Date .....

PYTHON PROGRAMMING

* Introducing Python -

→ Python is a high-level, general-purpose, and very popular programming language.

→ It is being used in web development, Machine learning applications, etc.

→ Python programs are generally smaller than the other programming languages.

→ Programmers have to type relatively less and the indentation requirement of the language, makes them readable at the time.

→ Very Simple Syntax. & Easy to learn.

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

→ General purpose language (Simple yet very powerful language)

- → Console Application & Scripts
- → Desktop Application
- → Web Application
- → Game Development
- → Machine Learning, Deep learning, AI, Big data etc

→ Multiparadigm Support

- → Procedural Style Programming like C.
- → OOPs like Java.
- → Functional programming like list.

Date .....

→ Portable or Platform Independent

- Programs are typically first compiled into an intermediate code, then the code is run by the interpreter.

→ Dynamically Typed

```
x = 10
y = "geeks"
x = "python"
```

This feature is different from C++ & Java. They are statically Typed.

In python we don't have to specify the variable name.

But the disadvantage is in this the type is assigned in the run time of the program, so there are more chances of run time error.

→ Automatic Garbage Collection

It means when you dynamically allocate memory so you don't have to worry to realising this memory.

→ Popular Application Built in Python

Youtube, Netflix, Quora, Instagram, DropBox

⇒ print () in Python -

A function is a set of instructions that take some input from us, do some work on those

Date .....

parameters and produce some output.

Example - print ("Hello")
print ("Welcome", "to", "GfG")
print ()
print ("Hope you are enjoying Python")

Output - Hello
Welcome to GfG

Hope you are enjoying Python

→ ~~end~~ end and sep parameters in print()

Eg - print ("Welcome", end = " ")
print (" to GfG")

Output - Welcome to GfG

End parameter creates space and it tells what should be printing after its printing.
And in sep parameter when we have multiple input parameters we use sep to separate them.
By default they are separated by space.

Eg - print ("10", "04", "2023", sep = "-")

Output - 10-04-2023

Spiral

Date .....

⇒ Variables in Python -

Variable is a name that is used to refer to memory location. Python variable is also known as an identi-fier and used to hold value.

In python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

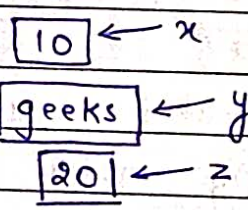
Example -

```
price = 100
tax = 18
total-price = price + tax
print (total-price)
```

Output - 118

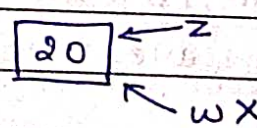
How variables work ?

```
x = 10
y = "geeks"
z = 20
```



if we say, w = z

then



but if, w = 30

```
print (x)
print (y)
print (z)
print (w)
```

O/P -

```
10
geeks
20
30
```

Spiral

Date .....

Python is Dynamically Typed

```
x = 10
print(x)
x = "geeks"
print(x)
```

O/P - 10
geeks

NOTE - Using a variable without assigning it cause error.

Eg- print(x)
O/P - Name Error : name 'x' is not assigned

=> input() in Python -

This function is used to take input from the user

Example -
```
name = input("Enter the name =")
print("welcome" + name)
```

O/P - Enter the name = Himani
Welcome Himani

Example - # Python program for addition

```
x = input("Enter First Number :")
y = input("Enter Second Number :")
x = int(x)
y = int(y)
res = x + y
print("Sum is", res)
```

Spiral

Date .....

Output - Enter First Number = 10
Enter Second Number = 20
Sum is 30

NOTE - input() always gives us a string so whenever we need mathematical expressions we have to convert it from string.

=> Type() in Python -

Type is a built-in function that tells you data type of a variable or a value.

Eg -
```
a = 10
print (type (a))
b = 10.5
print (type(b))
c = 2 + 3j
print (type (c))
```

O/P -
```
< class, 'int' >
< class, 'float' >
< class, 'complex' >
```

- In python we use "None" to indicate that we don't know the value yet or we didn't assign any value to this variable.

In python there is no char type, if you want char you should create single char string. (String is a sequence of char)

Eg -
```
str = "gfg"
print ( type (str))
J = [10, 20, 30]
print (type (J))
```

(array and dynamically add and remove items. Items are stored in continuous locations)

Spiral

Date .....

```
t = (10, 20, 30)
print (type(t))
```

(It is also like a list, the difference is you can't modify once you created a tuple. These are immutable)

```
s = {10, 20, 30}
print (type(s))
```

(Set is a collection of items where all items are distinct and set is like mathematical set)

```
d = {10 : "gfg", 20 : "ide"}
print (type(d))
```

(dictionaries are used to do mappings. This is a collection of key : value pairs)

Eg- items : prices , rollno : name.

=> Type conversion in Python -

Implicit

Eg-
```
a = 10
b = 1.5
c = a+b
print (c)
d = True
e = a+d
print (e)
```

O/P - 11.5
11

Explicit

Eg-
```
s = "135"
i = 1+ int(s)
f = float (s)
print (i)
print (f)
```

O/P - 145
135.0

Eg-
```
s = "geeks"
print ( list (s))
print ( tuple (s))
print ( set (s))
```

O/P- ['g', 'e', 'e', 'k', 's']
('g', 'e', 'e', 'k', 's') {'e', 'g', 'k', 's'}

Spiral

Date .....

Eg-3
```
a = 20
print (bin(a))
print (hex(a))
print (oct(a))
```

O/P -
```
0b10100
0x14
0024
```

⇒ Comments in Python -

We may wish to describe the code we develop. We might wish to take notes of why a section of script fn, for instance. Formulas, procedures, and sophisticated business logics are typically explained with comments.

Single-line comment - #
Multi-line comment - # or '''
'''

Doc-string - The string enclosed in triple quotes that comes immediately after the defined fn are called Python Docstring.

⇒ if, else and elif in Python -

There come situations in real life when we need to do some specific task and based on some specific conditions and, we decide what should we do next.

Similarly, there comes a situation in programming where a specific task is to be performed if a specific condition is true.

Date .....

