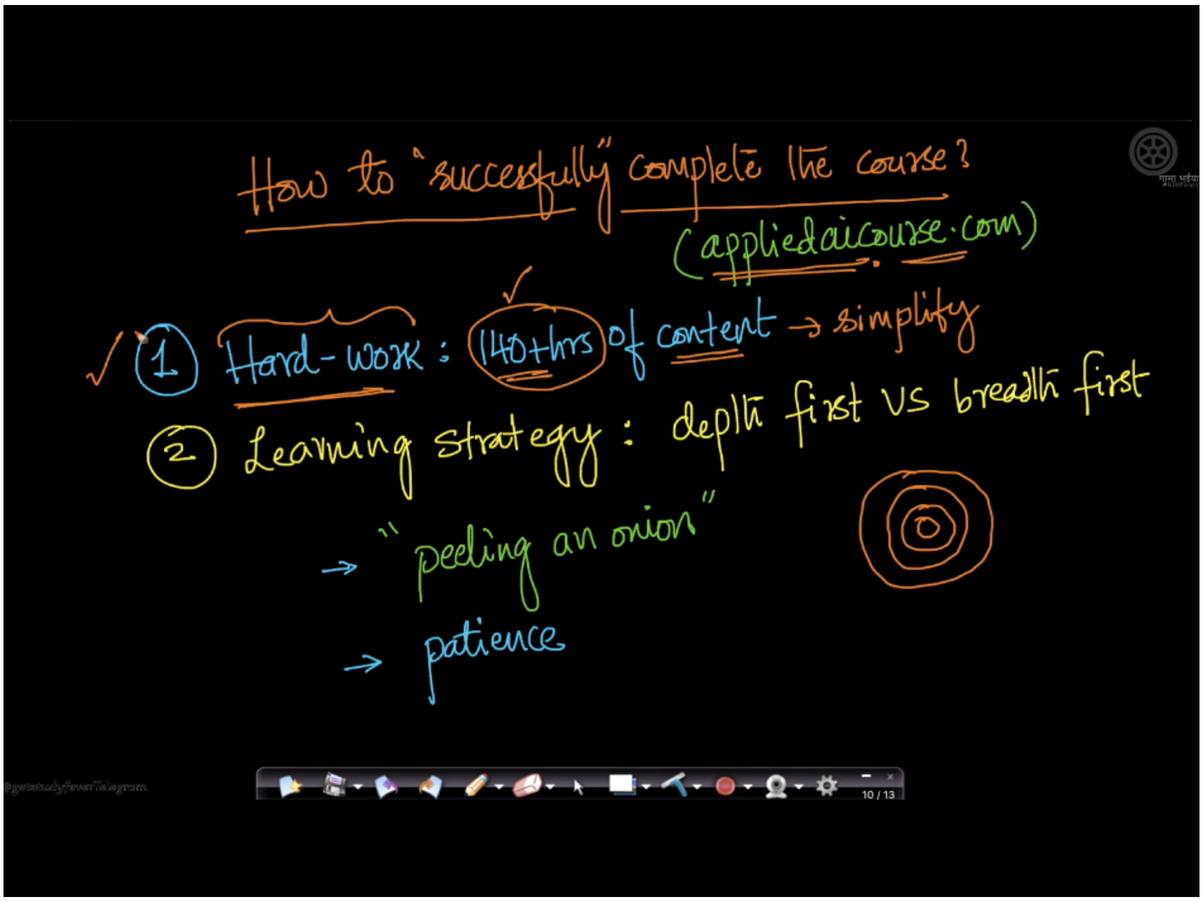
[AppliedAI]
Machine Learning Online Course

Module 1 _ Fundamentals of Programming

1. How to utilise AppliedAI Course



How to "successfully" complete the course?

(appliedaicourse.com)

✓ (1) Hard-work : 140+hrs of content → simplify

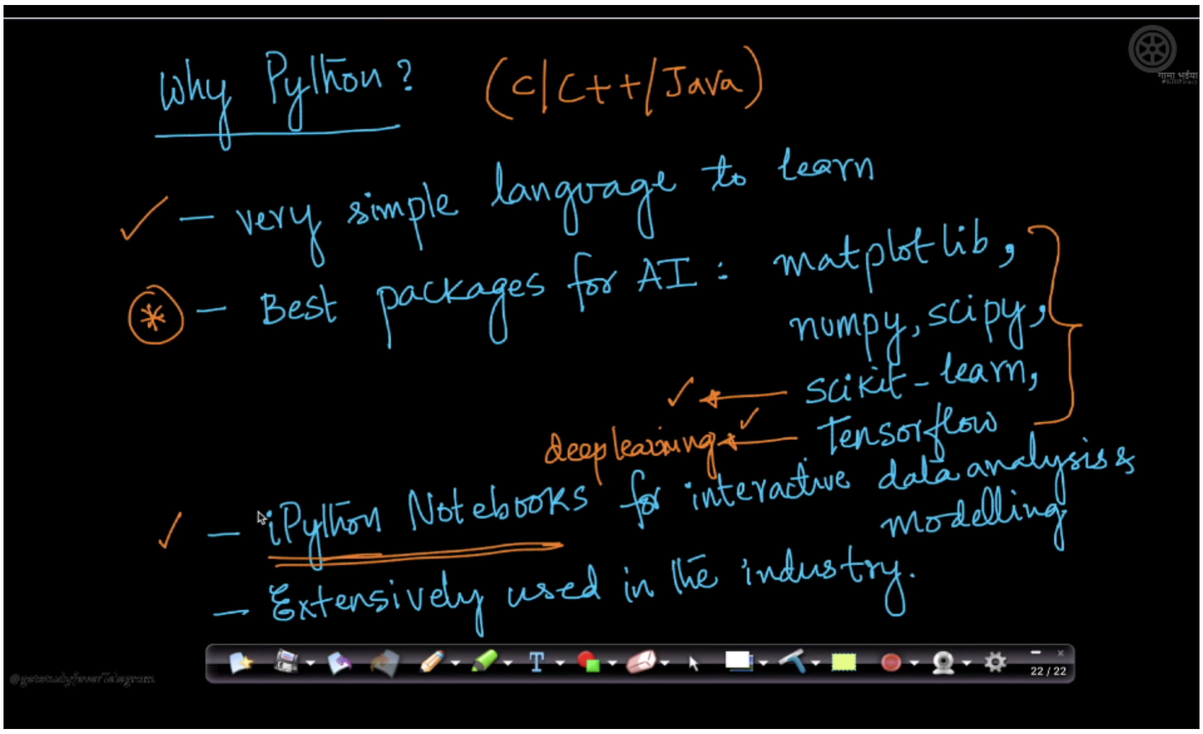
(2) Learning strategy : depth first vs breadth first

→ "peeling an onion"

→ patience

10 / 13

2. Python f



Why Python? (C|C++/Java)

✓ – very simple language to learn

⊛ – Best packages for AI: matplotlib, numpy, scipy, ✓ ← scikit-learn, deep learning ← ✓ Tensorflow

✓ – iPython Notebooks for interactive data analysis & modelling

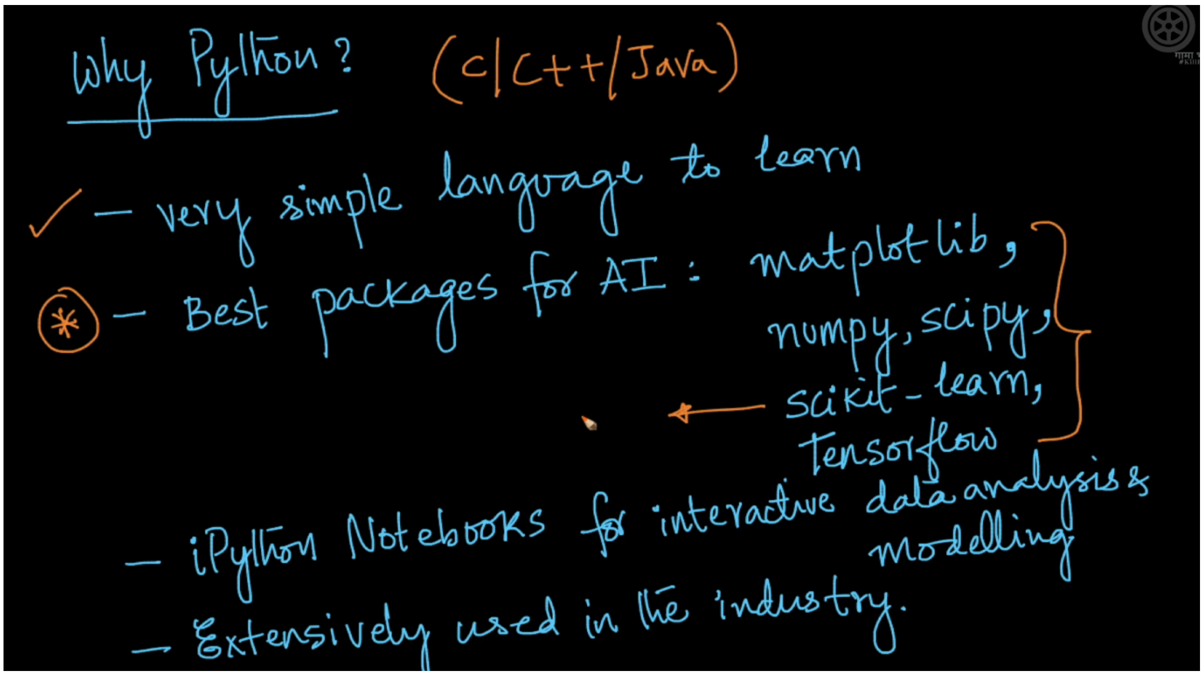
– Extensively used in the industry.

or Data Science Introduction

2. Python for Data Science Introduction

1. Python, Anaconda and relevant packages installations-

2. Why learn Python-



Why Python? (C/C++/Java)

✓ – very simple language to learn

(✳) – Best packages for AI : matplotlib, numpy, scipy, scikit-learn, Tensorflow

– iPython Notebooks for interactive data analysis & modelling

– Extensively used in the industry.

