THIS PYTHON COURSE TOTALLY BASED ON BESANT TECHNOLOGIES:

TOTAL(30-40 HRS)

INTRODUCTION(1HR):

python:
it is a general purpose language using in multiple departments

1.history
history author: guido van rossum 1989 came to the 1991

2.what kind of language

easy syntax:

1. java-7 lines to print
2.c -5 lines
3.single line print

free and open source:

highlevel programming language:

memory allocation about security etcet

platform independent:
if you are writing any codes in any platform it will
it will not depent on platform

portability

procedure oriented and object oriented
c,pascal there is no object

oops concept also will support

interpretted:
codes if you are writing it will execute line by line

extensible library:

embeded:
a program written in the other language it can be runnable when inserting inside of python program

dynamic language:
if you going mention a string in java 'str' integer 'int'
a =6 directly it will print it is a string

contents:

A.core python:

identfiers (variables):

a = 5

to mention a variable:

1.it will never starts with numbers (1a =5 #wrong)

2.$will never allow = ($a = 4)
3.a =2 ,A =5 print(a)
4.reserved words will not allow to mention as a varible(if,else,class,def,int,bine,True)
(if = 5)

1.data types(immuatable)
1.integers:
1.decimal
2.binary numbers
3.octal numbers
4.hexadecimals
2.string
3.float
4.complex
5.boolean
6.none

operators:
1.arithmetic operators--++,--,//,**
2.logical operators --and,or,not
3.comparison operators-->,<,<=,>=
4.membership -- is, is not a=(1,2,3) b =(1,2,3) a is b, in 2 in b
5.ternery operators == a=2 ,b=3 : a if a>b else b

2.data structures

1.tuple --(1,2,3) --immutable
2.list --[1,2,3]--mutable
3.set --{1,2,3}--mutable
4.dictionaries--{'a':3,'b':4}--mutable
6.bytes-[mutable]
7.bytearrays--[immutable]
8.range -- range(9)--0,1,2,3,4,5,6,7,8, range(3, 16) and (1,100,3)
9.frozenset --a ={1,2,3},frozenset(a) --immutable

type casting:
a = 'dsa;f'
bin(a),complex(b)
int(a)

3.control flow statments

1.if --the conditions is ther one of them might be true
2.if else
3.elif -else if  --there are more than one condition
4.while --condition will get satisfied until that it will iterate
5.break--when the condition get satisfies it will break the loop
6.continue -- pass current iteration to next iteration without  printion values
7.for loop --formally to access or itrate thruough a sequences each every element

4.error handling

1.try--find the condition is perfect
2.except-- to handle except
3.error calling  -- zero,ttype,value
4.multiple exceptions--zerod, string
5.else --if the condition is true then it will execute
6.finally --whatever you are printing inside of finally block it will never whether condition true or false
it will execute

5.file handling

1.text -- 'r','w','a','r+','x'
2.binary files == 'rb','wb'
3.csv files==import csv library , csv.readers,csv.writers

6.functions

```
def func(**a):

      add =a+b
      sub =a-b
      mul =a*b

      return add,sub,mul

print(func(a=2,b=4))
```

func(3,5)
1.normal func -- def method()
2.how to call a func

3.passing parameters
4.passing multiple parameters
5.argums(*a)
6.keyword parameters
7.kwargs(**a)

7.modules

concatenation fuctions,objects
whatever file that you are storing along with .py extension that is known modules

8.math, collections

math--sqrt(a),ceil(),floor(),log(),sin()
collections --namedtuple,concating

B.advance python:

1.oops

1.class
-blue print
instance method,class method,static method,

2.object
everything in python

3.polymorphism

1. operator overloading ,,++,--
2.operator overriding ++
3.duct typing --multiple methods in same name we are going to access all of them treat them as a same
4.method overloading--
5. method overriding

4.inheritence

1.single-one to one
2.multiple-a class inherit two class
3.multilevel inheritence-a ,b, c(a,b)
4.hybrid unheritence- c, d(c)

5.encapsulation

2.data base handling
store the value, retrive from the database,we can create table,insert some values to the table, join two tables

mysql,postgre,oracle,mongo

3.API
api -application programming interface,serialize json, execute seria

4.gui creation

graphical user interface --tkinter, matplotlib

5.tkinter

buttons,colors,opening,scrooling

6.network programming
django -- high level web frame work, how to create a projeect, make modules, how to serialize, how url patters
local machine link

CORE PYTHON:

1.WHAT CAN PYTHON DO:

1. AI and machine learning
Because Python is such a stable, flexible, and simple programming language, it’s perfect for various machine learning (ML)
and artificial intelligence (AI) projects.
In fact, Python is among the favourite languages among data scientists,
and there are many Python machine learning and AI libraries and packages available.

If you’re interested in this application of Python, our Deep Learning and Python Programming
for AI with Microsoft Azure ExpertTrack can help you develop your skills in these areas.
You can discover the uses of Python and deep learning while boosting your career in AI.

2. Data analytics
Much like AI and machine learning, data analytics is another rapidly developing field that utilises Python programming.
At a time when we’re creating more data than ever before, there is a need for those who can collect, manipulate and organise the information.

Python for data science and analytics makes sense. The language is easy-to-learn, flexible, and well-supported,
meaning it’s relatively quick and easy to use for analysing data. When working with large amounts of information,
it’s useful for manipulating data and carrying out repetitive tasks.

You can learn about data analytics using Python with our ExpertTrack, which will help you develop practical data analytics skills.

3. Data visualisation
Data visualisation is another popular and developing area of interest. Again, it plays into many of the strengths of Python.
As well as its flexibility and the fact it’s open-source, Python provides a variety of graphing libraries with all kinds of features.

Whether you’re looking to create a simple graphical representation or a more interactive plot, you can find a library to match your needs.
Examples include Pandas Visualization and Plotly. The possibilities are vast, allowing you to transform data into meaningful insights.

If data visualisation with Python sounds appealing, check out our 12-week ExpertTrack on the subject. You’ll learn how to leverage
Python libraries to interpret and analyse data sets.

4. Programming applications
You can program all kinds of applications using Python. The general-purpose language can be used to read and create file directories,
create GUIs and APIs, and more. Whether it’s blockchain applications, audio and video apps, or machine learning applications, you can build them all with Python.

We also have an ExpertTrack on programming applications with Python, which can help to kick-start your programming career.
Over the course of 12 weeks, you'll gain an introduction on how to use Python, and start programming your own applications using it.

5. Web development

Python is a great choice for web development. This is largely due to the fact that there are many Python web development frameworks
to choose from, such as Django, Pyramid, and Flask. These frameworks have been used to create sites and services such as Spotify, Reddit and Mozilla.

Thanks to the extensive libraries and modules that come with Python frameworks, functions such as database access, content management,
and data authorisation are all possible and easily accessible. Given its versatility, it's hardly surprising that Python is so widely used in web development.

6. Game development

Although far from an industry-standard in game development, Python does have its uses in the industry. It's possible to create simple
games using the programming language, which means it can be a useful tool for quickly developing a prototype. Similarly, certain functions
(such as dialogue tree creation) are possible in Python.

If you're new to either Python or game development, then you can also discover how to make a text-based game in Python. In doing so,
you can work on a variety of skills and improve your knowledge in various areas.

7. Language development

The simple and elegant design of Python and its syntax means that it has inspired the creation of new programming languages.
Languages such as Cobra, CoffeeScript, and Go all use a similar syntax to Python.

This fact also means that Python is a useful gateway language. So, if you're totally new to programming,
understanding Python can help you branch out into other areas more easily.

8. Finance

Python is increasingly being utilised in the world of finance, often in areas such as quantitative and qualitative analysis.
It can be a valuable tool in determining asset price trends and predictions, as well as in automating workflows across different data sources.

As mentioned already, Python is an ideal tool for working with big data sets, and there are many libraries available to help with
compiling and processing information. As such, it's one of the preferred languages in the finance industry.

9. SEO

Another slightly surprising entry on our list of Python uses is in the field of search engine optimisation (SEO).
It's an area that often benefits from automation, which is certainly possible through Python. Whether it's implementing
changes across multiple pages or categorising keywords, Python can help.

Emerging technologies such as natural language processing (NLP) are also likely to be relevant to those working in SEO.

Python can be a powerful tool in developing these NLP skills and understanding how people search and how search engines return results.

10. Design

When asking ‘what is Python used for?’ you probably weren’t expecting design to feature on the list. However,

Python can be used to develop graphic design applications. Surprisingly, the language is used across a range of 2D imaging software,
such as Paint Shop Pro and Gimp.

Python is even used in 3D animation software such as Lightwave, Blender, and Cinema 4D, showing just how versatile the language is.

Python projects for beginners

So, if you were wondering what to do with Python and who uses Python, we’ve given plenty of ideas for how it’s used.

But what about if you’re just starting out with the language and want to become a Python developer?

Below, we’ve outlined some Python project ideas for beginners. These can help you develop your knowledge and challenge
your abilities with the programming language:

Build a guessing game
Design a text-based adventure game
Create a simple Python calculator
Write a simple, interactive quiz
Build an alarm clock

Once you’ve mastered the basics of Python, each of these can challenge you and help you hone the skills you’ve already learned.

Final thoughts

That concludes our look at what Python programming can be used for. As you can see, there are many applications for this popular language, with a wide support network and a diverse range of libraries that can help.

There are many reasons why you might want to start learning Python. It’s a future-proof and in-demand skill that’s required across all kinds of industries. What’s more, we have a broad selection of Python courses that can help you either master the basics or develop some more specific skills.

2.WHY PYTHON

Python is a general purpose and high level programming language. You can use Python for developing desktop GUI applications,

websites and web applications. ... The simple syntax rules of the programming language further makes it easier for you to keep
the code base readable and application maintainable.

Readable and Maintainable Code

Multiple Programming Paradigms
Compatible with Major Platforms and Systems
Robust Standard Library
Many Open Source Frameworks and Tools

Simplify Complex Software Development
Adopt Test Driven Development

3.GOOD TO KNOW

What is Python?

Python is a backend programming language that's great for beginners.
Python is similar in many ways to Ruby, but is less verbose than other programming languages - a little less wordy.
Python is approachable. ...
Python can be used for scripting, web scraping, and creating data sets.

4.PYTHON SYNTAX COMPARED TO OTHER PROGRAMMING LANGUAGES

Java:

```
 public class HelloWorld
 {
 p s v main(String[] args)
 {
 SOP("Hello world");
 }
 }
```

c:

```
 #include<stdio.h>
 void main()
 {
 print("Hello world");
 }
```

Python:

```
print("Hello World")
```

5.INSTALLING PYTHON

go to the python offcial website and download

PYTHON BASICS(3HRS):

BEGINNING PYTHON BASICS

1.PRINT STATEMENT

We can use print() function to display output.

Form-1: print() without any argument
Just it prints new line character
Form-2:
print(String):

```
print("Hello World")
```

We can use escape characters also

```
print("Hello \n World")
print("Hello\tWorld")
```

We can use repetetion operator (*) in the string

```
print(10*"Hello")
print("Hello"*10)
```

We can use + operator also

```
 print("Hello"+"World")
```

2.COMMENTS

A comment in Python starts with the hash character, # , and extends to the end of the physical line.
 A hash character within a string value is not seen as a comment, though. To be precise, a comment
can be written in three ways - entirely on its own line, next to a statement of code, and as a multi-line comment block


A Python docstring is a string used to document a Python module, class, function or method,
 so programmers can understand what it does without having to read the details of the implementation.
Also, it is a common practice to generate online (html) documentation automatically from docstrings.

```
"I am a single-line comment"

'''
I am a
multi-line comment!
'''

print("Hello World")
```


IDENTIFIERS:

```
#1a ='apple'
#$b =5
a2b =7
a_b=  8
#b$d = 6   #special keywords not possible
#if = 7
print(a2b)
print(a_b)
```


3.PYTHON DATA STRUCTURES AND DATA TYPES

Among the basic data types and structures in Python are the following:
Logical: bool.
Numeric: int , float , complex.
Sequence: list , tuple , range.
Text Sequence: str.
Binary Sequence: bytes , bytearray , memoryview.
Map: dict.
Set: set , frozenset.

type()
 to check the type of variable
 id()
 to get address of object


ex:

```
a= 5
print(type(a))
```

```
b= 'doo'
print(type(b))


print(id(a))  #address finding
```

int data type:
We can use int data type to represent whole numbers (integral values)
Eg:

```
 a=10
 type(a) #int
```

Note:
In Python2 we have long data type to represent very large integral values.
But in Python3 there is no long type explicitly and we can represent long values also by
using int type only.
We can represent int values in the following ways
1. Decimal form
2. Binary form
3. Octal form
4. Hexa decimal form
1. Decimal form(base-10):
It is the default number system in Python
The allowed digits are: 0 to 9
Eg: a =10
2. Binary form(Base-2):
The allowed digits are : 0 & 1
Literal value should be prefixed with 0b or 0B

```
Eg: a = 0B1111
 a =0B123
 a=b111



a =4
b = 0b0011
o =0o12345
x = 0x258ab

print(bin(a))

print(b)

print(bin(o))

print(bin(x))
```

3. Octal Form(Base-8):

The allowed digits are : 0 to 7
Literal value should be prefixed with 0o or 0O.

```
#oct conversion

a =4
b = 0b0011
o =0o12345
x = 0x258ab

print(oct(a))

print(oct(b))

print((o))

print(oct(x))
```

4. Hexa Decimal Form(Base-16):
The allowed digits are : 0 to 9, a-f (both lower and upper cases are allowed)
Literal value should be prefixed with 0x or 0X
Eg:

```
 a =0XFACE
 a=0XBeef
 a =0XBeer
```

```
a=10
b=0o10
c=0X10
d=0B10
print(a)10
print(b)8
print(c)16
print(d)2
```

Base Conversions
Python provide the following in-built functions for base conversions
1.bin():
We can use bin() to convert from any base to binary

```
 bin(15)

 bin(0o11)

 bin(0X10)
```

oct():
We can use oct() to convert from any base to octal

```
oct(10)

oct(0B1111)

oct(0X123)
```

```
hex():
We can use hex() to convert from any base to hexa decimal
Eg:
hex(100)

hex(0B111111)
hex(0o12345)

#hex conversion

a =4
b = 0b0011
o =0o12345
x = 0x258ab

print(hex(a))

print(hex(b))

print(hex(o))

print((x))


example no:2:

'''
#int
#int type conversion
a= 5
b= 0B01001
b1 = 0b01010

o= 0o01234
o1 = 0O435


x= 0X01001
x1 = 0x01010
print(a,b,b1,o,o1,x,x1)

#nbinary coversion

a= bin(5)
b= 0B01001
b1 = 0b01010

o= bin(0o01234)
o1 = bin(0O435)


x= bin(0X01001)
x1 = bin(0x01010)
print(a,b,b1,o,o1,x,x1)


#oct coversion
```

```
a= oct(5)
b= oct(0B01001)
b1 = oct(0b01010)

o= bin(0o01234)
o1 = bin(0O435)


x= oct(0X01001)
x1 = oct(0x01010)
print(a,b,b1,o,o1,x,x1)
'''
#hex coversion

a= hex(5)
b= hex(0B01001)
b1 = hex(0b01010)

o= hex(0o01234)
o1 = hex(0O435)


x= bin(0X01001)
x1 = bin(0x01010)
print(a,b,b1,o,o1,x,x1)
```

float data type:
We can use float data type to represent floating point values (decimal values)
Eg: f=1.234
 type(f) float
We can also represent floating point values by using exponential form (scientific notation)
Eg: f=1.2e3
 print(f) 1200.0
instead of 'e' we can use 'E'

```
a =3.4567
print(type(a))
print(hex(a))
```

example:

```
a =2.34567898

print(type(a))

#b = bin(a)

#o = oct(a)

#x = hex(a)
```

Complex Data Type:

a and b contain intergers or floating point values

Eg:
3+5j
10+5.5j
0.5+0.1j
In the real part if we use int value then we can specify that either by decimal,octal,binary
or hexa decimal form.
But imaginary part should be specified only by using decimal form.
a=0B11+5j

a=3+0B11j

Even we can perform operations on complex type values.

```
a=10+1.5j
b=20+2.5j
c=a+b
print(c)
(30+4j)
type(c)

#addition

a = 4 + 7j
b =3 +11j
c =a + b
print(c)


#addition

a = 4j       # an integer we can add with a complex number img par also can addable

b =3 +11j
c =a + b
print(c)



a = 4r  #except j what ever charcter your adding it will be syntax
print(a)
```

: Complex data type has some inbuilt attributes to retrieve the real part and imaginary part
c=10.5+3.6j
c.real==>10.5
c.imag==>3.6

```
'''
a =3+4j
print(type(a))
print(hex(a))
```

```
a =3+0b001j
print(type(a))
'''
a =4
b =5+6j

c=a+b
print(c)

d =7j
e =5+6j

f=d+e
print(f)

print(a.real)
print(b.imag)

master example:
#complex

a =  4 +5j
print(type(a))

b= 4+3j
c =a+b


d =  2 +5j
print(type(a))

e=3j
f =d+e


g =  4
print(type(a))

h= 4+3j
i =h+g

print(c,f,i)

print(a.real)
print(a.imag)


#print(bin(a))

#print(oct(a))

#print(int(a))
#print(hex(a))


bool data type:
We can use this data type to represent boolean values.
The only allowed values for this data type are:
True and False
```

```
Internally Python represents True as 1 and False as 0
b=True
type(b) =>bool

a=10
b=20
c=a<b
print(c)==>True
True+True==>2
True-False==>1


'''
a =10
b= 20
print(a>b)

print(a<b)
'''
a =4
b =4

print(a<=b)
print(b<=a)

#finding the values of booleam

a = False

c = True
b = 5
print(c + b)

#converting into int, bin, oct,hex

a = False

c = True
b = 5
print(bin(c))


master example:


a =True +True
print(a)
b = False +False

print(b)
c,d = 2,3
print(d>c)

print(d<c)

f= True

print(bin(f))
```

```
print(int(f))


print(oct(f))


print(hex(f))


str type:
str represents String data type.
A String is a sequence of characters enclosed within single quotes or double
quotes.
s1='durga'
s1="durga"

#str concatenating two strings

a = '4'

b = 'two'

print(a + b)

#str concatenating two strings

a = '4'

b = "two"

c = 4

print(a + c) # can only concatenate str (not "int") to str

By using single quotes or double quotes we cannot represent multi line string
literals.
s1="durga
 soft"
For this requirement we should go for triple single quotes(''') or triple double
quotes(""")
s1='''durga
 soft'''
s1="""durga
 soft"""
We can also use triple quotes to use single quote or double quote in our String.
''' This is " character'''
' This i " Character '
We can embed one string in another string
'''This "Python class very helpful" for java students'''
a = '''this
        is
        the
        multiple
        lines'''

print(a*2) #multiplication of a string
```

Slicing of Strings:
slice means a piece
[ ] operator is called slice operator,which can be used to retrieve parts of String.
In Python Strings follows zero based index.
The index can be either +ve or -ve.
+ve index means forward direction from Left to Right
-ve index means backward direction from Right to Left

```
s="durga"
s[0]
s[1]
s[-1]
s[40]
s[1:40]
s[1:]
s[:4]
s[:]
s*3
len(s)
```

```
a  = 'google'

print(a[4])
print(a[3:])
print(a[:4])
print(a[-1])
print(a[-5:])

a = 'sting1'
b = "string2"
c ='''string3'''
d = """string4"""
print(type(a))

print(b + c + d)
print(type(c))
print(type(d))
print(a*3)

print(a[2])
print(b[2:])
print(c[:3])

print(d[1:4])
print(a[-1])
```

markup example:

```
#string

a = 'string1'


b = "string1"
```

```
c = '''string1'''


d = """string1"""

print(a)


print(b)

print(c)

print(d)

print(a[0])

print(b[3:])

print(c[:4])

print(d[-1])
#string

a = 'string1'


b = "string1"


c = '''string1'''


d = """string1"""

print(a)


print(b)

print(c)

print(d)

print(a[0])

print(b[3:])

print(c[:4])

print(d[1:4])
```

Note:
1. In Python the following data types are considered as Fundamental Data types
   - int
   - float
   - complex

¬ bool
¬ str

2. In Python,we can represent char values also by using str type and explicitly char type is
not available

```
c='a'
type(c)
```

3. long Data Type is available in Python2 but not in Python3. In Python3 long values also
we can represent by using int type only.
4. In Python we can present char Value also by using str Type and explicitly char Type is
not available.

What is the difference between data types and data structures in Python?
It is a collection of data types. It is a way of organizing the items in terms of memory, and also the way of accessing each item through some defined logic.
...
Difference between data type and data structure:
Data Types  Data Structures
Can hold values and not data, so it is data less  Can hold different kind and types of data within one single object

Implementation through Data Types is a form of abstract implementation
  Implementation through Data Structures is called concrete implementation
Can hold values and not data, so it is data less  Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity Time complexity comes into play when working with data structures
Examples: int, float, double  Examples: stacks, queues, tree

4.SIMPLE INPUT AND OUTPUT

The input() function helps to enter data at run time by the user and the output function print()
is used to display the result of the program on the screen after execution

```
city=input (“Enter Your City: ”)

Enter Your City:Madurai

print (“I am from “, city)

#input key word used to get some user inputs from users

a = input('enter the name:')
print(a)
print(type(a))
```

```
#input key word used to get some user inputs from users


a = int(input('enter the name:'))
print(a)
print(type(a))


#input key word used to get some user inputs from users

a = int(input('enter the first value:'))



b = int(input('enter the second value:'))

c = a + b
print(c)
print(type(c))

#input key word used to get some doing plusoperations for string also and interger

a = eval(input('enter the first value:'))



b = eval(input('enter the second value:'))

c = a + b
print(c)
print(type(c))
```

5.SIMPLE OUTPUT FORMATTING

There are several ways to present the output of a program, data can be printed in a human-readable form, or written to a file for future use or even in some other specified form. Sometimes user often wants more control the formatting of output than simply printing space-separated values. There are several ways to format output.

To use formatted string literals, begin a string with f or F before the opening quotation mark or triple quotation mark.
The str.format() method of strings help a user to get a fancier Output
Users can do all the string handling by using string slicing and concatenation operations to create any layout that the user wants.
The string type has some methods that perform useful operations for padding strings to a given column width.
Formatting output using String modulo operator(%) :
The % operator can also be used for string formatting. It interprets the left argument much like a printf()-style format
as in C language string to be applied to the right argument. In Python, there is no printf() function but the functionality
of the ancient printf is contained in Python. To this purpose, the modulo operator % is overloaded by the string class to
perform string formatting. Therefore, it is often called a string modulo (or sometimes even called modulus) operator.

The string modulo operator ( % ) is still available in Python(3.x) and the user is using it widely.

But nowadays the old style of formatting is removed from the language.

```
# Python program showing how to use
# string modulo operator(%) to print
# fancier output

# print integer and float value
print("Geeks : %2d, Portal : %5.2f" % (1, 05.333))




print("english : %2d, Portal : %5.2f" % (1, 05.333)) #5.2f is executing only 2
float values

# print integer value
print("Total students : %3d, Boys : %2d" % (240, 120))

# print octal value
print("%7.3o" % (25))

# print exponential value
print("%10.3E" % (356.08977))
```

6.OPERATORS IN PYTHON

Python Operators
Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values

```
print(10 + 5)
```

Python divides the operators in the following groups:

Arithmetic operators
Assignment operators
Comparison operators
Logical operators
Identity operators
Membership operators
Bitwise operators

Python Arithmetic Operators
Arithmetic operators are used with numeric values to perform common mathematical operations

Addition

```
x = 5
y = 3

print(x + y)
```

Subtraction

```
x = 5
```

```
y = 3

print(x - y)
```

Multiplication

```
x = 5
y = 3

print(x * y)
```

Division

```
x = 12
y = 3

print(x / y)
```

Modulus

```
x = 5
y = 2

print(x % y)
```

Exponentiation

```
x = 2
y = 5

print(x ** y) #same as 2*2*2*2*2
```

Floor division

```
x = 15
y = 2

print(x // y)

#the floor division // rounds the result down to the nearest whole number
```

Python Assignment Operators
Assignment operators are used to assign values to variables

normal:

```
x = 5

print(x)
```

method1:

```
x = 5
x = x-3
#same
x -= 3
```

```
print(x)


method2:

x = 5
#x =x+3 same

x += 3

print(x)


method3:

*=,   x *= 3,     x = x * 3


x = 5
#x =x*3 same

x *= 3

print(x)


method4:

/=,   x /= 3,     x = x / 3

x = 5
#x =x/3 same

x /= 3


method5:

%=,   x %= 3,     x = x % 3


x = 5
#x =x%3 same

x %= 3


method6:

//=   ,x //= 3,   x = x // 3


x = 5
#x =x//3 same

x //= 3

method7:
```

**=,  x **= 3     ,x = x ** 3

x = 5
#x =x**3 same

x *= 3

method8:

&=,   x &= 3,     x = x & 3

x = 5
#x =x&3 same

x &= 3

method9:

|=,   x |= 3,     x = x | 3

x = 5
#x =x|3 same

x |= 3


method10:

^=,   x ^= 3,     x = x ^ 3


x = 5
#x =x|3 same

x |= 3


method11:

>>=,  x >>= 3,    x = x >> 3

x = 5
#x =x>>3 same

x >>= 3

method12:

<<=,  x <<= 3,    x = x << 3

x = 5
#x =x<<3 same

x <<= 3

Python Comparison Operators
Comparison operators are used to compare two values:

Operator    Name  Example     Try it

==    Equal x == y
!=    Not equal   x != y
>     Greater than      x > y
<     Less than   x < y
>=    Greater than or equal to      x >= y
<=    Less than or equal to   x <= y

method1:

```
x = 5
y = 3

print(x == y)
```

method2:

```
x = 5
y = 3

print(x != y)
```

method3:

```
x = 5
y = 3

print(x > y)
```

method4:

```
x = 5
y = 3

print(x < y)
```

method5:

```
x = 5
y = 3

print(x >= y)
```

method6:

```
x = 5
y = 3

print(x <= y)
```

Python Logical Operators
Logical operators are used to combine conditional statements

Operator    Description Example     Try it

and Returns True if both statements are true x < 5 and x < 10
or Returns True if one of the statements is true x < 5 or x < 4
not Reverse the result, returns False if the result is true not(x < 5 and x < 10)

method1:

```
x = 5

print(x > 3 and x < 10)
```

method2:

```
x = 5

print(x > 3 and x >10)
```

method3:

```
x = 5

print(x > 3 and x < 10)
```

Python Identity Operators
Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator Description Example Try it
is Returns True if both variables are the same object x is y
is not Returns True if both variables are not the same object x is not y

method1:

```
x = 5
y = 5

print(x is not y)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have the same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison returns True because x is equal to y
```

method2:

```
print(x is  y)  #is operation is going to compare stored address unlike if give
'==' it will check the value



x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(id(x))

print(id(y))


print(x is  y)  #is operation is going to compare stored address unlike if give
'==' it will check the value

# returns False because z is the same object as x

print(x is not y)

# returns True because x is not the same object as y, even if they have the same
content

print(x != y)

# to demonstrate the difference betweeen "is not" and "!=": this comparison returns
False because x is equal to y

Python Membership Operators
Membership operators are used to test if a sequence is presented in an object:

Operator    Description Example     Try it
in                Returns True if a sequence with the specified value is present in
the object  x in y
not in            Returns True if a sequence with the specified value is not
present in the object   x not in y

method1:

x = ["apple", "banana"]

print("banana" in x)


# returns True because a sequence with the value "banana" is in the list



method2:

x = ["apple", "banana"]

print("pineapple" not in x)

# returns True because a sequence with the value "pineapple" is not in the list

Python Bitwise Operators
Bitwise operators are used to compare (binary) numbers:
```

| Operator | Name | Description |
|---|---|---|
| & | AND | Sets each bit to 1 if both bits are 1 |
| \| | OR | Sets each bit to 1 if one of two bits is 1 |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 |
| ~ | NOT | Inverts all the bits |
| << | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off |
| >> | Signed right shift | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off |

Bitwise Operators:
We can apply these operators bitwise.
These operators are applicable only for int and boolean types.
By mistake if we are trying to apply for any other type then we will get Error.
&,|,^,~,<<,>>
print(4&5) ==>valid
print(10.5 & 5.6) ==>
 TypeError: unsupported operand type(s) for &: 'float' and 'float'
print(True & True) ==>valid

print(4&5) ==>4
print(4|5) ==>5
print(4^5) ==>1

print(~5) ==>-6

PYTHON PROGRAM FLOW(3HRS)

1.IF STATEMENT AND IT RELATED STATEMENT

If and statement python
If-else conditional statement is used in Python when a situation leads to two conditions and one of them
 should hold true.
Syntax: if (condition): code1 else: code2 [on_true] if [expression] else [on_false]
Note: For more information, refer to Decision Making in Python (if , if..else, Nested if, if-elif) ...
Syntax:

```
if (condition):
    code1
else:
    code2
[on_true] if [expression] else [on_false]
```

in other words:

Python if Statement is used for decision-making operations. It contains a body of code which runs only

when the condition given in the if statement is true. If the condition is false, then the optional else
statement runs which contains some code for the else condition.

