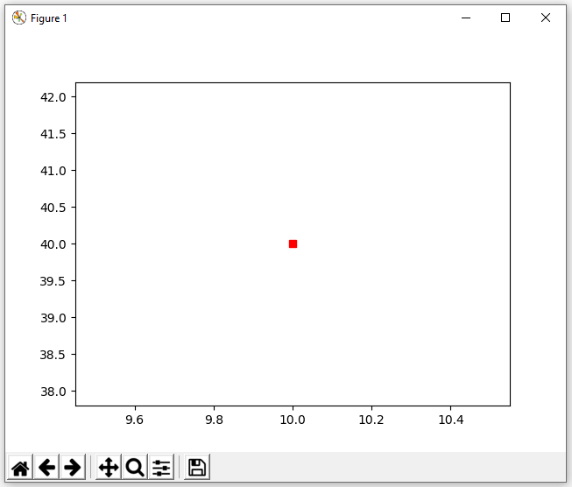


import matplotlib.pyplot as pp

pp.plot(10, 40, 'rs') #plots a red solid square

pp.show()

**Output produced by the script**



**Basic colour codes**: blue(b), green(g), red(r), cyan(c), magenta(m), yellow(y), black(k)

To use colour in plotting, either use colour name or the letter representing the colour.

**Basic point shapes**: '.' for point, 'o' for circle, '^' for triangle, 's' for square, 'D' for diamond

plot(yvalue) – will plot a line using values stored in a **list/tuple** yvalues to represent values for the **y-axis**. Values for the **x-axis** will 0 to len(yvalue)-1. **Solid** line will be plotted in **blue** colour.

plot(xvalue,yvalue) – will plot a line using values stored in a **list/tuple** xvalue as the values for the **x-axis** and values stored in **list/tuple** yvalue as the values for the **y-axis**. **Solid** line will be plotted in **blue** colour.

plot(xvalue,yvalue,'g') – will plot a line using values stored in a **list/tuple** xvalue as the values for the **x-axis** and values stored in **list/tuple** yvalue as the values for the **y-axis**. **Solid** line will be plotted in **green** colour. A Python script to plot a line is given in the next page:

import matplotlib.pyplot as pp

square=[1,4,9,16,25,36]

xvalue=[1,2,3,4,5,6]

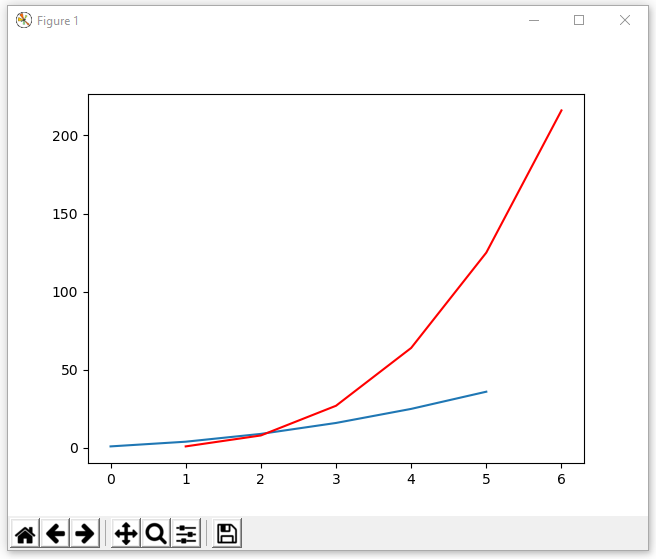
yvalue=[1,8,27,64,125,216]

pp.plot(square) #plots a line in blue

pp.plot(xvalue, yvalue, 'r') #plots a line in red

pp.show()

**Output produced by the script**



title(string) – string will display the title of the plot

xlablel(string) – string will display the label for the x-axis

ylablel(string) – string will display the label for the y-axis vertically

show() – will display the plot on the screen

A python script is given below showing title, label x-axis and label for y-axis:

import matplotlib.pyplot as pp

pp.title('y=x\*\*2')

pp.xlabel('x')

pp.ylabel('x\*\*2')

xvalue=[1,2,3,4,5,6]