3. Keywords and identifiers-

Keywords are th reserved ords in python
We can't use a keyword as variable name, function name or any other identifier
Keywords are case sensitive

```
#Get all keywords in python
import keyword
print ( keyword. kwlist )
print( " \nTota1 number of keywords: ",len ( keyword. kwlist) )
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

Tota1 number of keywords:  35
```

Identifiers-

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.
Rules for Writing Identifiers:
1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
2. An identifier cannot start with a digit. 1 variable is invalid, but variable 1 is perfectly fine.
3. Keywords cannot be used as identifiers.

We cannot use special symbols like !, @, #, $, % etc. in our identifier.

4. comments indentation and statements-

Comments are lines that exist in computer programs that are ignored by compilers and interpreters.

Including comments in programs makes code more readable for humans as it provides some information explanation about what each part of a program is doing.

In general, it is a good idea to write comments while you are writing or updating a program as it is easy to forget your thought process later on, and comments written later may be less useful in the long term.

In Python, we use the hash (#) symbol to start writing a comment.

```
#Print Hello world to console
print( "Hello, world" )
```

```
Hello    world
```

Multi Line Comments

If we have comments that extend multiple lines, one way of doing it is
to use hash (#) in the
beginning of each line.

```
#this is a long comment
#and it extends
#Mul tiple lines
```

Or

```
"""this is a long comment
and it extends
Multiple lines"""
```

Python Indentation

1. Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.
2. A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

3. Generally four whitespaces are used for indentation and is preferred over tabs.

Python Statement

Instructions that a Python interpreter can execute are called statements.

```
#statment
a = 1
```

Multi-Line Statement

In Python, end of a statement is marked by a newline character. But we can make a statement
extend over multiple lines with the line continuation character O.

```
#multi


a = 1+2+3+\
4+5+6\
+7+8

a = (1+2+3+4+5+6)
```

```
a = 10: b = 20: c= 30
```

5. Variables and data types in Python-

Variables
A variable is a location in memory used to store some data (value).
They are given unique names to differentiate between different memory locations. The rules for
writing a variable name is same as the rules for writing identifiers in Python.
We don't need to declare a variable before using it. In Python, we simply assign a value to a variable
and it will exist. We don't even have to declare the type of the variable. This is handled internally
according to the type of value we assign to the variable.

Variable Assignments

We use the assignment operator (=) to assign values to a variable

```
a =  10
b =  5.5
c = "ML"
```

Multiple Assignments

```
a,b,c = 10,5.5, "ML"
```

```
a=b=c ="AI" #assign the same value to multiple variables at once
```

Storage Locations

```
#Storage Locations
x = 3
print(id(x))
```

```
140710856064128
```

Data Types

Every value in Python has a datatype. Since everything is an object in Python programming, data
types are actually classes and variables are instance (object) of these classes.

Numbers

Integers, floating point numbers and complex numbers falls under Python numbers category. They
are defined as int, float and cgmplex class in Python.
We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

```
a = 5
print(a, "is of type ",type(a))
```

```
5 is of type  <class 'int'>
```

Boolean

Boolean represents the truth values False and True

```
a = True
print ( type (a) )
```

```
<class 'bool'>
```

Python Strings

String is sequence of Unicode characters.
We can use single quotes or double quotes to represent strings.
Multi-line strings can be denoted using triple quotes,'' or""
A string in Python consists of a series or sequence of characters - letters, numbers, and special
characters.

Strings can be indexed - often synonymously called subscripted as well.
Similar to C, the first character of a string has the index O.

Python List

List is an ordered sequence of items. It is one of the most used datatype in Python and is very
flexible. All the items in a list do not need to be of the same type.
Declaring a list is , Items separated by commas are enclosed within brackets [].

```
a = [10, 20.5, " Hello " ]
print (a)
```

```
[10, 20.5, ' Hello ']
```

Lists are mutable, meaning, value of elements of a list can be altered.

Python Tuple

Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable.
Tuples once created cannot be modified.

Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside
braces { }. Items in a set are not ordered.

Python Dictionary

Dictionary is an unordered collection of key-value pairs.
In Python, dictionaries are defined within braces O with each item being a pair in the form key:value.
Key and value can be of any type.

Python List-[]
Python Tuple-()
Python Set-{}
Python Dictionary-{key:value}

Conversion between Datatypes

We can convert between different data types by using different type conversion functions like into,
float(), str() etc.

6. Standard Input and Output-

Python Output

We use the print() function to output data to the standard output device

```
#pritn Output
print("Hello World")
```

```
Hello World
```

```
a = 10
print("The value of a is ", a )
print("The value of a is "+ str(a))
```

```
The value of a is  10
The value of a is 10
```

Output Formatting

```
#Output Formatting
a = 10; b = 20 #Multiple statments in single line.

print("The value of a is {} " .format(a,b))# default
```

```
The value of a is 10
```

```
#Output Formatting
a = 10; b = 20 #Multiple statments in single line.

print("The value of a is {1} and a is {0} " .format(a,b)) #specific
Position
```

```
#Can use keyword arguments to format the string
print("Hello {name}, {greeting}". format(name = "Piyush", greeting
="Good morning"))
```

```
Hello
Piyush          Good morning
```

```
# 'we can combine positional arguments with keyword arguments
print('The story of {0}, {1} and {other}' .format('Bill', 'Manfred',
other= 'Georg'))
```

Python Input

want to take the input from the user. In Python, we have the input() function to allow this.

```
num = input("Enter a number: ")
print(num)
```

7. Operators-

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

Operator Typesf

1. Arithmetic operators
2. Comparison (Relational) operators
3. Logical (Boolean) operators
4.Bitwise operators
5. Assignment operators
6. Special operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

+ , -, *, /, %, //, ** are arithmetic operators

```
x, y = 10, 20

#addition
print(x + y)

#subtraction(-)

#multiplication(*)

#division(/)

#modulo division (%)

#Floor Division (//)

#Exponent (**)
```

Comparision Operators

Comparison operators are used to compare values. It either returns True or False according to the condition.
  >, <, ==, !=, >=, <= are comparision operators
a, b = 10, 20

print(a < b) #check a is less than b

#check a is greater than b

#check a is equal to b

#check a is not equal to b (!=)

#check a greater than or equal to b

#check a less than or equal to b

Logical Operators

Logical operators are and, or, not
 Operators.

```
a, b = True, False

#print a and b
print(a and b)

#print a or b

#print not b
```

Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit
  &, |, ~, ^, >>, << are Bitwise operators

```
a, b = 10, 4

#Bitwise AND
print(a & b)

#Bitwise OR
```

```
#Bitwise NOT


#Bitwise XOR


#Bitwise rightshift


#Bitwise Leftshift
```

Assignment operators

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

=, +=, -=, *=, /=, %=, //=, **=, &=, |=, ^=, >>=, <<= are Assignment operators

```
a = 10

a += 10          #add AND
print(a)

#subtract AND (-=)

#Multiply AND (*=)

#Divide AND (/=)

#Modulus AND (%=)

#Floor Division (//=)

#Exponent AND (**=)
```

Special Operators

# Identity Operators
is and is not are the identity operators in Python.

They are used to check if two values (or variables) are located on the same part of the memory.

```
a = 5
b = 5
print(a is b)     #5 is object created once both a and b points to same
object

#check is not
```

```
True
```

```
l1 = [1, 2, 3]
l2 = [1, 2, 3]
print(l1 is l2)
```

```
False
```

```
s1 = "Satish"
s2 = "Satish"
print(s1 is not s2)
```

```
False
```

MemberShip Operators

in and not in are the membership operators in Python.

They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and   dictionary).

```
lst = [1, 2, 3, 4]
print(1 in lst)        #check 1 is present in a given list or not

#check 5 is present in a given list
```

```
True
```

```
d = {1: "a", 2: "b"}
print(1 in d)
```

```
True
```

8. Control flow_ if else-

Python if else Statement

The if...elif...else statement is used in Python for decision making.

if statement syntax

```
    if test expression:

        statement(s)
```

The program evaluates the test expression and will execute statement(s) only if the text expression is True.

If the text expression is False, the statement(s) is not executed.

Python interprets non-zero values as True. None and 0 are interpreted as False.

Flow Chart

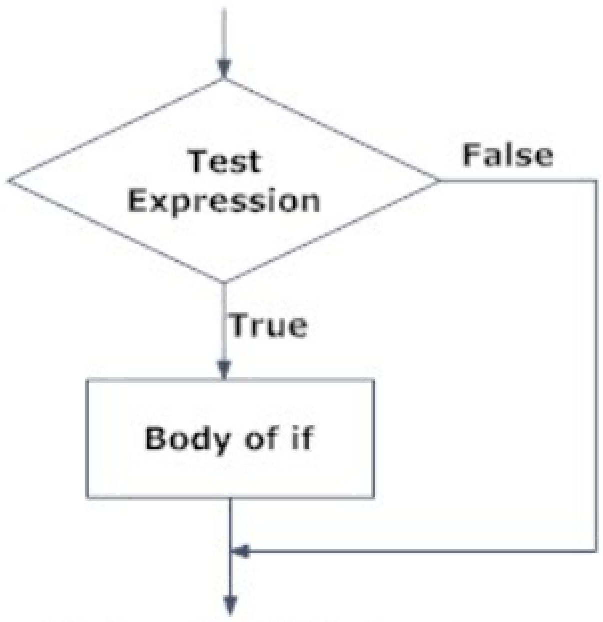


Fig: Operation of if statement

EX

```
num = 10

# try 0, -1 and None
if None:
    print("Number is positive")
print("This will print always")      #This print statement always print

#change number
```

if ... else Statement

Syntax:

```
    if test expression:

        Body of if

    else:

        Body of else
```