When you want to justify one condition while the other condition is not true, then you use Python
if else statement.

Python if Statement Syntax:

```
if expression
 Statement
else
 Statement
```

example:

```
>>> x = 0
>>> y = 5

>>> if x < y:                           # Truthy
...     print('yes')
...
yes
>>> if y < x:                           # Falsy
...     print('yes')
...

>>> if x:                               # Falsy
...     print('yes')
...
>>> if y:                               # Truthy
...     print('yes')
...
yes

>>> if x or y:                          # Truthy
...     print('yes')
...
yes
>>> if x and y:                         # Falsy
...     print('yes')
...

>>> if 'aul' in 'grault':               # Truthy
...     print('yes')
...
yes
>>> if 'quux' in ['foo', 'bar', 'baz']:  # Falsy
...     print('yes')
...
```

Python follows a convention known as the off-side rule, a term coined by British computer scientist Peter J. Landin. (The term is taken from the offside law in association football.) Languages that adhere to the off-side rule define blocks by indentation. Python is one of a relatively small set of off-side rule languages.

Recall from the previous tutorial on Python program structure that indentation has special significance in a Python program. Now you know why: indentation is used to define compound statements or blocks. In a Python program, contiguous statements that are indented to the same level are considered to be part of the same block.

Thus, a compound if statement in Python looks like this:

```
if <expr>:
    <statement>
    <statement>
    ...
    <statement>
<following_statement>
```

Here, all the statements at the matching indentation level (lines 2 to 5) are considered part of the same block. The entire block is executed if <expr> is true, or skipped over if <expr> is false. Either way, execution proceeds with <following_statement> (line 6) afterward.

example:

```
if 'foo' in ['bar', 'baz', 'qux']:
    print('Expression was true')
    print('Executing statement in suite')
    print('...')
    print('Done.')
print('After conditional')
```

The four print() statements on lines 2 to 5 are indented to the same level as one another. They constitute the block that would be executed if the condition were true. But it is false, so all the statements in the block are skipped. After the end of the compound if statement has been reached (whether the statements in the block on lines 2 to 5 are executed or not), execution proceeds to the first statement having a lesser indentation level: the print() statement on line 6.

Blocks can be nested to arbitrary depth. Each indent defines a new block, and each outdent ends the preceding block. The resulting structure is straightforward, consistent, and intuitive.

Here is a more complicated script file called blocks.py:

```
# Does line execute?                        Yes    No
#                                           ---    --
if 'foo' in ['foo', 'bar', 'baz']:        #  x
    print('Outer condition is true')      #  x

    if 10 > 20:                           #  x
        print('Inner condition 1')        #        x

    print('Between inner conditions')     #  x

    if 10 < 20:                           #  x
        print('Inner condition 2')        #  x

    print('End of outer condition')       #  x
print('After outer condition')            #  x
```

The else and elif Clauses

Now you know how to use an if statement to conditionally execute a single statement or a block of several statements. It’s time to find out what else you can do.

Sometimes, you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not. This is accomplished with an else clause:

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

If <expr> is true, the first suite is executed, and the second is skipped. If <expr> is false, the first suite is skipped and the second is executed. Either way, execution then resumes after the second suite. Both suites are defined by indentation, as described above.

In this example, x is less than 50, so the first suite (lines 4 to 5) are executed, and the second suite (lines 7 to 8) are skipped:

2.EXAMPLE OF IF STATMENT

```
>>> x = 20

>>> if x < 50:
...     print('(first suite)')
...     print('x is small')
... else:
...     print('(second suite)')
...     print('x is large')
...
(first suite)
x is small
```

Here, on the other hand, x is greater than 50, so the first suite is passed over, and the second suite executed:

```
>>> x = 120
>>>
>>> if x < 50:
...     print('(first suite)')
...     print('x is small')
... else:
...     print('(second suite)')
...     print('x is large')
...
(second suite)
x is large
```

There is also syntax for branching execution based on several alternatives. For this, use one or more elif (short for else if) clauses. Python evaluates each <expr> in turn and executes the suite corresponding to the first that is true. If none of the expressions are true, and an else clause is specified, then its suite is executed:

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

An arbitrary number of elif clauses can be specified. The else clause is optional. If it is present, there can be only one, and it must be specified last:

```
>>> name = 'Joe'
>>> if name == 'Fred':
...     print('Hello Fred')
... elif name == 'Xander':
...     print('Hello Xander')
... elif name == 'Joe':
...     print('Hello Joe')
... elif name == 'Arnold':
...     print('Hello Arnold')
... else:
...     print("I don't know who you are!")
...
Hello Joe
```

At most, one of the code blocks specified will be executed. If an else clause isn’t included, and all the conditions are false, then none of the blocks will be executed.

Note: Using a lengthy if/elif/else series can be a little inelegant, especially when the actions are simple statements like print(). In many cases, there may be a more Pythonic way to accomplish the same thing.

Here’s one possible alternative to the example above using the dict.get() method:

```
>>> names = {
...     'Fred': 'Hello Fred',
...     'Xander': 'Hello Xander',
...     'Joe': 'Hello Joe',
...     'Arnold': 'Hello Arnold'
... }

>>> print(names.get('Joe', "I don't know who you are!"))
Hello Joe
>>> print(names.get('Rick', "I don't know who you are!"))
I don't know who you are!
```

Recall from the tutorial on Python dictionaries that the dict.get() method searches a dictionary for the specified key and returns the associated value if it is found, or the given default value if it isn’t.

An if statement with elif clauses uses short-circuit evaluation, analogous to what you saw with the and and or operators. Once one of the expressions is found to be true and its block is executed, none of the remaining expressions are tested. This is demonstrated below:

```
>>> var  # Not defined
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    var
NameError: name 'var' is not defined

>>> if 'a' in 'bar':
...     print('foo')
... elif 1/0:
...     print("This won't happen")
... elif var:
...     print("This won't either")
...
```

```
foo
```
The second expression contains a division by zero, and the third references an undefined variable var. Either would raise an error, but neither is evaluated because the first condition specified is true.

## Conditional Expressions (Python’s Ternary Operator)

Python supports one additional decision-making entity called a conditional expression. (It is also referred to as a conditional operator or ternary operator in various places in the Python documentation.) Conditional expressions were proposed for addition to the language in PEP 308 and green-lighted by Guido in 2005.

In its simplest form, the syntax of the conditional expression is as follows:

```
<expr1> if <conditional_expr> else <expr2>
```
This is different from the if statement forms listed above because it is not a control structure that directs the flow of program execution. It acts more like an operator that defines an expression. In the above example, <conditional_expr> is evaluated first. If it is true, the expression evaluates to <expr1>. If it is false, the expression evaluates to <expr2>.

Notice the non-obvious order: the middle expression is evaluated first, and based on that result, one of the expressions on the ends is returned. Here are some examples that will hopefully help clarify:

```
>>> raining = False
>>> print("Let's go to the", 'beach' if not raining else 'library')
Let's go to the beach
>>> raining = True
>>> print("Let's go to the", 'beach' if not raining else 'library')
Let's go to the library

>>> age = 12
>>> s = 'minor' if age < 21 else 'adult'
>>> s
'minor'

>>> 'yes' if ('qux' in ['foo', 'bar', 'baz']) else 'no'
'no'

example:
>>> x = y = 40

>>> z = 1 + x if x > y else y + 2
>>> z
42

>>> z = (1 + x) if x > y else (y + 2)
>>> z

>>> x = y = 40

>>> z = 1 + (x if x > y else y) + 2
>>> z

example:
>>> x = y = 40

>>> z = 1 + (x if x > y else y) + 2
```

```
>>> z
```

example:

```
s = ('foo' if (x == 1) else
...      'bar' if (x == 2) else
...      'baz' if (x == 3) else
...      'qux' if (x == 4) else
...      'quux'
... )
```

The Python pass Statement
Occasionally, you may find that you want
 to write what is called a code stub: a placeholder for where you
 will eventually put a block of code that you haven’t implemented yet

example:

```
if True:
    pass

print('foo')
```

summarization:
With the completion of this tutorial, you are beginning to write Python
code that goes beyond simple sequential execution:

You were introduced to the concept of control structures. These are
compound statements that alter program control flow—the order of execution
 of program statements.
You learned how to group individual statements together into a block or suite.
You encountered your first control structure, the if statement, which makes
 it possible to conditionally execute a statement or block based on evaluation of
program data.
All of these concepts are crucial to developing more complex Python code.

The next two tutorials will present two new control structures: the while
 statement and the for statement. These structures facilitate iteration,
 execution of a statement or block of statements repeatedly

3.WHILE LOOP

Python Loops
Python has two primitive loop commands:

while loops
for loops

The while Loop
With the while loop we can execute a set of statements as long as a condition is
true.

Print i as long as i is less than 6

```
i = 1
while i < 6:
  print(i)
  i += 1
```

Note: remember to increment i, or else the loop will continue forever.
The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

The break Statement
With the break statement we can stop the loop even if the while condition is true

```
i = 1
while i < 6:
  print(i)
  if (i == 3):
    break
  i += 1
```

The continue Statement
With the continue statement we can stop the current iteration, and continue with the next

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)

# Note that number 3 is missing in the result
```

The else Statement
With the else statement we can run a block of code once when the condition no longer is true:

Example
Print a message once the condition is false

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

4.FOR LOOP
Python For Loops
A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set

ex:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
```

```
  print(x)
```

The for loop does not require an indexing variable to set beforehand.

Looping Through a String
Even strings are iterable objects, they contain a sequence of characters

```
for x in "banana":
  print(x)
```

The break Statement
With the break statement we can stop the loop before it has looped through all the items

ex:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

Exit the loop when x is "banana", but this time the break comes before the print

ex:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

The continue Statement
With the continue statement we can stop the current iteration of the loop, and continue with the next

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

Else in For Loop
The else keyword in a for loop specifies a block of code to be executed when the loop is finished

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Note: The else block will NOT be executed if the loop is stopped by a break statement.

Example
Break the loop when x is 3, and see what happens with the else block

```
for x in range(6):
```

```
  if x == 3: break
  print(x)
else:
  print("Finally finished!")

#If the loop breaks, the else block is not executed.
```

Nested Loops
A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop"

Example
Print each adjective for every fruit

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

The pass Statement
for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```
for x in [0, 1, 2]:
  pass

# having an empty for loop like this, would raise an error without the pass statement
```

5.RANGE STATEMENT

The range() Function
To loop through a set of code a specified number of times, we can use the range() function,
The range() function returns a sequence of numbers, starting from 0 by default, and increments
 by 1 (by default), and ends at a specified number.

Example
Using the range() function

ex:
```
for x in range(6):
  print(x)
```

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

The range() function defaults to 0 as a starting value, however it is possible to specify the
 starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6)

ex:

```
for x in range(2, 6):
  print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the
increment value by adding a third parameter: range(2, 30, 3)

```
for x in range(2, 30, 3):
  print(x)
```

6.BREAK AND CONTINUE



Python break and continue
In this article, you will learn to use break and continue statements to alter the flow of a loop.

Video: Python break and continue Statement

What is the use of break and continue in Python?
In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

The break and continue statements are used in these cases.

Join our newsletter for the latest updates.
Enter Email Address*
Join

Python break and continue
In this article, you will learn to use break and continue statements to alter the flow of a loop.

Video: Python break and continue Statement

What is the use of break and continue in Python?
In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

The break and continue statements are used in these cases.

Python break statement
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

ex:

```
# Use of break statement inside the loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

Python continue statement
The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

7.ASSERT

Python - Assert Statement
In Python, the assert statement is used to continue the execute if the given condition evaluates to True.
If the assert condition evaluates to False, then it raises the AssertionError exception with the specified
error message.

Syntax
assert condition [, Error Message]
The following example demonstrates a simple assert statement

```
x = 10
assert x > 0
print('x is a positive number.')
```

In the above example, the assert condition, x > 0 evalutes to be True, so it will continue to execute the next statement without any error.

The assert statement can optionally include an error message string, which gets displayed along with the AssertionError. Consider the following assert statement with the error message.

```
x = 0
assert x > 0, 'Only positive numbers are allowed'
print('x is a positive number.')
```

Above, x=0, so the assert condition x > 0 becomes False, and so it will raise the AssertionError with the specified message 'Only positive numbers are allowed'. It does not execute print('x is a positive number.') statement.

The following example uses the assert statement in the function.

```
def square(x):
    assert x>=0, 'Only positive numbers are allowed'
    return x*x

n = square(2) # returns 4
n = square(-2) # raise an AssertionError
```

8.EXAMPLES FOR LOOPING

example1:

```
n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
 print("* "*n)
```

example2:

```
n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
 for j in range(1,n+1):
      print(i,end=" ")
 print()
```

example3:

```
n=int(input("Enter the number of rows: "))
 for i in range(1,n+1):
      for j in range(1,n+1):
            print(j,end=" ")
      print()
```

example4:

```
n=int(input("Enter the number of rows: "))
 for i in range(1,n+1):
      for j in range(1,n+1):
            print(chr(64+i),end=" ")
      print()
```

example5:

```
n=int(input("Enter the number of rows: "))
 for i in range(1,n+1):
      for j in range(1,n+1):
```

```
            print(chr(64+j),end=" ")
      print()

example6:

n=int(input("Enter the number of rows: "))
 for i in range(1,n+1):
      for j in range(1,n+1):
            print(n+1-i,end=" ")
      print()

n=int(input("Enter the number of rows: "))
 for i in range(1,n+1):
      for j in range(1,n+1):
            print(chr(65+n-i),end=" ")
      print()

example7:

n=int(input("Enter the number of rows: "))
 for i in range(1,n+1):
      for j in range(1,n+1):
            print(chr(65+n-j),end=" ")
      print()

example8:

n=int(input("Enter the number of rows: "))
 for i in range(1,n+1):
      for j in range(1,n+1):
            print(chr(65+n-j),end=" ")
      print()

example9:

n=int(input("Enter the number of rows:"))
 for i in range(1,n+1):
      for j in range(1,i+1):
            print("*",end=" ")
      print()

example10:

n=int(input("Enter the number of rows: "))
for i in range(1,n+1):
      for j in range(1,i+1):
            print(i,end=" ")
      print()
```

FUNCTION MODULES(3HRS)

1.CREATING FUNCTIONS
A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function
In Python a function is defined using the def keyword

syntax:

```
def my_function():
  print("Hello from a function")
```

Calling a Function
To call a function, use the function name followed by parenthesis

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments
Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

2.FUNCTION PARAMETERS

Parameters or Arguments?
The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments
By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example
This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:
Example
This function expects 2 arguments, but gets only 1

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

Arbitrary Arguments, *args
If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

Example
If the number of arguments is unknown, add a * before the parameter name

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Arbitrary Arguments are often shortened to *args in Python documentations.

3.VARIABLE ARGUMENTS

Keyword Arguments
You can also send arguments with the key = value syntax.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Arbitrary Keyword Arguments, **kwargs
If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Arbitrary Kword Arguments are often shortened to **kwargs in Python documentations.

Default Parameter Value
The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Passing a List as an Argument
You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Return Values
To let a function return a value, use the return statement

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement
function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass

# having an empty function definition like this, would raise an error without the pass statement
```

Recursion
Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a

function calls itself.
 This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing
 a function which never terminates, or one that uses excess amounts of memory or processor power.
 However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, tri_recursion() is a function that we have defined to call itself ("recurse").
 We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out

is by testing and modifying it

```
ex 1;
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

This way the order of the arguments does not matter.

4.SCOPE OF A FUNCTION

A variable is only available from inside the region it is created. This is called scope.

Local Scope
A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

Example
A variable created inside a function is available inside that function

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

Function Inside Function
As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function:

Example
The local variable can be accessed from a function within the function

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()

myfunc()
```

Global Scope
A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example
A variable created outside of a function is global and can be used by anyone

```
x = 300

def myfunc():
  print(x)

myfunc()

print(x)
```

Naming Variables
If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example
The function will print the local x, and then the code will print the global x

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

Global Keyword
If you need to create a global variable, but are stuck in the local scope, you can use the global keyword.

The global keyword makes the variable global.

Example

If you use the global keyword, the variable belongs to the global scope

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

Also, use the global keyword if you want to make a change to a global variable inside a function.

Example
To change the value of a global variable inside a function, refer to the variable by using the global keyword

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

5.FUNCTION DOCUMENTATION

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. It's specified in source code that is used, like a comment, to document a specific segment of code.12-Aug-2020

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

It’s specified in source code that is used, like a comment, to document a specific segment of code. Unlike conventional source code comments, the docstring should describe what the function does, not how.

What should a docstring look like?

The doc string line should begin with a capital letter and end with a period.
The first line should be a short description.
If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.
Declaring Docstrings: The docstrings are declared using ”’triple single quotes”’ or “””triple double quotes””” just below the class, method or function declaration.
All functions should have a docstring.

Accessing Docstrings: The docstrings can be accessed using the __doc__ method of the object or using the help
 function.
The below examples demonstrates how to declare and access a docstring.

example 1:

```
def myfunction():
      '''Demonstrates triple double quotes
      docstrings and does nothing really.'''

      return None

print("Using __doc__:")
print(my_function.__doc__)

print("Using help:")
help(myfunction)
```

Example 2: Using triple double quotes

```
def my_function():
      """Demonstrates triple double quotes
      docstrings and does nothing really."""

      return None

print("Using __doc__:")
print(my_function.__doc__)

print("Using help:")
help(my_function)
```

The difference between Python comments and docstrings

You all must have got an idea about Python docstrings but have you ever wondered what is the difference between Python comments and docstrings. Let’s have a look at them.

Python Comments are the useful information that the developers provide to make the reader understand the source code. It explains the logic or a part of it used in the code. It is written by using # symbol.

6.LAMBDA FUNCTIONS AND MAP

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax
lambda arguments : expression
The expression is executed and the result is returned:

Example

Add 10 to argument a, and return the result

```
x = lambda a: a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example
Multiply argument a with argument b and return the result

```
x = lambda a, b: a * b
print(x(5, 6))
```

Example
Summarize argument a, b, and c and return the result

```
x = lambda a, b, c: a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?
The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be
 multiplied with an unknown number

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Or, use the same function definition to make a function that always triples the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

33

Or, use the same function definition to make both functions, in the same program

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

```
s=lambda a,b:a if a>b else b
print("The Biggest of 10,20 is:",s(10,20))
print("The Biggest of 100,200 is:",s(100,200))
```

filter() function:
We can use filter() function to filter values from the given sequence based on some condition.
filter(function,sequence)
 where function argument is responsible to perform conditional check
 sequence can be list or tuple or string.

```
l=[0,5,10,15,20,25,30]
l1=list(filter(lambda x:x%2==0,l))
print(l1) #[0,10,20,30]
l2=list(filter(lambda x:x%2!=0,l))
print(l2) #[5,15,25]
```

map() function:
For every element present in the given sequence,apply some functionality and generate
new element with the required modification. For this requirement we should go for map() function.
Eg: For every element present in the list perform double and generate new list of doubles.
Syntax:
map(function,sequence)
The function can be applied on each element of sequence and generates new sequence.

```
l=[1,2,3,4,5]
2) l1=list(map(lambda x:2*x,l))
3) print(l1) #
```

reduce() function:
reduce() function reduces sequence of elements into a single element by applying the
specified function.
reduce(function,sequence)
reduce() function present in functools module and hence we should write import statement.

```
from functools import *
2) l=[10,20,30,40,50]
```

```
3) result=reduce(lambda x,y:x+y,l)
4) print(result) # 150
```

8.CREATE A MODULE

What is a Module?
Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module
To create a module just save the code you want in a file with the file extension .py

Example
Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module
Now we can use the module we just created, by using the import statement:

Example
Import the module named mymodule, and call the greeting function

```
import mymodule

mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax:
module_name.function_name.

Variables in Module
The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example
Save this code in the file mymodule.py

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

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

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Naming a Module
You can name the module file whatever you like, but it must have the file extension .py

Re-naming a Module
You can create an alias when you import a module, by using the as keyword

Example
Create an alias for mymodule called mx

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Built-in Modules
There are several built-in modules in Python, which you can import whenever you like.

Example
Import and use the platform module

```
import platform

x = platform.system()
print(x)
```

Using the dir() Function
There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

Example
List all the defined names belonging to the platform module

```
import platform

x = dir(platform)
print(x)
```

Note: The dir() function can be used on all modules, also the ones you create yourself.

9.STANDARD MODULES
Python's standard library is very extensive, offering a wide range of functionalities. In this lesson, we will discuss how to use some of Python 3's standard modules such as the statistics module, the math module and the random module.

The Python Standard Library is a collection of script modules accessible to a Python program to simplify the programming process and removing the need to rewrite commonly used commands. They can be used by 'calling/importing' them at the beginning of a script

The Python Standard Library
While The Python Language Reference describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that

is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

standard modules

```
import math

print(math.pow(3.2,4))
```

EXCEPTION HANDLING(2HRS)

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause

1.ERRORS

Python - Error Types
The most common reason of an error in a Python program is when a certain statement is not in accordance with the prescribed usage. Such an error is called a syntax error. The Python interpreter immediately reports it, usually along with the reason.

```
>>> print "hello"
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("hello")?
```

In Python 3.x, print is a built-in function and requires parentheses. The statement above violates this usage and hence syntax error is displayed.

Many times though, a program results in an error after it is run even if it doesn't have any syntax error. Such an error is a runtime error, called an exception. A number of built-in exceptions are defined in the Python library. Let's see some common error types.

The following table lists important built-in exceptions in Python.

| Exception | Description |
| --- | --- |
| AssertionError | Raised when the assert statement fails. |
| AttributeError | Raised on the attribute assignment or reference fails. |

EOFError Raised when the input() function hits the end-of-file condition.
FloatingPointError Raised when a floating point operation fails.
GeneratorExit Raised when a generator's close() method is called.
ImportError Raised when the imported module is not found.
IndexError Raised when the index of a sequence is out of range.
KeyError Raised when a key is not found in a dictionary.
KeyboardInterrupt Raised wxhen the user hits the interrupt key (Ctrl+c or delete).
MemoryError Raised when an operation runs out of memory.
NameError Raised when a variable is not found in the local or global scope.
NotImplementedError Raised by abstract methods.
OSError Raised when a system operation causes a system-related error.
OverflowError Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError Raised when an error does not fall under any other category.
StopIteration Raised by the next() function to indicate that there is no further item to be returned by the iterator.
SyntaxError Raised by the parser when a syntax error is encountered.
IndentationError Raised when there is an incorrect indentation.
TabError Raised when the indentation consists of inconsistent tabs and spaces.
SystemError Raised when the interpreter detects internal error.
SystemExit Raised by the sys.exit() function.
TypeError Raised when a function or operation is applied to an object of an incorrect type.
UnboundLocalError Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError Raised when a Unicode-related error occurs during translation.
ValueError Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError Raised when the second operand of a division or module operation is zero.

error1:

```
>>> L1=[1,2,3]
>>> L1[3]
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>

L1[3]
IndexError: list index out of range
```

error2:

```
>>> import notamodule
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
```

```
import notamodule
ModuleNotFoundError: No module named 'notamodule'


error3:

>>> D1={'1':"aa", '2':"bb", '3':"cc"}
>>> D1['4']
Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>


D1['4']
KeyError: '4'



error4:

>>> from math import cube
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>

from math import cube
ImportError: cannot import name 'cube'


error4:

>>> it=iter([1,2,3])
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>

next(it)
StopIteration


error5:

>>> '2'+2
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>

'2'+2
TypeError: must be str, not int


error6:

>>> int('xyz')
Traceback (most recent call last):
```

```
File "<pyshell#14>", line 1, in <module>

int('xyz')
ValueError: invalid literal for int() with base 10: 'xyz'
```

error7:

```
>>> age
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>

age
NameError: name 'age' is not defined
```

error8:

```
>>> x=100/0
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>

x=100/0
ZeroDivisionError: division by zero
```

2.EXCEPTION HANDLING WITH TRY

Exception Handling in Python
The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

Syntax:

```
try :
    #statements in try block
except :
    #executed when error in try block
```

The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped.

If the exception does occur, the program flow is transferred to the except: block. The statements in the except: block are meant to handle the cause of the exception

appropriately. For example, returning an appropriate error message.

You can specify the type of exception after the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

The rest of the statements after the except block will continue to be executed, regardless if the exception is encountered or not.

The following example will throw an exception when we try to devide an integer by a string.

example:

```
try:
    a=5
    b='0'
    print(a/b)
except:
    print('Some error occurred.')
print("Out of try except blocks.")
```

You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate

ex2:

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

As mentioned above, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

ex3:

```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
print ('Out of try except blocks')
```

else and finally

In Python, keywords else and finally can also be used along with the try and except clauses.
While the except block is executed if the exception occurs inside the try block, the else block
gets processed if the try block is found to be exception free.

Syntax:

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occured or not
```

The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks

```
try:
    print('try block')
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
```

The first run is a normal case. The out of the else and finally blocks is displayed because the try block is error-free.