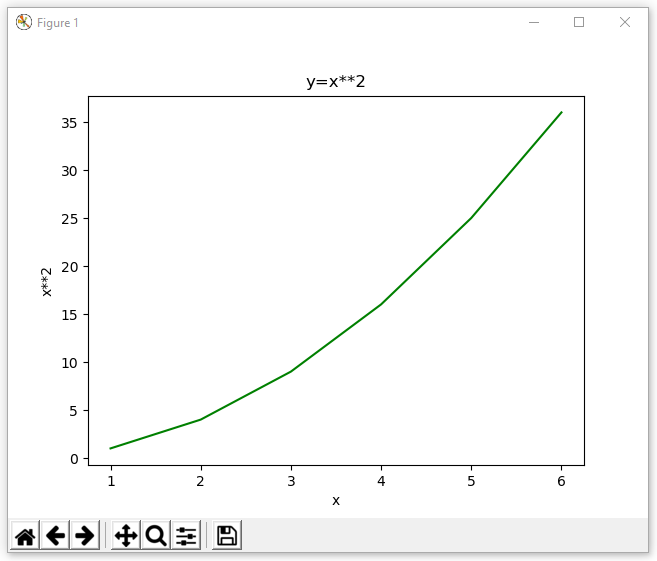
yvalue=[1,4,9,16,25,36]

pp.plot(xvalue, yvalue, 'g') #plots a line in green

pp.show()

**Output produced by the script**

**Title**



**Label for x-axis**

**Label for y-axis**

bar(xlist,ylist) – will plot a vertical bar chart with values in the **list / tuple** xlist as the values for **x-axis** and using the values in the **list / tuple** ylist as the **height** of the bar (values for **y-axis**). The bar will be plotted in **blue**.

bar(xlist,ylist,color='r') – will plot a vertical bar chart with values in the **list / tuple** xlist as the values for **x-axis** and using the values in the **list / tuple** ylist as the **height** of the bar (values for **y-axis**). **Keyword** **argument** color can be used to plot the bar chart using a **specific colour**. In this case bar chart will be plotted in **red**.

bar(xlist,ylist,color=['r','g','b','m','y']) – will plot a vertical bar chart with values in the **list / tuple** xlist as the values for **x-axis** and using the values in the **list / tuple** ylist as the **height** of the bar (values for **y-axis**). **Keyword** **argument** color can be used to plot the bar chart using **specific set of colours**. Assuming there are 5 values in both the **list/tuple**, five bars will be plotted with five different colours.

barh() – is almost similar to bar() function. Function barh() will plot horizontal bar chart. Parameters for barh() is exactly similar to bar() function. Python scripts are given below for plotting bar chart:

import matplotlib.pyplot as pp

pp.title('Employee Salary')

pp.xlabel('Employee Name')

pp.ylabel('Employee Salary (Lacs)')

name=['GAGAN','ROOPA','DIPAK','NEETA','KARAN']

basic=[1.25,1.35,1.28,1.32, 1.27]

pp.bar(name,basic) #bars will be plotted with blue colour

pp.show()

**Output produced by the script**



import matplotlib.pyplot as pp

pp.title('Employee Salary')

pp.xlabel('Employee Name')

pp.ylabel('Employee Salary (Lacs)')

name=['GAGAN','ROOPA','DIPAK','NEETA','KARAN']

basic=[1.25,1.35,1.28,1.32, 1.27]

pp.bar(name,basic,color='r') #bars will be plotted with blue colour

pp.show()

**Output produced by the script**



import matplotlib.pyplot as pp

pp.title('Employee Salary')

pp.xlabel('Employee Name')

pp.ylabel('Employee Salary (Lacs)')

name=['GAGAN','ROOPA','DIPAK','NEETA','KARAN']

basic=[1.25,1.35,1.28,1.32, 1.27]

pp.bar(name,basic,color=['r','g','b','m','y'])

#bars will be plotted with five different colours

pp.show()

**Output produced by the script**



pie(piedata) – will plot a pie chart with values stored in the **list/tuple** piedata. Number of pie slices will depend on number values present in the **list/tuple** piedata. There will be **no label** for the pie slices. Every pie slice will be plotted with **different colours**. Between the pie slices there will be **no gap**. A python script is given below to plot a pie chart:

import matplotlib.pyplot as pp

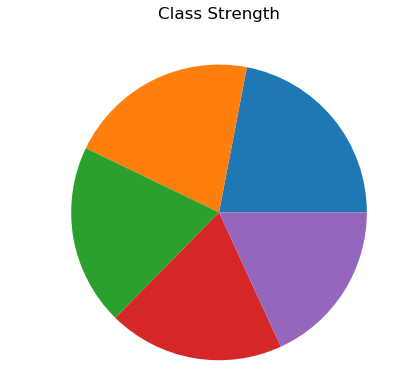
present=[40,38,36,35,33]

pp.title('Class Strength')

pp.pie(present)

pp.show()