Flow Chart

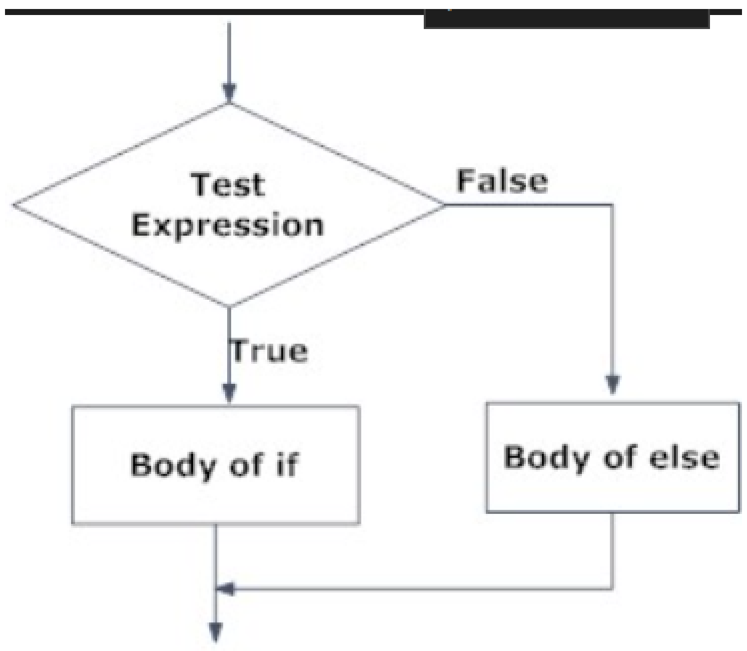


Fig: Operation of if...else statement

Example

```
num = 10
if num > 0:
    print("Positive number")
else:
    print("Negative Number")
```

if...elif...else Statement

Syntax:

```
    if test expression:

        Body of if
    elif test expression:

        Body of elif
    else:

        Body of else
```

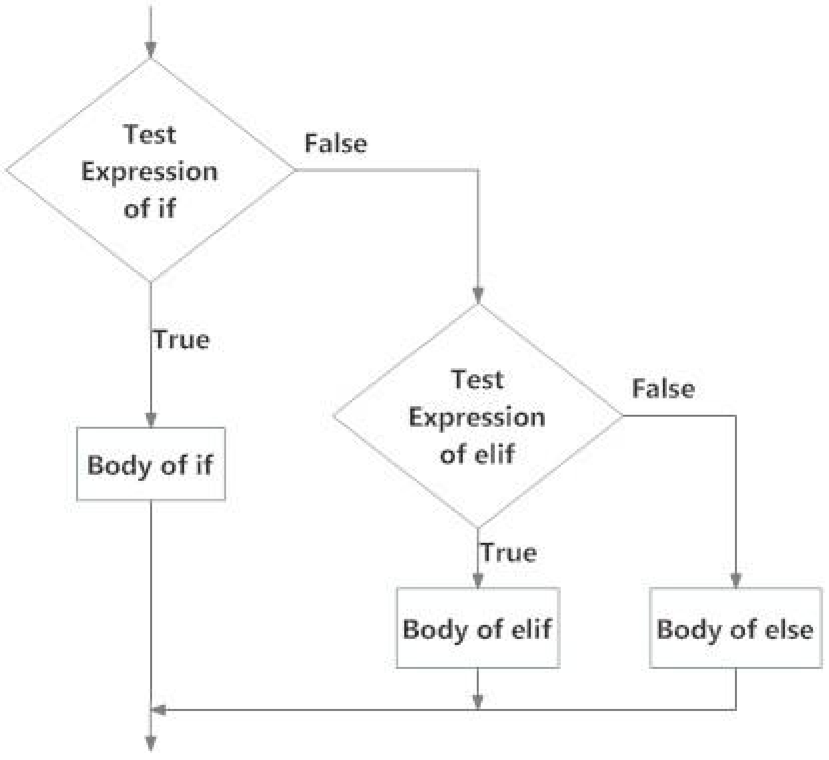


Fig: Operation of if...elif...else statement

Flow Chart

Example:

```
num = 0

if num > 0:
    print("Positive number")
elif num == 0:
    print("ZERO")
else:
    print("Negative Number")
```

Nested if Statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Example:

```
num = 10.5
```

```
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative Number")
```

Python program to find the largest element among three Numbers

```
num1 = 10
num2 = 50
num3 = 15

if (num1 >= num2) and (num1 >= num3):           #logical operator   and
    largest = num1
elif (num2 >= num1) and (num2 >= num3):
    largest = num2
else:
    largest = num3
print("Largest element among three numbers is: {}".format(largest))
```

```
Largest element among three numbers is: 50
```

9. Control flow_ while loop-

Python while Loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

Syntax:

```
    while test_expression:

        Body of while
```

The body of the loop is entered only if the test_expression evaluates to True.

After one iteration, the test expression is checked again.

This process continues until the test_expression evaluates to False.

Flow Chart

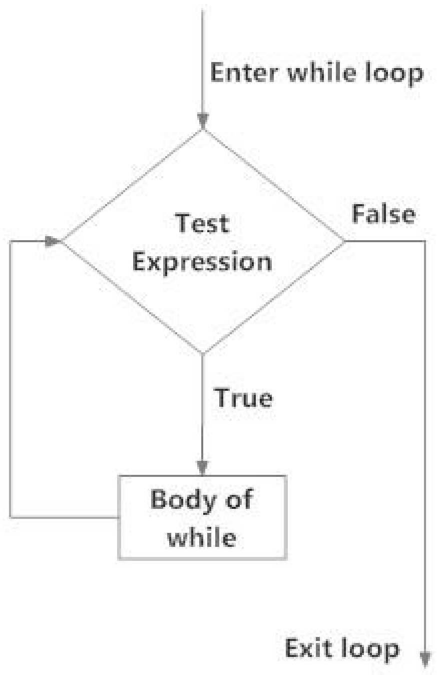


Fig: operation of while loop

Example

```
#Find product of all numbers present in a list

lst = [10, 20, 30, 40, 60]

product = 1
index = 0

while index < len(lst):
    product *= lst[index]
    index += 1

print("Product is: {}".format(product))
```

Product is: 14400000

Python Program to check given number is Prime number or not

```
num = int(input("Enter a number: "))        #convert string to int



isDivisible = False;


i=2;
```

```
while i < num:
    if num % i == 0:
        isDivisible = True;
        print ("{} is divisible by {}".format(num,i) )
    i += 1;

if isDivisible:
    print("{} is NOT a Prime number".format(num))
else:
    print("{} is a Prime number".format(num))
```

```
Enter a number: 33
33 is divisible by 3
33 is divisible by 11
33 is NOT a Prime number
```

10. Control flow_ for loop-

Python for Loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.

Iterating over a sequence is called traversal.

Syntax:

```
    for element in sequence :

        Body of for
```

Here, element is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence.

Flow Chart

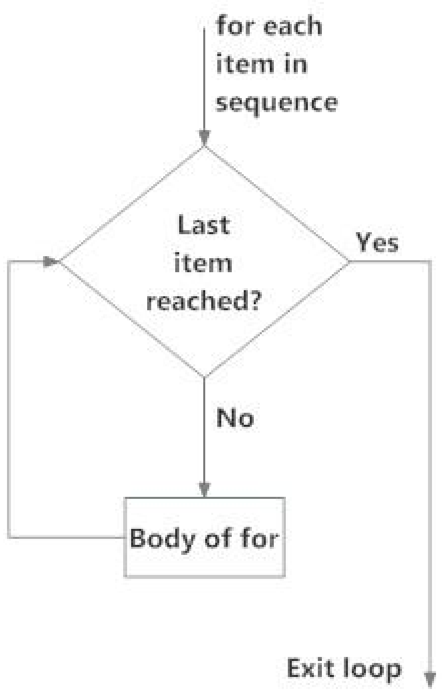


Fig: operation of for loop

Example

```
#Find product of all numbers present in a list

lst = [10, 20, 30, 40, 50]

product = 1
#iterating over the list
for ele in lst:
    print(type(ele))
    product *= ele

print("Product is: {}".format(product))
```

```
<class 'int'>
<class 'int'>
<class 'int'>
<class 'int'>
<class 'int'>
Product is: 12000000
```

range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

```
#print range of 10
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

for i in range (start, end, step)

```
#print range of numbers from 1 to 20 with step size of 2
for i in range(0, 20, 5):
    print(i)
```

```
0
5
10
15
```

```
lst = ["satish", "srinu", "murali", "naveen", "bramha"]

#iterate over the list using index
#for index in range(len(lst)):
#     print(lst[index])
for ele in lst:
    print(ele)
```

```
satish
srinu
murali
naveen
bramha
```

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

break statement can be used to stop a for loop. In such case, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

```
numbers = [1, 2, 3]

#iterating over the list
for item in numbers:
    print(item)
else:
    print("no item left in the list")
```

```
1
2
3
no item left in the list
```

```
for item in numbers:
    print(item)
    if item % 2 == 0:
        break
else:
    print("no item left in the list")
```

```
1
2
```

Python Program to display all prime numbers within an interval

```
index1 = 20
index2 = 50

print("Prime numbers between {0} and {1} are :".format(index1, index2))

for num in range(index1, index2+1):      #default step size is 1
    if num > 1:
        isDivisible = False;
        for index in range(2, num):
            if num % index == 0:
                isDivisible = True;
        if not isDivisible:
            print(num);
```