Raise an Exception
Python also provides the raise keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

The following code accepts a number from the user. The try block raises a ValueError exception if the number is outside the allowed range.

```
      try:
```

```
            x=int(input('Enter a number upto 100: '))
            if x > 100:
                  raise ValueError(x)
      except ValueError:
            print(x, "is out of allowed range")
      else:
            print(x, "is within the allowed range")
```

Here, the raised exception is a ValueError type. However, you can define your custom exception type to be raised. Visit Python docs to know more about user defined exceptions

3.HANDLING MULTIPLE EXCEPTIONS

Multiple Exception Handling in Python
Difficulty Level : Hard
Last Updated : 12 Jun, 2019
Given a piece of code that can throw any of several different exceptions, and one needs to account for all of the potential exceptions that could be raised without creating duplicate code or long, meandering code passages.

If you can handle different exceptions all using a single block of code, they can be grouped together in a tuple as shown in the code given below

ex2:

```
try:
   # do something
   pass

except ValueError:
   # handle ValueError exception
   pass

except (TypeError, ZeroDivisionError):
   # handle multiple exceptions
   # TypeError and ZeroDivisionError
   pass

except:
   # handle all other exceptions
   pass
```

4.WRITING YOUR OWN EXCEPTION

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, a hint is provided whether their guess is greater than or less than the stored number.

```
# define Python user-defined exceptions

#if you are print anythin in the class directly it will print for that we have to
```

```
follow module method that is the reason we used pass over here
class Error(Exception):
    """Base class for other exceptions"""
    pass


class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass


class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass


# you need to guess this number
number = 10

# user guesses a number until he/she gets it right
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()

print("Congratulations! You guessed it correctly.")
```

FILE HANDLING(1.5HRS)

1.FILE HANDLING MODES

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but alike other concepts of Python, this concept here is also easy and short. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let’s start with Reading and Writing files.

We use open () function in Python to open a file in read or write mode. As explained above, open ( ) will return a file object. To return a file object we use open() function along with two arguments, that accepts file name and the mode, whether to read or write. So, the syntax being: open(filename, mode). There are three kinds of mode, that Python provides and how files can be opened:

ex1:

```
# a file named "geek", will be opened with the reading mode.
file = open('text.txt', 'r')
# This will print every line one by one in the file
for each in file:
      print (each)
```

The open command will open the file in the read mode and the for loop will print each line present in the file

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

File Handling
The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax
To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

2.READING FILES
Open a File on the Server
Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
```

Good Luck!
To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

```
f = open("demofile.txt", "r")

print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Example
Open a file on a different location

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File
By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example
Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines
You can return one line by using the readline() method:

Example
Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling readline() two times, you can read the two first lines:

Example
Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Example
Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files
It is a good practice to always close the file when you are done with it.

Example
Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not
 show until you close the file.

3.WRITING AND ANPPENDING TO FILES

Python File Write

Write to an Existing File
To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example
Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Example
Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Create a New File
To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Example
Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
Result: a new empty file is created!

Example
Create a new file if it does not exist:

f = open("myfile.txt", "w")

Python Delete File

Delete a File
To delete a file, you must import the OS module, and run its os.remove() function:

Example
Remove the file "demofile.txt":

import os
os.remove("demofile.txt")

Check if File exist:
To avoid getting an error, you might want to check if the file exists before you
try to delete it:

Example
Check if file exists, then delete it:

import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")

Delete Folder
To delete an entire folder, use the os.rmdir() method:

Example
Remove the folder "myfolder":

import os
os.rmdir("myfolder")



The with statement:
The with statement can be used while opening a file.We can use this to group file
operation statements within a block.
The advantage of with statement is it will take care closing of file,after
completing all
operations automatically even in the case of exceptions also, and we are not
required to
close explicitly.


#using with statement
#with statement just like for loop as well while loop kind of operation that we
need follow space identation
with open('demo.txt','r') as file:
    print(file.read())
```

Eg:

```
1) with open("abc.txt","w") as f:
2) f.write("Durga\n")
3) f.write("Software\n")
4) f.write("Solutions\n")
5) print("Is File Closed: ",f.closed)
6) print("Is File Closed: ",f.closed)
7)
8) Output
9) Is File Closed: False
10) Is File Closed: True
```

The seek() and tell() methods:

tell():

We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. [ can you plese telll current cursor position]

The position(index) of first character in files is zero just like string index.

Eg:

```
1) f=open("abc.txt","r")
2) print(f.tell())
3) print(f.read(2))
4) print(f.tell())
5) print(f.read(3))
6) print(f.tell())
```

seek():

We can use seek() method to move cursor(file pointer) to specified location.

[Can you please seek the cursor to a particular location]

f.seek(offset, fromwhere)

offset represents the number of positions

The allowed values for second attribute(from where) are

0---->From beginning of file(default value)

1---->From current position

2--->From end of the file

Note: Python 2 supports all 3 values but Python 3 supports only zero.

Eg:

```
1) data="All Students are STUPIDS"
2) f=open("abc.txt","w")
3) f.write(data)
4) with open("abc.txt","r+") as f:
5) text=f.read()
6) print(text)
7) print("The Current Cursor Position: ",f.tell())
8) f.seek(17)
9) print("The Current Cursor Position: ",f.tell())
10) f.write("GEMS!!!")
11) f.seek(0)
12) text=f.read()
13) print("Data After Modification:")
14) print(text)
```

binary file:

To open a file in binary format, add 'b' to the mode parameter. Hence the "rb" mode opens the file in binary format for reading, while the "wb" mode opens the file in binary format for writing. Unlike text files, binary files are not human-readable. When opened using any text editor, the data is unrecognizable.

Handling Binary Data:
It is very common requirement to read or write binary data like images,video files,audio
files etc.
Q. Program to read image file and write to a new image file?

```
1) f1=open("rossum.jpg","rb")
2) f2=open("newpic.jpg","wb")
3) bytes=f1.read()
4) f2.write(bytes)
5) print("New Image is available with the name: newpic.jpg")
```

Handling csv files:
CSV==>Comma seperated values
As the part of programming,it is very common requirement to write and read data wrt csv
files. Python provides csv module to handle csv files.
Writing data to csv file:

```
1) import csv
2) with open("emp.csv","w",newline='') as f:
3) w=csv.writer(f) # returns csv writer object
4) w.writerow(["ENO","ENAME","ESAL","EADDR"])
5) n=int(input("Enter Number of Employees:"))
6) for i in range(n):
7) eno=input("Enter Employee No:")
8) ename=input("Enter Employee Name:")
9) esal=input("Enter Employee Salary:")
10) eaddr=input("Enter Employee Address:")
11) w.writerow([eno,ename,esal,eaddr])
12) print("Total Employees data written to csv file successfully")
```

Note: Observe the difference with newline attribute and without

4.HANDLING FILE EXCEPTIONS

Exception Handling in File Processing. To handle exceptions, we can use the Try, Catch, and Throw keywords.
These allowed us to perform normal assignments in a Try section and then handle an exception, if any, in a
Catch block.

CLASSES IN PYTHON(3.5HRS)

Python Classes/Objects
Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Example
Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5

print(MyClass)
```

Create Object
Now we can use the class named MyClass to create objects:

Example
Create an object named p1, and print the value of x

```
class MyClass:
  x = 5

p1 = MyClass()
print(p1.x)
```

2.CREATING CLASSES

The __init__() Function
The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example
Create a class named Person, use the __init__() function to assign values for name and age

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The __init__() function is called automatically every time the class is being used to create a new object.

## Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example
Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Note: The self parameter is a reference to the current instance of the class, and is used to access variables
 that belong to the class.

## The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables
 that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the
 first parameter of any function in the class:

Example

Use the words mysillyobject and abc instead of self:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

## Modify Object Properties

You can modify properties on objects like this:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)

p1.age = 40

print(p1.age)
```

Delete Object Properties

You can delete properties on objects by using the del keyword:

Example
Delete the age property from the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)

del p1.age

print(p1.age)
```

Delete Objects
You can delete objects by using the del keyword:

Example
Delete the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)

del p1

print(p1)
```

The pass Statement
class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

Example

```
class Person:
  pass

# having an empty class definition like this, would raise an error without the pass statement
```

1. Instance Variables:
If the value of a variable is varied from object to object, then such type of variables are called
instance variables.
For every object a separate copy of instance variables will be created.

Where we can declare Instance variables:
1. Inside Constructor by using self variable
2. Inside Instance Method by using self variable
3. Outside of the class by using object reference variable

1. Inside Constructor by using self variable:
We can declare instance variables inside a constructor by using self keyword. Once we creates
object, automatically these variables will be added to the object.
Example:

```
1) class Employee:
2)
3) def __init__(self):
4) self.eno=100
5) self.ename='Durga'
6) self.esal=10000
7)
8) e=Employee()

 print(e.__dict__)
```

2. Inside Instance Method by using self variable:
We can also declare instance variables inside instance method by using self variable. If any
instance variable declared inside instance method, that instance variable will be added once we
call taht method.

Example:

```
1) class Test:
2)
3) def __init__(self):
4) self.a=10
5) self.b=20
6)
7) def m1(self):
```

```
8) self.c=30
9)
10) t=Test()
11) t.m1()
12) print(t.__dict__)
```

We can also add instance variables outside of a class to a particular object.

```
1) class Test:
2)
3) def __init__(self):
4) self.a=10
5) self.b=20
6)
7) def m1(self):
8) self.c=30
9)
10) t=Test()
11) t.m1()
12) t.d=40
13) print(t.__dict__)
```

We can access instance variables with in the class by using self variable and outside of the class by
using object reference.

```
1) class Test:
2)
3) def __init__(self):
4) self.a=10
5) self.b=20
6)
7) def display(self):
8) print(self.a)
9) print(self.b)
10)
11) t=Test()
12) t.display()
13) print(t.a,t.b)
```

How to delete instance variable from the object:

1. Within a class we can delete instance variable as follows

```
 del self.variableName
```

2. From outside of class we can delete instance variables as follows

```
 del objectreference.variableName
```

Example:

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40

    def m1(self):
        del self.d

t=Test()
```

```
print(t.__dict__)#after this we will see d has been deleted

t.m1()
print(t.__dict__)
del t.c# after this we can see c has been deleted
print(t.__dict__)
```

1. Static variables:
If the value of a variable is not varied from object to object, such type of variables we have to
declare with in the class directly but outside of methods. Such type of variables are called Static
variables.
For total class only one copy of static variable will be created and shared by all objects of that
class.
We can access static variables either by class name or by object reference. But recommended to
use class name.

Instance Variable vs Static Variable:
Note: In the case of instance variables for every object a seperate copy will be created,but in the
case of static variables for total class only one copy will be created and shared by every object of
that class.

```
class Test:

 x=10

 def __init__(self):
     self.y=20

t1=Test()
t2=Test()
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
Test.x=888
t1.y=999
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
```

Various places to declare static variables:
1. In general we can declare within the class directly but from out side of any method
2. Inside constructor by using class name
3. Inside instance method by using class name
4. Inside classmethod by using either class name or cls variable
5. Inside static method by using class name

```
class Test:
    a = 10

    def __init__(self):
        Test.b = 20  # using the class name mention  static variable
```

```
    def m1(self):
        Test.c = 30

    @classmethod
    def m2(cls):
        cls.d1 = 40
        Test.d2 = 400

    @staticmethod
    def m3():
        Test.e = 50


print(Test.__dict__)
t = Test()
print(Test.__dict__)
t.m1()
print(Test.__dict__)
Test.m2()
print(Test.__dict__)
Test.m3()
print(Test.__dict__)
Test.f = 60
print(Test.__dict__)
```

How to access static variables:
1. inside constructor: by using either self or classname
2. inside instance method: by using either self or classname
3. inside class method: by using either cls variable or classname
4. inside static method: by using classname
5. From outside of class: by using either object reference or classnmae

```
class Test:
   a=10
   def __init__(self):
     print(self.a)
     print(Test.a)

   def m1(self):
     print(self.a)

     print(Test.a)
   @classmethod #used to access class variables using 'cls' keyword
   def m2(cls):
       print(cls.a)
       print(Test.a)
   @staticmethod # here we are not able to use self
   def m3():
     print(Test.a)
t=Test()
print(Test.a)
print(t.a)
t.m1()
t.m2()
t.m3()
```

Where we can modify the value of static variable:
Anywhere either with in the class or outside of class we can modify by using

classname.
But inside class method, by using cls variable.
Example:

```
class Test:
   a=777

   @classmethod
   def m1(cls):
     cls.a=888 # we can modify a static vaiable but this modification only
availbale in this method
   @staticmethod
   def m2():
     Test.a=999
print(Test.a)

Test.method()
print(Test.a)
Test.m1()
print(Test.a)
Test.m2()
print(Test.a)
```

If we change the value of static variable by using either self or object reference variable, then the
value of static variable won't be changed,just a new instance variable with that name will be
added to that particular object.
Example 1:

```
class Test:
  a=10
  def m1(self):  #if you want to call a instance method try to call through an
object
    self.a=888
t1=Test()

t1.m1()
print(Test.a)
print(t1.a)
```

```
class Test:


  x = 10
  def __init__(self):  #here you can change the static variable using constructor
     self.y = 20

t1 = Test()
t2 = Test()
print('t1:', t1.x, t1.y)
print('t2:', t2.x, t2.y)
t1.x = 888
t1.y = 999
print('t1:', t1.x, t1.y)
print('t2:', t2.x, t2.y)
```

How to delete static variables of a class:

We can delete static variables from anywhere by using the following syntax
del classname.variablename
But inside classmethod we can also use cls variable
del cls.variablename

```
class Test:
 a=10
 @classmethod
 def m1(cls):
   del cls.a
Test.m1()
print(Test.__dict__)
```

```
class Test:
  a=10
  def __init__(self):
     Test.b=20
     del Test.a

  def m1(self):
   Test.c=30
   del Test.b
  @classmethod
  def m2(cls):
   cls.d=40
   del Test.c
  @staticmethod
  def m3():
   Test.e=50
   del Test.d
print(Test.__dict__)
t=Test()
print(Test.__dict__)
t.m1()
print(Test.__dict__)
Test.m2()
print(Test.__dict__)
Test.m3()
print(Test.__dict__)
Test.f=60
print(Test.__dict__)
del Test.e
print(Test.__dict__)
```

1. Instance Methods:
Inside method implementation if we are using instance variables then such type of methods are
called instance methods.
Inside instance method declaration,we have to pass self variable.

```
 def m1(self):
```

By using self variable inside method we can able to access instance variables.
Within the class we can call instance method by using self variable and from outside of the class
we can call by using object reference.

```
class studentt:
    def __init__(self, name, marks):
```

```
        self.name = name
        self.marks = marks

    def display(self):
        print(self.name)
        print(self.marks)

    def grade(self):
        if self.marks>=60:
            print(self.marks)
        else:
            print(self.marks)

n = int(input('enter student no:'))
for x in range(n):
    name = input('name:')
    marks = int(input('marks:'))
    s = studentt(name,marks)
    s.display()
    s.grade()
```

Uses of classmethod() classmethod() function is used in factory design patterns where we want to call many functions with the class name rather than object.

The @classmethod Decorator:

inside method implementation if we are using only class variables that kind of method is class method

```
class dog:
    legs = 4

    @classmethod
    def walk(cls,name):
        print('{} walks with{} legs..'.format(name,cls.legs))

dog.walk('doggies')
```

Python staticmethod()
inside of this method  we will nott use any instance or class variables that kind of method static method

Example

```
#staic method

class math:
    @staticmethod
    def add(x,y):
        print('sum of the numbers:',x+y)

    @staticmethod
    def sub(x,y):
        print('sub of the numbers:', x - y)

    @staticmethod
```

```
    def mul(x,y):
        print('mul of the numbers:', x * y)


math.add(4,5)
math.sub(7,5)
math.mul(4,2)
```

ex:2:

```
class A:

    def add(a,b):
        return a+b

    def mul(a,b):
        return a*b

print(A.mul(5,6))
```

GETTER METHOD & SETTER METHOD:

SETTER METHOD:
setter method used to set values to the instance variables,in other word we can mutating

GETTER METHOD:
getter method used to get vallues of the instance variables, in other accessoor method

```
class student:
    def setname(self, name):
        self.name = name


    def getname(self):
        return self.name

    def setmarks(self,marks):
        self.marks = marks

    def getmarks(self):
        return self.marks

n=2
for x in range(n):
    s = student()
    name = input('name:')
    s.setname(name)
    marks = int(input('marks:'))
    s.setmarks(marks)
    print('hi',s.getname())
    print('your marks',s.getmarks())
```

INNER Class:

a class which is availbale inside of another class that is known as inner class

```
class outer:
    def __init__(self):
        print('this is outer')

    class inner:
        def __init__(self):
            print('this is inner')

        def m1(self):
            print('this is inner method')

o =outer()
i=o.inner()
i.m1()
```

4.INHERITENCE

Python Inheritance
Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class
Any class can be a parent class, so the syntax is the same as creating any other class:

Example

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print('this is first',self.firstname,'this is last', self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)

x = Student("Mike", "Olsen")
x.printname()
```

Add Properties
Example
Add a property called graduationyear to the Student class:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019

x = Student("Mike", "Olsen")
print(x.graduationyear)
print(x.lastname)
print(x.firstname)

x.printname()

Add Methods
Example
Add a method called welcome to the Student class:

class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)

x = Student("Mike", "Olsen", 2019)
x.welcome()


MULTIPLE INHERITENC:


#multiple inheritence:
'''

when a class is derived or inherited from more than one base class it is called
multiple inheritence
'''

class Class1:
    def m(self):
        print('class first')
```

```
class Class2:
    def m(self):
        print('class second')



class Class3:
    def m(self):
        print('class third')



class Class4(Class1,Class2):
    def m(self):
        print('class four')
        Class1.m(self)
        Class2.m(self)

obj = Class4()
obj.m()
```

MULTI LEVEl INHERITENCE:

a class which is inheritening methods and feauturres from already inherited class that is know multilevel inheritence

```
class Class1:
    def m(self):
        print('class first')



class Class2(Class1):
    def m(self):
        print('class second')



class Class3(Class1):
    def m(self):
        print('class third')



class Class4(Class3,Class2):
    def m(self):
        print('class four')
        Class1.m(self)
        Class2.m(self)
        Class3.m(self)

obj = Class4()
obj.m()
```

5.POLYMORPHISM

Polymorphism in Python
What is Polymorphism: The word polymorphism means having many forms.
In programming, polymorphism means the same function
name (but different signatures) being used for different types.

Polymorphism with class methods:
The below code shows how Python can use two different class types, in the same way.

We create a for loop that iterates through a tuple of objects. Then call the
methods
without being concerned about which class type each object is. We assume that these

methods actually exist in each class.
duck typing:

```
class India():
      def capital(self):
            print("New Delhi is the capital of India.")

      def language(self):
            print("Hindi is the most widely spoken language of India.")

      def type(self):
            print("India is a developing country.")

class USA():
      def capital(self):
            print("Washington, D.C. is the capital of USA.")

      def language(self):
            print("English is the primary language of USA.")

      def type(self):
            print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
      country.capital()
      country.language()
      country.type()


class Duck():
  def talk(self):
    print(' quack..')

class cat():
  def talk(self):
    print(' meow')

class Dog():
  def talk(self):
    print('bow bow')
```

```
class wolf():
  def talk(self):
    print('yelling')


def f1(obj):
  obj.talk()


i = [Duck(),cat(),Dog(),wolf()]

for x in i:
    f1(x)
```

Polymorphism with Inheritance:
In Python, Polymorphism lets us define methods in the child class that have the same name as the
methods in the parent class. In inheritance, the child class inherits the methods from the parent class.
However, it is possible to modify a method in a child class that it has inherited from the parent class.
This is particularly useful in cases where the method inherited from the parent class doesn’t quite fit
the child class. In such cases, we re-implement the method in the child class. This process of
re-implementing a method in the child class is known as Method Overriding.

```
class Bird:

def intro(self):
      print("There are many types of birds.")

def flight(self):
      print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
def flight(self):
      print("Sparrows can fly.")

class ostrich(Bird):
def flight(self):
      print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

operator overloading:

we can use same operator for multiple pupose that kind of operators known operator overloading

```
class Book:
  def __init__(self,pages):
    self.pages = pages

  def __add__(self, other):
    return self.pages+other.pages

b1 = Book(100)
b2 = Book(300)
print(b1+b2)
```

use of the operator overloading here useed to add two objects which is not at all possible thrugh arithmetic option

method overloading:

```
#method 2 methods having same name but different type of arguments then those methods are calling overloaded
 methods

#the last method is going override the previous methods for get a result we have to pass two arguments
class Test:
  def m1(self):
    print('no-arguments')


  def m1(self,a):
    print('one-arguments')


  def m1(self,a,b):
    print('two-arguments')

t = Test()
t.m1(10,20)



#method 2 methods having same name but different type of arguments then those methods are calling overloaded methods

#the last method is going override the previous methods for get a result we have to pass two arguments
class Test:
  def sum(self, a = None, b =None, c = None):
    if a!=None and b !=None and c !=None:

      print('sum of the 3 numbers:', a+b+c)
    elif a != None and b != None :
      print('sum of the 2 numbers:', a + b)
    else:
      print('passed number',a)

t = Test()
t.sum(3,4,5)
```

```
t.sum(10,20)
t.sum(2)


#constructor overloading:

#like wise method if we are mentioning multiple constructors  then the last
constructor will get execute


class Test:
  def __init__(self):
    print('first constructor')


  def __init__(self,a):
    print('second constructor')


  def __init__(self,a,b):
    print('third constructor')

#Test()
#Test(2)

Test(2,3)


#method overriding

'''
if method is available in the parent class  while inheritence if the child class
not satisfied with parent class we can
mention  our own child class method this kind overriding method overriding
'''


class a:
  def property(self):
    print('dollar')

  def aim(self):
    print('python')

class b(a):
  def aim(self):
    print('javascript')
c = b()
c.property()
c.aim()#here aim method got overriden by class b


#constructor overriding

class a:
  def __init__(self):
    print('parent')

class b(a):
```

```
  def __init__(self):
    print('child')

b()
```

GENERATORS AND ITERATORS(2HRS)

Python generators are a simple way of creating iterators. All the work we mentioned above are automatically
handled by generators in Python. Simply speaking, a generator is a function that returns an object (iterator)
which we can iterate over (one value at a time).

Let's see the difference between Iterators and Generators in python. ... An iterator does not make use of
local variables, all it needs is iterable to iterate on. A generator may have any number of 'yield' statements.
You can implement your own iterator using a python class; a generator does not need a class in python.

1.ITERATORS

Python Iterators
An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__()
and __next__().

Iterator vs Iterable
Lists, tuples, dictionaries, and sets are all iterable objects.
They are iterable containers which you can get an iterator from.
.
All these objects have a iter() method which is used to get an iterator:

Example
Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

Even strings are iterable objects, and can return an iterator:

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Example
Iterate the values of a tuple

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

Example
Iterate the characters of a string:

```
mystr = "banana"

for x in mystr:
  print(x)
```

Create an Iterator
To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called __init__(), which allows you
to do some initializing when the object is being created.

The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator
object itself.

The __next__() method also allows you to do operations, and must return the next item in the sequence.

Example
Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self //we have to return self(reference variable only
```

```
  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration
The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration to go on forever, we can use the StopIteration statement.

In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example
Stop after 20 iterations

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration  ,  # without giving stop iteration it is going to loop againg and againg

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

2.GENERATORS
Generators in Python
Difficulty Level : Easy
Last Updated : 31 Mar, 2020
Prerequisites: Yield Keyword and Iterators

There are two terms involved when we discuss generators.

Generator-Function : A generator-function is defined like a normal function, but whenever it needs to generate
 a value,

```
 it does so with the yield keyword rather than return.
 If the body of a def contains yield, the function automatically becomes a
generator function.
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
      yield 1
      yield 2
      yield 3

# Driver code to check above generator function
 for value in simpleGeneratorFun():
      print(value)


example:



def simpleGeneratorFun():
      yield 1
      yield 2
      yield 3
'''
whenever instead printing values the print() function returning address of a
sequence use for
 loop to print the values

print(simpleGeneratorFun())

print(simpleGeneratorFun())

print(simpleGeneratorFun())
'''
for s in simpleGeneratorFun():
  print(s)




Generator-Object : Generator functions return a generator object. Generator objects
are used either by calling the
next method on the generator object or using the generator object in a “for in”
loop (as shown in the above program).
# A Python program to demonstrate use of
# generator object with next()

# A generator function
def simpleGeneratorFun():
      yield 1
      yield 2
      yield 3

# x is a generator object
x = simpleGeneratorFun()

# Iterating over the generator object using next
print(x.next()) # In Python 3, __next__()
```

```
print(x.next())
print(x.next())
```

example:

```
# A generator function
def simpleGeneratorFun():
      yield 1
      yield 2
      yield 3

# x is a generator object
x = simpleGeneratorFun()

# Iterating over the generator object using next
print(x.__next__()) # In Python 3, __next__()
print(x.__next__())
print(x.__next__())

#print(x.__next__())  #whenever we are crossing our bound it will give us
stopiteration



# A simple generator for Fibonacci Numbers
def fib(limit):

      # Initialize first two Fibonacci Numbers
      a, b = 0, 1

      # One by one yield next Fibonacci Number
      while a < limit:
            yield a
            a, b = b, a + b

# Create a generator object
x = fib(5)

# Iterating over the generator object using next
print(x.next()) # In Python 3, __next__()
print(x.next())
print(x.next())
print(x.next())
print(x.next())

# Iterating over the generator object using for
# in loop.
print("\nUsing for in loop")
for i in fib(5):
      print(i)
```

3.THE FUNCTIONS ANY AND ALL

Any All in Python

Any

Returns true if any of the items is True. It returns False if empty or all are false.
 Any can be thought of as a sequence of OR operations on the provided iterables. It short circuit the execution i.e. stop the execution as soon as the result is known.

Syntax : any(list of iterables)

```
# Since all are false, false is returned
print (any([False, False, False, False]))

# Here the method will short-circuit at the
# second item (True) and will return True.
print (any([False, True, False, False]))

# Here the method will short-circuit at the
# first (True) and will return True.
print (any([True, False, False, False]))
```

All
Returns true if all of the items are True (or if the iterable is empty).
All can be thought of as a sequence of AND operations on the provided iterables.
 It also short circuit the execution i.e. stop the execution as soon as the result is known.

Syntax : all(list of iterables)

```
# Here all the iterables are True so all
# will return True and the same will be printed
print (all([True, True, True, True]))

# Here the method will short-circuit at the
# first item (False) and will return False.
print (all([False, True, True, False]))

# This statement will return False, as no
# True is found in the iterables
print (all([False, False, False]))


# This code explains how can we
# use 'any' function on list
list1 = []
list2 = []

# Index ranges from 1 to 10 to multiply
for i in range(1,11):
      list1.append(4*i)

# Index to access the list2 is from 0 to 9
for i in range(0,10):
      list2.append(list1[i]%5==0)

print('See whether at least one number is divisible by 5 in list 1=>')
print(any(list2))

# Illustration of 'all' function in python 3
```

```
# Take two lists
list1=[]
list2=[]

# All numbers in list1 are in form: 4*i-3
for i in range(1,21):
      list1.append(4*i-3)

# list2 stores info of odd numbers in list1
for i in range(0,20):
      list2.append(list1[i]%2==1)

print('See whether all numbers in list1 are odd =>')
print(all(list2))
```

Whereas, if any() checked the values, the output would have been False.

The method any() is often used in combination with the map() method and list comprehensions:

```
old_list = [2, 1, 3, 8, 10, 11, 13]
list_if_even = list(map(lambda x: x % 2 == 0, old_list))
list_if_odd = [x % 2 != 0 for x in old_list]

print(list_if_even)
print(list_if_odd)

print("Are any of the elements even? " + str(any(list_if_even)))
print("Are any of the elements odd? " + str(any(list_if_odd)))
```

5.DATA COMPRESSION

In python, the data can be archived, compressed using the modules like zlib, gzip, bz2,lzma,zipfile
and tarfile. ... To use the respective module, you need to import the module first.

Example:

```
import zlib

s = b' hi hello guys this is the python class'

print(len(s))

z= zlib.compress(s)

print(z)

t = zlib.decompress(z)
print(t)
```

7.SPECIALIZED SORTS

Custom Sorting using the key parameter
sorted() function has an optional parameter called 'key' which takes a function as its value.
 This key function transforms each element before sorting, it takes the value and returns 1 value
 which is then used within sort instead of the original value.