**Output produced by the script**



pie(piedata, labels=llist) – will plot a pie chart with values stored in the **list/tuple** piedata. Number of pie slices will depend on number values present in the **list/tuple** piedata. **Keyword** **argument** lables will be used to label the pie slices and it will receive values from the **list/tuple** llist. Every pie slice will be plotted with **different colours**. Between the pie slices there will be **no gap**. A python script is given below to plot a pie chart:

import matplotlib.pyplot as pp

present=[40,38,36,35,33]

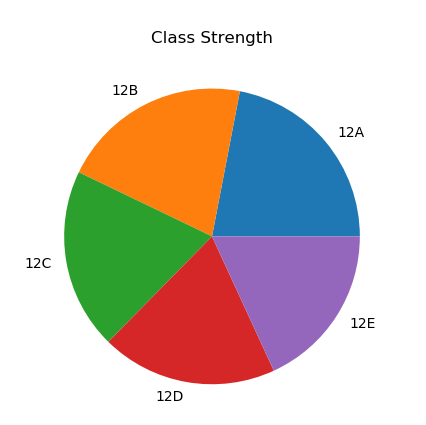
secs=['12A','12B','12C','12D','12E']

pp.title('Class Strength')

pp.pie(present, labels=secs)

pp.show()

**Output produced by the script**



pie(piedata, labels=llist, colors=clist) – will plot a pie chart with values stored in the **list/tuple** piedata. Number of pie slices will depend on number values present in the **list/tuple** piedata. **Keyword** **argument** lables will be used to label the pie-slices and it will receive values from the list/tuple llist. Using **keyword argument** colors, it is possible to colour the pie slices with specific sets of colours. Between the pie slices there will be **no gap**. A python script is given below to plot a pie chart:

import matplotlib.pyplot as pp

present=[40,38,36,35,33]

secs=['12A','12B','12C','12D','12E']

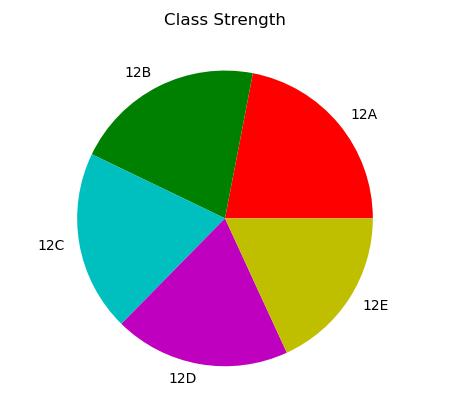
mycol=['r','g','c','m','y']

pp.title('Class Strength')

pp.pie(present, labels=secs, colors=mycol)

pp.show()

**Output produced by the script**



pie(piedata, labels=llist, colors=clist, explode=explist) – will plot a pie chart with values stored in the **list/tuple** piedata. Number of pie slices will depend on number values present in the list/tuple piedata. **Keyword argument** lables will be used to label the pie slices and it will receive values from the **list/tuple** llist. Using **keyword argument** colors, it is possible to colour the pie slices with specific sets of colours. Using **keyword argument** explode, it is possible to have space (gap) between the pie slices. Also, it is possible to pull out a particular pie slice. A python script is given below to plot a pie chart:

import matplotlib.pyplot as pp

present=[40,38,36,35,33]

secs=['12A','12B','12C','12D','12E']

mycol=['r','g','c','m','y']

Pie slice '12A' will be pulled out. For the remaining pie slices, there be will have no gap.