```
Prime numbers between 20 and 50 are :
23
29
31
37
41
43
47
```

11. Control flow_ break and continue-

Python break and continue Statements

In Python, break and continue statements can alter the flow of a normal loop.

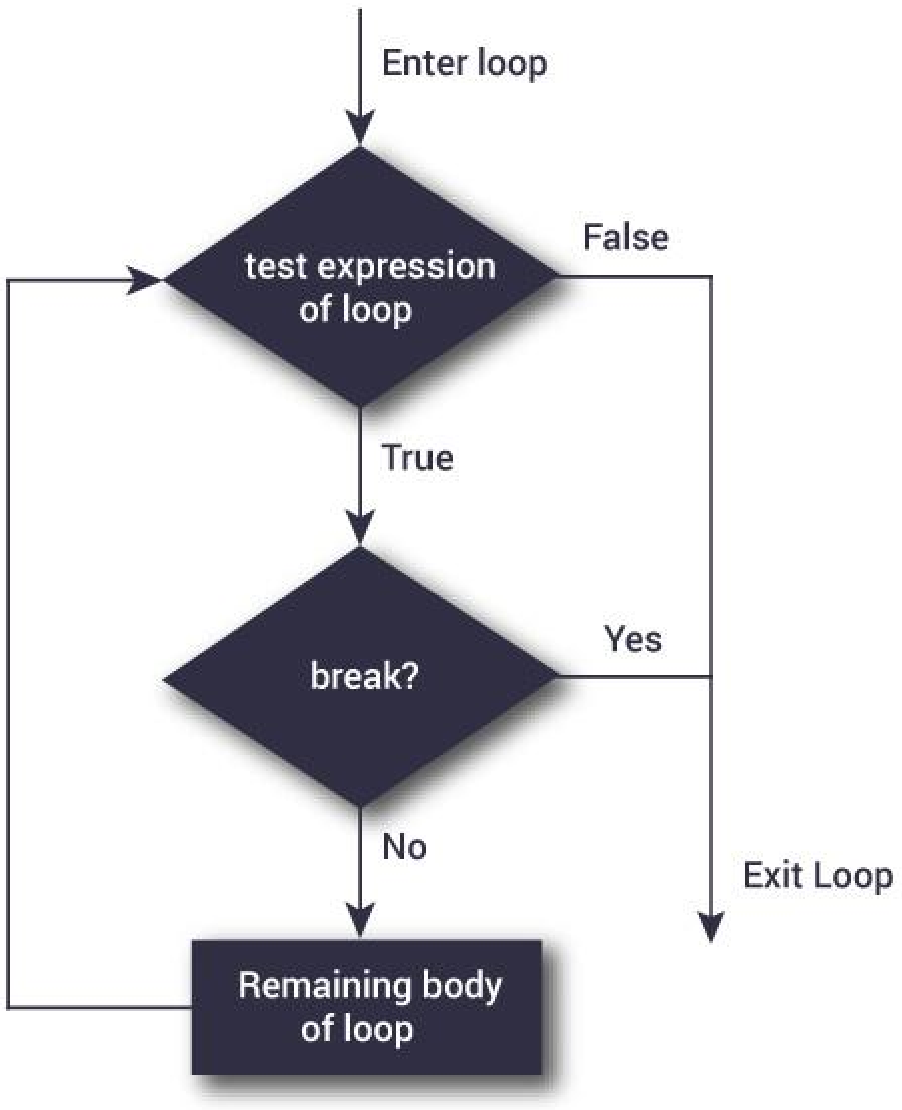
Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without cheking test expression.

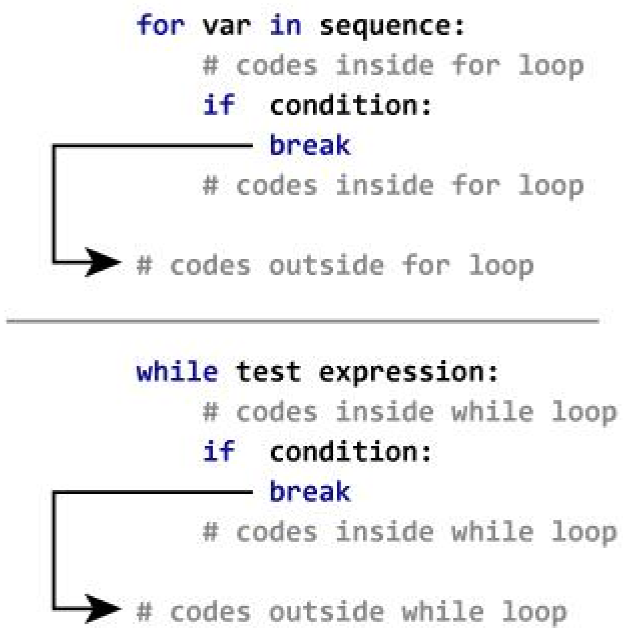
The break and continue statements are used in these cases.

Python break Statement

Syntax:

```
    break
```


Enter loop
test expression of loop
False
True
break?
Yes
No
Exit Loop
Remaining body of loop



```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Example

```
1
2
3
Outside of for loop
```

Python Program to check given number is Prime number or not (using break)

```
num = int(input("Enter a number: "))        #convert string to int


isDivisible = False;

i=2;
while i < num:
    if num % i == 0:
        isDivisible = True;
        print ("{} is divisible by {}".format(num,i) )
        break; # this line is the only addition.
    i += 1;

if isDivisible:
    print("{} is NOT a Prime number".format(num))
else:
    print("{} is a Prime number".format(num))
```

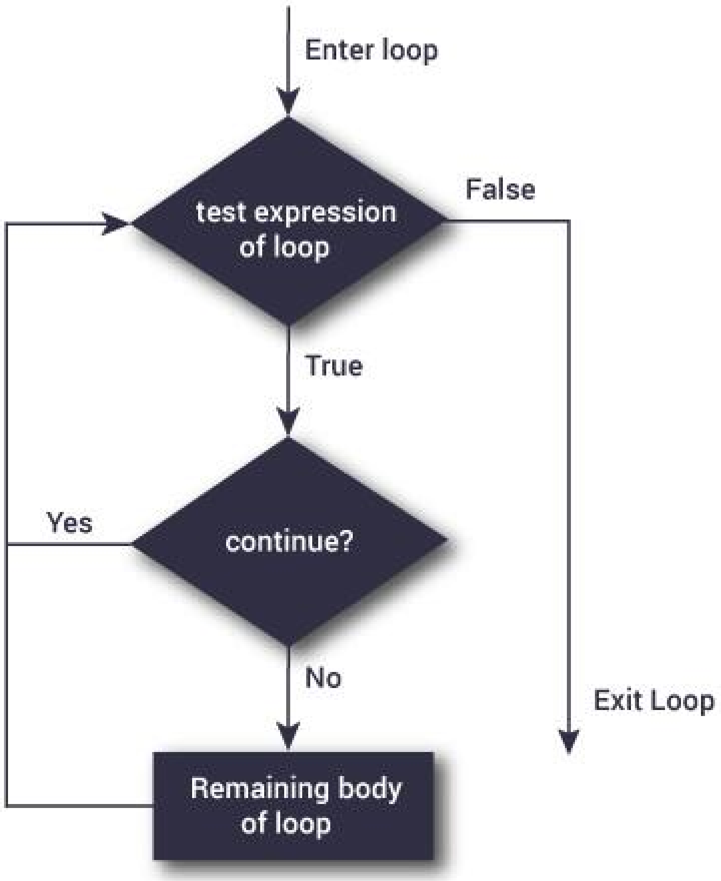
```
Enter a number: 16
```

```
16 is divisible by 2
16 is NOT a Prime number
```

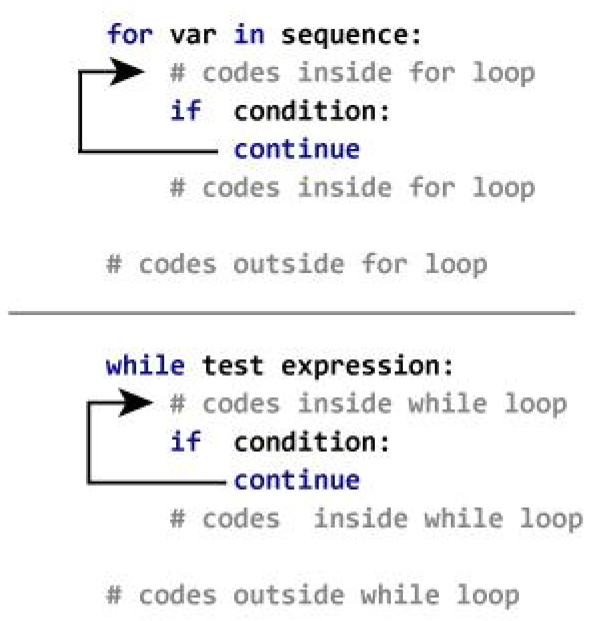
Python Continue Statement

Syntax:

```
    continue
```



Flow Chart


```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```
```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

Example

```
#print odd numbers present in a list
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num % 2 == 0:
        continue
    print(num)
else:
    print("else-block")
```

```
1
3
5
else-block
```

3. Python for Data Science_ Data Structures

1. Lists-

Data Structure:

A data structure is a collection of data elements (such as numbers or characters—or even other data structures) that is structured in some way, for example, by numbering the elements. The most basic data structure in Python is the "sequence".

-> List is one of the Sequence Data structure

-> Lists are collection of items (Strings, integers or even other lists)

-> Lists are enclosed in [ ]

-> Each item in the list has an assigned index value.

-> Each item in a list is separated by a comma

-> Lists are mutable, which means they can be changed.