```
# List
x = ['q', 'w', 'r', 'e', 't', 'y']
print (sorted(x))

# Tuple
x = ('q', 'w', 'e', 'r', 't', 'y')
print (sorted(x))


x = "python"
print (sorted(x))

# Dictionary
x = {'q':1, 'w':2, 'e':3, 'r':4, 't':5, 'y':6}
print (sorted(x))

# Set
x = {'q', 'w', 'e', 'r', 't', 'y'}
print (sorted(x))

# Frozen Set
x = frozenset(('q', 'w', 'e', 'r', 't', 'y'))
print (sorted(x))


L = ["cccc", "b", "dd", "aaa"]

print ("Normal sort :", sorted(L))

print ("Sort with len :", sorted(L, key = len))
```

Key also takes user-defined functions as its value for the basis of sorting.

```
# Sort a list of integers based on
# their remainder on dividing from 7

def func(x):
      return x % 7

L = [15, 3, 11, 7]

print ("Normal sort :", sorted(L))
print ("Sorted with key:", sorted(L, key = func))
```

COLLECTIONS:(2HR)

Collections is a built-in Python module that implements specialized container datatypes providing
 alternatives to Python's general purpose built-in containers such as dict , list , set , and tuple

Table of Content:



## Counters

A counter is a sub-class of the dictionary. It is used to keep the count of the elements in an iterable in
the form of an unordered dictionary where the key represents the element in the iterable and value represents
 the count of that element in the iterable.

Note: It is equivalent to bag or multiset of other languages.

Initializing Counter Objects
The counter object can be initialized using the counter() function and this function can be called
 in one of the following ways:

With a sequence of items
With a dictionary containing keys and counts
With keyword arguments mapping string names to counts

```
# A Python program to show different
# ways to create Counter
from collections import Counter

# With sequence of items
print(Counter(['B','B','A','B','C','A','B',
                  'B','A','C']))

# with dictionary
print(Counter({'A':3, 'B':5, 'C':2}))

# with keyword arguments
print(Counter(A=3, B=5, C=2))
```

## OrderedDict

An OrderedDict is also a sub-class of dictionary but unlike dictionary,
it remembers the order in which the keys were inserted.

```
# A Python program to demonstrate working
# of OrderedDict

from collections import OrderedDict

print("This is a Dict:\n")
d = {}
d['a'] = 1
d['b'] = 2
d['c'] = 3
d['d'] = 4

for key, value in d.items():
      print(key, value)

print("\nThis is an Ordered Dict:\n")
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4

for key, value in od.items():
      print(key, value)
```

While deleting and re-inserting the same key will push the key to the last to maintain the order of insertion of the key.

Example:

```
# A Python program to demonstrate working
# of OrderedDict

from collections import OrderedDict


od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4

print('Before Deleting')
for key, value in od.items():
    print(key, value)

# deleting element
od.pop('a')

# Re-inserting the same
od['a'] = 1

print('\nAfter re-inserting')
for key, value in od.items():
    print(key, value)
```

DefaultDict:

DefaultDict
Defaultdict is a container like dictionaries present in the module collections. Defaultdict is
 a sub-class of the dictionary class that returns a dictionary-like object. The functionality of
 both dictionaries and defualtdict are almost same except for the fact that defualtdict never raises a KeyError.
 It provides a default value for the key that does not exists.

```
# Python program to demonstrate
# defaultdict


from collections import defaultdict


# Function to return a default
# values for keys that is not
# present
def def_value():
      return "Not Present"

# Defining the dict
d = defaultdict(def_value)
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

LAMDA:

```
# Python program to demonstrate
# default_factory argument of
# defaultdict


from collections import defaultdict


# Defining the dict and passing
# lambda as default_factory argument
d = defaultdict(lambda: "Not Present")
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

__missing__(): This function is used to provide the default value for the dictionary. This function takes
 default_factory as an argument and if this argument is None, a KeyError is raised otherwise it provides a
 default value for the given key. This method is basically called by the __getitem__() method of the dict
 class when the requested key is not found. __getitem__() raises or return the value returned by the __missing__(). method.

```
# Python program to demonstrate
# defaultdict


from collections import defaultdict


# Defining the dict
d = defaultdict(lambda: "Not Present")
d["a"] = 1
d["b"] = 2

# Provides the default value
# for the key
print(d.__missing__('a'))
print(d.__missing__('d'))
```

ChainMap
A ChainMap encapsulates many dictionaries into a single unit and returns a list of dictionaries.

```
# Python program to demonstrate
# ChainMap


from collections import ChainMap

example:
#chainmap

d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}
d3 = {'e': 5, 'f': 6}
d4 = {'x':3,'y':7}

# Defining the chainmap if you are trying to print in dictionary method you will find same result with or without chain map
c = ChainMap(d1, d2, d3)

print(c)
#the key will fetchable from the reverse method
b = list(ChainMap(d1,d2,d3,d4))
print(b)
```

Values from ChainMap can be accessed using the key name. They can also be accessed

by using the keys() and values() method.

```
# Python program to demonstrate
# ChainMap


from collections import ChainMap


d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}
d3 = {'e': 5, 'f': 6}

# Defining the chainmap
c = ChainMap(d1, d2, d3)

#single key can be accesible
# Accessing Values using key name
print(c['a'])

#here we are not able to split key and values using values or key method we will
get the tottal result

# Accesing values using values()
# method
print(c.values())

# Accessing keys using keys()
# method
print(c.keys())
```

Adding new dictionary
A new dictionary can be added by using the new_child() method. The newly added dictionary is added at the beginning of the ChainMap.

```
# Python code to demonstrate ChainMap and
# new_child()

import collections

# initializing dictionaries
dic1 = { 'a' : 1, 'b' : 2 }
dic2 = { 'b' : 3, 'c' : 4 }
dic3 = { 'f' : 5 }

# initializing ChainMap
chain = collections.ChainMap(dic1, dic2)

# printing chainMap
print ("All the ChainMap contents are : ")
print (chain)

# using new_child() to add new dictionary
chain1 = chain.new_child(dic3)
```

```
# printing chainMap
print ("Displaying new ChainMap : ")
print (chain1)
```

example:

```
dic1 = { 'a' : 1, 'b' : 2 }
dic2 = { 'b' : 3, 'c' : 4 }
dic3 = { 'f' : 5 }


chain = ChainMap(dic1, dic2)


print ("All the ChainMap contents are : ")
print (chain)

# using new_child() to add new dictionary in the first place
chain1 = chain.new_child(dic3)

# printing chainMap
print ("Displaying new ChainMap : ")
print (chain1)
```

NamedTuple
A NamedTuple returns a tuple object with names for each position which the ordinary tuples lack.
For example, consider a tuple names student where the first element represents fname, second represents
lname and the third element represents the DOB. Suppose for calling fname instead of remembering the index

position you can actually call the element by using the fname argument, then it will be really easy for
accessing tuples element. This functionality is provided by the NamedTuple.

```
# Python code to demonstrate namedtuple()

from collections import namedtuple

# Declaring namedtuple()
Student = namedtuple('Student',['name','age','DOB'])

# Adding values
S = Student('Nandini','19','2541997')

# Access using index
print ("The Student age using index is : ",end ="")
print (S[1])

# Access using name
print ("The Student name using keyname is : ",end ="")
print (S.name)
```

Conversion Operations
1. _make(): This function is used to return a namedtuple() from the iterable passed as argument.

2. _asdict(): This function returns the OrdereDict() as constructed from the mapped values of namedtuple().

```
# Python code to demonstrate namedtuple() and
# _make(), _asdict()


from collections import namedtuple

# Declaring namedtuple()
Student = namedtuple('Student',['name','age','DOB'])

# Adding values
S = Student('Nandini','19','2541997')

# initializing iterable
li = ['Manjeet', '19', '411997' ]

# initializing dict
di = { 'name' : "Nikhil", 'age' : 19 , 'DOB' : '1391997' }

# using _make() to return namedtuple()
print ("The namedtuple instance using iterable is : ")
print (Student._make(li))

# using _asdict() to return an OrderedDict()
print ("The OrderedDict instance using namedtuple is : ")
print (S._asdict())
```

Deque
Deque (Doubly Ended Queue) is the optimized list for quicker append and pop operations from both sides of the
container. It provides O(1) time complexity for append and pop operations as compared to list with O(n) time
 complexity.

```
# Python code to demonstrate deque


from collections import deque

# Declaring deque
queue = deque(['name','age','DOB'])

print(queue)
```

Inserting Elements
Elements in deque can be inserted from both ends. To insert the elements from right append() method is used and to insert the elements from the left appendleft()

method is used.

```
# Python code to demonstrate working of
# append(), appendleft()


from collections import deque

# initializing deque
de = deque([1,2,3])

# using append() to insert element at right end
# inserts 4 at the end of deque
de.append(4)

# printing modified deque
print ("The deque after appending at right is : ")
print (de)

# using appendleft() to insert element at right end
# inserts 6 at the beginning of deque
de.appendleft(6)

# printing modified deque
print ("The deque after appending at left is : ")
print (de)
```

Removing Elements
Elements can also be removed from the deque from both the ends. To remove elements from right use pop() method and to remove elements from the left use popleft() method.

```
# Python code to demonstrate working of
# pop(), and popleft()

from collections import deque

# initializing deque
de = deque([6, 1, 2, 3, 4])

# using pop() to delete element from right end
# deletes 4 from the right end of deque
de.pop()

# printing modified deque
print ("The deque after deleting from right is : ")
print (de)

# using popleft() to delete element from left end
# deletes 6 from the left end of deque
de.popleft()

# printing modified deque
print ("The deque after deleting from left is : ")
print (de)
```

UserDict

Python supports a dictionary like a container called UserDict present in the collections module. This class acts as a wrapper class around the dictionary objects. This class is useful when one wants to create a dictionary of their own with some modified functionality or with some new functionality. It can be considered as a way of adding new behaviors for the dictionary. This class takes a dictionary instance as an argument and simulates a dictionary that is kept in a regular dictionary. The dictionary is accessible by the data attribute of this class.

```
# Python program to demonstrate
# userdict


from collections import UserDict


d = {'a':1,
      'b': 2,
      'c': 3}

# Creating an UserDict
userD = UserDict(d)
print(userD.data)


# Creating an empty UserDict
userD = UserDict()
print(userD.data)
```

Example 2: Let’s create a class inherting from UserDict to implement a customised dictionary.

```
# Python program to demonstrate
# userdict


from collections import UserDict


# Creating a Dictionary where
# deletion is not allowed
class MyDict(UserDict):

      # Function to stop deletion
      # from dictionary
      def __del__(self):
            raise RuntimeError("Deletion not allowed")

      # Function to stop pop from
      # dictionary
      def pop(self, s = None):
            raise RuntimeError("Deletion not allowed")
```

```
      # Function to stop popitem
      # from Dictionary
      def popitem(self, s = None):
            raise RuntimeError("Deletion not allowed")

# Driver's code
d = MyDict({'a':1,
      'b': 2,
      'c': 3})

print("Original Dictionary")
print(d)

d.pop(1)
```

UserList
Collections.UserList
Python supports a List like a container called UserList present in the collections module. This class acts as a wrapper class around the List objects. This class is useful when one wants to create a list of their own with some modified functionality or with some new functionality. It can be considered as a way of adding new behaviors for the list. This class takes a list instance as an argument and simulates a list that is kept in a regular list. The list is accessible by the data attribute of the this class.

```
# Python program to demonstrate
# userlist


from collections import UserList


L = [1, 2, 3, 4]

# Creating a userlist
userL = UserList(L)
print(userL.data)


# Creating empty userlist
userL = UserList()
print(userL.data)
```

ex :2:

```
class list(UserList):
    def remove(self, s=None):
        raise  RuntimeError('Deletion is not allowed')

    def pop(self, s =None):
        raise RuntimeError('pop is not allowed')
```

```
l = list([2,4,5,6])
print(l)
l.append(5)
print(l)
#l.remove(5)
#print(l)

l.pop()
print(l)
```

Strings are the arrays of bytes representing Unicode characters. However, Python does not support the character data type. A character is a string of length one.

```
# Python program to demonstrate
# string

# Creating a String
# with single Quotes
String1 = 'Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)

# Creating a String
# with double Quotes
String1 = "I'm a Geek"
print("\nString with the use of Double Quotes: ")
print(String1)
```

Python supports a String like a container called UserString present in the collections module. This class acts as a wrapper class around the string objects. This class is useful when one wants to create a string of their own with some modified functionality or with some new functionality. It can be considered as a way of adding new behaviors for the string. This class takes any argument that can be converted to string and simulates a string whose content is kept in a regular string. The string is accessible by the data attribute of this class.

```
# Python program to demonstrate
# userstring


from collections import UserString


d = 12344

# Creating an UserDict
userS = UserString(d)
print(userS.data)


# Creating an empty UserDict
userS = UserString("")
print(userS.data)
```

ADVANCE PYTHON(14HRS)

WRITING GUIS IN PYTHON (TKINTER)(2HRS)

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.

Tkinter − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option
in this chapter.

what is graphicaluser interface:
it is a desktop application which is helping you to interact with computer
1.text editor
2.games
3.apps
use of gui is reducing coding to open anything on your system without gui we have to use cmd prompt to interact with your computer

GUi python libraries:

1.kivy
2.wx pythoon
3.tkinter
4.matplotlib

1.INTRODUCTION

Tkinter Programming

Tkinter -- is a inbuild library of python you dont have to intall that externally

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to
create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

in other words:
Tkinter in python GUI programming is standard GUI library it will give us object oriented interface to the TK
GUI toolkit

fundamentals to used tkinter:

1.import tkinter
2.create GUI app in main window
3.add widgets
4.enter the main event loop

basic example:

```
import tkinter
```

```
window = tkinter.Tk()
window.title("My first")//title for your GUI
label = tkinter.Label(window, text = "hello world").pack()//content of your gui
window.mainloop()//it is a infinite loop which can hold your gui forever until you
click the cut button


#mainloop is used  as an infinite loop used to run the application until you are
going to close

#lable - is used to implement display boxes  where you can place text, image
anything that you want
a user can change the content


pack():
manager to code with tkinter.particular location we dont have to mention pack()
will automatically take care
 about that



Creating a GUI application using Tkinter is an easy task. All you need to do is
perform the following steps −


widgets:
      a widget is an element of a graphical user interface that displays
information or provides spacific way
      for a user to

interact with the operating system or an app

      1.label -- you can set the label font so you can make it bigger or bold
      2.button
      3.entry
      4.combobox
      5.checkbutton
      6.radio
      7.scrolledbox
      8.spinbox
      9.menubar
      10.NOtebook



label:




Python offers multiple options for developing GUI (Graphical User Interface). Out
of all the GUI methods, tkinter is the most commonly used method. It is a standard
Python interface to the Tk GUI toolkit shipped with Python. Python with tkinter is
the fastest and easiest way to create the GUI applications. Creating a GUI using
tkinter is an easy task.
To create a tkinter app:

button
```

eexample:

```
import tkinter *

from tkinter import ttk
window = Tk()
window.title("My first")
label = Label(window, text = "hello world" ,font = ("Arial Bold",30)).pack()
window.geometry('350x200')

def clicked():
    print("Button clicked")

bt = ttk.Button(window, text = "click" ,command = clicked)
bt.pack(side = 'top')
#bt.grid(column=20,row=0)
window.mainloop()
```

entry:

used to getting user inputs

```
import tkinter as tk

var = tk.Tk()

var.geometry('360x360')

nam = tk.StringVar()
pas = tk.StringVar()

def submit():
    name = nam.get()
    password = pas.get()
    print(name)
    print(password)
    nam.set("")
    pas.set("")


nam_label = tk.Label(var,text = 'enter your name')
nam_entry = tk.Entry(var,textvar = nam)
pas_label = tk.Label(var,text = 'enter your password')
pas_entry = tk.Entry(var,textvar = pas)

sub = tk.Button(var, text = 'Subbmit', command = submit )

nam_label.grid(row = 0,column = 0)
nam_entry.grid(row = 1,column = 0)
pas_label.grid(row = 2,column = 0)
pas_entry.grid(row = 3,column = 0)
sub.grid(row = 4,column = 0)

var.mainloop()
```

combobox:
Python provides a variety of GUI (Graphic User Interface)

types such as PyQT, Tkinter, Kivy, WxPython, and PySide. Among them,
 tkinter is the most commonly used GUI module in Python since it is
 simple and easy to understand. The word Tkinter comes from the Tk interface.
 The tkinter module is available in Python standard library which has to be
 imported while writing a program in Python to generate a GUI.
      # python program demonstrating


```
# Combobox widget using tkinter


import tkinter as tk
from tkinter import ttk

# Creating tkinter window
window = tk.Tk()
window.title('Combobox')
window.geometry('500x250')

# label text for title
ttk.Label(window, text = "GFG Combobox Widget",
            background = 'green', foreground ="white",
            font = ("Times New Roman", 15)).grid(row = 0, column = 1)

# label
ttk.Label(window, text = "Select the Month :",
            font = ("Times New Roman", 10)).grid(column = 0,
            row = 5, padx = 10, pady = 25)

# Combobox creation
n = tk.StringVar()
monthchoosen = ttk.Combobox(window, width = 27, textvariable = n)

# Adding combobox drop down list
monthchoosen['values'] = (' January',
                                    ' February',
                                    ' March',
                                    ' April',
                                    ' May',
                                    ' June',
                                    ' July',
                                    ' August',
                                    ' September',
                                    ' October',
                                    ' November',
                                    ' December')

monthchoosen.grid(column = 1, row = 5)
monthchoosen.current()//if you mention any values it choose the month as default
window.mainloop()
```

We can also set the initial default values in the Combobox widget as shown in the below sample code.

```
import tkinter as tk
from tkinter import ttk
```

```
# Creating tkinter window
window = tk.Tk()
window.geometry('350x250')
# Label
ttk.Label(window, text = "Select the Month :",
            font = ("Times New Roman", 10)).grid(column = 0,
            row = 15, padx = 10, pady = 25)

n = tk.StringVar()
monthchoosen = ttk.Combobox(window, width = 27,
                                        textvariable = n)

# Adding combobox drop down list
monthchoosen['values'] = (' January',
                                    ' February',
                                    ' March',
                                    ' April',
                                    ' May',
                                    ' June',
                                    ' July',
                                    ' August',
                                    ' September',
                                    ' October',
                                    ' November',
                                    ' December')

monthchoosen.grid(column = 1, row = 15)

# Shows february as a default value
monthchoosen.current(1)
window.mainloop()
```

5.checkbutton

The Checkbutton widget is a standard Tkinter widget that is used to implement on/off selections.

Checkbuttons can contain text or images.

When the button is pressed, Tkinter calls that function or method.

EXP 1:

```
from tkinter import *
master = Tk()
var1 = IntVar()
Checkbutton(master, text="male", variable=var1).grid(row=0, sticky=W)
var2 = IntVar()
Checkbutton(master, text="female", variable=var2).grid(row=1, sticky=W)
mainloop()
```

EXAMPLE 2:

```
from tkinter import *

root = Tk()
root.geometry("300x200")

w = Label(root, text ='GeeksForGeeks', font = "50")
w.pack()

Checkbutton1 = IntVar()
```

```
Checkbutton2 = IntVar()
Checkbutton3 = IntVar()

Button1 = Checkbutton(root, text = "Tutorial",
                              variable = Checkbutton1,
                              onvalue = 1,
                              offvalue = 0,
                              height = 2,
                              width = 10)

Button2 = Checkbutton(root, text = "Student",
                              variable = Checkbutton2,
                              onvalue = 1,
                              offvalue = 0,
                              height = 2,
                              width = 10)

Button3 = Checkbutton(root, text = "Courses",
                              variable = Checkbutton3,
                              onvalue = 1,
                              offvalue = 0,
                              height = 2,
                              width = 10)

Button1.pack()
Button2.pack()
Button3.pack()

mainloop()
#root.mainloop
```

6.RADIOBUTTON:

The Radiobutton is a standard Tkinter widget used to implement one-of-many selections.
Radiobuttons can contain text or images, and you can associate a Python function or
method with each button. When the button is pressed, Tkinter automatically calls that
function or method.

Igrid() method:It organizes the widgets in grid (table-like structure) before placing in the parent widget.
place() method:It organizes the widgets by placing them on specific positions directed by the programmer.
There are a number of widgets which you can put in your tkinter application. Some of the major widgets are explained below:

activebackground: to set the background color when button is under the cursor.
activeforeground: to set the foreground color when button is under the cursor.
bg: to set he normal background color.
command: to call a function.
font: to set the font on the button label.
image: to set the image on the button.
width: to set the width of the button.
height: to set the height of the button.

button EXP1:

```
import tkinter as tk
r = tk.Tk()
r.title('Counting Seconds')
button = tk.Button(r, text='Stop', width=25, command=r.destroy) //used destroy or
close the GUI
button.pack()
r.mainloop()
```

EXP2:

```
# Importing Tkinter module
from tkinter import *
# from tkinter.ttk import *

# Creating master Tkinter window
master = Tk()
master.geometry("175x175")

# Tkinter string variable
# able to store any string value
v = StringVar(master, "1")

# Dictionary to create multiple buttons
values = {"RadioButton 1" : "1",
            "RadioButton 2" : "2",
            "RadioButton 3" : "3",
            "RadioButton 4" : "4",
            "RadioButton 5" : "5"}

# Loop is used to create multiple Radiobuttons
# rather than creating each button separately
for (text, value) in values.items():
      Radiobutton(master, text = text, variable = v,
                        value = value, indicator = 0,
                        background = "light blue").pack(fill = X, ipady = 5)

# Infinite loop can be terminated by
# keyboard or mouse interrupt
# or by any predefined function (destroy())
mainloop()
```

Changing button boxes into standard radio buttons. For this remove indicatoron option.

```
 # Importing Tkinter module
from tkinter import *
from tkinter.ttk import *

# Creating master Tkinter window
master = Tk()
master.geometry("175x175")

# Tkinter string variable
# able to store any string value
v = StringVar(master, "1")
```

```
# Dictionary to create multiple buttons
values = {"RadioButton 1" : "1",
            "RadioButton 2" : "2",
            "RadioButton 3" : "3",
            "RadioButton 4" : "4",
            "RadioButton 5" : "5"}

# Loop is used to create multiple Radiobuttons
# rather than creating each button separately
for (text, value) in values.items():
      Radiobutton(master, text = text, variable = v,
            value = value).pack(side = TOP, ipady = 5)

# Infinite loop can be terminated by
# keyboard or mouse interrupt
# or by any predefined function (destroy())
mainloop()
```

Adding Style to Radio Button using style class.