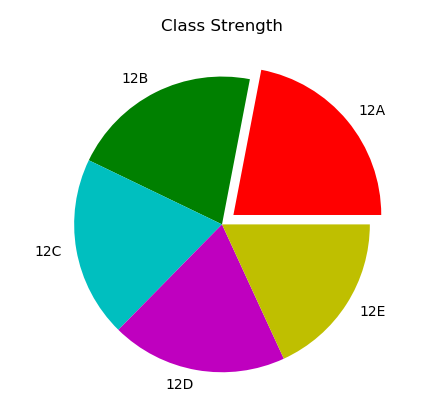
explist=[0.1,0.0,0.0,0.0,0.0]

pp.title('Class Strength')

pp.pie(present,labels=secs, colors=mycol, explode=explist)

pp.show()

**Output produced by the script**



import matplotlib.pyplot as pp

present=[40,38,36,35,33]

secs=['12A','12B','12C','12D','12E']

mycol=['r','g','c','m','y']

Pie slice '12A' will be pulled out. For the remaining pie slices, there be will be little gap.

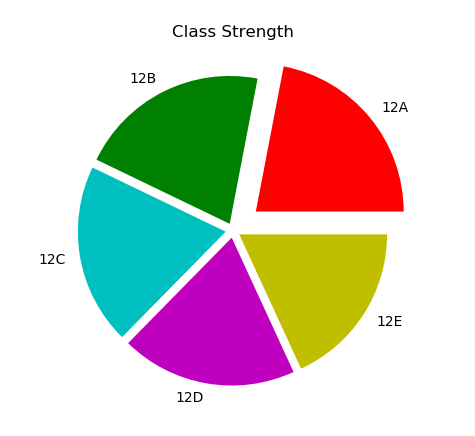
explist=[0.2,0.05,0.05,0.05,0.05]

pp.title('Class Strength')

pp.pie(present,labels=secs, colors=mycol, explode=explist)

pp.show()

**Output produced by the script**



pie(piedata, labels=llist, colors=clist, explode=explist) – will plot a pie chart with values stored in the list/tuple piedata. Number of pie slices will depend on number values present in the list/tuple data. **Keyword argument** lables will be used to label the pie-slices and it will receive values from the **list/tuple** llist. Using **keyword argument** colors, it is possible to colour the pie slices with specific sets of colours. Using **keyword argument** explode it is possible to have space (gap) between the pie slices. Also, it is possible to pull out a particular pie slice. Using **keyword argument** autopct we can display size of every pie slice in percentage. Adding all the values will give 100. A python script is given below to plot a pie chart:

import matplotlib.pyplot as pp

present=[40,38,36,35,33]

secs=['12A','12B','12C','12D','12E']

mycol=['r','g','c','m','y']

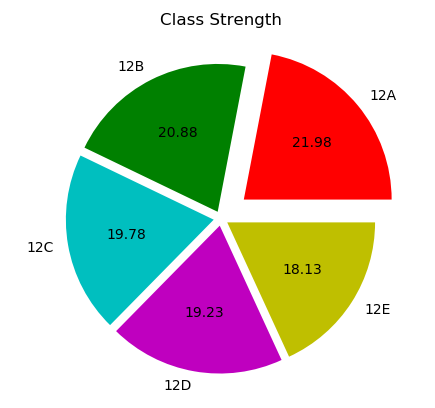
explist=[0.2,0.05,0.05,0.05,0.05]

pp.title('Class Strength')

pp.pie(present,labels=secs,colors=mycol,explode=explist,autopct='%1.2f')

pp.show()

**Output produced by the script**



**Python MySQLdb**

MySQLdb is a Python package used as an interface for connecting to a MySQL database server from Python. To use MySQLdb one has to install mysqlclient. So important functions from MySQLdb. So important functions from MySQLdb:

db=MySQLdb.connect() will connect Python to MySQL server and creates a **connector** object db. This function has at least three parameters - host, user, passwd and **optional** database.

curs=db.cursor() will create curs as **cursor** object using **connector** object db. All the SQL command will be executed through **cursor** **object**.

curs.execute() will execute **SQL** **query** as a string. **SQL** query string does not require terminating semi-colon(;).

curs.fetchall() will retrieve **all the rows** as a **nested tuple** after a **select** query.

curs.fetchone() will retrieve **one row** as a **tuple** after a **select** query.

curs.fetchmany() will retrieve **many row** as a nested tuple after a **select** query.

db.commit() is needed after a **DML** commands like **INSERT** **INTO**, **DELETE** and **UPDATE** so that data in the table gets inserted, deleted and updated.