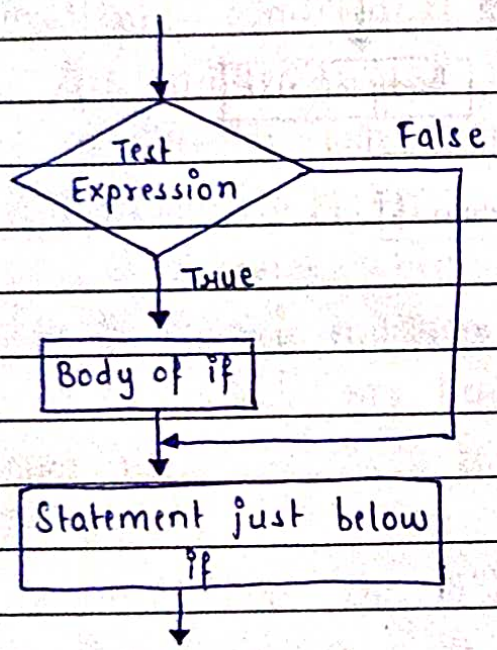
In such cases, conditional statements can be used. And the conditional statements are -

- if
- if .. else
- Nested if
- if - elif statements.

<u>if - statement</u> - in this if the condition is true then if part executes else the else part executes.

<u>Syntax</u> - if condition :
Statements to execute if
condition is true

<u>Flowchart</u> -



<u>Example</u> - if 10 > 5 :
print ("10 is greater than 5")
print ("Program ended")

<u>O/P</u> - 10 is greater than 5.
Program ended

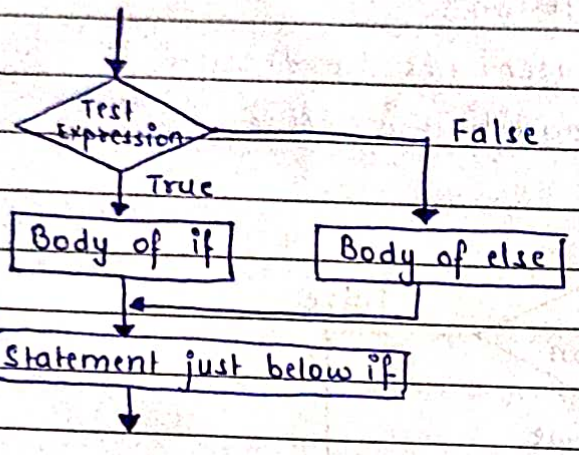
Date .....

if-else statement - In conditional if statement the additional block of code is merged as else statement which is performed when if condition is false.

Syntax -

```
if (condition):
    # Executes this block if
    # condition is true
else :
    # Executes this block if
    # condition is false
```

Flowchart -



Example -

```
x = 3
if x == 4 :
    print ("Yes")
else:
    print ("No")
```

Example - You can also chain if...else statement with more than one condition.

```
letter = "A"
if letter == "B":
    print (" letter is B")
else :
    if letter == "C":
        print (" letter is C")
```

Date .....

```
else :
    if letter == "A":
        print ("letter is A")
    else
        print ("letter isn't A, B and C")
```

O/P - letter is A

Nested if Statement - if statement can also be checked inside other if statement. This conditional statement is called a nested if statement. This means that inner if condition will be checked only if outer if condition is true and by this, we can see multiple conditions to be satisfied.

Syntax -
```
if (condition 1):
    # Executes when condition 1 is true
    if (condition 2):
        # Executes when condition 2 is true
    # if Block is end here
    # if Block is end here
```

Example -
```
num = 10
if num > 5:
    print ("Bigger than 5")
    if num <= 15:
        print ("Between 5 and 15")
```

O/P -
```
Bigger than 5
    Between 5 and 15
```

if-elif Statement - shortcut of if-else chain, while using if-elif statement at the end else block is added which is performed if none of the above if-elif statement is true.

Date .....