```
# Importing Tkinter module
from tkinter import *
from tkinter.ttk import *

# Creating master Tkinter window
master = Tk()
master.geometry('175x175')

# Tkinter string variable
# able to store any string value
v = StringVar(master, "1")

# Style class to add style to Radiobutton
# it can be used to style any ttk widget
style = Style(master)
style.configure("TRadiobutton", background = "light green",  #trying to modify the
widget properties
                        foreground = "red", font = ("arial", 10, "bold"))

# Dictionary to create multiple buttons
values = {"RadioButton 1" : "1",
            "RadioButton 2" : "2",
            "RadioButton 3" : "3",
            "RadioButton 4" : "4",
            "RadioButton 5" : "5"}

# Loop is used to create multiple Radiobuttons
# rather than creating each button separately
for (text, value) in values.items():
      Radiobutton(master, text = text, variable = v,  #variable in tkinter used to
manipulate some vallues
                        value = value).pack(side = TOP, ipady = 5)

# Infinite loop can be terminated by
# keyboard or mouse interrupt
# or by any predefined function (destroy())
mainloop()
```

Scrollbar Widget
The scrollbar widget is used to scroll down the content.
 We can also create the horizontal scrollbars to the Entry widget.

Methods:
Methods used in this widgets are as follows:

get(): This method is used to returns the two numbers a and b which represents the current position
of the scrollbar.
set(first, last): This method is used to connect the scrollbar to the other widget w.
 The yscrollcommand or xscrollcommand
 of the other widget to this method.


example:

```
from tkinter import *

root = Tk()
root.geometry("150x200")

w = Label(root, text ='GeeksForGeeks',
            font = "50")

w.pack()

scroll_bar = Scrollbar(root)

scroll_bar.pack( side = RIGHT,  #setting bar side scroling style as vertically
                         fill = Y )

mylist = Listbox(root,
                         yscrollcommand = scroll_bar.set )

for line in range(1, 26):
      mylist.insert(END, "Geeks " + str(line))

mylist.pack( side = LEFT, fill = BOTH )

scroll_bar.config( command = mylist.yview )

root.mainloop()
```

example:

```
from tkinter import *

root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack( side = RIGHT, fill = Y )

mylist = Listbox(root, yscrollcommand = scrollbar.set )
for line in range(100):
   mylist.insert(END, "This is line number " + str(line))

mylist.pack( side = LEFT, fill = BOTH )  #fill the widget wants fill the entire
space assigned to it.
```

```
scrollbar.config( command = mylist.yview ) #used to access an objects attribute
after its initializing

mainloop()
```

spinbox:

The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select
from a fixed number of values.

example:

```
from tkinter import *

master = Tk()

w = Spinbox(master, from_=0, to=10)
w.pack()

mainloop()
```

example 2:

```
import tkinter as tk
from tkinter import ttk #used to style the tkinter widgets inside of it spinbox is
available


# root window
root = tk.Tk()
root.geometry('300x200')
root.resizable(False, False)
root.title('Spinbox Demo')

# Spinbox
current_value = tk.StringVar(value=0)
spin_box = ttk.Spinbox(
    root,
    from_=0,
    to=30,
    textvariable=current_value,
    wrap=True)

spin_box.pack()

root.mainloop()
```

Python Tkinter Menu
The Menu widget is used to create various types of menus (top level, pull down, and pop up) in the python application.

The top-level menus are the one which is displayed just under the title bar of the parent window.
We need to create a new instance of the Menu widget and add various commands
to it by using the add() method.

example:

```
from tkinter import *

top = Tk()


def hello():
    print("hello!")


# create a toplevel menu
menubar = Menu(top)
menubar.add_command(label="Hello!", command=hello)
menubar.add_command(label="Quit!", command=top.quit)

# display the menu
top.config(menu=menubar)

top.mainloop()
```

example:

```
from tkinter import Toplevel, Button, Tk, Menu

top = Tk()
menubar = Menu(top)
file = Menu(menubar, tearoff=0)  #permits you to detach menus for most window
making floating menus
file.add_command(label="New")
file.add_command(label="Open")
file.add_command(label="Save")
file.add_command(label="Save as...")
file.add_command(label="Close")

file.add_separator()

file.add_command(label="Exit", command=top.quit)

menubar.add_cascade(label="File", menu=file)
edit = Menu(menubar, tearoff=0)
edit.add_command(label="Undo")

edit.add_separator()

edit.add_command(label="Cut")
edit.add_command(label="Copy")
edit.add_command(label="Paste")
edit.add_command(label="Delete")
edit.add_command(label="Select All")

menubar.add_cascade(label="Edit", menu=edit)
help = Menu(menubar, tearoff=0)
help.add_command(label="About")
menubar.add_cascade(label="Help", menu=help) #used to create multiple types of
menus

top.config(menu=menubar)
```

```
top.mainloop()
```

example2:

```
from tkinter import Toplevel, Button, Tk, Menu

top = Tk()
menubar = Menu(top)
file = Menu(menubar, tearoff=0)
file.add_command(label="New")
file.add_command(label="Open")
file.add_command(label="Save")
file.add_command(label="Save as...")
file.add_command(label="Close")

file.add_separator()

file.add_command(label="Exit", command=top.quit)

menubar.add_cascade(label="File", menu=file)
edit = Menu(menubar, tearoff=0)
edit.add_command(label="Undo")

edit.add_separator()

edit.add_command(label="Cut")
edit.add_command(label="Copy")
edit.add_command(label="Paste")
edit.add_command(label="Delete")
edit.add_command(label="Select All")

menubar.add_cascade(label="Edit", menu=edit)
help = Menu(menubar, tearoff=0)
help.add_command(label="About")
menubar.add_cascade(label="Help", menu=help)

top.config(menu=menubar)
top.mainloop()
```

Frame: It acts as a container to hold the widgets. It is used for grouping and organizing the widgets.
The general syntax is:
w = Frame(master, option=value)
master is the parameter used to represent the parent window.
There are number of options which are used to change the format of the widget.
Number of options can be passed as parameters separated by commas. Some of them are listed below.

highlightcolor: To set the color of the focus highlight when widget has to be focused.
bd: to set the border width in pixels.
bg: to set the normal background color.
cursor: to set the cursor used.
width: to set the width of the widget.
height: to set the height of the widget.

EXAMPLE:

```
from tkinter import *
```

```
root = Tk()
frame = Frame(root)
frame.pack()
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )
redbutton = Button(frame, text = 'Red', fg ='red')
redbutton.pack( side = LEFT)
greenbutton = Button(frame, text = 'Brown', fg='brown')
greenbutton.pack( side = LEFT )
bluebutton = Button(frame, text ='Blue', fg ='blue')
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text ='Black', fg ='black')
blackbutton.pack( side = BOTTOM)
root.mainloop()
```

Listbox: It offers a list to the user from which the user can accept any number of options.
The general syntax is:
w = Listbox(master, option=value)
master is the parameter used to represent the parent window.
There are number of options which are used to change the format of the widget.
Number of options can be passed as parameters separated by commas. Some of them are listed below.

highlightcolor: To set the color of the focus highlight when widget has to be focused.
bg: to set he normal background color.
bd: to set the border width in pixels.
font: to set the font on the button label.
image: to set the image on the widget.
width: to set the width of the widget.
height: to set the height of the widget.

EXAMPLE:

```
from tkinter import *

top = Tk()
Lb = Listbox(top)
Lb.insert(1, 'Python')
Lb.insert(2, 'Java')
Lb.insert(3, 'C++')
Lb.insert(4, 'Any other')
Lb.pack()
top.mainloop()
```

Menu: It is used to create all kinds of menus used by the application.
The general syntax is:
w = Menu(master, option=value)
master is the parameter used to represent the parent window.
There are number of options which are used to change the format of this widget.
Number of options can be passed as parameters separated by commas. Some of them are listed below.

title: To set the title of the widget.
activebackground: to set the background color when widget is under the cursor.
activeforeground: to set the foreground color when widget is under the cursor.

bg: to set he normal background color.
command: to call a function.
font: to set the font on the button label.
image: to set the image on the widget.

EXAMPLE:

```
from tkinter import *

root = Tk()
menu = Menu(root)
root.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New')
filemenu.add_command(label='Open...')
filemenu.add_separator()
filemenu.add_command(label='Exit', command=root.quit)
helpmenu = Menu(menu)
menu.add_cascade(label='Help', menu=helpmenu)
helpmenu.add_command(label='About')
mainloop()
```

#pane ==layour or blanket
PannedWindowIt is a container widget which is used to handle number of panes arranged in it.
The general syntax is:
w = PannedWindow(master, option=value)
master is the parameter used to represent the parent window.
There are number of options which are used to change the format of the widget.
Number of options can be passed as parameters separated by commas. Some of them are listed below.

bg: to set he normal background color.
bd: to set the size of border around the indicator.
cursor: To appear the cursor when the mouse over the menubutton.
width: to set the width of the widget.
height: to set the height of the widget.

EXAMPLE:

```
from tkinter import *
m1 = PanedWindow()
m1.pack(fill = BOTH, expand = 1)
left = Entry(m1, bd = 5)
m1.add(left)
m2 = PanedWindow(m1, orient = VERTICAL)
m1.add(m2)
top = Scale( m2, orient = HORIZONTAL)
m2.add(top)
mainloop()
```

PYTHON SQL DATABASE ACCESS(3HRS)
Python MySQL

MySQL is an open-source relational database management system. Its name is a combination of "My",
the name of co-founder Michael Widenius's daughter, and "SQL", the abbreviation for

Structured
Query Language

MySQL is a database management system.

MySQL is a widely used relational database management system (RDBMS).

MySQL is free and open-source.

MySQL is ideal for both small and large applications.

MySQL is a very popular open-source relational database management system (RDBMS).

What is MySQL?
MySQL is a relational database management system
MySQL is open-source
MySQL is free
MySQL is ideal for both small and large applications
MySQL is very fast, reliable, scalable, and easy to use
MySQL is cross-platform
MySQL is compliant with the ANSI SQL standard
MySQL was first released in 1995
MySQL is developed, distributed, and supported by Oracle Corporation
MySQL is named after co-founder Monty Widenius's daughter: My

Who Uses MySQL?
Huge websites like Facebook, Twitter, Airbnb, Booking.com, Uber, GitHub, YouTube, etc.
Content Management Systems like WordPress, Drupal, Joomla!, Contao, etc.
A very large number of web developers around the world
Show Data On Your Web Site
To build a web site that shows data from a database, you will need:

An RDBMS database program (like MySQL)
A server-side scripting language, like PHP
To use SQL to get the data you want
To use HTML / CSS to style the page

What is RDBMS?
RDBMS stands for Relational Database Management System.

RDBMS is a program used to maintain a relational database.

RDBMS is the basis for all modern database systems such as MySQL, Microsoft SQL Server, Oracle, and Microsoft Access.

RDBMS uses SQL queries to access the data in the database.

What is a Database Table?

A table is a collection of related data entries, and it consists of columns and rows.

A column holds specific information about every record in the table.

A record (or row) is each individual entry that exists in a table.

Look at a selection from the Northwind "Customers" table:

What is SQL?
SQL is the standard language for dealing with Relational Databases.

SQL is used to insert, search, update, and delete database records.

How to Use SQL
The following SQL statement selects all the records in the "Customers" table:

Keep in Mind That...
SQL keywords are NOT case sensitive: select is the same as SELECT
In this tutorial we will write all SQL keywords in upper-case.

Semicolon after SQL Statements?
Some database systems require a semicolon at the end of each SQL statement.

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

In this tutorial, we will use semicolon at the end of each SQL statement.

to check all available databases we could use 'show databases' command on workbencj

inside of a database to check all tables inside of and databases ' show tables' in workbennch

to fetch all the data from the tables use 'select * from (tablename)'

we can fetch particular column also using the column instead of using '*' using column Name

CREATE TABLE: USE IT IN MY SQL WORKBENCH:

first you have to create database using 'create database DB (databasename)'
example
USE DB;

```
CREATE TABLE Persons(
PersonID int,
LastName varchar(255),
FirstName varchar(255),
Address varchar(255),
City varchar(255)
);
```

INSERT DATA:

```
use db;

#inserting data to a databases:

insert into  city(cityID,LastName,FirstName,state,nation)
values('3','angeles','los','uk','canada')
```

ex2: inserting multiple data at a time

```
use dbs;


insert into employ(employ_id, firstName,LastName,salary,experiance) values
(2,'jermy','renner',50000,13);
insert into employ(employ_id, firstName,LastName,salary,experiance) values
(3,'jos','brolin',50000,13);
insert into employ(employ_id, firstName,LastName,salary,experiance) values
(4,'tom','hiddleston',50000,13);
insert into employ(employ_id, firstName,LastName,salary,experiance) values
(5,'chris','hemsworth',50000,13);
insert into employ(employ_id, firstName,LastName,salary,experiance) values
(6,'chris','evans',50000,13);
insert into employ(employ_id, firstName,LastName,salary,experiance) values
(7,'rdj','junior',50000,13)
```

select key:

```
example:
USE DB;

SELECT LastName FROM Persons;
```

to select everything from a db

example:

```
SELECT * FROM Customers;
```

The following SQL statement counts and returns the number of different (distinct) countries in the "Customers" table:

example

```
SELECT DISTINCT first_name FROM actor;
```

The WHERE clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

The WHERE clause is not only used in SELECT statements, it is also used in UPDATE, DELETE,

example1:

```
SELECT * FROM actor
WHERE first_name='JENNIFER';
```

Equal_example:

```
SELECT * FROM actor
WHERE actor_id = 6;
```

greater_example:

```
SELECT * FROM actor
WHERE actor_id > 6;
```

lesser_example:

```
SELECT * FROM actor
WHERE actor_id < 6;
```

greaterOrEqual_example:

```
SELECT * FROM actor
WHERE actor_id => 6;
```

lesserrOrEqual_example:

```
SELECT * FROM actor
WHERE actor_id =< 6;
```

NotEqual_example://it will give all values except 6: in sql <>stands for notequal

```
SELECT * FROM actor
WHERE actor_id <> 6;
```

lesserrOrEqual_example:

```
use sakila;

select * from actor
where actor_id <>'6' and actor_id <>'7'



SELECT * FROM actor
WHERE actor_id BETWEEN 3 AND 6;
```

like_example://Search for a pattern

from the front:
ex:

```
use dbs;

select * from employ
where firstName like 'j%'
```

from the last:

ex:

```
SELECT * FROM actor
WHERE first_name like '%s' ;
```

IN_like://To specify multiple possible values for a column

```
SELECT * FROM actor
WHERE first_name IN ('ELVIS','JAMES') ;
```

The MySQL AND, OR and NOT Operators
The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

The AND operator displays a record if all the conditions separated by AND are TRUE.
The OR operator displays a record if any of the conditions separated by OR is TRUE.
The NOT operator displays a record if the condition(s) is NOT TRUE.

AND_EXAMPLE

```
SELECT * FROM actor
WHERE first_name='ED' AND last_name='CHASE';
```

The following SQL statement selects all fields from "Customers" where country is "Germany" OR "Spain":

or_EXAMPLE  //# used to comment a line in mysql also

```
SELECT * FROM actor
#WHERE first_name='ED' or last_name='CHASE';
WHERE first_name='ED' or last_name='DAVIS';
```

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

NOT_EXAMPLE:

```
SELECT * FROM actor

WHERE NOT actor_id = 3;
```

You can also combine the AND, OR and NOT operators.

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "Stuttgart"
(use parenthesis to form complex expressions):

MULTIPLE_EXAMPLE:

```
SELECT * FROM actor

WHERE first_name = 'ed' and (last_name = 'CHASE' or last_name = 'CHASE');
```

MULTIPLE_EXAMPLE1:

```
SELECT * FROM actor
```

```
WHERE NOT first_name = 'ed' and NOT last_name = 'CHASE' or last_name = 'CHASE';
```

The MySQL ORDER BY Keyword
The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records
in descending order, use the DESC keyword.

example:

```
SELECT * FROM actor

ORDER BY first_name;
```

ORDER BY DESC Example
The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

reverse example:

```
SELECT * FROM actor

ORDER BY first_name desc;
```

ORDER BY Several Columns Example
The following SQL statement selects all customers from the "Customers" table, sorted by the
"Country" and the "CustomerName" column. This means that it orders by Country, but if some
rows have the same Country, it orders them by CustomerName:

multiple_columns:

```
SELECT * FROM actor

ORDER BY first_name , last_name;
```

example1:

```
SELECT * FROM actor
ORDER BY first_name ASC, last_name DESC;
```

The MySQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax
It is possible to write the INSERT INTO statement in two ways:

1. Specify both the column names and the values to be inserted

insertvalue_example:

```
USE DB;

SELECT * FROM Persons;
INSERT INTO Persons (PersonID, LastName, FirstName,Address, City)
VALUES ('1', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', 'Norway');
```

insertSpecifiedcolumn_example:

```
USE DB;

SELECT * FROM Persons;
INSERT INTO Persons (PersonID, LastName, FirstName)
VALUES ('1', 'Tom B. Erichsen', 'Skagen 21');
```

What is a NULL Value?
A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record
 without adding a value to this field. Then, the field will be saved with a NULL value.

We will have to use the IS NULL and IS NOT NULL operators instead.

example:

```
USE DB;

SELECT * FROM Persons
where address is null
```

example_not_null:

```
USE DB;

SELECT * FROM Persons
where address is not null
```

membership operators:

used to check and display what and all matching values available
using in operators:

```
#in
use sakila;

select * from actor
where first_name in ('ed','nick')
```

#not in except the mentioned values it will fetch data

```
use sakila;

select * from actor
where first_name not in ('ed','nick')
```

UPDATE to specify which columns and values that should be updated in a table.

The following SQL updates the first customer (CustomerID = 1) with a new ContactName and a new City:

```
use world;

UPDATE City
SET District= 'Alfred Schmidt', Population= '3164938'
WHERE CountryCode = 'AFG';
```

The MySQL DELETE Statement
The DELETE statement is used to delete existing records in a table.

example:

```
use db;

DELETE FROM Persons
ORDER BY city
LIMIT 10;
```

3.DB CONNECTION
to connect use your user name and password along with correctt data base name
example:

```
import pymysql

#database connection
connection =
pymysql.connect(host="localhost",user="root",passwd="pugal@001",database="DB")
cursor = connection.cursor()
# some other statements  with the help of cursor
connection.close()
```

4.CREATING DB TABLE

Creating a Database
To create a database in MySQL, use the "CREATE DATABASE" statement:

cursor:

a cursor allows row by row processing of the result sets. a cursor is used for the result set
and returned from a query

Example

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="",
database="Employee")
cursor = connection.cursor()
# Query for creating table
EmpTableSql = """CREATE TABLE Besant(
ID INT(20) PRIMARY KEY AUTO_INCREMENT,
EMPID INT(20),
NAME  CHAR(20) NOT NULL,
DESIGNATION CHAR(10),
EXPERIENCE INT(20),
SALARY INT(20))"""
cursor.execute(EmpTableSql)
connection.close()
```

to check whether if it  is table has been created not type on my sql like this:

```
use DB;

show tables
```

Check if Table Exists
You can check if a table exist by listing all tables in your database with the "SHOW TABLES" statement:

Example

Return a list of your system's databases:

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()

cursor.execute("SHOW DATABASES")

for i in cursor:
    print(i)
```

5.INSERT, READ, UPDATE,DELETE OPERATIONS

Python MySQL Insert Into Table

Insert Into Table
To fill a table in MySQL, use the "INSERT INTO" statement.

Example
Insert a record in the "customers" table:

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()

# queries for inserting values
insert1 = "INSERT INTO Besant(EMPID,NAME,DESIGNATION,EXPERIENCE,SALARY)
VALUES(1004,'geeni', 'SE',2,200000 );"
insert2 = "INSERT INTO Besant(EMPID,NAME,DESIGNATION,EXPERIENCE,SALARY)
VALUES(1001,'Kesavan', 'SSE',3,100000 );"
insert3 = "INSERT INTO Besant(EMPID,NAME,DESIGNATION,EXPERIENCE,SALARY)
VALUES(1002,'Vikash', 'SE',2,200000 );"
insert4 = "INSERT INTO Besant(EMPID,NAME,DESIGNATION,EXPERIENCE,SALARY)
VALUES(1003,'Disha', 'SE',2,200000 );"


#executing the quires
cursor.execute(insert1)
cursor.execute(insert2)
cursor.execute(insert3)
cursor.execute(insert4)


#commiting the connection then closing it.
connection.commit()
connection.close()
```

Example
#Select all records from the "Besant" table, and display the result:

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()

cursor.execute("SELECT * FROM Besant")
result = cursor.fetchall()

for i in result:
    print(i)
```

Note: We use the fetchall() method, which fetches all rows from the last executed statement.

Selecting Columns
To select only some of the columns in a table, use the "SELECT" statement followed

by the column name(s):

Example
Select only the name and SALARY columns:

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()

cursor.execute("SELECT NAME,SALARY FROM Besant")
result = cursor.fetchall()

for i in result:
    print(i)
```

Using the fetchone() Method
If you are only interested in one row, you can use the fetchone() method.

The fetchone() method will return the first row of the result:

Example
Fetch only one row:

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()

cursor.execute("SELECT NAME,SALARY FROM Besant")
result = cursor.fetchone()

#for i in result:
#    print(i)
print(result)
```

Python MySQL Where

Select With a Filter
When selecting records from a table, you can filter the selection by using the "WHERE" statement:

Example
Select record(s) where the NAME result:

```
import pymysql
```

```
#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()


sql = "SELECT * FROM Besant WHERE NAME = 'Kesavan'"


cursor.execute(sql)
result = cursor.fetchone()

for i in result:
    print(i)
```

Wildcard Characters
You can also select the records that starts, includes, or ends with a given letter or phrase.

Use the %  to represent wildcard characters:

Example
Select records where the name contains the word "kesavan":

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()


sql = "SELECT * FROM Besant WHERE NAME LIKE '%Kesavan%'"


cursor.execute(sql)
result = cursor.fetchone()

for i in result:
    print(i)
```

Python MySQL Order By

Sort the Result
Use the ORDER BY statement to sort the result in ascending or descending order.

The ORDER BY keyword sorts the result ascending by default. To sort the result in descending order, use the DESC keyword.

Example
Sort the result alphabetically by name: result:

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()


sql = "SELECT * FROM Besant ORDER BY NAME"


cursor.execute(sql)
result = cursor.fetchall()

for i in result:
    print(i)
```

ORDER BY DESC
Use the DESC keyword to sort the result in a descending order.

Example
Sort the result reverse alphabetically by name:

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="DB")
cursor = connection.cursor()


sql = "SELECT * FROM Besant ORDER BY NAME DESC"


cursor.execute(sql)
result = cursor.fetchall()

for i in result:
    print(i)
```

Delete Record
You can delete records from an existing table by using the "DELETE FROM" statement:

Example
Delete any record where the address is "Mountain 21":

rowcount:

returns the number of rows affected by the last execute method for the same cursor object

```
import pymysql

#database connection
connection = pymysql.connect(host="localhost", user="root", passwd="pugal@001",
database="db")
cursor = connection.cursor()



sql = "DELETE FROM besant WHERE EMPID = '1001'"



cursor.execute(sql)
connection.commit()
print(cursor.rowcount, 'deleted')
```

Important!: Notice the statement: mydb.commit(). It is required to make the changes, otherwise no changes are made
 to the table.

Notice the WHERE clause in the DELETE syntax: The WHERE clause specifies which record(s)
 that should be deleted. If you omit the WHERE clause, all records will be deleted!

Prevent SQL Injection
It is considered a good practice to escape the values of any query, also in delete statements.

This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.