1. Write a menu driven Python program to do the following:

a) Write a Python function to create a database **school**, create a table **teacher** with following columns:

**CODE INT** teacher code

**NAME CHAR (20)** teacher name

**SEX CHAR** Gender

**DEPT CHAR (15)** department

**NOP INT** number of periods

b) Add following rows in the **teacher** table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CODE** | **NAME** | **SEX** | **DEPT** | **NOP** |
| 2011 | MR. JATIN | M | MATHEMATICS | 27 |
| 2013 | MR. PRANAV | M | CHEMISTRY | 26 |
| 2015 | MS. GEETA | F | PHYSICS | 27 |
| 2017 | MS. MARIA | F | CHEMISTRY | 24 |
| 2019 | MS. PARUL | F | MATHEMATICS | 25 |
| 2012 | MR. ADITYA | M | PHYSICS | 24 |
| 2014 | MS. ROOPA | F | PHYSICS | 25 |
| 2016 | MS. MALATI | F | MATHEMATICS | 24 |
| 2018 | MR. DEEPAK | M | CHEMISTRY | 27 |
| 2020 | MR. KUNAL | M | PHYSICS | 27 |

c) Write a Python function to execute following query (with menu):

i) Display the **teacher** table on the screen

ii) Display the **teacher** table sorted in descending order of **NOP**

iii) Input CODE and search for CODE in the teacher table. If inputted CODE does not exist then display an error message.

iv) Input DEPT and displays all the teacher's details for the inputted DEPT. If inputted DEPT does not exist then display an error message.

v) Display number teachers DEPT-wise

vi) Display number teachers SEX-wise

d) Write a Python function to add following columns in the teacher table:

**DESIG** **CHAR** **(3)** **DESIG** – designation

**BONUS** **FLOAT** **BONUS** – Dussehra / Diwali bonus

e) Write a Python function to edit column DESIG as:

|  |  |
| --- | --- |
| **NAME** | **DESIG** |
| MR. JATIN | TGT |
| MR. PRANAV | TGT |
| MS. GEETA | PGT |
| MS. MARIA | HOD |
| MS. PARUL | PGT |
| MR. ADITYA | HOD |
| MS. ROOPA | PGT |
| MS. MALATI | HOD |
| MR. DEEPAK | TGT |
| MR. KUNAL | TGT |

To edit column BONUS:

**DESIG BONUS**

HOD 2000

PGT 1750

TGT 1500

f) Write a Python function to execute following query (with menu):

i) Input CODE and delete the row containing inputted CODE from the teacher table. If inputted CODE does not exist then display an error message.

ii) Input NAME and delete the row containing inputted NAME from the teacher table. If inputted NAME does not exist then display an error message.

import MySQLdb

db=MySQLdb.connect(host='localhost', user='faipskwt', passwd='P@ssw0rd')

curs=db.cursor()

def createtable():

sql='create database school'

curs.execute(sql)

curs.execute('use school')

sql='create table teacher(code int, name char(20), sex char, dept char(15), nop int)'

curs.execute(sql)

print('Table: teacher created')

def addrows():

curs.execute('use school')

for x in range(10):

co=int(input('Teacher code? '))

na=input('Teacher name? ')

se=input('Sex? ')

de=input('Department? ')

np=int(input('No. of periods? '))

na=na.upper()

se=se.upper()

de=de.upper()

sql='insert into teacher values({},"{}","{}","{}",{})'.format(co,na,se,de,np)

curs.execute(sql)

db.commit()

def displayrows():

curs.execute('use school')

while True:

print('Display Menu')

print('1. Display Table')

print('2. Display Table in descending order of nop')

print('3. Display details by searching for tcode')

print('4. Display details by searching for dept')

print('5. Display number of teachers SEX-wise')

print('6. Display number of teachers DEPT-wise')

print('0. Exit menu')

ch=int(input('Choose [0-6]? '))

if ch==1:

sql='select \* from teacher'

curs.execute(sql)

data=curs.fetchall()

for teacher in data:

for col in teacher:

print(col,end='\t')

print()

elif ch==2:

sql='select \* from teacher order by nop desc'

curs.execute(sql)

data=curs.fetchall()

for teacher in data:

for col in teacher:

print(col, end='\t')

print()

elif ch==3:

code=int(input('Teacher code to search? '))

sql='select \* from teacher where tcode={}'.format(code)

count=curs.execute(sql)