List Creation

```
emptyList = []

lst = ['one', 'two', 'three', 'four'] # list of strings

lst2 = [1, 2, 3, 4] #list of integers

lst3 = [[1, 2], [3, 4]] # list of lists

lst4 = [1, 'ramu', 24, 1.24] # list of different datatypes

print(lst4)
```

```
[1, 'ramu', 24, 1.24]
```

List Length

```
lst = ['one', 'two', 'three', 'four']

#find length of a list
print(len(lst))
```

```
4
```

List Append

```
lst = ['one', 'two', 'three', 'four']

lst.append('five') # append will add the item at the end

print(lst)
```

```
['one', 'two', 'three', 'four', 'five']
```

List Insert

```
#syntax: lst.insert(x, y)

lst = ['one', 'two', 'four']

lst.insert(2, "three") # will add element y at location x

print(lst)
```

```
['one', 'two', 'three', 'four']
```

List Remove

```
#syntax: lst.remove(x)

lst = ['one', 'two', 'three', 'four', 'two']

lst.remove('two') #it will remove first occurence of 'two' in a given
list

print(lst)
```

```
['one', 'three', 'four', 'two']
```

List Append & Extend

```
lst = ['one', 'two', 'three', 'four']

lst2 = ['five', 'six']

#append
lst.append(lst2)

print(lst)
```

```
['one', 'two', 'three', 'four', ['five', 'six']]
```

```
lst = ['one', 'two', 'three', 'four']

lst2 = ['five', 'six']

#extend will join the list with list1

```

```
lst.extend(lst2)

print(lst)
```

```
['one', 'two', 'three', 'four', 'five', 'six']
```

List Delete

```
#del to remove item based on index position

lst = ['one', 'two', 'three', 'four', 'five']

del lst[1]
print(lst)

#or we can use pop() method
a = lst.pop(1)
print(a)

print(lst)
```

```
['one', 'three', 'four', 'five']
three
['one', 'four', 'five']
```

```
lst = ['one', 'two', 'three', 'four']

#remove an item from list
lst.remove('three')

print(lst)
```

```
['one', 'two', 'four']
```

List related keywords in Python

```
#keyword 'in' is used to test if an item is in a list
lst = ['one', 'two', 'three', 'four']

if 'two' in lst:
    print('AI')

#keyword 'not' can combined with 'in'
if 'six' not in lst:
    print('ML')
```

AI
ML

List Reverse

```
#reverse is reverses the entire list

lst = ['one', 'two', 'three', 'four']

lst.reverse()

print(lst)
```

```
['four', 'three', 'two', 'one']
```

List Sorting

The easiest way to sort a List is with the sorted(list) function.

That takes a list and returns a new list with those elements in sorted order.

The original list is not changed.

The sorted() optional argument reverse=True, e.g. sorted(list, reverse=True), makes it sort backwards.

```
#create a list with numbers
numbers = [3, 1, 6, 2, 8]

sorted_lst = sorted(numbers)


print("Sorted list :", sorted_lst)

#original list remain unchanged
print("Original list: ", numbers)
```

```
Sorted list : [1    2    3    6    8]
  Original list: [3    1    6    2    8]
```

```
#print a list in reverse sorted order
print("Reverse sorted list :", sorted(numbers, reverse=True))
```

```
#orginal list remain unchanged
print("Original list :",  numbers)
```

```
Reverse sorted list :      6    3    2   1]
[8
         Original list : [3    1    6    2    8]
```

```
lst = [1, 20, 5, 5, 4.2]

#sort the list and stored in itself
lst.sort()

# add element 'a' to the list to show an error

print("Sorted list: ", lst)
```

```
Sorted list: [1    4.2    5    5    20]
```

```
lst = [1, 20, 'b', 5, 'a']
print(lst.sort()) # sort list with element of different datatypes.
```

List Having Multiple References

```
lst = [1, 2, 3, 4, 5]
abc = lst
abc.append(6)

#print original list
print("Original list: ", lst)
```

```
Original list:      2    3    4    5    6]
[1
```

String Split to create a list

```
#let's take a string

s = "one,two,three,four,five"
slst = s.split(',')
print(slst)
```

```
['one', 'two', 'three', 'four', 'five']
```

```
s = "This is applied AI Course"
split_lst = s.split() # default split is white-character: space or tab
print(split_lst)
```

```
['This', 'is', 'applied', 'AI', 'Course']
```

List Indexing

Each item in the list has an assigned index value starting from 0.

Accessing elements in a list is called indexing.

```
lst = [1, 2, 3, 4]
print(lst[1]) #print second element

#print last element using negative index
print(lst[-2])
```

```
2
3
```

List Slicing

Accessing parts of segments is called slicing.

The key point to remember is that the :end value represents the first value that is not in the selected slice.

```
numbers = [10, 20, 30, 40, 50,60,70,80]

#print all numbers
print(numbers[:])

#print from index 0 to index 3
print(numbers[0:4])
```

```
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 30, 40]
```

```
print (numbers)
#print alternate elements in a list
print(numbers[::2])
```

```python
#print elemnts start from 0 through rest of the list
print(numbers[2::2])
```

```
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 30, 50, 70]
[30, 50, 70]
```

List extend using "+"

```python
lst1 = [1, 2, 3, 4]
lst2 = ['varma', 'naveen', 'murali', 'brahma']
new_lst = lst1 + lst2

print(new_lst)
```

```
[1, 2, 3, 4, 'varma', 'naveen', 'murali', 'brahma']
```

List Count

```python
numbers = [1, 2, 3, 1, 3, 4, 2, 5]

#frequency of 1 in a list
print(numbers.count(1))

#frequency of 3 in a list
print(numbers.count(3))
```

```
2
2
```

List Looping

```python
#loop through a list

lst = ['one', 'two', 'three', 'four']

for ele in lst:
    print(ele)
```

```
one
two
three
four
```

# List Comprehensions

List comprehensions provide a concise way to create lists.

Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
# without list comprehension
squares = []
for i in range(10):
    squares.append(i**2)   #list append
print(squares)
```

```
#using list comprehension
squares = [i**2 for i in range(10)]
print(squares)
```

```
#example

lst = [-10, -20, 10, 20, 50]

#create a new list with values doubled
new_lst = [i*2 for i in lst]
print(new_lst)

#filter the list to exclude negative numbers
new_lst = [i for i in lst if i >= 0]
print(new_lst)


#create a list of tuples like (number, square_of_number)
new_lst = [(i, i**2) for i in range(10)]
print(new_lst)
```

```
[-20, -40, 20, 40, 100]
[10, 20, 50]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81)]
```

Nested List Comprehensions

```
#let's suppose we have a matrix

matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]

#transpose of a matrix without list comprehension
transposed = []
for i in range(4):
    lst = []
    for row in matrix:
        lst.append(row[i])
    transposed.append(lst)

print(transposed)
```

```
0 1 2
1 5 9
2 6 10
3 7 11
4 8 12
```

```
#with list comprehension
transposed = [[row[i] for row in matrix] for i in range(4)]
print(transposed)
```

```
0 1 2
1 5 9
2 6 10
3 7 11
4 8 12
```

2

# Tuples
-> A tuple is similar to list

-> The diffence between the two is that we can't change the elements of tuple once it is assigned whereas in the list, elements can be changed
# Tuple creation
#empty tuple
t = ()

#tuple having integers
t = (1, 2, 3)
print(t)

```
#tuple with mixed datatypes
t = (1, 'raju', 28, 'abc')
print(t)

#nested tuple
t = (1, (2, 3, 4), [1, 'raju', 28, 'abc'])
print(t)
#only parenthesis is not enough
t = ('satish')
type(t)
#need a comma at the end
t = ('satish',)
type(t)
#parenthesis is optional
t = "satish",
print(type(t))

print(t)
# Accessing Elements in Tuple
t = ('satish', 'murali', 'naveen', 'srinu', 'brahma')

print(t[1])
#negative index
print(t[-1]) #print last element in a tuple
#nested tuple
t = ('ABC', ('satish', 'naveen', 'srinu'))

print(t[1])
print(t[1][2])
#Slicing
t = (1, 2, 3, 4, 5, 6)

print(t[1:4])

#print elements from starting to 2nd last elements
print(t[:-2])

#print elements from starting to end
print(t[:])
# Changing a Tuple
#unlike lists, tuples are immutable
#This means that elements of a tuple cannot be changed once it has been assigned. But, if
the element is itself a mutable datatype like list, its nested items can be changed.
#creating tuple
t = (1, 2, 3, 4, [5, 6, 7])

t[2] = 'x' #will get TypeError
t[4][1] = 'satish'
```

```
print(t)
#concatinating tuples

t = (1, 2, 3) + (4, 5, 6)
print(t)
#repeat the elements in a tuple for a given number of times using the * operator.
t = (('satish', ) * 4)
print(t)
# Tuple Deletion
#we cannot change the elements in a tuple.
# That also means we cannot delete or remove items from a tuple.

#delete entire tuple using del keyword
t = (1, 2, 3, 4, 5, 6)

#delete entire tuple
del t



# Tuple Count
t = (1, 2, 3, 1, 3, 3, 4, 1)

#get the frequency of particular element appears in a tuple
t.count(1)
# Tuple Index
t = (1, 2, 3, 1, 3, 3, 4, 1)

print(t.index(3)) #return index of the first element is equal to 3

#print index of the 1



# Tuple Memebership
#test if an item exists in a tuple or not, using the keyword in.
t = (1, 2, 3, 4, 5, 6)

print(1 in t)
print(7 in t)
# Built in Functions
# Tuple Length
t = (1, 2, 3, 4, 5, 6)
print(len(t))
# Tuple Sort
t = (4, 5, 1, 2, 3)

new_t = sorted(t)
print(new_t) #Take elements in the tuple and return a new sorted list
        #(does not sort the tuple itself).
```

```
#get the largest element in a tuple
t = (2, 5, 1, 6, 9)

print(max(t))
#get the smallest element in a tuple
print(min(t))
#get sum of elments in the tuple
print(sum(t))
```

3

# Tuples
-> A tuple is similar to list

-> The diffence between the two is that we can't change the elements of tuple once it is assigned whereas in the list, elements can be changed

```
# Tuple creation
#empty tuple
t = ()

#tuple having integers
t = (1, 2, 3)
print(t)

#tuple with mixed datatypes
t = (1, 'raju', 28, 'abc')
print(t)

#nested tuple
t = (1, (2, 3, 4), [1, 'raju', 28, 'abc'])
print(t)
#only parenthesis is not enough
t = ('satish')
type(t)
#need a comma at the end
t = ('satish',)
type(t)
#parenthesis is optional
t = "satish",
print(type(t))

print(t)
# Accessing Elements in Tuple
t = ('satish', 'murali', 'naveen', 'srinu', 'brahma')

print(t[1])
#negative index
```

```
print(t[-1]) #print last element in a tuple
#nested tuple
t = ('ABC', ('satish', 'naveen', 'srinu'))

print(t[1])
print(t[1][2])
#Slicing
t = (1, 2, 3, 4, 5, 6)

print(t[1:4])

#print elements from starting to 2nd last elements
print(t[:-2])

#print elements from starting to end
print(t[:])
# Changing a Tuple
#unlike lists, tuples are immutable
#This means that elements of a tuple cannot be changed once it has been assigned. But, if
the element is itself a mutable datatype like list, its nested items can be changed.
#creating tuple
t = (1, 2, 3, 4, [5, 6, 7])

t[2] = 'x' #will get TypeError
t[4][1] = 'satish'
print(t)
#concatinating tuples

t = (1, 2, 3) + (4, 5, 6)
print(t)
#repeat the elements in a tuple for a given number of times using the * operator.
t = (('satish', ) * 4)
print(t)
# Tuple Deletion
#we cannot change the elements in a tuple.
# That also means we cannot delete or remove items from a tuple.