Syntax -
```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```

Example -
```
letter = "A"
if letter == "B":
    print ("letter is B")
elif letter == "C":
    print ("letter is C")
elif letter == "A":
    print ("letter is A")
else:
    print ("letter isn't A, B or C")
```

O/P - letter is A

⇒ Arithmetic operators - They are used to perform mathem-atical operations like additional, subtraction, multip-lication, division.

There are 7 arithmetic operators in Python -

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modules
6. Exponentiation
7. Floor division

Date .....

① Addition Operator - In python, + is the addition operator. It is used to add 2 values.

Example -
```
val1 = 2
val2 = 3
res = val1 + val2
print(res)
```
o/p - 5

② Subtraction Operator - used to subtract the second value from the first value.

Example -
```
val1 = 3
val2 = 2
res = val1 - val2
print(res)
```

③ Modulus Operator - % is the modulus operator used to find the remainder when first operand is divided by the second.

Example -
```
val1 = 3
val2 = 2
res = val1 % val2
print(res)
```
o/p - 1

④ Exponentiation Operator - ** is used to raise the first operand to power of second.

Example -
```
val1 = 2
val2 = 3
res = val1 ** val2
print(res)
```
o/p - 8

Spiral

Date .....

⑤ Floor division - // is used to find the floor of the quotient when first operand is divided by the second.

Example -
```
val1 = 3
val2 = 2
res = val1 // val2
print (res)
```
o/p - 1

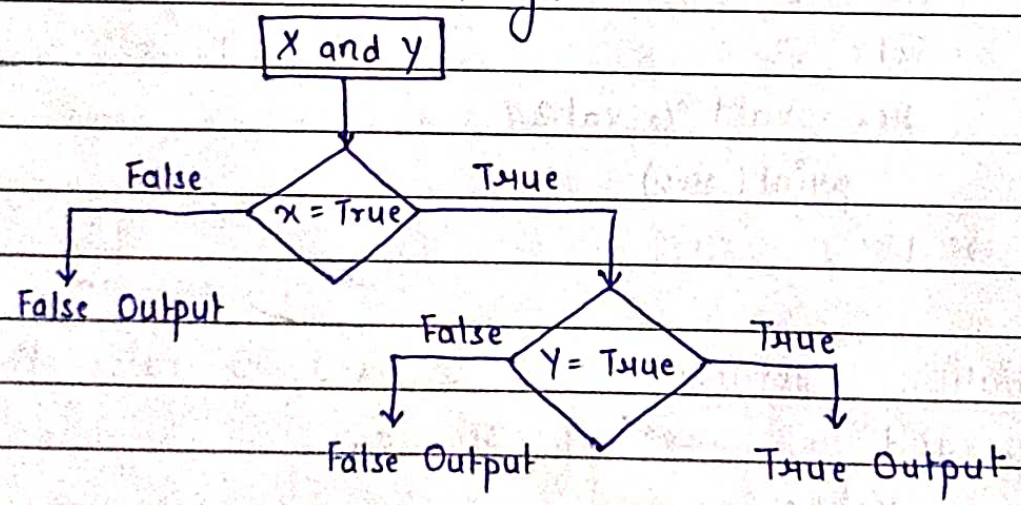
⇒ Python logical Operators with Examples -

Operators are used to perform operations on values and variables. These are the special symbols that carry out arithmetic and logical computations. The value the operator operates on is known as Operand.

- Logical And - Logical or - Logical not

- Logical and - True if both the operands are true.

x and y.



Example -
```
a = 10
b = 10
c = -10
if a > 0 and b > 0:
```

Spiral

Date .....

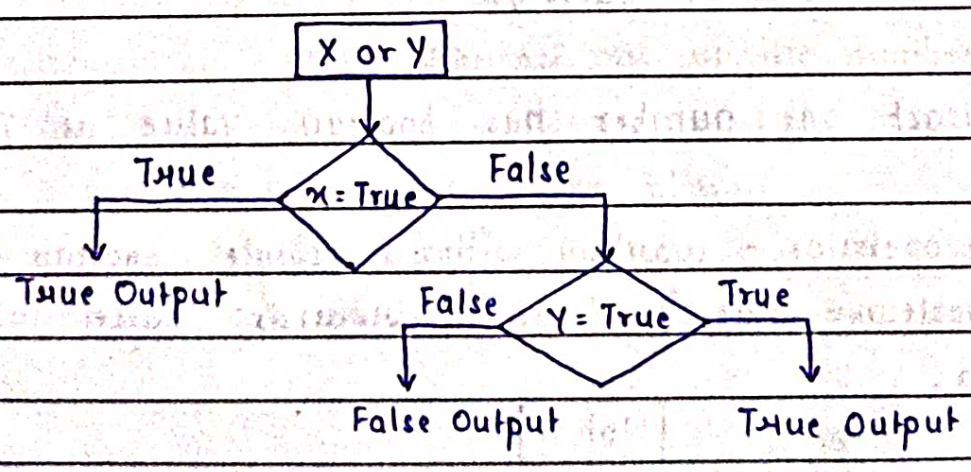
```
print (" The numbers are greater than 0")
 if a>0 and b>0 and c>0:
print (" The numbers are greater than 0")
 else:
    print ("Atleast one number is not greater than 0")
```

O/P - The numbers are greater than 0
Atleast one number is not greater than 0.

Example -
```
a = 10
b = 12
c = 0
if a and b and c:
    print ("All the numbers have boolean value as True")
else:
    print ("Atleast one number has boolean value as False")
```

O/P - Atleast one number has boolean value as False.

- Logical OR - It returns True if either of the operands is True.



Example -
```
a = 10
b = -10
c = 0
```

Date .....

```
if a>0 or b>0:
    print ("Either of the number is greater than 0")
else:
    print ("No number is greater than 0")
if b>0 or c>0:
    print ("Either of the number is greater than 0")
else:
    print ("No number is greater than 0")
```

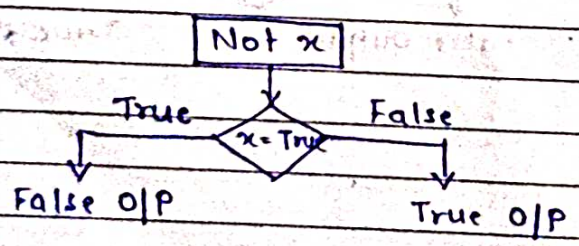
O/P - Either of the number is greater than 0
No number is greater than 0

Example -

```
a = 10
b = 12
c = 0
if a or b or c:
    print ("Atleast one number has boolean value as True")
else:
    print ("All the numbers have boolean value as False")
```

O/P - Atleast one number has boolean value as True.

- Logical NOT operator - works with a single boolean value. If the boolean value is True it returns False and vice-versa.



Date .....

Example - a = 10

```
if not a :
    print ("Boolean value of a is True")
if not ( a % 3 == 0 or a % 5 == 0):
    print ("10 is not divisible by either 3 or 5")
else:
    print ("10 is divisible by either 3 or 5")
```

O/P - 10 is divisible by either 3 or 5

- Order of evaluation of logical operator -

In the case of multiple operators, Python always evaluates the expression from left to right.

Example -

```
def order(x):
    print ("Method called for value:", x)
    return True if x > 0 else False
a = order
b = order
c = order
if a(-1) or b(5) or c(10):
    print ("Atleast one of the number is positive")
```

O/P - Method called for value = -1
Method called for value = 5
Atleast one of the number is positive.

for

Spiral

Date .....

⇒ Python Membership and Identity Operators -

Python offers two membership operators to check or validate the membership of a value. It tests for membership in a sequence, such as strings, lists or tuples.

in operator - The 'in' operator is used to check if a character / substring / element exists in a sequence or not. Evaluate to True if it finds the specified element in a sequence otherwise False. For eg -

```
'G' in 'GeeksforGeeks'      # Checking 'G' in string
True
'g' in 'GeeksforGeeks'      # Checking 'g' in string since pytho
is case-sensitive, return False
False
```

Example -

```
list1 = [1, 2, 3, 4, 5]
list2 = [6, 7, 8, 9]
for item in list1:
    if item in list2:
        print("overlapping")
    else:
        print("not overlapping")
```

O/P -