if count>0:

data=curs.fetchall()

for teacher in data:

for col in teacher:

print(col, end='\t')

print()

else:

print(code,'does not exist in the table')

elif ch==4:

dept=input('Department details to be displayed? ')

sql='select \* from teacher where dept="{}"'.format(dept)

count=curs.execute(sql)

if count>0:

data=curs.fetchall()

for teacher in data:

for col in teacher:

print(col, end='\t')

print()

else:

print(dept,'does not exist in the table')

elif ch==5:

sql='select sex, count(\*) from teacher group by sex'

curs.execute(sql)

data=curs.fetchall()

for row in data:

print(row[0], row[1], sep='\t')

elif ch==6:

sql='select dept, count(\*) from teacher group by dept'

curs.execute(sql)

data=curs.fetchall()

for row in data:

print(row[0], row[1], sep='\t')

elif ch==0: break

def addcols():

curs.execute('use school')

curs.execute('alter table teacher add desig char(3)')

curs.execute('alter table teacher add bonus float')

print('Columns: desig and bonus added in the table: teacher')

def edit():

curs.execute('use school')

curs.execute('update teacher set desig="HOD" where code in (2017, 2012, 2016)')

curs.execute('update teacher set desig="PGT" where code in (2015, 2019, 2014)')

sql='update teacher set desig="TGT" where code in (2011,2013,2018,2020,2021,2022)'

curs.execute(sql)

curs.execute('update teacher set bonus=2000 where desig="HOD"')

curs.execute('update teacher set bonus=1750 where desig="PGT"')

curs.execute('update teacher set bonus=1500 where desig= "TGT"')

db.commit()

print('Columns desig and bonus are updated')

def delrows():

curs.execute('use school')

while True:

print('Delete Menu')

print('1. Delete row by Code')

print('2. Delete row by Name')

print('0. Exit menu')

ch=int(input('Choose [0-2]? '))

if ch==1:

code=int(input('Teacher code to delete? '))

sql='delete from teacher where code={}'.format(code)

count=curs.execute(sql)

if count>0:

db.commit()

print('Row with code={} deleted'.format(code))

else:

print('Row with code={} not found'.format(code))

elif ch==2:

name=input('Teacher name to delete? ')

sql='delete from teacher where name="{}"'.format(name)

count=curs.execute(sql)

if count>0:

db.commit()

print('Row with name={} deleted'.format(name))

else:

print('Row with name={} not found'.format(name))

elif ch==0: break

while True:

print('Menu')

print('1. Create Table')

print('2. Add Rows')

print('3. Display Table')

print('4. Add Columns')

print('5. Updae Table')

print('6. Delete Rows')

print('0. Exit menu')

ch=int(input('Choose [0-6]? '))

if ch==1: createtable()

elif ch==2: addrows()

elif ch==3: displayrows()

elif ch==4: addcols()

elif ch==5: edit()

elif ch==6: delrows()

elif ch==0: break

db.close()

**HTML**

* HTML stands for Hyper Text Markup Language
* HTML describes the structure of a Web page
* HTML consists of a series of elements
* HTML elements tell the browser how to display the content
* HTML elements are represented by tags
* Browsers do not display the HTML tags, but use them to render the content of the page

**HTML tags**

HTML tags are element names surrounded by angle brackets

<tagname> Content goes here... </tagname>

* HTML tags normally come **in pairs** like <p> and </p>
* The first tag in a pair is the **start tag,** the second tag is the **end tag**
* The end tag is written like the start tag, but with a **forward slash** inserted before the tag name

The start tag is also called the **opening tag**, and the end tag the **closing tag**.

The HTML document itself begins with <html> and ends with </html>. The visible part of the HTML document is between <body> and </body>. HTML headings are defined with the <h1> to <h6> tags. <h1> defines the most important heading (largest font size). <h6> defines the least important heading (smallest font size) .