The mysql.connector module uses the placeholder %s to escape values in the delete statement:

Example
Escape values by using the placeholder %s method:

```
import pymysql
mydb = pymysql.connect(
  host="localhost",
  user="root",
  password="pugal@001",
  database="db"
)
mycursor = mydb.cursor()
sql = "DELETE FROM newtable WHERE name = %s"
adr = ("geeni", )
mycursor.execute(sql, adr)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Python MySQL Drop Table

Delete a Table
You can delete an existing table by using the "DROP TABLE" statement:

Example
Delete the table "customers":

```
import pymysql

mydb = pymysql.connect(
  host="localhost",
  user="root",
  password="pugal@001",
  database="db"
)

mycursor = mydb.cursor()

sql = "DROP TABLE emp"

mycursor.execute(sql)


#If this page was executed with no error(s), you have successfully deleted the
"customers" table.
```

Drop Only if Exist
If the the table you want to delete is already deleted, or for any other reason does not exist, you can use the IF EXISTS keyword to avoid getting an error.

Example
Delete the table "customers" if it exists:

```
import pymysql

mydb = pymysql.connect(
  host="localhost",
  user="root",
  password="pugal@001",
  database="db"
)

mycursor = mydb.cursor()

sql = "DROP TABLE IF EXISTS besant"

mycursor.execute(sql)

#If this page was executed with no error(s), you have successfully deleted the
"customers" table.
```

Python MySQL Limit

Limit the Result
You can limit the number of records returned from the query, by using the "LIMIT" statement:

Example
Select the 5 first records in the "customers" table:

```
import pymysql
mydb = pymysql.connect(
  host="localhost",
  user="root",
  password="pugal@001",
  database="sakila"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM actor LIMIT 5")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Start From Another Position
If you want to return five records, starting from the third record, you can use the "OFFSET" keyword:

Example
Start from position 3, and return 5 records:

```
import pymysql
mydb = pymysql.connect(
  host="localhost",
  user="root",
  password="pugal@001",
  database="sakila"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM actor LIMIT 5 offset 2") #from starting 3 it will fetch five records

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

NETWORK PROGRAMMING(1 HR)

1.INTRODUCTION

Python Network Programming is about using python as a programming language to handle
 computer networking requirements. For example, if we want to create and run a local
 web server or automatically download some files from a URL with a pattern.

Python plays an essential role in network programming. The standard library of Python
 has full support for network protocols, encoding, and decoding of data and
 other networking concepts, and it is simpler to write network programs in Python than that of C++.

2.sockets:
      is one endpoint of a two way communication link between two programs runnin on the network
      inter-process

sockets:
a client and a server
prot number and type of the connections also we need to concentrate

ports
a server will have an ip address to interact with internets it can have name

ex:

cnn.com #domain name
every service will havee different port number

we dont use universal already booked port number we have to use some unreserved port number

ex:9999

to do network programming we have library named as 'socket'

type of the connection:

    tcp and udp one is connection oriented another connection less network
      gprs-general packet radio service--switching the technology that enables the data transfer

            udp -- user datagram protocol- connectionless
    tcp = transmission control protocol- it will use connection oriented protocol- which means
    first we have to create a connection then only you can communicate
    in udp we dont have to create a connection we just have to send the packet based on the
    network it will reach destination but the drawback we dont know to sure whether reached the destination
    or not
    here we care about tcp only we creating connection then only we will send the packets

3.creating server:

```
import socket

#creating a socket

s = socket.socket()#inside of () first one is type of connection like ipv4 or ipv6
```

```
second network type here by default we use tcp

print('Socket created')

#to connectt or bind with a socket number we could use bind
#here our machine only server and client thats why we are using local host

s.bind(('localhost',9999))  #bind takes one argument only thats why we have given
double brackets to convert one element
#here we can mention client that we used to get connections
s.listen(3)

print('waiting for connection')

#to process that continuely we can use a loop

while True:

    #why we are mention two values one for socket another one for address
    c, addr=s.accept()

    print("connections establisheed", addr)

    #to send info client
    c.send('hello world')

    #dont forget to close the connection
    c.close()


this program will not execute connection accept bcuz we did not used any client

4.THE CLIENT PROGRAM

The Python API client is an open source, production-ready API wrapper that
 abstracts from the complexity of directly interfacing with the Algolia Search API.
 It handles, for example, network retry strategy, record batching, and reindexing
strategy.

import socket

#we have to create socket for client

c= socket.socket()

#here we dont need to binde it is server job we used to simply connect it

#here we need to mention server and portnumber to establish connect in single
variable using double brackets

c.connect(('localhost',9999))

#client will print nothing we did given anything we can give receive like as for
# but we need accept whatever server sends us then it will give the result along
with size
#

#print(c.recv(1024))
```

utf-8:

variable width-character encoding datas  for electronic communication defined by the unicode standard

encoding:

the process of putting a sequence of characters(letters,numbers...) into specialized format for efficient
transformation

bytes:
basic unit of information in computer storage and processing

ex:2

server:

```
import socket

#creating a socket

s = socket.socket()#inside of () first one is type of connection like ipv4 or ipv6 second network type here by default we use tcp

print('Socket created')

#to connectt or bind with a socket number we could use bind
#here our machine only server and client thats why we are using local host

s.bind(('localhost',9999))  #bind takes one argument only thats why we have given double brackets to convert one element
#here we can mention client that we used to get connections
s.listen(3)

print('waiting for connection')

#to process that continuely we can use a loop

while True:

    #why we aremention to values one for socket another one for address
    c, addr=s.accept()

    print("connections establisheed", addr)

    #to send info client but if you are send like this the client never understand a string we have to convert into
    #bytes in utf-8 formant
    c.send(bytes('hello world','utf-8'))

    #dont forget to close the connection
    c.close()
    #every time you rerun the server we will give different port number
```

client:

```
import socket
#we have to create socket for client
c= socket.socket()

#here we dont need to binde it is server job we used to simply connect it
#here we need to mention server and portnumber to establish connect in single
variable using double brackets
c.connect(('localhost',9999))

#client will print nothing we did given anything we can give receive like as for
# but we need accept whatever server sends us then it will give the result along
with size
#
#print(c.recv(1024))

#if you dont want to print along with by bytes we can give decode method
print(c.recv(1024).decode())
```

ex:3 to send data from the client:

SERVER:

```
import socket


s = socket.socket()
print('Socket created')

s.bind(('localhost',9999))
s.listen(3)

print('waiting for connection')

while True:

    c, addr=s.accept()
    name = c.recv(1024).decode()
    print("connections establisheed", addr,name)

    c.send(bytes('hello world','utf-8'))

    c.close()
    #every time you rerun the server we will give different port number
```

decoding:

the process of turning communication into inderstanding format or normal format

CLIENT:

```
import socket
#we have to create socket for client
c= socket.socket()
```

```
c.connect(('localhost',9999))

name = input("enter your name")
c.send(bytes(name,'utf-8'))#sometime it may cause error connection first time that
time you can run client only again then give the name
print(c.recv(1024).decode())
```

DATE AND TIME(1HR)

In Python, date and time are not a data type of its own, but a module named datetime can be imported
to work with the date as well as time. Datetime module comes built into Python, so there is no need
to install it externally. Datetime module supplies classes to work with date and time

Python Datetime

Python Dates
A date in Python is not a data type of its own, but we can import a module named datetime
to work with dates as date objects.

Example
Import the datetime module and display the current date:

```
import datetime

x = datetime.datetime.now()

print(x)
```

Date Output

When we execute the code from the example above the result will be:

```
2021-09-10 13:10:39.403464
```

The date contains year, month, day, hour, minute, second, and microsecond.

The datetime module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

Example

Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
```

```
print(x.strftime("%A"))
```

Creating Date Objects

To create a date, we can use the datetime() class (constructor) of the datetime module.

The datetime() class requires three parameters to create a date: year, month, day.

Example

Create a date object:

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

The strftime() Method

The datetime object has a method for formatting date objects into readable strings.

The method is called strftime(), and takes one parameter, format, to specify the format of the returned string:

Example

Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

1.SLEEP

Python sleep()
The sleep() function suspends (waits) execution of the current thread for a given number of seconds.

Python has a module named time which provides several useful functions to handle time-related tasks. One of the popular functions among them is sleep().

The sleep() function suspends execution of the current thread for a given number of seconds.

Example 1: Python sleep()

```
import time

print("Printed immediately.")

time.sleep(2.4)
```

```
print("Printed after 2.4 seconds.")
```

Here's how this program works:

"Printed immediately" is printed

Suspends (Delays) execution for 2.4 seconds.
"Printed after 2.4 seconds" is printed.
As you can see from the above example, sleep() takes a floating-point number as an argument.

Before Python 3.5, the actual suspension time may be less than the argument specified to the time() function.

Since Python 3.5, the suspension time will be at least the seconds specified.

Example 2: Python create a digital clock

```
import time

while True:
  localtime = time.localtime()
  result = time.strftime("%I:%M:%S %p", localtime)
  print(result)
  time.sleep(1)
```

In the above program, we computed and printed the current local time inside the infinite while loop.
Then, the program waits for 1 second. Again, the current local time is computed and printed. This process goes on.

When you run the program, the output will be something like:

```
02:10:50 PM
02:10:51 PM
02:10:52 PM
02:10:53 PM
02:10:54 PM
... .. ...
```

Here is a slightly modified better version of the above program.

2.PROGRAM EXECUTION TIME

flush:

used to buffer or clear the internal buffer used to flush the output stream it will true and false
default will be false if you ever mentioned flush it will stay in false, if you specifies true it flushes stream

```
import time

while True:
  localtime = time.localtime()
  result = time.strftime("%I:%M:%S %p", localtime)
```

```
  print(result, end="", flush=True)
  print("\r", end="", flush=True) #used to clear the result of the program if you
enable this line you will
                                                        never see the result
  time.sleep(1)
```

Multithreading in Python

Before talking about sleep() in multithreaded programs, let's talk about processes and threads.

A computer program is a collection of instructions. A process is the execution of those instructions.

A thread is a subset of the process. A process can have one or more threads.

MULTITHEREADING:

to use execute several tasks simultaneously the concept is multitasking

1,process based multitasking
2,thread based multitasking

PROCESS BASED:

EXECuting several tasks simultaneously where eash task is seperate independent process this known as
process based

thread based:

EXECuting several tasks simultaneously where eash task is seperate independent part of the same program

mentioning way:

1.used to create withoout a class
2.extending from the class
3.creating without extending the class

Example 3: Python multithreading

All the programs above in this article are single-threaded programs. Here's an example of a
multithreaded Python program.

```
import threading

def print_hello_three_times():
  for i in range(3):
    print("Hello")

def print_hi_three_times():
    for i in range(3):
      print("Hi")
```

```
t1 = threading.Thread(target=print_hello_three_times)
t2 = threading.Thread(target=print_hi_three_times)

t1.start()
t2.start()
```
When you run the program, the output will be something like:

```
Hello
Hello
Hi
Hello
Hi
Hi
```

The above program has two threads t1 and t2. These threads are run using t1.start() and t2.start() statements.

Note that, t1 and t2 run concurrently and you might get different output.

Visit this page to learn more about Multithreading in Python.

time.sleep() in multithreaded programs

The sleep() function suspends execution of the current thread for a given number of seconds.

In case of single-threaded programs, sleep() suspends execution of the thread and process. However, the function suspends a thread rather than the whole process in multithreaded programs.

Example 4: sleep() in a multithreaded program

```
import threading
import time

def print_hello():
  for i in range(4):
    time.sleep(0.5)
    print("Hello")

def print_hi():
    for i in range(4):
      time.sleep(0.7)
      print("Hi")

t1 = threading.Thread(target=print_hello)
t2 = threading.Thread(target=print_hi)
t1.start()
t2.start()
```
The above program has two threads. We have used time.sleep(0.5) and time.sleep(0.75) to suspend execution of
 these two threads for 0.5 seconds and 0.7 seconds respectively.

3.MORE METHODS ON DATE/TIME

```
%a    Weekday, short version  Wed
%A    Weekday, full version   Wednesday
%w    Weekday as a number 0-6, 0 is Sunday     3
%d    Day of month 01-31      31
%b    Month name, short version     Dec
%B    Month name, full version      December
%m    Month as a number 01-12 12
%y    Year, short version, without century     18
%Y    Year, full version      2018
%H    Hour 00-23  17
%I    Hour 00-12  05
%p    AM/PM PM
%M    Minute 00-59      41
%S    Second 00-59      08
%f    Microsecond 000000-999999     548513
%z    UTC offset  +0100
%Z    Timezone    CST
%j    Day number of year 001-366    365
%U    Week number of year, Sunday as the first day of week, 00-5352
%W    Week number of year, Monday as the first day of week, 00-5352
%c    Local version of date and time      Mon Dec 31 17:41:00 2018
%C    Century     20
%x    Local version of date   12/31/18
%X    Local version of time   17:41:00
%%    A % character     %
%G    ISO 8601 year     2018
%u    ISO 8601 weekday (1-7)  1
%V    ISO 8601 weeknumber (01-53)   01
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%a"))
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%A"))
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%w"))
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%d"))
```

```
import datetime
```

```
x = datetime.datetime.now()
print(x.strftime("%b"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%B"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%m"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%y"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%Y"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%H"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%I"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%p"))
```

```
import datetime
x = datetime.datetime.now()
print(x.strftime("%M"))
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%S"))
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%c"))
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%x"))
```

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%X"))
```

FEW MORE TOPICS IN DETAILED(1HR)

1.FILTER

Python filter() Function

lamda:

```
l = [1,2,5,4,6]

l1 = list(filter(lambda x:x%2!=0,l))
print(l1)
```

Example
Filter the array, and return a new array with only the values equal to or above 18:

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
```

```
    return False
  else:
    return True

adults = filter(myFunc, ages)

for x in adults:
  print(x)
```

ex:2

```
l = [2,5,7,9,8,12]

def fun(x):
    if x%2 != 0:
        return True
    else:
        return False

l1 = filter(fun,l)

for i in l1:
    print(i)
```

Definition and Usage
The filter() function returns an iterator were the items are filtered through a function to
 test if the item is accepted or not.

Syntax

```
filter(function, iterable)
```

Parameter Values

| Parameter | Description |
|---|---|
| function | A Function to be run for each item in the iterable |
| iterable | The iterable to be filtered |

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

# returns True if number is even
def check_even(number):
    if number % 2 == 0:
          return True

    return False

# Extract elements from the numbers list for which check_even() returns True
even_numbers_iterator = filter(check_even, numbers)

# converting to list
even_numbers = list(even_numbers_iterator)

print(even_numbers)

# Output: [2, 4, 6, 8, 10]
```

ex:2

```
ages = [5, 12, 17, 18, 24, 32]

def fun(s):
    if s<18:
        return True


    return False

l1 = list(filter(fun,ages))
print(l1)
```

Join our newsletter for the latest updates.
Enter Email Address*
Join

Python map()
In this tutorial, we will learn about the Python map() function with the help of examples.

The map() function applies a given function to each item of an iterable (list, tuple etc.)
 and returns an iterator.

Example

```
numbers = [2, 4, 6, 8, 10]

# returns square of a number
def square(number):
  return number * number

# apply square() function to each item of the numbers list
squared_numbers_iterator = map(square, numbers)

# converting to list
squared_numbers = list(squared_numbers_iterator)
print(squared_numbers)

# Output: [4, 16, 36, 64, 100]
```

map() Syntax
Its syntax is:

```
ex:2
#map --applying the given condition to all the sequence elements
'''
l = [2,3,5,6,8,9]

l1 = list(map(lambda x: x*x,l))

print(l1)

'''
ex:3
l = [3,5,7]
```

```
def square(x):
    return x*x

l1 = list(map(square,l))
print(l1)
```

map(function, iterable, ...)
map() Parameter
The map() function takes two parameters:

function - a function that perform some action to each element of an iterable
iterable - an iterable like sets, lists, tuples, etc
You can pass more than one iterable to the map() function.

map() Return Value
The map() function returns an object of map class. The returned value can be passed to functions like

list() - to convert to list
set() - to convert to a set, and so on.
Example 1: Working of map()
Output

```
<map object at 0x7f722da129e8>
{16, 1, 4, 9}
```

In the above example, each item of the tuple is squared.

Since map() expects a function to be passed in, lambda functions are commonly used while working with map() functions.

A lambda function is a short function without a name. Visit this page to learn more about Python lambda Function.

Example 2: How to use lambda function with map()?

```
numbers = (1, 2, 3, 4)
result = map(lambda x: x*x, numbers)
print(result)

# converting map object to set

numbersSquare = set(result)
print(numbersSquare)
```

Output

```
<map 0x7fafc21ccb00>
{16, 1, 4, 9}
```

There is no difference in functionalities of this example and Example 1.

Example 3: Passing Multiple Iterators to map() Using Lambda
In this example, corresponding items of two lists are added.

```
num1 = [4, 5, 6]
num2 = [5, 6, 7]

result = map(lambda n1, n2: n1+n2, num1, num2)
print(list(result))
```

3.REDUCE

reduce() in Python
Difficulty Level : Medium
Last Updated : 26 May, 2021
The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.This function is defined in “functools” module.

Working :

At first step, first two elements of sequence are picked and the result is obtained.
Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
This process continues till no more elements are left in the container.
The final returned result is returned and printed on console.

```
# python code to demonstrate working of reduce()

# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2, ]

# using reduce to compute sum of list
print("The sum of the list elements is : ", end="")
print(functools.reduce(lambda a, b: a+b, lis))

# using reduce to compute maximum element from list
print("The maximum element of the list is : ", end="")
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

Output
```
The sum of the list elements is : 17
The maximum element of the list is : 6
```

```
# python code to demonstrate working of reduce()
# using operator functions

# importing functools for reduce()
import functools

# importing operator for operator functions
import operator

# initializing list
lis = [1, 3, 5, 6, 2, ]

# using reduce to compute sum of list
# using operator functions
print("The sum of the list elements is : ", end="")
print(functools.reduce(operator.add, lis))
```

```
# using reduce to compute product
# using operator functions
print("The product of list elements is : ", end="")
print(functools.reduce(operator.mul, lis))

# using reduce to concatenate string
print("The concatenated product is : ", end="")
print(functools.reduce(operator.add, ["geeks", "for", "geeks"]))
```

reduce() vs accumulate()

Both reduce() and accumulate() can be used to calculate the summation of a sequence elements.
But there are differences in the implementation aspects in both of these.

reduce() is defined in “functools” module, accumulate() in “itertools” module.
reduce() stores the intermediate result and only returns the final summation value. Whereas, accumulate() returns a
iterator containing the intermediate results. The last number of the iterator returned is summation
value of the list.
reduce(fun,seq) takes function as 1st and sequence as 2nd argument.
In contrast accumulate(seq,fun) takes sequence as 1st argument and function as 2nd argument

```
# python code to demonstrate summation
# using reduce() and accumulate()

# importing itertools for accumulate()
import itertools

# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 4, 10, 4]

# printing summation using accumulate()
print("The summation of list using accumulate is :", end="")
print(list(itertools.accumulate(lis, lambda x, y: x+y)))

# printing summation using reduce()
print("The summation of list using reduce is :", end="")
print(functools.reduce(lambda x, y: x+y, lis))
```

4.DECORATORS

Python Decorators
A decorator takes in a function, adds some functionality and returns it. In this tutorial,
you will learn how you can create a decorator and why you should use it.

Python has an interesting feature called decorators to add functionality to an existing code.

This is also called metaprogramming because a part of the program tries to modify another
 part of the program at compile time.

Prerequisites for learning decorators
In order to understand about decorators, we must first know a few basic things in Python.

We must be comfortable with the fact that everything in Python (Yes! Even classes), are objects.
 Names that we define are simply identifiers bound to these objects. Functions are no exceptions,
 they are objects too (with attributes). Various different names can be bound to the same function object.

Here is an example.

Getting back to Decorators

Functions and methods are called callable as they can be called.

In fact, any object which implements the special __call__() method is termed callable.
So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")
```

In the example shown above, make_pretty() is a decorator. In the assignment step:

```
pretty = make_pretty(ordinary)
```

The function ordinary() got decorated and the returned function was given the name pretty.

We can see that the decorator function added some new functionality to the original function.
 This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object

 that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

ordinary = make_pretty(ordinary).
This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the @ symbol along with the name of the decorator function and place it above the
 definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

Decorating Functions with Parameters
The above decorator was simple and it only worked with functions that did not have any parameters.
What if we had functions that took in parameters like:

```
def divide(a, b):
    return a/b
```

This function has two parameters, a and b. We know it will give an error if we pass in b as 0.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner


@smart_divide
def divide(a, b):
    print(a/b)
```

This new implementation will return None if the error condition arises.

```
>>> divide(2,5)
I am going to divide 2 and 5
```

0.4

```
>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

In this manner, we can decorate functions that take parameters.

A keen observer will notice that parameters of the nested inner() function inside the decorator
is the same as the parameters of functions it decorates. Taking this into account, now we can
make general decorators that work with any number of parameters.

In Python, this magic is done as function(*args, **kwargs). In this way, args will be the tuple
 of positional arguments and kwargs will be the dictionary of keyword arguments. An example of such
 a decorator will be:

```
def works_for_all(func):
    def inner(*args, **kwargs):
        print("I can decorate any function")
        return func(*args, **kwargs)
    return inner
```

Chaining Decorators in Python

Multiple decorators can be chained in Python.

This is to say, a function can be decorated multiple times with different (or same) decorators.
 We simply place the decorators above the desired function.

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner  #we are not suppose to use parentheses here  or else we will get error argument missing


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner


@star
@percent
def printer(msg):
    print(msg)


printer("Hello")
```

ex:2

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)   #this line come before the function values
        func(*args, **kwargs)
        print("*" * 30)    #this line come after the function values
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)#this line comes first
        func(*args, **kwargs)
        print("%" * 30)#this line comes second but inside of the first decorator
    return inner




def dollar(func):
    def inner(*args, **kwargs):
        print("$" * 30)#this line comes first
        func(*args, **kwargs)
        print("$" * 30)#this line comes second but inside of the first decorator
    return inner



@star
@percent
@dollar
def printer(*msg):
    print(msg)

printer('hello world','this is pugal')
```

5.FROZENSET

Frozen sets are a native data type in Python that have the qualities of sets — including class
methods — but are immutable like tuples. To use a frozen set, call the function frozenset() and
pass an iterable as the argument. If you pass a set to the function, it will return the same set,
which is now immutable.

Python frozenset() Function

Example

Freeze the list, and make it unchangeable:

```
mylist = ['apple', 'banana', 'cherry']
x = frozenset(mylist)
print(x)
```

Definition and Usage

The frozenset() function returns an unchangeable frozenset object (which is like a set object, only unchangeable).

Syntax

```
frozenset(iterable)
```

Parameter Values

| Parameter | Description |
|---|---|
| iterable | An iterable object, like list, set, tuple etc. |

More Examples

Example
Try to change the value of a frozenset item.

This will cause an error:

```
mylist = ['apple', 'banana', 'cherry']
x = frozenset(mylist)
x[1] = "strawberry"
print(x)
```

REGULAR EXPRESSION:(1HR)

A regular expression is a special sequence of characters that helps you match or find
 other strings or sets of strings, using a specialized syntax held in a pattern. ...
 The Python module re provides full support for Perl-like regular expressions in Python.
 The re module raises the exception re.

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

RegEx Module

Python has a built-in package called re, which can be used to work with Regular Expressions.

Import the re module:

import re:

RegEx in Python
When you have imported the re module, you can start using regular expressions:

Example
Search the string to see if it starts with "The" and ends with "Spain":

^ --used to starts with
$--used to define ends with
*--used find more occurences
{}--  Exactly the specified number of occurrences
+     One or more occurrences
|     Either or

()    Capture and group

\     Signals a special sequence (can also be used to escape special characters)

[]    A set of characters

.     Any character (except newline character)

#symbol--[]

```
import re

txt = "The rain in Spain"

#Find all lower case characters alphabetically between "a" and "m":

x = re.findall("[a-m]", txt)
print(x)
```

#^    Starts with

```
import re

txt = "hello world"

#Check if the string starts with 'hello':
t ="hi hello"
x = re.findall("^hello", txt)

y = re.findall("^hello", t)

if y:
  print("Yes, the string starts with 'hello'")
```

```
else:
  print("No match")


#$    Ends with


import re

txt = "this is the dangerous world"

#Check if the string ends with 'world':

x = re.findall("world$", txt)
if x:
  print("Yes, the string ends with 'world'")
else:
  print("No match")



#*    Zero or more occurrences


import re

txt = "The rain in Spain falls mainly in the plain!"

#Check if the string contains "ai" followed by 0 or more "x" characters:

x = re.findall("aix*", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")



#{}   One or more occurrences

import re

txt = "The rain in Spain falls mainly in the plain!"

#Check if the string contains "a" followed by exactly two "l" characters:

x = re.findall("al{2}", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

|     Either or

```
import re

txt = "The rain in Spain falls mainly in the plain!"

#Check if the string contains either "falls" or "stays":

x = re.findall("falls|stays", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

find all:

def:The findall() function returns a list containing all matches.

```
import re

txt = "The rain in Spain"

#Check if "Portugal" is in the string:

x = re.findall("Portugal", txt)
print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

The search() Function
The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned

\s used to search operation while keyword as search
ex:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())
```

If no matches are found, the value None is returned

```
import re

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)
```

The split() Function
The split() function returns a list where the string has been split at each match
split them put in a list

ex:

```
import re

#Split the string at every white-space character:

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

sub():
used to replace or substitute the required value

The sub() Function
The sub() function replaces the matches with the text of your choice

ex:

```
import re

#Replace all white-space characters with the digit "9":

txt = "The rain in Seain"
x = re.sub("\s", "9", txt)
print(x)
```

You can control the number of replacements by specifying the count parameter

ex2:

```
import re

#Replace the first two occurrences of a white-space character with the digit 9:

txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
```

Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character Description Example Try it

\A Returns a match if the specified characters are at the beginning of the string "\AThe"
\b Returns a match where the specified characters are at the beginning or at the end of a word
(the "r" in the beginning is making sure that the string is being treated as a "raw string") r"\bain"
r"ain\b"
\B Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a
word
(the "r" in the beginning is making sure that the string is being treated as a "raw string") r"\Bain"
r"ain\B"
\d Returns a match where the string contains digits (numbers from 0-9) "\d"
\D Returns a match where the string DOES NOT contain digits "\D"
\s Returns a match where the string contains a white space character "\s"
\S Returns a match where the string DOES NOT contain a white space character "\S"
\w Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9,
and the underscore _ character) "\w"
\W Returns a match where the string DOES NOT contain any word characters "\W"
\Z Returns a match if the specified characters are at the end of the string "Spain\Z"

\A:

```
import re

txt = "The rain in Spain"

#Check if the string starts with "The":

x = re.findall("\AThe", txt)

print(x)

if x:
  print("Yes, there is a match!")
else:
  print("No match")
```

Returns a match where the specified characters are at the beginning or at the end of a word

```
#Check if "ain" is present at the beginning of a WORD:

import re
```

```
txt = "The rain in Spain"

#Check if "ain" is present at the beginning of a WORD:

x = re.findall(r"\bain", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

CHecking end of the word:

```
import re

txt = "The rain in Spain"

#Check if "ain" is present at the end of a WORD:

x = re.findall(r"ain\b", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match where the specified characters are present, but NOT at the beginning (or at the end)
of a word

```
import re

txt = "The rain in Spain"

#Check if "ain" is present, but NOT at the beginning of a word:

x = re.findall(r"\Bain", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

checkin match not at the end:

```
import re
```

```
txt = "The rain in Spain"

#Check if "ain" is present, but NOT at the end of a word:

x = re.findall(r"ain\B", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match where the string DOES NOT contain digits

```
import re

txt = "The rain in Spain"

#Return a match at every no-digit character:

x = re.findall("\D", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match where the string DOES NOT contain a white space character

```
import re

txt = "The rain in Spain"

#Return a match at every NON white-space character:

 x = re.findall("\S", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match where the string contains any word characters
not special char

```
import re
```

```
txt = "The rain45$ in Spain"

#Return a match at every word character (characters from a to Z, digits from 0-9,
and the underscore _ character):

x = re.findall("\w", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match where the string DOES NOT contain any word characters

```
import re

txt = "The rain#$% in Spain"

#Return a match at every NON word character (characters NOT between a and Z. Like
"!", "?" white-space etc.):

x = re.findall("\W", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match if the specified characters are at the end of the string

```
import re

txt = "The rain in Spain"

#Check if the string ends with "Spain":

x = re.findall("Spain\Z", txt)

print(x)

if x:
  print("Yes, there is a match!")
else:
  print("No match")
```

Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

Returns a match where one of the specified characters (a, r, or n) are present

```
import re

txt = "The rain in Spain"

#Check if the string has any a, r, or n characters:

x = re.findall("[arn]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
import re

txt = "The rain in Spain"

#Check if the string has any characters between a and n:

x = re.findall("[a-n]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match for any character EXCEPT a, r, and n

```
import re

txt = "The rain in Spain"

#Check if the string has other characters than a, r, or n:

x = re.findall("[^arn]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match where any of the specified digits (0, 1, 2, or 3) are present

```
import re

t = 'a world is flat'
txt = 'altogether the worlds biggest falls nayagara while the compare with all
falls wallls'


x = re.findall('[*ats]' , txt)
y = re.findall('flat$',t)
print(x)
print(y)




import re

txt = "The rain in Spain"

#Check if the string has any 0, 1, 2, or 3 digits:

x = re.findall("[0123]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")



Returns a match for any digit between 0 and 9

import re

txt = "8 times before 11:45 AM"

#Check if the string has any digits:

x = re.findall("[0-9]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")


Returns a match for any two-digit numbers from 00 and 59

import re

txt = "8 times before 11:45 AM"
```

```
#Check if the string has any two-digit numbers, from 00 to 59:

x = re.findall("[0-5][0-9]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Returns a match for any character alphabetically between a and z, lower case OR upper case

```
import re

txt = "8 times before 11:45 AM"

#Check if the string has any characters from a to z lower case, and A to Z upper case:

x = re.findall("[a-zA-Z]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means:
 return a match for any + character in the string

for example here we can try ['+'] it will find only plus not special char

```
import re

txt = "8 times before 11:45 AM"

#Check if the string has any + characters:

x = re.findall("[+]", txt)

print(x)

if x:
  print("Yes, there is at least one match!")
else:
  print("No match")
```

about the search, and the result:

.span() returns a tuple containing the start-, and end positions of the match.
.string returns the string passed into the function
.group() returns the part of the string where there was a match
Example

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

```
import re

#Search for an upper case "S" character in the beginning of a word, and print its
position:

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

Example
Print the string passed into the function:
the condition gets satisfied it will print the whole string

```
import re

#The string property returns the search string:

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

group():
unlike printing the wholse sentence groupt will print the particular words

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

```
import re

#Search for an upper case "S" character in the beginning of a word, and print the
word:

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

Important Methods of Queue:
1. put(): Put an item into the queue.
2. get(): Remove and return an item from the queue.

Types of Queues:
Python Supports 3 Types of Queues.
1. FIFO Queue:
q = queue.Queue()
This is Default Behaviour. In which order we put items in the queue, in the same order the items
will come out (FIFO-First In First Out).
Eg:
1) import queue

```
2) q=queue.Queue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
8) print(q.get(),end=' ')
```

2. LIFO Queue:
The removal will be happend in the reverse order of insertion(Last In First Out)
Eg:

```
1) import queue
2) q=queue.LifoQueue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
8) print(q.get(),end=' ')
```

3. Priority Queue:
The elements will be inserted based on some priority order.