```
not 100
not
not
not
not
```

Date .....

- 'not in' operator - Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Example -
```
x = 24
y = 20
list = [10, 20, 30, 40, 50]
if (x not in list):
    print ("x is NOT present in given list")
else:
    print ("x is present in given list")
if (y in list):
    print ("y is present in given list")
else:
    print ("y is NOT present in given list")
```

O/P - x is NOT present in given list.
y is present in given list

- Identity operators - They are used to compare the object if both the objects are actually of the same data type and share the same memory location.
There are different identity operators such as

'is' operator - Evaluate to True if the variables on either side of the operator point to the same object and false otherwise.

Example -
```
x = 5
y = 5
print (x is y)
id (x)
id (y)
```

Spiral

Date .....

O/P - 140704586672032
140704586672032

Here, in the given example, both the variable x and y have value 5 assigned to it and both share the same memory location, which is why return True.

'is not' operator - Evaluates to false if the variables on either side of the operator point to a different object and true otherwise.

Example -

```
x = 5
if (type(x) is not int):
    print ("true")
else:
    print ("false")
# Prints True
x = 5.6
if (type(x) is not int):
    print ("true")
else:
    print ("false")
```

O/P - False
True

⇒ Python Bitwise Operators -

Operators are used to perform operations on values and variables. These are the special symbols that carry out arithmetic and logical computations. The value the operator operates on is known as operand.

Date .....

① Bitwise operators -

a)- **Bitwise AND (&)** - Return 1 if both the bits are 1 else 0.

Example - a = 10 = 1010 (Binary)
b = 4 = 0100 (Binary)
a & b = 1010
&
0100
= 0000
= 0 (Decimal)

b)- **Bitwise OR (|)** - Returns 1 if either of the bit is 1 else 0.

Example - a = 10 = 1010 (Binary)
b = 4 = 0100 (Binary)
a|b = 1010
|
0100
= 1110
= 14 (Decimal)

c)- **Bitwise NOT operator (~)** - Returns one's complement of the no

Example - a = 10 = 1010 (Binary)
~a = ~1010
= -(1010 + 1)
= -(1011)
= -11 (Decimal)

d)- **Bitwise (XOR) (^)** - Return 1 if one of the bits is 1 and the other is 0 else return 0.

Date .....

Example - a = 10 = 1010 (Binary)
b = 4 = 0100 (Binary)
a^b = 1010
^
0100
= 1110
= 14 (Decimal)

Example -
```
a = 10
b = 4
# Print bitwise AND operation
print ("a & b=", a & b)
# Print bitwise OR operation
print ("a|b =", a|b)
# Print bitwise NOT operation
print ("~a =", ~a)
# Print bitwise XOR operation
print ("a ^b =", a ^b)
```

o/p - a & b = 0 , a|b = 14 , ~a = -11
a^b = 14

(3) Shift Operators - These operators are used to shift the bits of a number left or right thereby multiplying or dividing the no. by two resp. They can be used when we have to multiply or divide a number by two.

a)- Bitwise Right ~~op~~ Shift (>>) - Shifts the bits of the no. to the right and fills 0 on void left (fills 1 in the case of a negative number) as a result. Similar effect as of dividing the no. with some power of two.

Spiral

Date .....

Example - a = 10 = 0000 1010 (Binary)
a >> 1 = 0000 0101 = 5

Example - a = -10 = 1111 0110 (Binary)
a >> 1 = 1111 1011 = -5

b)- Bitwise Left Shift - (<<)
Shifts the bits of the number to the left and fills 0 on void right as a result. Similar effect as of multiplying the no. with some power of two.

Example - a = 5 = 0000 0101 (Binary)
a << 1 = 0000 1010 = 10
a << 2 = 0001 0100 = 20

Example - b = -10 = 1111 0110 (Binary)
b << 1 = 1110 1100 = -20
b << 2 = 1101 1000 = -40

③ Bitwise Operator Overloading - Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two string and merge two lists. It is achievable because the '+' operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

Example -
```
class Geek():
    def_init_(self, value):
        self.value = value
    def_and_(self, obj):
        print("And operator overloaded")
```

Date .....

```
        if isinstance (obj, Geek):
            return self.value & obj.value
        else:
            raise ValueError ("Must be a object of class Geek")
    def __or__(self, obj):
        print ("Or operator overloaded")
        if isinstance (obj, Geek):
            return self.value | obj.value
        else:
            raise ValueError ("Must be a object of class Geek")
    def __xor__(self, obj):
        print ("XOR operator overloaded")
        if isinstance (obj, Geek):
            return self.value ^ obj.value
        else:
            raise ValueError ("Must be a object of class Geek")
    def __lshift__(self, obj):
        print ("lshift operator overloaded")
        if isinstance (obj, Geek):
            return self.value << obj.value
        else:
            raise ValueError ("Must be a object of class
                                Geek")
    def __invert__(self):
        print ("Invert operator overloaded")
        return ~self.value
# Driver's code
    if __name__ == "__main__":
        a = Geek (10)
        b = Geek (12)
        print (a & b)
```

Spiral

Date .....

```
print (a|b)
print (a^b)
print (a<<b)
print (a>>b)
print (~a)
```

O/P - And operator overloaded
8
Or operator overloaded
14
Xor operator overloaded
8
lshift operator overloaded
40960
rshift operator overloaded
8
Invert operator overloaded
-11

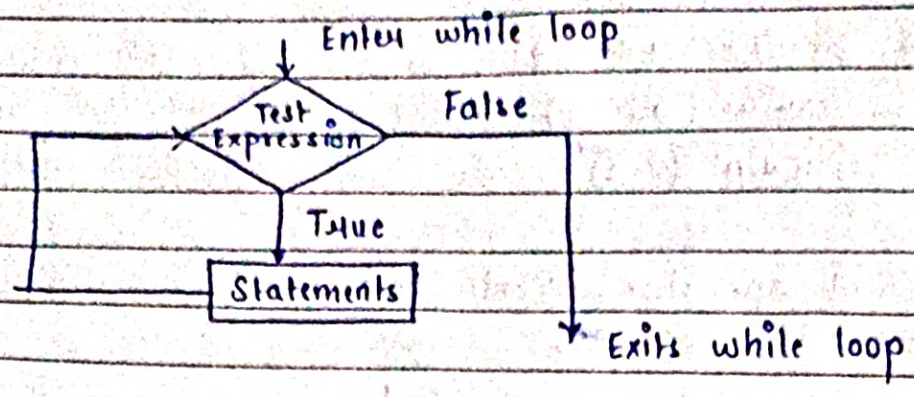
* Loops in Python -

- While Loop - It is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

Syntax - while expression :
statement (s)

Date .....

Flowchart



While loop falls under the category of indefinite iteration. Indefinite iteration means that the number of times the loop is executed isn't specified explicitly in advance.

Statements represents all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its methods of grouping statements. When a while loop is executed, expr is first evaluated in a Boolean context and if it is true, the loop body is executed. Then the expr is checked again, if it is still true then the body is executed again and this continues until the expression becomes false.

Example - 1