#delete entire tuple using del keyword
t = (1, 2, 3, 4, 5, 6)

#delete entire tuple
del t



# Tuple Count
t = (1, 2, 3, 1, 3, 3, 4, 1)

#get the frequency of particular element appears in a tuple
```

```
t.count(1)
# Tuple Index
t = (1, 2, 3, 1, 3, 3, 4, 1)

print(t.index(3)) #return index of the first element is equal to 3

#print index of the 1


# Tuple Memebership
#test if an item exists in a tuple or not, using the keyword in.
t = (1, 2, 3, 4, 5, 6)

print(1 in t)
print(7 in t)
# Built in Functions
# Tuple Length
t = (1, 2, 3, 4, 5, 6)
print(len(t))
# Tuple Sort
t = (4, 5, 1, 2, 3)

new_t = sorted(t)
print(new_t) #Take elements in the tuple and return a new sorted list
           #(does not sort the tuple itself).
#get the largest element in a tuple
t = (2, 5, 1, 6, 9)

print(max(t))
#get the smallest element in a tuple
print(min(t))
#get sum of elments in the tuple
print(sum(t))
```

4

```
# Sets
```

-> A set is an unordered collection of items. Every element is unique (no duplicates).

-> The set itself is mutable. We can add or remove items from it.

-> Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

```
# Set Creation
#set of integers
s = {1, 2, 3}
print(s)
```

```
#print type of s
print(type(s))
#set doesn't allow duplicates. They store only one instance.
s = {1, 2, 3, 1, 4}
print(s)
#we can make set from a list
s = set([1, 2, 3, 1])
print(s)
#initialize a set  with set() method
s = set()

print(type(s))
# Add element to a Set
#we can add single element using add() method and
#add multiple elements using update() method
s = {1, 3}

#set object doesn't support indexing
print(s[1]) #will get TypeError
#add element
s.add(2)
print(s)
#add multiple elements
s.update([5, 6, 1])
print(s)
#add list and set
s.update([8, 9], {10, 2, 3})
print(s)
# Remove elements from a Set
#A particular item can be removed from set using methods,
#discard() and remove().

s = {1, 2, 3, 5, 4}
print(s)

s.discard(4)    #4 is removed from set s

print(s)
#remove an element
s.remove(2)

print(s)
#remove an element not present in a set s
s.remove(7) # will get KeyError
#discard an element not present in a set s
s.discard(7)
print(s)
```

```
#we can remove item using pop() method

s = {1, 2, 3, 5, 4}

s.pop() #remove random element

print(s)
s.pop()
print(s)
s = {1, 5, 2, 3, 6}

s.clear()   #remove all items in set using clear() method

print(s)
# Python Set Operations
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

#union of 2 sets using | operator

print(set1 | set2)
#another way of getting union of 2 sets
print(set1.union(set2))
#intersection of 2 sets using & operator
print(set1 & set2)
#use intersection function
print(set1.intersection(set2))
#set Difference: set of elements that are only in set1 but not in set2

print(set1 - set2)
#use differnce function
print(set1.difference(set2))
"""symmetric difference: set of elements in both set1 and set2
#except those that are common in both."""

#use ^ operator

print(set1^set2)
#use symmetric_difference function
print(set1.symmetric_difference(set2))
#find issubset()
x = {"a","b","c","d","e"}
y = {"c","d"}

print("set 'x' is subset of 'y' ?", x.issubset(y)) #check x is subset of y

#check y is subset of x
print("set 'y' is subset of 'x' ?", y.issubset(x))
```

# Frozen Sets
Frozen sets has the characteristics of sets, but we can't be changed once it's assigned.
While tuple are immutable lists, frozen sets are immutable sets
Frozensets can be created using the function frozenset()
Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.
This datatype supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have method that add or remove elements.

```
set1 = frozenset([1, 2, 3, 4])
set2 = frozenset([3, 4, 5, 6])

#try to add element into set1 gives an error
set1.add(5)
print(set1[1]) # frozen set doesn't support indexing
print(set1 | set2) #union of 2 sets
#intersection of two sets
print(set1 & set2)

#or
print(set1.intersection(set2))
#symmetric difference
print(set1 ^ set2)

#or
print(set1.symmetric_difference(set2))
```

5

# Dictionary
Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

# Dict Creation

```
#empty dictionary
my_dict = {}

#dictionary with integer keys
my_dict = {1: 'abc', 2: 'xyz'}
print(my_dict)

#dictionary with mixed keys
my_dict = {'name': 'satish', 1: ['abc', 'xyz']}
print(my_dict)


#create empty dictionary using dict()
my_dict = dict()
my_dict = dict([(1, 'abc'), (2, 'xyz')])    #create a dict with list of tuples
print(my_dict)
# Dict Access
my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#get name
print(my_dict['name'])
#if key is not present it gives KeyError
print(my_dict['degree'])
#another way of accessing key
print(my_dict.get('address'))
#if key is not present it will give None using get method
print(my_dict.get('degree'))
# Dict Add or Modify  Elements
my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#update name
my_dict['name'] = 'raju'

print(my_dict)
#add new key
my_dict['degree'] = 'M.Tech'

print(my_dict)
# Dict Delete or Remove Element
#create a dictionary
my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#remove a particular item
print(my_dict.pop('age'))

print(my_dict)
my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#remove an arbitarty key
my_dict.popitem()

print(my_dict)
squares = {2: 4, 3: 9, 4: 16, 5: 25}

#delete particular key
del squares[2]

print(squares)
#remove all items
squares.clear()
```

```
print(squares)
squares = {2: 4, 3: 9, 4: 16, 5: 25}

#delete dictionary itself
del squares

print(squares) #NameError because dict is deleted
# Dictionary Methods
squares = {2: 4, 3: 9, 4: 16, 5: 25}

my_dict = squares.copy()
print(my_dict)
#fromkeys[seq[, v]] -> Return a new dictionary with keys from seq and value equal to v
(defaults to None).
subjects = {}.fromkeys(['Math', 'English', 'Hindi'], 0)
print(subjects)
subjects = {2:4, 3:9, 4:16, 5:25}
print(subjects.items()) #return a new view of the dictionary items (key, value)
subjects = {2:4, 3:9, 4:16, 5:25}
print(subjects.keys()) #return a new view of the dictionary keys
subjects = {2:4, 3:9, 4:16, 5:25}
print(subjects.values()) #return a new view of the dictionary values
#get list of all available methods and attributes of dictionary
d = {}
print(dir(d))
# Dict Comprehension
#Dict comprehensions are just like list comprehensions but for dictionaries

d = {'a': 1, 'b': 2, 'c': 3}
for pair in d.items():
   print(pair)
#Creating a new dictionary with only pairs where the value is larger than 2
d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
new_dict = {k:v for k, v in d.items() if v > 2}
print(new_dict)
#We can also perform operations on the key value pairs
d = {'a':1,'b':2,'c':3,'d':4,'e':5}
d = {k + 'c':v * 2 for k, v in d.items() if v > 2}
print(d)
```

6

```
# Strings
   A string is a sequence of characters.
```

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, string is a sequence of Unicode character.
For more details about unicode

https://docs.python.org/3.3/howto/unicode.html
# How to create a string?
Strings can be created by enclosing characters inside a single quote or double quotes.

Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
myString = 'Hello'

print(myString)


myString = "Hello"
print(myString)


myString = '''Hello'''
print(myString)
```
# How to access characters in a string?
We can access individual characters using indexing and a range of characters using slicing.

Index starts from 0.

Trying to access a character out of index range will raise an IndexError.

The index must be an integer. We can't use float or other types, this will result into TypeError.

Python allows negative indexing for its sequences.
```
myString = "Hello"

#print first Character
print(myString[0])

#print last character using negative indexing
print(myString[-1])

#slicing 2nd to 5th character
print(myString[2:5])
```

If we try to access index out of the range or use decimal number, we will get errors.

```
print(myString[15])
print(myString[1.5])
# How to change or delete a string ?
```

Strings are immutable. This means that elements of a string cannot be changed once it has been assigned.

We can simply reassign different strings to the same name.

```
myString = "Hello"
myString[4] = 's' # strings are immutable
```

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the keyword del.

```
del myString # delete complete string
print(myString)
# String Operations
# Concatenation
```

Joining of two or more strings into a single one is called concatenation.

The + operator does this in Python. Simply writing two string literals together also concatenates them.