```
1) import queue
2) q=queue.PriorityQueue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
8) print(q.get(),end=' ')
```

PYTHON DEBUGGING BY USING ASSERTIONS

Types of assert statements:
There are 2 types of assert statements
1. Simple Version
2. Augmented Version
1. Simple Version:

```
assert conditional_expression
```

2. Augmented Version:

```
assert conditional_expression,message
```

error:

```
def squareIt(x):
2) return x**x


   assert squareIt(2)==4,"The square of 2 should be 4"
4) assert squareIt(3)==9,"The square of 3 should be 9"
5) assert squareIt(4)==16,"The square of 4 should be 16"
6) print(squareIt(2))
7) print(squareIt(3))
8) print(squareIt(4))
```

ex:

```
) def squareIt(x):
17) return x*x
18) assert squareIt(2)==4,"The square of 2 should be 4"
19) assert squareIt(3)==9,"The square of 3 should be 9"
20) assert squareIt(4)==16,"The square of 4 should be 16"
21) print(squareIt(2))
22) print(squareIt(3))
23) print(squareIt(4))
```

Working with random module:

This module defines several functions to generate random numbers.
We can use these functions while developing games,in cryptography and to generate random numbers on fly for authentication.

1. random() function:
 This function always generate some float value between 0 and 1 ( not inclusive)
 0<x<1

```
from random import *
2) for i in range(10):
3) print(random())
```

2. randint() function:
To generate random integer beween two given numbers(inclusive)

```
 from random import *
2) for i in range(10):
3) print(randint(1,100))
```

3. uniform():
 It returns random float values between 2 given numbers

```
 from random import *
2) for i in range(10):
3) print(uniform(1,10))
```

4. randrange([start],stop,[step])
 returns a random number from range
 start<= x < stop
 start argument is optional and default value is 0
 step argument is optional and default value is 1

ex:1

```
 from random import *
2) for i in range(10):
3) print(randrange(10))
```

ex:2

```
from random import *
2) for i in range(10):
3) print(randrange(1,11))
```

ex:3

```
from random import *
2) for i in range(10):
3) print(randrange(1,11,2))
```

DJANGO ESSENTIAL:(1HR)

Web Application:
The applications which will provide services over the web are called web applications.
Eg: gmail.com, facebook.com, durgasoftvideos.com etc
Every web application contains 2 main components
1) Front-End
2) Back-End
1) Front-End:
- It represents what user is seeing on the website
- We can develop Front-End content by using the following technologies:
- HTML, JS, CSS, JQuery and BootStrap
- JQuery and Bootstrap are advanced front-end technologies, which are developed by using HTML, CSS and JavaScript only.

HTML:
- HyperText Markup Language
- Every web application should contain HTML. i.e HTML is the mandatory technology for web development.It represents structure of web page

CSS: Cascading Style Sheets
- It is optional technology, still every web application contains CSS.
- The main objective of CSS is to add styles to the HTML Pages like colors, fonts, borders etc.

Java Script:
- It allows to add interactivity to the web application including programming logic.
- The main objective of Java Script is to add functionality to the HTML Pages. ie to add dynamic nature to the HTML Pages

Back End:
- It is the technology used to decide what to show to the end user on the Front-End.
- ie Backend is responsible to generate required response to the end user,which is displayed by the Front-End.
- Back-End has 3 important components:

1) The Language like Java,Python etc
2) The Framework like DJango,Pyramid,Flask etc
3) The Database like SQLite,Oralce,MySQL etc

- For the Backend language Python is the best choice b'z of the following reasons: Simple and easy to learn, libraries and concise code.

¬ For the Framework DJango is the best choice b'z it is Fast, Secure and Scalable. Django
is the most popular web application framework for Python.
¬ DJango provides inbuilt database which is nothing but SQLite, which is the best choice
for database.
¬ The following are various popular web applications which are developed by using Python and DJango

DJango:
- Django is a free and open-source web framework.
- It is written in Python.
- It follows the Model-View-Template (MVT) architectural pattern.
- It is maintained by the Django Software Foundation (DSF)

Top 5 Features of Django Framework:
Django was invented to meet fast-moving newsroom deadlines, while satisfying the tough
requirements of experienced Web developers.
The following are main important features of Django
1) Fast:
Django was designed to help developers take applications from concept to completion as quickly as possible.
2) Fully loaded:
Django includes dozens of extras we can use to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS
feeds, and many more tasks.
3) Security:
Django takes security seriously and helps developers avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery and
clickjacking. Its user authentication system provides a secure way to manage user accounts and passwords

Django Project vs Django Application:
A Django project is a collection of applications and configurations which forms a full web
application.
Eg: Bank Project
A Dango Application is responsible to perform a particular task in our entire web application.

How to create Django Project:
Once we installed django in our system, we will get 'django-admin' command line tool,
which can be used to create our Django project.
```
django-admin startproject firstProject
```

__init__.py:
It is a blank python script.Because of this special file name, Django treated this folder as
python package.
Note: If any folder contains __init__.py file then only that folder is treated as Python

package.But this rule is applicable unti

settings.py:
In this file we have to specify all our project settings and and configurations like
installed applications, middileware configurations, database configurations etc
urls.py:
¬ Here we have to store all our url-patterns of our project.
¬ For every view (web page), we have to define separate url-pattern. End user can use
url-patterns to access our webpages.
wsgi.py:
¬ wsgi ▭ Web Server Gateway Interface.
¬ We can use this file while deploying our application in production on online server.
manage.py:
¬ The most commonly used python script is manage.py
¬ It is a command line utility to interact with Django project in various ways like to run
development server, run tests, create migrations etc.

How to Run Django Development Server:
We have to move to the manage.py file location and we have to execute the following command.
py manage.py runserver

Creation of First Web Application:
Once we creates Django project, we can create any number of applications in that project.
The following is the command to create application.
python manage.py startapp firstApp

1) __init__.py:
It is a blank Python script. Because of this special name,Python treated this folder as a
package.
2) admin.py:
We can register our models in this file. Django will use these models with Django's admin interface.
3) apps.py:
In this file we have to specify application's specific configurations.
4) models.py:
In this file we have to store application's data models.
5) tests.py:
In this file we have to specify test functions to test our code.

6) views.py:
In this file we have to save functions that handles requests and return required responses.
7) Migrations Folder:
This directory stores database specific information related to models.

In settings.py:
1) INSTALLED_APPS = [
2) 'django.contrib.admin',
3) 'django.contrib.auth',

```
4) 'django.contrib.contenttypes',
5) 'django.contrib.sessions',
6) 'django.contrib.messages',
7) 'django.contrib.staticfiles',
8) 'firstApp'
9) ]
```

views.py:

```
1) from django.shortcuts import render
2) from django.http import HttpResponse
3)
4) # Create your views here.
5) def display(request):

s='<h1>Hello Students welcome to DURGASOFT Django classes!!!</h1>'
      7) return HttpResponse(s)
```

Note:

1) Each view will be specified as one function in views.py.
2) In the above example display is the name of function which is nothing but one view.
3) Each view should take atleast one argument (request)
4) Each view should return HttpResponse object with our required response.

View can accept request as input and perform required operations and provide proper response to the end user.

PROJECT LEVEL.URLS.PY:

```
from django.contrib import admin
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('newapp.urls')),
```

APP LEVELS URLS.PY:

```
from django.urls import path

from . import views

urlpatterns = [

    path('', views.viewing),

]
```

Various Practice Applications:

1. Write Django Application just to send Helloworld message as response.
2. Write Django application to send server time as response
3. Single application with multiple views

views.py:

```
1) from django.shortcuts import render
2) from django.http import HttpResponse
3)
4) # Create your views here.
5) def viewing(request):
6) return HttpResponse('<h1>Hello Fr  iend Good Morning!!!</h1>')
7)
8) def good_evening_view(request):
9) return HttpResponse('<h1>Hello Friend Good Evening !!!</h1>')
10)
11) def good_afternoon_view(request):
12) return HttpResponse('<h1>Hello Friend Good Afternoon!!!</h1>')
```

APPLEVEL:

urls.py

```
from django.urls import path

from . import views

urlpatterns = [

    path('', views.viewing),
    path('good', views.GM),
    path('goode', views.GE),

]
```

MULTIPLEAPPS:

APP.1:VIEWS:

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def viewing(request):
    return render(request,'hello.html')


def GM(request):
    return HttpResponse("<h1>good morning</h1>")



def GE(request):
    return HttpResponse("<h1>good evening</h1>")
```

APP:1 URLS.PY

```
from django.urls import path
```

```
from . import views

urlpatterns = [

    path('', views.viewing),
    path('good', views.GM),
    path('goode', views.GE),

]
```

APP.2 VIEWS.PY:

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def viewing(request):
   return HttpResponse("<h1>hi this is pugal from app2</h1>")
   #return render(request,'color.html')

def GM(request):
    return HttpResponse("<h1>good morning from app2</h1>")



def GE(request):
    return HttpResponse("<h1>good evening from app2</h1>")
```

APPS2:URLS.PY

```
urlpatterns = [

    path('', views.viewing),
    path('good2', views.GM),
    path('goode2', views.GE),
```

PROJECT LEVEL URLS.PY:

```
from django.contrib import admin
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('app', include('newapp.urls')),
    path('',include('newapp2.urls'))# to access second app multiple view
]
```

Django Templates:
It is not recommended to write html code inside python script (views.py file) because:
1) It reduces readability because Python code mixed with html code
2) No seperation of roles. Python developer has to concentrate on both python code and

HTML Code:

3) It does not promote reusability of code

We can overcome these problems by seperating html code into a seperate html file.This
html file is nothing but template.
From the Python file (views.py file) we can use these templates based on our requirement.

We have to write templates at project level only once and we can use these in multiple
applications.

TO SET TEMPLATES:

in settings foldr:

```
TEMPLATES = [
                  .........
                  'DIRS': ['templates'],
                  .........
                  ]
```

it has to then create a templates folder inside of the project
inside of this you can write you html files:

hello.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>hi</title>
</head>
<body>
<h1>hello from html </h1>
</body>
</html>
```

views.py:

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def viewing(request):
    return render(request,'hello.html')
```

rest of them same runserver you can see the message from html

giving css styles:

color.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>coloring</title>
    <style>
        h1{
            color: white;
            background-color: orange;
            }

        p{
            color:yello;
            background-color: blue;
            font-size:40px
        }
    </style>
</head>
<body>
<h1>hello this is headinng</h1>

<p>this is the paragraph</p>

</body>
</html>
```

views.py:

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def viewing(request):
   #return HttpResponse("<h1>hi this is pugal from app2</h1>")
   return render(request,'color.html')
```

substitute values django to html:

sub.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>substitution</title>
    <style>
        h1{
            color:white;
            background-color:black;
    </style>
</head>
<body>
<h1>Hello Server Current Date and Time : <br>
     {{insert_date}}</h1>
```

```
</body>
</html>
```

views.py:

```
import datetime
# Create your views here.
from django.shortcuts import render
from django.http import HttpResponse

def sub(request):

    date=datetime.datetime.now()
    my_dict={'insert_date':date}

    return render(request,'sub.html',context=my_dict)
```

sharing integer and string values from django to html:

student.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>student</title>

   <style>
     body{
       background: green;
       color:white;
          }
     h1{
       border:10px solid yellow;
       }
   </style>
</head>
<body>

<h1>Hello this response from Template File.I will take care everything about pres
entation</h1><hr>
 <h2>The Server Date and Time is:{{date}}</h2>
 <ol>
 <li>Name:{{name}}</li>
 <li>Rollno:{{rollno}}</li>
 <li>Marks:{{marks}}</li>
 </ol>

</body>
</html>
```

views.py:

```
import datetime
# Create your views here.
from django.shortcuts import render
from django.http import HttpResponse
```

```
def stud(request):
    dt = datetime.datetime.now()


    name = 'rani mangamma angala paraeshwari'
    rollno = 101
    marks = 100
    my_dict = {'date': dt, 'name': name, 'rollno': rollno, 'marks': marks}
    return render(request, 'student.html', my_dict)
```

DATA STRUCTURES:(3HR)
==========

1.LIST COMPREHENSIONS

List Data Structure

If we want to represent a group of individual objects as a single entity where insertion
order preserved and duplicates are allowed, then we should go for List.
insertion order preserved.

duplicate objects are allowed

heterogeneous objects are allowed.

List is dynamic because based on our requirement we can increase the size and decrease
the size.

In List the elements will be placed within square brackets and with comma seperator.

We can differentiate duplicate elements by using index and we can preserve insertion

order by using index. Hence index will play very important role.

Python supports both positive and negative indexes. +ve index means from left to right

where as negative index means right to left
[10,"A","B",20, 30, 10]

List objects are mutable.i.e we can change the content.
Creation of List Objects:

1. We can create empty list object as follows...

```
 list=[]
 print(list)
```

```
 print(type(list))
```

2. If we know elements already then we can create list as follows

```
      list=[10,20,30,40]
```

3. With dynamic input:

```
 list=eval(input("Enter List:"))
 print(list)
 print(type(list))
```

if you are doing this you will get error:
to mention a list we have to use 'list' keyword because as default it will be in the 'tuple' method
3.proper way:

```
list1 = list(eval(input("Enter List:")))
print(list1)
print(type(list1))
```

4. With list() function:

```
l=list(range(0,10,2))
print(l)
print(type(l))
```

Eg:

```
s="durga"
l=list(s)
print(l)
```

result:

```
 ['d', 'u', 'r', 'g', 'a']
```

5. with split() function:

```
s="Learning Python is very very easy !!!"
l=s.split()
print(l)
print(type(l))
```

result:

```
 ['Learning', 'Python', 'is', 'very', 'very', 'easy', '!!!']
 <class 'list'>
```

Note:
Sometimes we can take list inside another list,such type of lists are called nested lists.

NESSTED LISTS:

EX:
[10,20,[30,40]]

acessing nested list values:

```
A =[10,20,[30,40]]
print(A[2][0])
```

Accessing elements of List:

We can access elements of the list either by using index or by using slice operator(:)

1. By using index:

List follows zero based index. ie index of first element is zero.

List supports both +ve and -ve indexes.

+ve index meant for Left to Right

-ve index meant for Right to Left

```
list=[10,20,30,40]

print(list[0]) ==>10

print(list[-1]) ==>40

print(list[10]) ==>IndexError: list index out of range
```

2. By using slice operator:

Syntax:
list2= list1[start:stop:step]

start ==>it indicates the index where slice has to start
 default value is 0

stop ===>It indicates the index where slice has to end
 default value is max allowed index of list ie length of the list

step ==>increment value
 default value is 1

Eg:

```
n=[1,2,3,4,5,6,7,8,9,10]
print(n[2:7:2])
print(n[4::2])
print(n[3:7])
print(n[8:2:-2])
print(n[4:100])
```

 Output
 [3, 5, 7]

```
 [5, 7, 9]
 [4, 5, 6, 7]
 [9, 7, 5]
 [5, 6, 7, 8, 9, 10]
```

List vs mutability:

Once we creates a List object,we can modify its content. Hence List objects are mutable.

Eg:

```
1) n=[10,20,30,40]
2) print(n)
3) n[1]=777
4) print(n)
```

Traversing the elements of List:

The sequential access of each element in the list is called traversal.

1. By using while loop:

```
1) n=[0,1,2,3,4,5,6,7,8,9,10]
2) i=0
3) while i<len(n):
4) print(n[i])
5) i=i+1
```

2. By using for loop:

```
1) n=[0,1,2,3,4,5,6,7,8,9,10]
2) for n1 in n:
3) print(n1)
```

3. To display only even numbers:

```
1) n=[0,1,2,3,4,5,6,7,8,9,10]
2) for n1 in n:
3) if n1%2==0:
4) print(n1)
```

4. To display elements by index wise:

```
1) l=["A","B","C"]
2) x=len(l)
3) for i in range(x):
4) print(l[i],"is available at positive index: ",i,"and at negative index: ",i-x)
```

result:

```
 A is available at positive index: 0 and at negative index: -3
 B is available at positive index: 1 and at negative index: -2
 C is available at positive index: 2 and at negative index: -1
```

Important functions of List:

I. To get information about list:

1. len():
 returns the number of elements present in the list

Eg: n=[10,20,30,40]
 print(len(n))==>4

2. count():
 It returns the number of occurrences of specified item in the list

1) n=[1,2,2,2,2,3,3]

2) print(n.count(1))

3) print(n.count(2))

4) print(n.count(3))

5) print(n.count(4))

Output

1
4
2
0

3. index() function:
returns the index of first occurrence of the specified item.

Eg:
1) n=[1,2,2,2,2,3,3]
2) print(n.index(1)) ==>0
3) print(n.index(2)) ==>1
4) print(n.index(3)) ==>5
5) print(n.index(4)) ==>ValueError: 4 is not in list

Note: If the specified element not present in the list then we will get ValueError.Hence
before index() method we have to check whether item present in the list or not by using in
operator.

print( 4 in n)==>False

II. Manipulating elements of List:

1. append() function:

We can use append() function to add item at the end of the list.

Eg:
1) list=[]
2) list.append("A")
3) list.append("B")
4) list.append("C")
5) print(list)
result:

```
 ['A', 'B', 'C']
```

Eg: To add all elements to list upto 100 which are divisible by 10

```
list=[]
for i in range(101):
    if i%10==0:
        list.append(i)
        print(list)
```

```
result:
 [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

2. insert() function:

To insert item at specified index position

```
1) n=[1,2,3,4,5]
2) n.insert(1,888)
3) print(n)
```

```
result:
 [1, 888, 2, 3, 4, 5]
```

Eg:

```
1) n=[1,2,3,4,5]
2) n.insert(10,777)
3) n.insert(-10,999)
4) print(n)
5)
```

```
result:
 [999, 1, 2, 3, 4, 5, 777]
```

Note: If the specified index is greater than max index then element will be inserted at last
position. If the specified index is smaller than min index then element will be inserted at
first position.

Differences between append() and insert()

3. extend() function:

To add all items of one list to another list

```
l1.extend(l2)
 all items present in l2 will be added to l1
```

Eg:

```
1) order1=["Chicken","Mutton","Fish"]
2) order2=["RC","KF","FO"]
3) order1.extend(order2)
4) print(order1)
```

result:

['Chicken', 'Mutton', 'Fish', 'RC', 'KF', 'FO']

Eg:

```
1) order=["Chicken","Mutton","Fish"]
2) order.extend("Mushroom")
3) print(order)

6) ['Chicken', 'Mutton', 'Fish', 'M', 'u', 's', 'h', 'r', 'o', 'o', 'm']
```

4. remove() function:

We can use this function to remove specified item from the list.If the item present multiple times then only first occurrence will be removed.

```
1) n=[10,20,10,30]
2) n.remove(10)
3) print(n)

result:
 [20, 10, 30]
```

If the specified item not present in list then we will get ValueError

```
1) n=[10,20,10,30]
2) n.remove(40)
3) print(n)

result
 ValueError: list.remove(x): x not in list
```

Note: Hence before using remove() method first we have to check specified element present in the list or not by using in operator.

5. pop() function:

It removes and returns the last element of the list.
This is only function which manipulates list and returns some element.

Eg:

```
1) n=[10,20,30,40]
2) print(n.pop())
3) print(n.pop())
4) print(n)

result:
 40
 30
 [10, 20]
```

If the list is empty then pop() function raises IndexError

Eg:

```
1) n=[]
```

2) print(n.pop()) ==> IndexError: pop from empty list

Note:

1. pop() is the only function which manipulates the list and returns some value

2. In general we can use append() and pop() functions to implement stack datastructure
by using list,which follows LIFO(Last In First Out) order.
In general we can use pop() function to remove last element of the list. But we can use to
remove elements based on index.
n.pop(index)===>To remove and return element present at specified index.
n.pop()==>To remove and return last element of the list

1) n=[10,20,30,40,50,60]
2) print(n.pop()) #60
3) print(n.pop(1)) #20
4) print(n.pop(10)) ==>IndexError: pop index out of range

Differences between remove() and pop()

Note:
List objects are dynamic. i.e based on our requirement we can increase and decrease the
size.
append(),insert() ,extend() ===>for increasing the size/growable nature
remove(), pop() ======>for decreasing the size /shrinking nature
remove() pop()

1) We can use to remove special element from the List. 1) We can use to remove last element

from the List.

2) It can’t return any value. 2) It returned removed element.

3) If special element not available then we get VALUE ERROR.

3) If List is empty then we get Error.

III. Ordering elements of List:

1. reverse():

We can use to reverse() order of elements of list.

1) n=[10,20,30,40]

2) n.reverse()

3) print(n)

result:
6) [40, 30, 20, 10]

2. sort() function:

In list by default insertion order is preserved. If want to sort the elements of list according
to default natural sorting order then we should go for sort() method.
For numbers ==>default natural sorting order is Ascending Order
For Strings ==> default natural sorting order is Alphabetical Order

```
1) n=[20,5,15,10,0]

2) n.sort()

3) print(n) #[0,5,10,15,20]

5) s=["Dog","Banana","Cat","Apple"]

6) s.sort()


7) print(s) #['Apple','Banana','Cat','Dog']
```

Note: To use sort() function, compulsory list should contain only homogeneous elements.
otherwise we will get TypeError

Eg:

```
1) n=[20,10,"A","B"]

2) n.sort()

3) print(n)
```

result:

```
5) TypeError: '<' not supported between instances of 'str' and 'int'
```

Note: In Python 2 if List contains both numbers and Strings then sort() function first sort

numbers followed by strings

```
1) n=[20,"B",10,"A"]

2) n.sort()


3) print(n)# [10,20,'A','B']
```

But in Python 3 it is invalid.

To sort in reverse of default natural sorting order:

We can sort according to reverse of default natural sorting order by using reverse=True
argument.

Eg:

1. n=[40,10,30,20]

2. n.sort()

3. print(n) ==>[10,20,30,40]

4. n.sort(reverse=True)

5. print(n) ===>[40,30,20,10]

6. n.sort(reverse=False)

7. print(n) ==>[10,20,30,40]

Aliasing and Cloning of List objects:
The process of giving another reference variable to the existing list is called aliasing.
Eg:
1) x=[10,20,30,40]
2) y=x
3) print(id(x))
4) print(id(y))
The problem in this approach is by using one reference variable if we are changing content,then those changes will be reflected to the other reference variable.
1) x=[10,20,30,40]
2) y=x
3) y[1]=777
4) print(x) ==>[10,777,30,40]
To overcome this problem we should go for cloning.
The process of creating exactly duplicate independent object is called cloning.
We can implement cloning by using slice operator or by using copy() function
10 20 30 40
x
y
10 20
777
30 40
x
y

1. By using slice operator:
1) x=[10,20,30,40]
2) y=x[:]
3) y[1]=777
4) print(x) ==>[10,20,30,40]
5) print(y) ==>[10,777,30,40]

2. By using copy() function:
1) x=[10,20,30,40]
2) y=x.copy()
3) y[1]=777
4) print(x) ==>[10,20,30,40]
5) print(y) ==>[10,777,30,40]

Q. Difference between = operator and copy() function

= operator meant for aliasing

copy() function meant for cloning
10 20 30 40
x
10 20
777
30 40
y
10 20 30 40
x
10 20
777
30 40
y

Using Mathematical operators for List Objects:
We can use + and * operators for List objects.
1. Concatenation operator(+):
We can use + to concatenate 2 lists into a single list
1) a=[10,20,30]
2) b=[40,50,60]
3) c=a+b
4) print(c) ==>[10,20,30,40,50,60]
Note: To use + operator compulsory both arguments should be list objects,otherwise we
will get TypeError.

Eg:

c=a+40 ==>TypeError: can only concatenate list (not "int") to list

c=a+[40] ==>valid


2. Repetition Operator(*):

We can use repetition operator * to repeat elements of list specified number of times

Eg:

1) x=[10,20,30]
2) y=x*3
3) print(y)==>[10,20,30,10,20,30,10,20,30]

Comparing List objects

We can use comparison operators for List objects.

Eg:

1. x=["Dog","Cat","Rat"]
2. y=["Dog","Cat","Rat"]
3. z=["DOG","CAT","RAT"]

4. print(x==y) True

5. print(x==z) False

6. print(x != z) True

Note:

Whenever we are using comparison operators(==,!=) for List objects then the following
should be considered

1. The number of elements

2. The order of elements

3. The content of elements (case sensitive)

Note: When ever we are using relatational operators(<,<=,>,>=) between List objects,only
first element comparison will be performed.

Eg:
```
1. x=[50,20,30]

2. y=[40,50,60,100,200]
3. print(x>y) True
4. print(x>=y) True
5. print(x<y) False
6. print(x<=y) False
```

Eg:
```
1. x=["Dog","Cat","Rat"]
2. y=["Rat","Cat","Dog"]
3. print(x>y) False
4. print(x>=y) False
5. print(x<y) True
6. print(x<=y) True
```

Membership operators:

We can check whether element is a member of the list or not by using memebership

operators.

in operator

not in operator

Eg:
```
1. n=[10,20,30,40]
2. print (10 in n)
3. print (10 not in n)
4. print (50 in n)
5. print (50 not in n)
```

7. Output
8. True
9. False
10. False
11. True

clear() function:

We can use clear() function to remove all elements of List.