```
count = 0
while (count < 3):
    count = count + 1
    print ("Hello Geek")
```

O/P -

```
Hello Geek
Hello Geek
Hello Geek
```

Date .....

Example - 2

```
a = [1, 2, 3, 4]
while a:
    print (a.pop())
```

O/P -

```
4
3
2
1
```

In this, we have run a while loop over a list that will run until there is an element present in the list.

Example - 3 Single statement while block

Just like the if block, if the while block consists of single statement we can declare the entire loop in a single line. If there are multiple statements in the block that make up the loop body, they can be separated by semicolons (;).

```
count = 0
while (count < 5): count += 1; print ("Hello Geek")
```

O/P -

```
Hello Geek
Hello Geek
Hello Geek
Hello Geek
Hello Geek
```

Spiral

Date .....

– range() in Python –

The python range() fn returns a sequence of numbers, in a given range. The most common use of it is to iterate sequences on a sequence of numbers using Python loops.

Syntax – range(start, stop, step)

Parameter – start: [optional]
stop:
step:

Return:

Example –
```
for i in range(5):
    print(i, end = " ")
print()
```

O/P – 0, 1, 2, 3, 4

What is the use of the range() in Python?

In simple terms, range() allows the user to generate a series of no. within a given range. Depending on how many arguments the user is passing to the fn, the user can decide where that series of numbers will begin & end, as well as how big the difference will be between one no. and the next. Python range() fn takes can be initialized in 3 ways –

- range(stop) takes one argument.
- range(start, stop) takes two argument.
- range(start, stop, step) takes three argument.

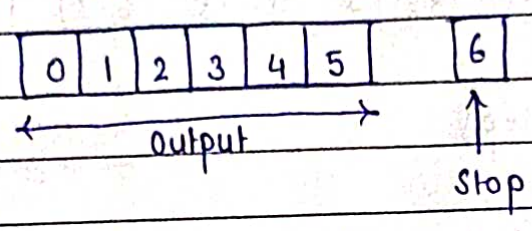
Date .....

a)- range (stop) -

When the user call range() with one argument, the user will get a series of numbers that starts at 0 and includes every whole number up to, but not including, the number that the user has provided as the stop.

Example -

Python range(6)



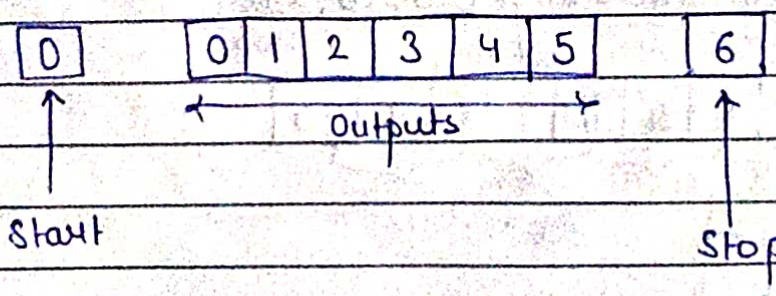
Example -
```
for i in range(6):
    print(i, end=" ")
    print()
```

O/P - 0 1 2 3 4 5

b)- range (start, stop) -

When the user call range() with two arguments, the user gets to decide not only where the series of no. stops but also where it starts, so the user don't have to start at 0 all the time. Users can use range() to generate a series of numbers from X to Y.

Python range (0, 6)



Date .....

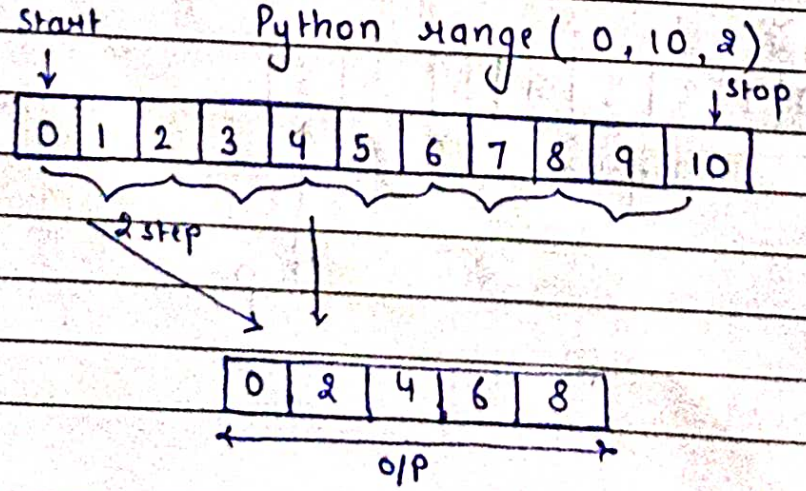
Example - Demonstration of Python range (start, stop)

```
# printing a natural
# number from 5 to 20
for i in range (5, 20):
    print (i, end = " ")
```

O/P - 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

c)- range (start, stop, step)

When the user called range() with three arguments, the user can choose not only where the series of numbers will start and stop, but also how big the difference will be between the one no. and the next. If the user doesn't provide the step, then range() will automatically behave as if the step is 1. In this example, we are printing even no. between 0 and 10, so we choose our starting point from 0 (start = 0) and stop the series at 10 (stop = 10). For printing an even number the difference between one number and the next number must be 2 (step = 2) after providing a step we get the following output (0, 2, 4, 6, 8).



Spiral

Date .....

Example - Demonstration of Python range (start, stop, step)