The * operator can be used to repeat the string for a given number of times.

```
s1 = "Hello "
s2 = "Satish"

#concatenation of 2 strings
print(s1 + s2)

#repeat string n times
print(s1 * 3)
# Iterating Through String
count = 0
for l in "Hello World":
    if l == 'o':
        count += 1
print(count, ' letters found')
# String Membership Test
print('l' in 'Hello World') #in operator to test membership
print('or' in 'Hello World')
# String Methods
```

Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc

```
"Hello".lower()
"Hello".upper()
"This will split all words in a list".split()
' '.join(['This', 'will', 'split', 'all', 'words', 'in', 'a', 'list'])
"Good Morning".find("Mo")
s1 = "Bad morning"
```

```
s2 = s1.replace("Bad", "Good")

print(s1)
print(s2)
# Python Program to Check where a String is Palindrome or not ?
myStr = "Madam"

#convert entire string to either lower or upper
myStr = myStr.lower()

#reverse string
revStr = reversed(myStr)


#check if the string is equal to its reverse
if list(myStr) == list(revStr):
   print("Given String is palindrome")
else:
   print("Given String is not palindrome")

# Python Program to Sort Words in Alphabetic Order?
myStr = "python Program to Sort words in Alphabetic Order"

#breakdown the string into list of words
words = myStr.split()

#sort the list
words.sort()

#print Sorted words are
for word in words:
   print(word)
```

4. Python for Data Science_ Functions

1. Introduction-

# Python Functions
Function is a group of related statements that perform a specific task.
Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

It avoids repetition and makes code reusable.
# Syntax:
```
   def function_name(parameters):
        """
        Doc String
        """

        Statement(s)
```
1. keyword "def" marks the start of function header

2. Parameters (arguments) through which we pass values to a function. These are optional

3. A colon(:) to mark the end of funciton header

4. Doc string describe what the function does. This is optional

5. "return" statement to return a value from the function. This is optional

```
# Example:
def print_name(name):
    """
    This function prints the name
    """
    print("Hello " + str(name))
```

# Function Call
Once we have defined a function, we can call it from anywhere
print_name('satish')

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

Although optional, documentation is a good programming practice, always document your code
Doc string will be written in triple quotes so that docstring can extend up to multiple lines
print(print_name.__doc__) # print doc string of the function

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

```
    return [expression]
```
-> return statement can contain an expression which gets evaluated and the value is returned.

-> if there is no expression in the statement or the return statement itself is not present inside a function, then the function will return None Object

```
def get_sum(lst):
    """
    This function returns the sum of all the elements in a list
    """
    #initialize sum
    _sum = 0

    #iterating over the list
    for num in lst:
        _sum += num
    return _sum

s = get_sum([1, 2, 3, 4])
print(s)

#print doc string
print(get_sum.__doc__)

# How Function works in Python?
![title](function_works.jpg)
# Scope and Life Time of Variables
-> Scope of a variable is the portion of a program where the variable is recognized
-> variables defined inside a function is not visible from outside. Hence, they have a local
scope.
-> Lifetime of a variable is the period throughout which the variable exits in the memory.

-> The lifetime of variables inside a function is as long as the function executes.
-> Variables are destroyed once we return from the function.
# Example:
global_var = "This is global variable"

def test_life_time():
    """
    This function test the life time of a variables
    """
    local_var = "This is local variable"
    print(local_var)       #print local variable local_var

    print(global_var)      #print global variable global_var



#calling function
test_life_time()

#print global variable global_var
print(global_var)

#print local variable local_var
print(local_var)
```

```
# Python program to print Highest Common Factor (HCF) of two numbers
def computeHCF(a, b):
    """
    Computing HCF of two numbers
    """
    smaller = b if a > b else a  #consice way of writing if else statement

    hcf = 1
    for i in range(1, smaller+1):
        if (a % i == 0) and (b % i == 0):
            hcf = i
    return hcf

num1 = 6
num2 = 36

print("H.C.F of {0} and {1} is: {2}".format(num1, num2, computeHCF(num1, num2)))
```

2. Types of functions-

```
# Types Of Functions
1. Built-in Functions

2. User-defined Functions
# Built-in Functions
# 1. abs()
# find the absolute value

num = -100

print(abs(num))

# 2. all()
#return value of all() function

True: if all elements in an iterable are true

False: if any element in an iterable is false
lst = [1, 2, 3, 4]
print(all(lst))

lst = (0, 2, 3, 4)    # 0 present in list
print(all(lst))

lst = []              #empty list always true
print(all(lst))
```

```
lst = [False, 1, 2]   #False present in a list so all(lst) is False
print(all(lst))
```

```
# dir()
```
The dir() tries to return a list of valid attributes of the object.

If the object has __dir__() method, the method will be called and must return the list of attributes.

If the object doesn't have __dir()__ method, this method tries to find information from the __dict__ attribute (if defined), and from type object. In this case, the list returned from dir() may not be complete.

```
numbers = [1, 2, 3]

print(dir(numbers))
```

```
# divmod()
```
The divmod() method takes two numbers and returns a pair of numbers (a tuple) consisting of their quotient and remainder.

```
print(divmod(9, 2)) #print quotient and remainder as a tuple

#try with other number
```

```
# enumerate()
```
The enumerate() method adds counter to an iterable and returns it
syntax: enumerate(iterable, start=0)

```
numbers = [10, 20, 30, 40]

for index, num in enumerate(numbers,10):
    print("index {0} has value {1}".format(index, num))
```

```
# filter()
```
The filter() method constructs an iterator from elements of an iterable for which a function returns true.
syntax: filter(function, iterable)

```
def find_positive_number(num):
    """
    This function returns the positive number if num is positive
    """
    if num > 0:
        return num

number_list = range(-10, 10) #create a list with numbers from -10 to 10
print(list(number_list))

positive_num_lst = list(filter(find_positive_number, number_list))
```

```
print(positive_num_lst)

# isinstance()
```
The isinstance() function checks if the object (first argument) is an instance or subclass of classinfo class (second argument).
syntax: isinstance(object, classinfo)
```
lst = [1, 2, 3, 4]
print(isinstance(lst, list))

#try with other datatypes tuple, set
t = (1,2,3,4)
print(isinstance(t, list))
# map()
```
Map applies a function to all the items in an input_list.
syntax: map(function_to_apply, list_of_inputs)
```
numbers = [1, 2, 3, 4]

#normal method of computing num^2 for each element in the list.
squared = []
for num in numbers:
    squared.append(num ** 2)

print(squared)

numbers = [1, 2, 3, 4]

def powerOfTwo(num):
    return num ** 2

#using map() function
squared = list(map(powerOfTwo, numbers))
print(squared)

# reduce()
```
reduce() function is for performing some computation on a list and returning the result.

It applies a rolling computation to sequential pairs of values in a list.
```
#product of elemnts in a list
product = 1
lst = [1, 2, 3, 4]

# traditional program without reduce()
for num in lst:
    product *= num
print(product)

#with reduce()
from functools import reduce # in Python 3.
```

```
def multiply(x,y):
   return x*y;

product = reduce(multiply, lst)
print(product)
```

# 2. User-defined Functions
Functions that we define ourselves to do certain specific task are referred as user-defined functions
If we use functions written by others in the form of library, it can be termed as library functions.
# Advantages
1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.

2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.

3. Programmars working on large project can divide the workload by making different functions.
# Example:
```
def product_numbers(a, b):
   """
   this function returns the product of two numbers
   """
   product = a * b
   return product

num1 = 10
num2 = 20
print "product of {0} and {1} is {2} ".format(num1, num2, product_numbers(num1, num2))
# Python program to make a simple calculator that can add, subtract, multiply and division
def add(a, b):
   """
   This function adds two numbers
   """
   return a + b

def multiply(a, b):
   """
   This function multiply two numbers
   """
   return a * b

def subtract(a, b):
   """
   This function subtract two numbers
```

```
    """
    return a - b

def division(a, b):
    """
    This function divides two numbers
    """
    return a / b

print("Select Option")
print("1. Addition")
print ("2. Subtraction")
print ("3. Multiplication")
print ("4. Division")

#take input from user
choice = int(input("Enter choice 1/2/3/4"))

num1 = float(input("Enter first number:"))
num2 = float(input("Enter second number:"))
if choice == 1:
    print("Addition of {0} and {1} is {2}".format(num1, num2, add(num1, num2)))
elif choice == 2:
    print("Subtraction of {0} and {1} is {2}".format(num1, num2, subtract(num1, num2)))
elif choice == 3:
    print("Multiplication of {0} and {1} is {2}".format(num1, num2, multiply(num1, num2)))
elif choice == 4:
    print("Division of {0} and {1} is {2}".format(num1, num2, division(num1, num2)))
else:
    print("Invalid Choice")
```

3. Function arguments-

```
# Function Arguments

def greet(name, msg):
    """
    This function greets to person with the provided message
    """
    print("Hello {0} , {1}".format(name, msg))

#call the function with arguments
greet("satish", "Good Morning")

#suppose if we pass one argument

greet("satish") #will get an error
```

```
# Different Forms of Arguments
# 1. Default Arguments
```
We can provide a default value to an argument by using the assignment operator (=).
```
def greet(name, msg="Good Morning"):
    """
    This function greets to person with the provided message
    if message is not provided, it defaults to "Good Morning"
    """
    print("Hello {0} , {1}".format(name, msg))

greet("satish", "Good Night")

#with out msg argument
greet("satish")
```

Once we have a default argument, all the arguments to its right must also have default values.
```
def greet(msg="Good Morning", name)

#will get a SyntaxError : non-default argument follows default argument
# 2. Keyword Arguments
```
kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function
```
# Example:
def greet(**kwargs):
    """
    This function greets to person with the provided message
    """
    if kwargs:
        print("Hello {0} , {1}".format(kwargs['name'], kwargs['msg']))
greet(name="satish", msg="Good Morning")

# 3. Arbitary Arguments
```
Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.
```
# Example:
def greet(*names):
    """
    This function greets all persons in the names tuple
    """
    print(names)

    for name in names:
        print("Hello,  {0} ".format(name))

greet("satish", "murali", "naveen", "srikanth")
```

4. Recursive functions-

# Recurison
We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.
# Example:

```
#python program to print factorial of a number using recurion

def factorial(num):
    """
    This is a recursive function to find the factorial of a given number
    """
    return 1 if num == 1 else (num * factorial(num-1))

num = 5
print ("Factorial of {0} is {1}".format(num, factorial(num)))
```