Eg:
```
1. n=[10,20,30,40]
2. print(n)
3. n.clear()
4. print(n)


6. Output
7. D:\Python_classes>py test.py
8. [10, 20, 30, 40]
9. []
```

Nested Lists:
Sometimes we can take one list inside another list. Such type of lists are called nested
lists.

Eg:
```
1. n=[10,20,[30,40]]
2. print(n)
3. print(n[0])
4. print(n[2])
5. print(n[2][0])
6. print(n[2][1])
7.
8. Output
9. D:\Python_classes>py test.py
10. [10, 20, [30, 40]]
11. 10
12. [30, 40]
13. 30
14. 40
```
Note: We can access nested list elements by using index just like accessing multi dimensional array elements.

2.NESTED LIST COMPREHENSIONS

Nested List as Matrix:

In Python we can represent matrix by using nested lists.

```
a = [[10, 20, 30], [40, 50, 60], [70, 80, 90]]

for i in a:
  for j in i:
    print(j,end = '  ')
  print()
```

Output

```
[[10, 20, 30], [40, 50, 60], [70, 80, 90]]
Elements by Row wise:
[10, 20, 30]
[40, 50, 60]
[70, 80, 90]
Elements by Matrix style:
10 20 30
40 50 60
70 80 90
```

List Comprehensions:

It is very easy and compact way of creating list objects from any iterable objects(like
list,tuple,dictionary,range etc) based on some condition.

Syntax:

```
list=[expression for item in list if condition]
```

Eg:

```
1) s=[ x*x for x in range(1,11)]
2) print(s)
3) v=[2**x for x in range(1,6)]
4) print(v)
5) m=[x for x in s if x%2==0]
6) print(m)
7)
8) Output
9) D:\Python_classes>py test.py
10) [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

11) [2, 4, 8, 16, 32]
12) [4, 16, 36, 64, 100]
```

Eg:

```
1) words=["Balaiah","Nag","Venkatesh","Chiranjeevi"]
2) l=[w[0] for w in words]
3) print(l)
4)
5) Output['B', 'N', 'V', 'C']
```

Eg:

```
1) num1=[10,20,30,40]
2) num2=[30,40,50,60]
3) num3=[ i for i in num1 if i not in num2]
4) print(num3) [10,20]
5)
6) common elements present in num1 and num2
7) num4=[i for i in num1 if i in num2]
8) print(num4) [30, 40]
```

Eg:

```
1) words="the quick brown fox jumps over the lazy dog".split()
```

```
2) print(words)
3) l=[[w.upper(),len(w)] for w in words]
4) print(l)
5)
6) Output
7) ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
8) [['THE', 3], ['QUICK', 5], ['BROWN', 5], ['FOX', 3], ['JUMPS', 5], ['OVER', 4],
9) ['THE', 3], ['LAZY', 4], ['DOG', 3]]
```

3.DICTIONARY COMPREHENSIONS

Dictionary Data Structure
We can use List,Tuple and Set to represent a group of individual objects as a single entity.
If we want to represent a group of objects as key-value pairs then we should go for

Dictionary.

Eg:
 rollno----name
 phone number--address
 ipaddress---domain name

Duplicate keys are not allowed but values can be duplicated.

Hetrogeneous objects are allowed for both key and values.

insertion order is not preserved

Dictionaries are mutable

Dictionaries are dynamic

indexing and slicing concepts are not applicable

Note: In C++ and Java Dictionaries are known as "Map" where as in Perl and Ruby it is
known as "Hash"

How to create Dictionary?
d={} or d=dict()

we are creating empty dictionary. We can add entries as follows

```
d[100]="durga"

d[200]="ravi"

d[300]="shiva"

print(d) #{100: 'durga', 200: 'ravi', 300: 'shiva'}
```

If we know data in advance then we can create dictionary as follows

```
d={100:'durga' ,200:'ravi', 300:'shiva'}

d={key:value, key:value}
```

How to access data from the dictionary?

We can access data by using keys.

```
d={100:'durga' ,200:'ravi', 300:'shiva'}
print(d[100]) #durga
print(d[300]) #shiva
```

If the specified key is not available then we will get KeyError

```
print(d[400]) # KeyError: 400
```

We can prevent this by checking whether key is already available or not by using has_key() function or by using in operator.

d.has_key(400) ==> returns 1 if key is available otherwise returns 0

But has_key() function is available only in Python 2 but not in Python 3. Hence compulsory we have to use in operator.

```
if 400 in d:
print(d[400])
```

Q. Write a program to enter name and percentage marks in a dictionary and display information on the screen

```
rec={}
n=int(input("Enter number of students: "))
i=1
while i <=n:
 name=input("Enter Student Name: ")
 marks=input("Enter % of Marks of Student: ")
 rec[name]=marks
 i=i+1
 print("Name of Student","\t","% of marks")
for x in rec:
    print("\t",x,"\t\t",rec[x])
```

13) Output
14) D:\Python_classes>py test.py
15) Enter number of students: 3

```
16) Enter Student Name: durga
17) Enter % of Marks of Student: 60%
18) Enter Student Name: ravi
19) Enter % of Marks of Student: 70%
20) Enter Student Name: shiva
21) Enter % of Marks of Student: 80%
22) Name of Student % of marks
23) durga 60%
24) ravi 70 %
25) shiva 80%
```

How to update dictionaries?

```
d[key]=value
```

If the key is not available then a new entry will be added to the dictionary with the
specified key-value pair

If the key is already available then old value will be replaced with new value.

Eg:

```
1. d={100:"durga",200:"ravi",300:"shiva"}
2. print(d)

3. d[400]="pavan"
4. print(d)

5. d[100]="sunny"
6. print(d)
7.
8. Output
9. {100: 'durga', 200: 'ravi', 300: 'shiva'}
10. {100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
11. {100: 'sunny', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
```

How to delete elements from dictionary?

```
del d[key]
 It deletes entry associated with the specified key.
 If the key is not available then we will get KeyError
```

Eg:

```
1. d={100:"durga",200:"ravi",300:"shiva"}
2. print(d)

3. del d[100]
4. print(d)

5. del d[400]
6.
7. Output
8. {100: 'durga', 200: 'ravi', 300: 'shiva'}
9. {200: 'ravi', 300: 'shiva'}

10. KeyError: 400
```

```
d.clear()
```

To remove all entries from the dictionary

Eg:
```
1. d={100:"durga",200:"ravi",300:"shiva"}
2. print(d)

3. d.clear()
4. print(d)
5.
6. Output
7. {100: 'durga', 200: 'ravi', 300: 'shiva'}
8. {}
```

del d

To delete total dictionary.Now we cannot access d

Eg:
```
1. d={100:"durga",200:"ravi",300:"shiva"}
2. print(d)

3. del d
4. print(d)
5.
6. Output
7. {100: 'durga', 200: 'ravi', 300: 'shiva'}

8. NameError: name 'd' is not defined
```
Important functions of dictionary:

1. dict():
To create a dictionary

d=dict() ===>It creates empty dictionary
d=dict({100:"durga",200:"ravi"}) ==>It creates dictionary with specified elements

d=dict([(100,"durga"),(200,"shiva"),(300,"ravi")])==>It creates dictionary with the given
list of tuple elements

2. len()
Returns the number of items in the dictionary

3. clear():
To remove all elements from the dictionary

4. get():
To get the value associated with the key

d.get(key)
 If the key is available then returns the corresponding value otherwise returns None.It
wont raise any error.

d.get(key,defaultvalue)
 If the key is available then returns the corresponding value otherwise returns default
value.

Eg:
d={100:"durga",200:"ravi",300:"shiva"}
print(d[100]) ==>durga

print(d[400]) ==>KeyError:400

print(d.get(100)) ==durga

print(d.get(400)) ==>None

print(d.get(100,"Guest")) ==durga

print(d.get(400,"Guest")) ==>Guest

3. pop():

d.pop(key)
 It removes the entry associated with the specified key and returns the corresponding
value

 If the specified key is not available then we will get KeyError
Eg:

1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d.pop(100))

3) print(d)
4) print(d.pop(400))
5)
6) Output

7) durga
8) {200: 'ravi', 300: 'shiva'}
9) KeyError: 400

4. popitem():
It removes an arbitrary item(key-value) from the dictionaty and returns it.

Eg:
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d)
3) print(d.popitem())
4) print(d)
5)
6) Output
7) {100: 'durga', 200: 'ravi', 300: 'shiva'}
8) (300, 'shiva')
9) {100: 'durga', 200: 'ravi'}

If the dictionary is empty then we will get KeyError

d={}

print(d.popitem()) ==>KeyError: 'popitem(): dictionary is empty'

5. keys():

It returns all keys associated eith dictionary
Eg:

```
1) d={100:"durga",200:"ravi",300:"shiva"}
2) print(d.keys())

3) for k in d.keys():
4) print(k)
5)
6) Output
7) dict_keys([100, 200, 300])
8) 100
9) 200
10) 300
```

6. values():
It returns all values associated with the dictionary

Eg:

```
1. d={100:"durga",200:"ravi",300:"shiva"}
2. print(d.values())
3. for v in d.values():
4. print(v)
5.
6. Output
7. dict_values(['durga', 'ravi', 'shiva'])
8. durga
9. ravi
10. shiva
```

7. items():

It returns list of tuples representing key-value pairs.

[(k,v),(k,v),(k,v)]

Eg:

```
1. d={100:"durga",200:"ravi",300:"shiva"}
2. for k,v in d.items():
3. print(k,"--",v)
4.
5. Output
6. 100 -- durga
7. 200 -- ravi
8. 300 -- shiva
```

8. copy():

To create exactly duplicate dictionary(cloned copy)

```
d1=d.copy();
```

9. setdefault():

```
d.setdefault(k,v)
```

If the key is already available then this function returns the corresponding value.

If the key is not available then the specified key-value will be added as new item to the
dictionary.
Eg:

```
1. d={100:"durga",200:"ravi",300:"shiva"}
2. print(d.setdefault(400,"pavan"))
3. print(d)
4. print(d.setdefault(100,"sachin"))
5. print(d)
6.
7. Output
8. pavan
9. {100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
10. durga
11. {100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
```

tuple data type:

tuple data type is exactly same as list data type except that it is immutable.i.e we cannot

chage values.

Tuple elements can be represented within parenthesis.

Eg:

```
1) t=(10,20,30,40)
2) type(t)
3) <class 'tuple'>
4) t[0]=100

5) TypeError: 'tuple' object does not support item assignment

6) >>> t.append("durga")

7) AttributeError: 'tuple' object has no attribute 'append'

8) >>> t.remove(10)

9) AttributeError: 'tuple' object has no attribute 'remove'
```

Note: tuple is the read only version of list

range Data Type:

range Data Type represents a sequence of numbers.

The elements present in range Data type are not modifiable. i.e range Data type is

immutable.

Form-1: range(10)

generate numbers from 0 to 9
Eg:

```
r=range(10)
for i in r : print(i) 0 to 9
```

Form-2: range(10,20)

generate numbers from 10 to 19

r = range(10,20)

for i in r : print(i) 10 to 19

Form-3: range(10,20,2)

2 means increment value
r = range(10,20,2)

for i in r : print(i) 10,12,14,16,18

We can access elements present in the range Data Type by using index.

r=range(10,20)

r[0]==>10

r[15]==>IndexError: range object index out of range
We cannot modify the values of range data type

Set Data Structure

¬ If we want to represent a group of unique values as a single entity then we should go
for set.
¬ Duplicates are not allowed.
¬ Insertion order is not preserved.But we can sort the elements.
¬ Indexing and slicing not allowed for the set.
¬ Heterogeneous elements are allowed.
¬ Set objects are mutable i.e once we creates set object we can perform any changes in
that object based on our requirement.
¬ We can represent set elements within curly braces and with comma seperation
¬ We can apply mathematical operations like union,intersection,difference etc on set
objects.

Creation of Set objects:
Eg:
1. s={10,20,30,40}
2. print(s)
3. print(type(s))
4.
5. Output
6. {40, 10, 20, 30}
7. <class 'set'>
We can create set objects by using set() function
s=set(any sequence)
Eg 1:
1. l = [10,20,30,40,10,20,10]
2. s=set(l)

```
3. print(s) # {40, 10, 20, 30}
Eg 2:
1. s=set(range(5))
2. print(s) #{0, 1, 2, 3, 4}
```
Note: While creating empty set we have to take special care.
Compulsory we should use set() function.

s={} ==>It is treated as dictionary but not empty set.
Eg:
```
1. s={}
2. print(s)
3. print(type(s))
4.
5. Output
6. {}
7. <class 'dict'>
```
Eg:
```
1. s=set()
2. print(s)
3. print(type(s))
4.
5. Output
6. set()
7. <class 'set'>
```
Important functions of set:
1. add(x):
Adds item x to the set
Eg:
```
1. s={10,20,30}
2. s.add(40);
3. print(s) #{40, 10, 20, 30}
```
2. update(x,y,z):

To add multiple items to the set.
Arguments are not individual elements and these are Iterable objects like List,range etc.
All elements present in the given Iterable objects will be added to the set.

Eg:
```
1. s={10,20,30}
2. l=[40,50,60,10]
3. s.update(l,range(5))
4. print(s)
```
Q. What is the difference between add() and update() functions in set?
We can use add() to add individual item to the Set,where as we can use update() function
to add multiple items to Set.
add() function can take only one argument where as update() function can take any number of arguments but all arguments should be iterable objects.
Q. Which of the following are valid for set s?
```
1. s.add(10)
2. s.add(10,20,30) TypeError: add() takes exactly one argument (3 given)
3. s.update(10) TypeError: 'int' object is not iterable
4. s.update(range(1,10,2),range(0,10,2))
```
3. copy():
Returns copy of the set.

It is cloned object.

```
s={10,20,30}
s1=s.copy()
print(s1)
```

4. pop():

It removes and returns some random element from the set.

Eg:

```
1. s={40,10,30,20}
2. print(s)
3. print(s.pop())
4. print(s)
5.
6. Output
7. {40, 10, 20, 30}
8. 40
9. {10, 20, 30}
```

5. remove(x):

It removes specified element from the set.

If the specified element not present in the Set then we will get KeyError

```
s={40,10,30,20}
s.remove(30)
print(s) # {40, 10, 20}
s.remove(50) ==>KeyError: 50
```

6. discard(x):

It removes the specified element from the set.

If the specified element not present in the set then we won't get any error.

```
s={10,20,30}
s.discard(10)
print(s) ===>{20, 30}
s.discard(50)
print(s) ==>{20, 30}
```

Q. What is the difference between remove() and discard() functions in Set?

Q. Explain differences between pop(),remove() and discard() functionsin Set?

7.clear():

To remove all elements from the Set.

```
1. s={10,20,30}
2. print(s)
3. s.clear()
4. print(s)
5.
6. Output
7. {10, 20, 30}
8. set()
```

Mathematical operations on the Set:

1.union():

x.union(y) ==>We can use this function to return all elements present in both sets

x.union(y) or x|y

Eg:
```
x={10,20,30,40}
y={30,40,50,60}
print(x.union(y)) #{10, 20, 30, 40, 50, 60}
print(x|y) #{10, 20, 30, 40, 50, 60}
```

2. intersection():
x.intersection(y) or x&y
Returns common elements present in both x and y
Eg:
```
x={10,20,30,40}
y={30,40,50,60}
print(x.intersection(y)) #{40, 30}
print(x&y) #{40, 30}
```

3. difference():
x.difference(y) or x-y
returns the elements present in x but not in y
Eg:
```
x={10,20,30,40}
y={30,40,50,60}
print(x.difference(y)) #{10, 20}
print(x-y) #{10, 20}
print(y-x) #{50, 60}
```

4.symmetric_difference():
x.symmetric_difference(y) or x^y
Returns elements present in either x or y but not in both
Eg:
```
x={10,20,30,40}
y={30,40,50,60}
print(x.symmetric_difference(y)) #{10, 50, 20, 60}
print(x^y) #{10, 50, 20, 60}
```

Membership operators: (in , not in)
Eg:
```
1. s=set("durga")
2. print(s)
3. print('d' in s)
4. print('z' in s)
5.
6. Output
7. {'u', 'g', 'r', 'd', 'a'}
8. True
9. False
```

Set Comprehension:
Set comprehension is possible.

```
s={x*x for x in range(5)}
print(s) #{0, 1, 4, 9, 16}
s={2**x for x in range(2,10,2)}
print(s) #{16, 256, 64, 4}
```
set objects won't support indexing and slicing:
Eg:
```
s={10,20,30,40}
print(s[0]) ==>TypeError: 'set' object does not support indexing
print(s[1:3]) ==>TypeError: 'set' object is not subscriptable
```

Formatting the Strings:
We can format the strings with variable values by using replacement operator {} and format()
method.
The main objective of format() method to format string into meaningful output form.
Case- 1: Basic Formatting for default, positional and keyword arguments

```
name='durga'
salary=10000
age=48
print("{} 's salary is {} and his age is {}".format(name,salary,age))
print("{0} 's salary is {1} and his age is {2}".format(name,salary,age))
print("{x} 's salary is {y} and his age is {z}".format(z=age,y=salary,x=name))
```

```
d--->Decimal IntEger
f----->Fixed point number(float).The default precision is 6
b-->Binary format
o--->Octal Format
x-->Hexa Decimal Format(Lower case)
X-->Hexa Decimal Format(Upper case)
```

Eg-1:

```
print("The intEger number is: {}".format(123))
print("The intEger number is: {:d}".format(123))
print("The intEger number is: {:5d}".format(123))
print("The intEger number is: {:05d}".format(123))
```

Output:

```
The intEger number is: 123
The intEger number is: 123
The intEger number is: 123
The intEger number is: 00123
```

Eg-2:

```
print("The float number is: {}".format(123.4567))
print("The float number is: {:f}".format(123.4567))

print("The float number is: {:8.3f}".format(123.4567))
print("The float number is: {:08.3f}".format(123.4567))
print("The float number is: {:08.3f}".format(123.45))
print("The float number is: {:08.3f}".format(786786123.45))
```

Output:

```
The float number is: 123.4567
The float number is: 123.456700
The float number is: 123.457
The float number is: 0123.457
The float number is: 0123.450
The float number is: 786786123.450
```

Note:

{:08.3f}
Total positions should be minimum 8.
After decimal point exactly 3 digits are allowed.If it is less then 0s will be placed in the last
positions
If total number is < 8 positions then 0 will be placed in MSBs
If total number is >8 positions then all intEgral digits will be considered.
The extra digits we can take only 0
Note: For numbers default alignment is Right Alignment(>)

Eg-3: Print Decimal value in binary, octal and hexadecimal form

```
print("Binary Form:{0:b}".format(153))
print("Octal Form:{0:o}".format(153))
print("Hexa decimal Form:{0:x}".format(154))
print("Hexa decimal Form:{0:X}".format(154))
```

Output:

```
Binary Form:10011001
Octal Form:231
Hexa decimal Form:9a
Hexa decimal Form:9A
```

Note: We can represent only int values in binary, octal and hexadecimal and it is not possible for
float values.

Note:

{:5d} It takes an intEger argument and assigns a minimum width of 5.

{:8.3f} It takes a float argument and assigns a minimum width of 8 including "." and after decimal
point excatly 3 digits are allowed with round operation if required

{:05d} The blank places can be filled with 0. In this place only 0 allowed.

Case-3: Number formatting for signed numbers

While displaying positive numbers,if we want to include + then we have to write {:+d} and {:+f}

Using plus for -ve numbers there is no use and for -ve numbers - sign will come automatically.

```
print("int value with sign:{:+d}".format(123))
print("int value with sign:{:+d}".format(-123))
print("float value with sign:{:+f}".format(123.456))
print("float value with sign:{:+f}".format(-123.456))
```

Output:

```
int value with sign:+123
int value with sign:-123
float value with sign:+123.456000
float value with sign:-123.456000
```

ACCESSING API ESSENTIAL(1HR)

1. INTRODUCTION

What are API in Python?

Image result for api essentials in python

An API, or Application Programming Interface, is a server that you can use to retrieve and send data to using code. APIs are most commonly used to retrieve data, and that will be the focus of this beginner tutorial. When we want to receive data from an API, we need to make a request.

Python API Tutorials

In this section we collect tutorials related to API design or interacting with APIs using Python.

REST APIs in web applications would be one example where Python shines.

Free Bonus: Click here to download a copy of the "REST API Examples" Guide and get a hands-on introduction to
Python + REST API principles with actionable examples.

There’s an amazing amount of data available on the Web. Many web services, like YouTube and GitHub, make their
data accessible to third-party applications through an application programming interface (API). One of the most

popular ways to build APIs is the REST architecture style. Python provides some great tools not only to get data
from REST APIs but also to build your own Python REST APIs.

In this tutorial, you'll learn:

What REST architecture is
How REST APIs provide access to web data
How to consume data from REST APIs using the requests library
What steps to take to build a REST API
What some popular Python tools are for building REST APIs
By using Python and REST APIs, you can retrieve, parse, update, and manipulate the data provided by
any web service you're interested in.

Free Bonus: Click here to download a copy of the "REST API Examples" Guide and get a hands-on introduction to Python + REST API principles with actionable examples.

REST Architecture
REST stands for representational state transfer and is a software architecture style that defines a
pattern for client and server communications over a network. REST provides a set of constraints for
software architecture to promote performance, scalability, simplicity, and reliability in the system.

REST defines the following architectural constraints:

Stateless: The server won't maintain any state between requests from the client.
Client-server: The client and server must be decoupled from each other, allowing each to develop independently.
Cacheable: The data retrieved from the server should be cacheable either by the client or by the server.
Uniform interface: The server will provide a uniform interface for accessing resources without defining their
representation.
Layered system: The client may access the resources on the server indirectly through other layers such as a
proxy or load balancer.
Code on demand (optional): The server may transfer code to the client that it can run, such as JavaScript for
a single-page application.
Note, REST is not a specification but a set of guidelines on how to architect a network-connected software system.

Django rest framework:

step1:first create a project
step2:change the directory to the project name

step3: then give the command "pip install djangorestframework" to install restframework then
step 4:create a application
step5: mention the app name in settings file
step6:then migrate the application after that create models inside of the application folder

step7:then start creating  fileds for your json

before that we need to import model from django

CharField   A field to store text based values.

EmailField  It is a CharField that checks that the value is a valid email address.

URLField    A CharField for a URL, validated by URLValidator.

models.py:

```
from django.db import models

# Create your models here.

class samp(models.Model):
    name = models.CharField(max_length=40)
    id = models.CharField(max_length=34)
    sal = models.IntegerField(default=0)
    exp = models.IntegerField(default=0)
```

serializers:

to converting model data in another streaming method

create serializer file inside of your app folder

serializers.py:

```
from rest_framework import serializers
from .models import samp


class sampserializer(serializers.ModelSerializer):
    class Meta:
        model = samp
        fields = '__all__'
```

views.py:

```
from rest_framework import generics
from .serializer import instserializer,courseserializer
from .models import samp

class sampListView(generics.ListCreateAPIView):
    serializer_class = sampserializer
    queryset = inst.objects.all()
```

for application:

urls.py:

```
from django.urls import path
from .views import sampListView, courseListView

urlpatterns = [

    path('instruc', InstructListView.as_view()),

      ]
```

then mention in on your link on project level

PROJECTS:

PROJECT NO1:(WORD GUESSING GAME):

```
import random
# library that we use in order to choose
# on random words from a list of words

name = input("What is your name? ")
# Here the user is asked to enter the name first

print("Good Luck ! ", name)

words = ['rainbow', 'computer', 'science', 'programming',
            'python', 'mathematics', 'player', 'condition',
            'reverse', 'water', 'board', 'HELLO']

# Function will choose one random
# word from this list of words
word = random.choice(words)


print("Guess the characters")

guesses = ''

# any number of turns can be used here
turns = 12


while turns > 0:

      # counts the number of times a user fails
      failed = 0

      # all characters from the input
      # word taking one at a time.
      for char in word:

            # comparing that character with
```

```
            # the character in guesses
            if char in guesses:
                  print(char)

            else:
                  print("_")

                  # for every failure 1 will be
                  # incremented in failure
                  failed += 1


      if failed == 0:
            # user will win the game if failure is 0
            # and 'You Win' will be given as output
            print("You Win")

            # this print the correct word
            print("The word is: ", word)
            break

      # if user has input the wrong alphabet then
      # it will ask user to enter another alphabet
      guess = input("guess a character:")

      # every input character will be stored in guesses
      guesses += guess

      # check input with the character in word
      if guess not in word:

            turns -= 1

            # if the character doesn’t match the word
            # then “Wrong” will be given as output
            print("Wrong")

            # this will print the number of
            # turns left for the user
            print("You have", + turns, 'more guesses')


            if turns == 0:
                  print("You Loose")



Project 2:(CALCULATOR):

def addition ():
    print("Addition")
    n = float(input("Enter the number: "))
    t = 0 //Total number enter
    ans = 0
    while n != 0:
        ans = ans + n
        t+=1
        n = float(input("Enter another number (0 to calculate): "))
    return [ans,t]
```

```
def subtraction ():
    print("Subtraction");
    n = float(input("Enter the number: "))
    t = 0 //Total number enter
    sum = 0
    while n != 0:
        ans = ans - n
        t+=1
        n = float(input("Enter another number (0 to calculate): "))
    return [ans,t]
def multiplication ():
    print("Multiplication")
    n = float(input("Enter the number: "))
    t = 0 //Total number enter
    ans = 1
    while n != 0:
        ans = ans * n
        t+=1
        n = float(input("Enter another number (0 to calculate): "))
    return [ans,t]
def average():
    an = []
    an = addition()
    t = an[1]
    a = an[0]
    ans = a / t
    return [ans,t]
// main...
while True:
    list = []
    print(" My first python program!")
    print(" Simple Calculator in python by Malik Umer Farooq")
    print(" Enter 'a' for addition")
    print(" Enter 's' for substraction")
    print(" Enter 'm' for multiplication")
    print(" Enter 'v' for average")
    print(" Enter 'q' for quit")
    c = input(" ")
    if c != 'q':
        if c == 'a':
            list = addition()
            print("Ans = ", list[0], " total inputs ",list[1])
        elif c == 's':
            list = subtraction()
            print("Ans = ", list[0], " total inputs ",list[1])
        elif c == 'm':
            list = multiplication()
            print("Ans = ", list[0], " total inputs ",list[1])
        elif c == 'v':
            list = average()
            print("Ans = ", list[0], " total inputs ",list[1])
        else:
            print ("Sorry, invilid character")
    else:
        break
```