```
for i in range (0,10,2):
    print (i, end = " ")
    print()
```

O/P - 0, 2, 4, 6, 8

For Loop in Python -

It is used for sequential traversal i.e., it is used for iterating over an iterable like String, Tuple, List, Set or Dictionary.

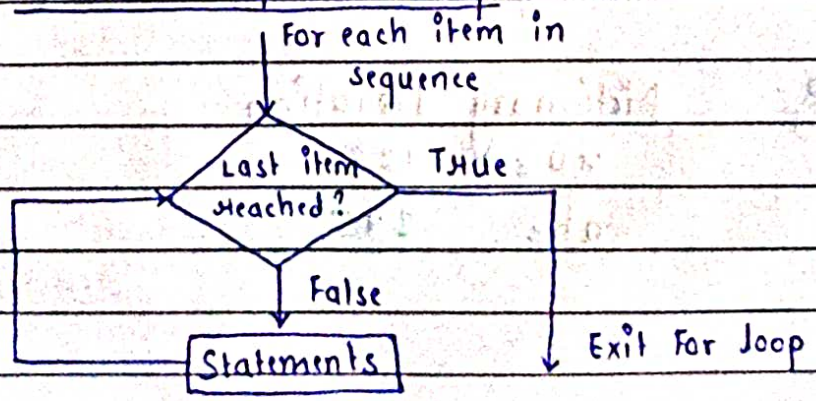
In python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is a "for" loop which is similar to each loop in other languages.

NOTE - In Python, for loops only implements the collection-based iteration.

Syntax -

```
for var in iterable:
    # statements
```

Flowchart of For loop



Date .....

Here the iterable is a collection of objects like lists, tuples. The indented statements inside the for loop are executed once for each item in an iterable. The variable var takes the value of the next item of the iterable each time through the loop.

Example -

```
# Python program to illustrate
# iterating over a list
l = ["geeks", "for", "geeks"]
for i in l:
    print(i)
```

O/P -

```
geeks
for
geeks
```

Example - Loop in Dictionary

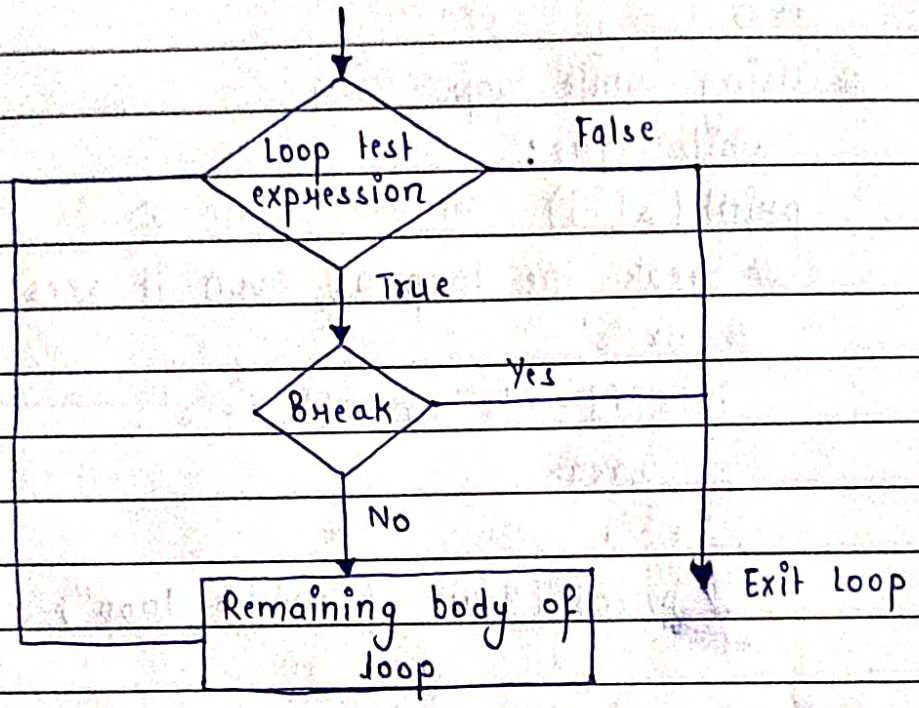
```
# iterating over dictionary
print ("Dictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d:
    print ("%s %d" % (i, d[i]))
```

O/P -

```
Dictionary Iteration
xyz 123
abc 345
```

Date .....

* break in python - It is used to bring the control out of the loop when some external condition is triggered. break statement is put inside the loop body (generally after if condition). It terminates the current loop, i.e., the loop in which it appears, and resumes execution at the next statement immediately after the end of that loop. If the break statement is inside a nested loop, the break will terminate the innermost loop.



Example -
```
for i in range (10):
    print(i)
    if i == 2:
        break
```

O/P - 0
1
2

Example - 2
```
s = 'geeksforgeeks'
# Using for loop
```

Date .....

```
for letter in s:
    print(letter)
# break the loop as soon it sees 'e'
 # or 's'
  if letter == 'e' or letter == 's'
        break
print ("Out of for loop")
  print()
   i=0
 # Using while loop
   while True:
  print(s[i])
   # break the loop as soon it sees 'e'
    # or 's'
     if s[i] == 'e' or s[i] == 's':
        break
     i+=1
      print ("Out of while loop")
```

O/P -