# Advantages
1. Recursive functions make the code look clean and elegant.

2. A complex task can be broken down into simpler sub-problems using recursion.

3. Sequence generation is easier with recursion than using some nested iteration.

# Disadvantages
1. Sometimes the logic behind recursion is hard to follow through.

2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

3. Recursive functions are hard to debug.

```
# Python program to display the fibonacci sequence up to n-th term using recursive function
def fibonacci(num):
    """
    Recursive function to print fibonacci sequence
    """
    return num if num <= 1 else fibonacci(num-1) + fibonacci(num-2)

nterms = 10
print("Fibonacci sequence")
for num in range(nterms):
    print(fibonacci(num))
```

5. Lambda functions-

# Anonymous / Lambda Function
In Python, anonymous function is a function that is defined without a name.

While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Lambda functions are used extensively along with built-in functions like filter(), map()
syntax:

```
    lambda arguments: expression
# Example:
double = lambda x: x*2

print(double(5))

def double(x):
    return x * 2

print(double(5))

#Example use with filter()
lst = [1, 2, 3, 4, 5]
even_lst = list(filter(lambda x: (x%2 == 0), lst))
print(even_lst)

#Example use with map()
lst = [1, 2, 3, 4, 5]
new_lst = list(map(lambda x: x ** 2, lst))
print(new_lst)

#Example use with reduce()
from functools import reduce

lst = [1, 2, 3, 4, 5]
product_lst = reduce(lambda x, y: x*y, lst)
print(product_lst)
```

6. Modules-

# Modules
Modules refer to a file containing Python statements and definitions.

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

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.
# How to import a module?
We use the import keyword to do this.
```
import example #imported example module
```

Using the module name we can access the function using dot (.) operation.
```
example.add(10, 20)
```

Python has a lot of standard modules available.

https://docs.python.org/3/py-modindex.html
```
# Examples:
import math
print(math.pi)

import datetime
datetime.datetime.now()

# import with renaming
import math as m
print(m.pi)
```

# from...import statement
We can import specific names form a module without importing the module as a whole.
```
from datetime import datetime
datetime.now()

# import all names
from math import *
print("Value of PI is " + str(pi))
```

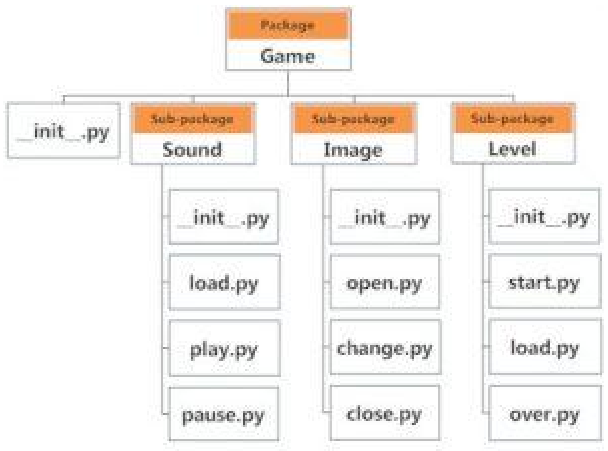
# dir() built in function
We can use the dir() function to find out names that are defined inside a module.
```
dir(example)

print(example.add.__doc__)
```

7. Packages-

# Package
Packages are a way of structuring Python’s module namespace by using “dotted module names”.

A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.



# importing module from a package
We can import modules from packages using the dot (.) operator.
# import Gate.Image.open

8. File Handling-

# FILE I/O
File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

File operation:

1. Open a file

2. Read or write (perform operation)

3. Close the file
# Opening a File
Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.
f = open('example.txt') #open file in current direcotry
We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.
# Python File Modes
'r' Open a file for reading. (default)

'w' Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.

'x' Open a file for exclusive creation. If the file already exists, the operation fails.

'a' Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.

't' Open in text mode. (default)

'b' Open in binary mode.

'+' Open a file for updating (reading and writing)

```
f = open('example.txt') #equivalent to 'r'
f = open('example.txt', 'r')

f = open('test.txt', 'w')
```

The default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux. So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

# Closing a File

Closing a file will free up the resources that were tied with the file and is done using the close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

```
f = open('example.txt')
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

```
try:
   f = open("example.txt")
   # perform file operations

finally:
   f.close()
```

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

The best way to do this is using the with statement. This ensures that the file is closed when the block inside with is exited.

We don't need to explicitly call the close() method. It is done internally.

```
with open("example.txt",encoding = 'utf-8') as f:
   #perform file operations
```
# Writing to a File
In order to write into a file we need to open it in **write 'w', append 'a' or exclusive creation 'x' mode**.

We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using **write()** method. This method returns the number of characters written to the file.
```
f = open("test.txt", "w")
f.write("This is a First File\n")
f.write("Contains two lines\n")
f.close()
```

This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten.

# Reading From a File
There are various methods available for this purpose. We can use the read(size) method to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file.
```
f = open("test.txt", "r")
f.read()
f = open("test.txt", "r")
f.read(4)
#f = open("test.txt","r")
f.read(10)
```
We can change our current file cursor (position) using the seek() method.

Similarly, the **tell()** method returns our current position (in number of bytes).
```
f.tell()
f.seek(0) #bring the file cursor to initial position
print(f.read()) #read the entire file
```
We can read a file line-by-line using a for loop. This is both efficient and fast.
```
f.seek(0)
for line in f:
   print(line)
```
Alternately, we can use readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.
```
f = open("test.txt", "r")
f.readline()
f.readline()
```

```
f.readline()
```
The **readlines()** method returns a list of remaining lines of the entire file. All these reading method return empty values when end of file (EOF) is reached.
```
f.seek(0)
f.readlines()
```
# Renaming And Deleting Files In Python.
While you were using the **read/write** functions, you may also need to **rename/delete** a file in Python. So, there comes a **os** module in Python which brings the support of file **rename/delete** operations.

So, to continue, first of all, you should import the **os** module in your Python script.
```
import os

#Rename a file from test.txt to sample.txt
os.rename("test.txt", "sample.txt")
f = open("sample.txt", "r")
f.readline()
#Delete a file sample.txt
os.remove("sample.txt")
f = open("sample.txt", "r")
f.readline()
```
# Python Directory and File Management
If there are a large number of files to handle in your Python program, you can arrange your code within different directories to make things more manageable.

A directory or folder is a collection of files and sub directories. Python has the os module, which provides us with many useful methods to work with directories (and files as well).
Get current Directory
We can get the present working directory using the getcwd() method.

This method returns the current working directory in the form of a string.
```
import os
os.getcwd()
```
Changing Directory
We can change the current working directory using the chdir() method.

The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements.
```
os.chdir("/Users/varma/")
os.getcwd()
```
List Directories and Files
All files and sub directories inside a directory can be known using the listdir() method.
```
os.listdir(os.getcwd())
```
Making New Directory
We can make a new directory using the mkdir() method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

```
os.mkdir('test')
```
However, note that rmdir() method can only remove empty directories.

In order to remove a non-empty directory we can use the rmtree() method inside the shutil module.
```
os.rmdir('test')
import shutil

os.mkdir('test')
os.chdir('./test')
f = open("testfile.txt",'w')
f.write("Hello World")
os.chdir("../")
os.rmdir('test')

# remove an non-empty directory
shutil.rmtree('test')
os.getcwd()
```

9. Exception Handling-

# Python Errors and Built-in-Exceptions
When writing a program, we, more often than not, will encounter errors.

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.
```
if a < 3
```
Errors can also occur at runtime and these are called exceptions.

They occur, for example, when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module we try to import is not found (ImportError) etc.
Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.
```
1 / 0
open('test.txt')
```
# Python Built-in Exceptions
```
dir(__builtins__)
```
# Python Exception Handling - Try, Except and Finally
Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.

When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.

If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

# Catching Exceptions in Python

In Python, exceptions can be handled using a try statement.

A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

```
# import module sys to get the type of exception
import sys

lst = ['b', 0, 2]

for entry in lst:
    try:
        print("The entry is", entry)
        r = 1 / int(entry)
    except:
        print("Oops!", sys.exc_info()[0],"occured.")
        print("Next entry.")
        print("****************************")
print("The reciprocal of", entry, "is", r)
```

# Catching Specific Exceptions in Python

In the above example, we did not mention any exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch.

A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.

```
import sys

lst = ['b', 0, 2]

for entry in lst:
    try:
        print("****************************")
        print("The entry is", entry)
        r = 1 / int(entry)
    except(ValueError):
        print("This is a ValueError.")
    except(ZeroDivisionError):
        print("This is a ZeroError.")
    except:
        print("Some other error")
print("The reciprocal of", entry, "is", r)
```

# Raising Exceptions
In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise.

We can also optionally pass in value to the exception to clarify why that exception was raised.

```
raise KeyboardInterrupt
raise MemoryError("This is memory Error")
try:
    num = int(input("Enter a positive integer:"))
    if num <= 0:
        raise ValueError("Error:Entered negative number")
except ValueError as e:
    print(e)
```

# try ... finally
The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.

```
try:
    f = open('sample.txt')
    #perform file operations

finally:
    f.close()
```

10. Debugging Python-

# Debugging
pdb implements an interactive debugging environment for Python programs. It includes features to let you pause your program, look at the values of variables, and watch program execution step-by-step, so you can understand what your program actually does and find bugs in the logic.
# Starting the Debugger
From the Command Line

```
def seq(n):
    for i in range(n):
        print(i)
    return

seq(5)
```

From Within Your Program

```
import pdb

#interactive debugging
def seq(n):
    for i in range(n):
        pdb.set_trace() # breakpoint
        print(i)
    return
```

```
seq(5)


# c : continue
# q: quit
# h: help
# list
# p: print
# p locals()
# p globals()
```

# Debugger Commands
1. h(elp) [command]

Without argument, print the list of available commands. With a command as argument, print help about that command. help pdb displays the full documentation (the docstring of the pdb module). Since the command argument must be an identifier, help exec must be entered to get help on the ! command.
2. w(here)

Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.
3. d(own) [count]

Move the current frame count (default one) levels down in the stack trace (to a newer frame).
4.c(ont(inue))

Continue execution, only stop when a breakpoint is encountered.
5. q(uit)

Quit from the debugger. The program being executed is aborted.
Termial/Command prompt based debugging