**Python Django**

Django is a free and open source web application framework written in Python. A Python framework is a collection of Python libraries. They are grouped together, and allow you to create applications or websites from an existing source, instead of from scratch and to development easier. Django is high-level Python Web-framework that helps to create rapid development, clean and pragmatic design. How to start a **Django** project? To start a Django project we need to execute following command:

django-admin startproject projname

projname must be **valid Python identifier name**. **Two** folder will be created after the execution of the above command. **Outer** folder projname and **inner** folder projname.

|  |  |
| --- | --- |
| **Outer** folder projname contains: | **Inner** folder projname contains: |
| * Inner folder projname * manage.py | * \_\_init\_\_.py * settings.py (module) * urls.py (module) * wsgi.py (module) |

Inner folder projname is actually a **package**. One user defined module has to be created inside the inner folder projname. Generally, module name is views.py but any other name can be given. To begin with we create a module views.py (or any other **file name**) in the **inner** folder projname. Module views.py contains at least one function. That function name is usually home() or index(). Every function inside views.py represents a **Web page**. An example is given below:

#views.py

from django.http inport HttpResponse

def home(request):

html='<body><h1 align=center>Django - Python Web Framwork</h1></body>'

return HttpResponse(html)

Function HttpResponse() has to be imported django.http. Kindly note the **mixed** **case** identifier name for HttpResponse. Function HttpResponse() returns a **HTML** **code** as a string to the Web browser. Variable name html is a string and it represents a line of **HTML** code. Single line string is enough for a single line of HTML code. For multi-line HTML code, it is better to use multi-line string. Next, we have edit the module urls.py.

#urls.py

from django.urls import path

from . import views

urlpatterns= [

path('',views.home)

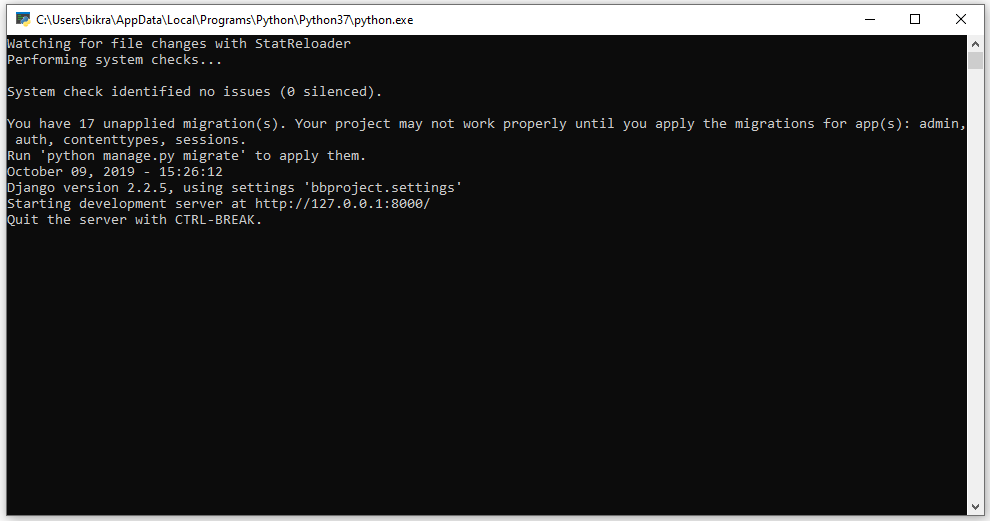
]

Function path() has to be imported from django.urls and the user defined **module** views has to imported as well. The dot (.) after the **keyword** **from** signifies current **working directory (folder)**. **Django** server is constantly reading the module urls.py. Web browser sends a request for a page to the **Django** server. Django server reads the module urls.py to see which user-defined module contains the requested Web Page. Function path() then tells the **Django** server which function to be invoked from the user-defined module for requested Web page. If the function name is either home() or index(), then the first parameter in the function path() is an **empty string** and the second parameter is the name of the function to be invoked from the module name (views.home). In our example, function home() is to invoked from the module views.

Next, we need to start the **Django** server. For that we need to go the **outer** folder projname and execute the following command:

python manage.py runserver

If the is (are) no error(s) in the module urls.py, following window will appear on the screen:



If there is any error (normally in the module urls.py), then the Window disappear in a flash. After execution of this command a new file db.sqlite3, will be created in the **outer** folder projname. Next, we need to start the Web browser. In the address bar of the Web browser we will type the **URL** of the **Django** server:

localhost:8000

OR

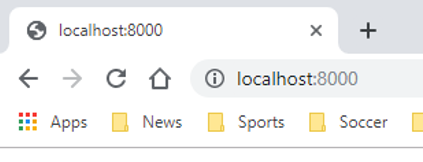
127.0.0.1:8000

Web browser will display the following Web page on the screen:



**URL of the Django Server** **Address Bar of the Web browser**

Web browser is requesting for the **Home**/**Index** page of the **Django** server to be displayed through the Web browser. **Django** server will receive the request made by the Web browser. **Django** server reads the module urls.py and identifies the function to be invoked from the module views.py. Next, **Django** server reads the module views.py and invokes the function home(). Function home() returns the **HTML** code as string through HttpResponse() to the Web browser. Web browser will receive the HTML code from the **Django** server and the Web browser will display the Web page on the screen.