```
g
e
Out of for loop
g
e
Out of while loop
```

* Continue Statement in Python - It is a loop control statement that forces to execute the next iteration of the loop while skipping the rest of the code inside the loop for the current iteration only, i.e.,

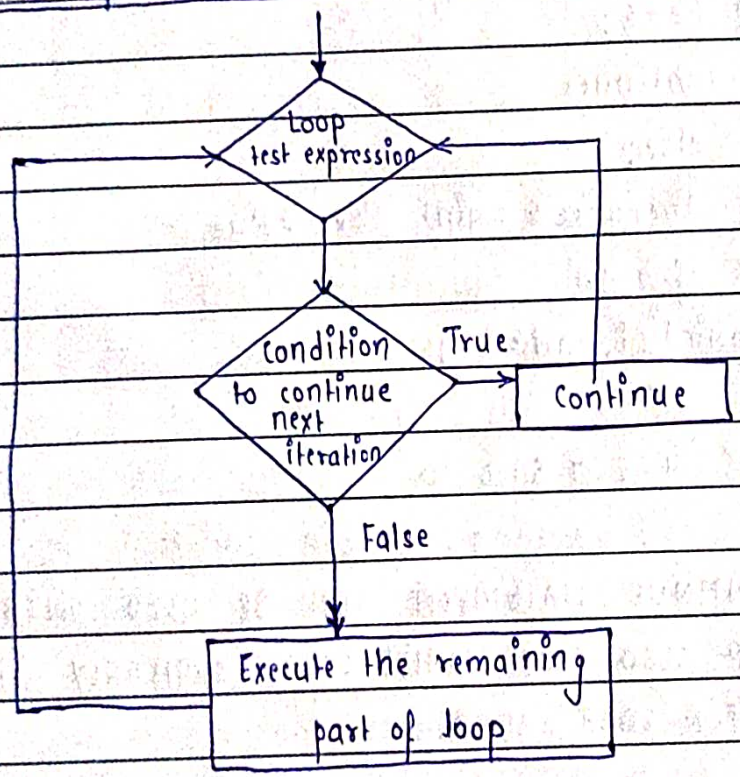
Date .....

when the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped for the current iteration and the next iteration of the loop will begin.

Syntax -

```
while True :
    .....
    if x = = 10 :
        continue
    print (x)
```

Flowchart of Continue Statement -



Example -

```
for var in "Geeksforgeeks":
    if var = = "e":
        continue
    print (var)
```

O/P -

G	o
K	r
S	g
f	k
	s

Date .....

Explanation - Here we are skipping the print of character 'e' using if-condition checking and continue statement

Example - 2

Consider a situation when you need to write a program which prints the number from 1 to 10, but not 6.

```
# loop from 1 to 10
for i in range (1, 11)
    # if i is equal to 6,
    # continue to next iteration
    # without printing
    if i==6:
        continue
        else:
    # otherwise print the value
    # of i
    print (i, end=" ")
```

O/P - 1 2 3 4 5 7 8 9 10

NOTE - THE CONTINUE STATEMENT CAN BE USED WITH ANY OTHER LOOP ALSO LIKE "WHILE LOOP" SIMILARLY AS IT IS USED WITH 'FOR LOOP" ABOVE.

* Nested Loop - With for and while loops we can create nested loop. Nested loop means loops inside a loop. For example, while loop inside the for loop, for loop inside the for loop, etc.

Date .....

Syntax - Outer-loop Expression :

Inner-loop Expression :
Statement inside inner-loop

Statement inside Outer-loop

Example -
```
x = [1,2]
y = [4,5]
for i in x :
  for j in y :
    print ( i,j)
```

O/P -
```
14
15
24
25
```

Example - 2
```
# Multiplication Table
# Running outer loop from 2 to 3
for i in range (2,4):
  # Printing inside the outer loop
  # Running inner loop from 1 to 10
  for j in range (1,11):
    # Printing inside the inner loop
    print (i, "*", j, "=", i*j)
  # Printing inside the outer loop
  print ()
```

O/P - 2 * 1 = 2 - - - - - - - - - 2 * 10 = 20
3 * 1 = 3 - - - - - - - - - - 3 * 10 = 30

Date .....

Example -
```
# Initialize list1 and list2
# with some strings
list1 = ['I am', 'You are']
list2 = ['healthy', 'fine', 'geek']
# Store length of list2 in list2_size
list2_size = len(list2)
# Running outer for loop to
# iterate through a list1
for item in list1:
    # Printing outside inner loop
    print("start outer for loop")
    # initialize couter i with 0
    i=0
    # Running inner while loop to
    # iterate through a list2
    while (i< list2_size):
        # Printing inside inner loop
        print(item, list2[i])
        # Incrementing the value of i
        i = i+1
    # printing outside inner loop
    print("end for loop")
```

O/P -
```
start outer for loop
I am healthy
I am fine
I am geek
end for loop
start outer for loop
You are healthy
You are fine
```

Spiral

Date .....

You are geek
end for loop

In this example, we are initializing two lists with some strings. Store the size of list 2 in 'list 2_size' using len() function and using it in the while loop as a counter. After that run an outer for loop to iterate over list 1 and inside that loop run an inner while loop to iterate over list 2 using list indexing inside that we are printing each value of list 2 for every value of list 1.

* Functions in Python -

Python Functions is a block of statements that return the specific task.

The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Syntax - Python functions

Keyword Function name Parameters

```
def function_name (parameters):

    # statement ] ---------> Body of Statement

    return expression
```

function return

Date .....

Example -

```
def fun():
    print("Welcome to GFG")
# Driver code to call a function
fun()
```

O/P - Welcome to GFG.

- Defining and calling a function with parameters -

Syntax -

```
def function_name(parameter : data_type) ->
    return_type :
        """ Doctring """
    # body of the function
        return expression
```

Example -