The Web browser requesting home or Index page of the **Django** Server.

**Django Server**

**urls.py**

**views.py**

Now we will add another function in the **module** views.py. The second function page1() will represent another Web page. The edited code for the **module** views.py is give below:

#views.py

from django.http inport HttpResponse

def home(request):

html='<body><h1 align=center>Django - Python Web Framwork</h1></body>'

return HttpResponse(html)

def page1(request):

html='''

<body>

<h1 align=center> What is Django?</h1>

Django is Python Framework that

encourages rapid development and

clean, pragmatic design. Built by

experienced developers, it takes

care of much of the hassle of Web

development, so you can focus on

writing your app without needing

to reinvent the wheel. It's free

and open source.

</body>

'''

return HttpResponse(html)

Function page1() contains HTML code as a multi-line string. Since we have another function in the **module** views.py, we need to edit the **module** urls.py also. The edited code for the **module** urls.py is given in the next page:

#urls.py

from django.urls import path

from . import views

urlpatterns= [

path('', views.home),

path('page1', views.page1)

]

By typing localhost:8000/page1 in the address bar of a Web browser, will display the following Web page on the screen (Web page is zoomed to 150%):



The Web page to be render by the module views.py can be a HTML file also. To render a Web Page stored as **HTML** file, we have to make changes in the modules views.py, urls.py and settings.py. Also, a HTML file has to created and stored in a **sub-folder** created under the **outer** folder projname. First, let us add new function in the module views.py to render a HTML file. Edited module views.py is given below:

#views.py

from django.http inport HttpResponse

from django.shortcuts import render

def home(request):

html='<body><h1 align=center>Django - Python Web Framwork</h1></body>'

return HttpResponse(html)

def page1(request):

html='''

<body>

<h1 align=center> What is Django?</h1>

Django is Python Framework that encourages rapid

development and clean, pragmatic design. Built by

experienced developers, it takes care of much of

the hassle of Web development, so you can focus

on writing your app without needing to reinvent

the wheel. It's free and open source.

</body>

'''

return HttpResponse(html)

def page2(request):

return render(request, 'page2.html')

A new statement has been added in the module views.py to import function render(). Function render(), renders a Web page stored as a HTML file. Function render() has two parameters – request and the Web page to rendered as a string. Since a new function has been added to the module views.py, the module urls.py is to be edited as well. Edited module urls.py is given below:

#urls.py

from django.urls import path

from . import views

urlpatterns= [

path('', views.home),

path('page1', views.page1),

path('page2/', views.page2)

]

Now we need tell the Django server, the location of the Web page. A new sub-folder webpage will be created under the outer folder projname. In the sub-folder webpage, a HTML file page2.html to created and saved. Django server needs to know the location of Web Page. For that, we need to edit the module settings.py. A line has to be added in the module settings.py. Edited settings.py is given below (only the part of code is shown where a change has to be made):

**Change to be made in the module** settings.py

TEMPLATES=[

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [**os.path.join(BASE\_DIR, 'WebPages')**],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

Function os.path.join() has two parameters - BASE\_DIR and the sub-folder name WebPages as a string. Content of the HTML file page2.html is given below:

<html>

<body>

<h1 align=center>Why Django?</h1>

<b>Ridiculously fast</b><br>

Django was designed to help developers take applications from

concept to completion as quickly as possible.<p>

<b>Reassuringly secure</b><br>

Django takes security seriously and helps developers avoid many common security mistakes.<p>

<b>Exceedingly scalable</b><br>

Some of the busiest sites on the Web leverage Django's ability to quickly and flexibly scale.<p>

</body>

</html>

By typing localhost:8000/page2 in the address bar of a Web browser, will display the following Web page on the screen (Web page is zoomed to 150%):