```
# some more functions
def is_prime(n):
    if n in [2, 3]:
        return True
    if (n==1) or (n % 2 == 0):
        return False
    r = 3
    while r * r <= n:
        if n % r == 0:
            return False
        r += 2
        return True
print(is_prime(78), is_prime(79))
```

O/P - False True

Spiral

Date .....

⇒ Arguments of a Python Function -

Arguments are the values passed inside the paranthesis of the function. A function can be have any number of arguments separated by a comma.

Example -
```
# A simple Python Function to check
# whether x is even or odd
def evenOdd(x):
    print("even")
else:
    print ("odd")
# Driver code to call the function
evenOdd(2)
evenOdd(3)
```

O/P - even
odd

- Types of Arguments -

① Default argument - It is a parameter that assumes a default value if a value is not provided in the function call for that argument.

Example -
```
def myFun (x, y=50):
    print ("x:", x)
    print ("y:", y)
# Driver code (We call myFun() with
# only argument)
myFun(10)
```

O/P - x=10
y=50

Spiral

Date .....

② Keyword arguments -

The idea is to allow the caller to specify the argument name with values so that caller does not need to remember the order of parameters.

Example -
```
def student (firstname, lastname):
    print (firstname, lastname)
# Keyword arguments
student( firstname = 'Geeks', lastname = 'for')
student (lastname = 'for', firstname = 'Geeks')
```

O/P -
```
Geeks for
Geeks for
```

③ Variable-length arguments -

We can pass a variable number of argum-ents to a function using special symbol. There are two special symbols -

- * args (Non-keyword Arguments)
- ** kwargs (Keyword Arguments)

Example -
```
def myFun (*argv):
    for arg in argv:
        print (arg)
myFun ('Hello', 'Welcome', 'to', Geeksfor Geeks')
```

O/P -
```
Hello
Welcome
to
GeeksforGeeks
```

Spiral

Date .....

* Doctstring - The first string after the function is called the Document string or Doctring in short. This is used to describe the functionality of the function. The use of docstring in fn is optional but it is considered a good practice.

Syntax - print(function_name.__doc__)

Example -
```
def evenOdd(x):
    """Function to check whether x is even or odd"""
    if (x % 2 == 0):
        print("even")
    else:
        print("odd")
# Driven code to call the function
print(evenOdd.__doc__)
```

O/P - Function to check if the number is even or odd.

* Return statement in Python function -
The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller.

Syntax - return [expression_list]

The return statement can consist of a variable, an

Date .....

expression, or a constant which is returned to the end of the function execution. If none of the above is present with the return statement a None object is returned.

Example -
```
def square_value(num):
    """This function returns the square value
       of the entered number"""
    return num **2
print (square_value(2))
print (square_value(4))
```

O/P -
```
4
16
```

- Pass by Reference or pass by value -

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created.

Example -
```
def myFun(x):
    x[0] = 20
# Driver Code (Note the lst is modified after
# function call
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print (lst)
```

O/P - [20, 10, 11, 12, 13, 14, 15]

* Strings in Python -

Strings are array of bytes representing Unicode characters.

Date .....

Example - "Geeksforgeeks" or 'Geeksforgeeks'

Python does not have a character data type, a single character is simply a string with a length of 1. Square braces can be used to access elements in string.

Example - print ("A Computer Science portal for geeks")

o/p - A Computer Science portal for geeks.

- Creating a String in Python -

String in python can be created using single, double or even triple quotes.

Example -

```
String 1 = 'Welcome to the Geeks World'
print ("String with the use of single Quotes:")
print ( String 1)
# Creating a String
# with double Quotes
String 1 = "I m a Geek"
print ("\n String with the use of Double Quote
print ( String 1)
# Creating a String with triple Quotes
String 1 = '''I'm a "Geek"'''
print ("\n String with the use of Triple
                                Quotes :")
print (String 1)
# Creating String with triple
# Quotes allows multiple lines
String 1 = '''Geeks
For
Life'''
print ("\n Creating a multiline String:")
```

Spiral

Date .....

print (String 1)

O/P - String with the use of single quotes :
Welcome to the Geeks world
String with the use of double quotes :
I'm a Geek
String with the use of Triple quotes :
I'm a "Geek"
Creating a multiline String :
Geeks
For
Life

- Accessing characters in Python String -

In python, individual characters of a string can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the string, eg. -1 refers to the last char, -2 refers to the second last char and so on.

While accessing an index out of the range will cause an IndexError. Only Integers are allowed to be passed as an index, float or other types that will cause a TypeError.

Example -
```
String 1 = "Geeks For Geeks"
print ("Initial String :")
print (String 1)
# Printing First character
print ("\n First character of String is :")
print (String 1 [0])
```

Date .....

```
# Printing Last Character
print ("\n Last character of String is :")
print (String1[-1])
```

O/P - Initial String :
GeeksForGeeks
First character of String is :
G
Last character of String is :
S

- Reversing a Python String -

```
# Program to reverse a string
gfg = "geeksforgeeks"
print (gfg [::-1])
```

O/P - skeegrofskeeg

- String Slicing -

To access a range of characters in the String, the method of slicing is used. Slicing in a String is done by using a Slicing operator (colon).

Example -
```
String 1 = "GeeksforGeeks"
print (" Initial String :")
print (String 1)
print ("\n Slicing characters for 3-12 :")
print ( String 1 [ 3 : 12])
print ("\n Slicing characters between" +
        "3rd and 2nd last character :")
```

Date .....

print(String1[3:-2])

O/P - Initial String:
GeeksforGeeks
Slicing characters from 3-12:
ksforGeek
Slicing characters between 3rd and 2nd last char:
ksforGee

- char() function -
It returns a string from a Unicode code integer.
Eg - print(char(97))

O/P - a

* Escape Sequencing in Python -

While printing String with single and double quote in it cause SyntaxError because string already contains Single and Double Quotes and hence cannot be printed with the use of either of these. Hence, to print such a String either Triple Quotes are used to Escape sequences are used to print such strings.

- String Concatenation using + Operator -
This operator can be used to add multiple string together. However, the arguments must be a string. Here, the + operator combines the string that is stored in the var1 and var2 and stores in another variable var3.

NOTE - STRINGS ARE IMMUTABLE, THEREFORE, WHENEVER IT IS

Spiral

Date .....

CONCATENATED, IT IS ASSIGNED A NEW VARIABLE.

- index() method - It allows a user to find the Index of the first occurence of an existing substring inside a given string.

  Syntax - string-obj.index(substring, begp, endp)

  Parameters -
  - substring - The string to be searched for.
  - begp - (default : 0) : This f^n specifies the position from where search has to be started.
  - endp (default : length of string) : This f^n specifies the position from where search has to end.

  Return - Returns the 1st position of substring found.

  Exception - Raise valueError if argument string is not found or index is out of range.

- rindex() method - It returns the highest index of the substring inside the string if the substring is found. Otherwise, it raises ValueError.

⇒ Case Changing of Strings -

- lower() - Convert all uppercase characters in a string into lowercase.
- upper() - Convert all lowercase characters in a string into uppercase.
- title() - Convert string to title case.

Date .....

⇒ Built in methods - starts with() & endswith()

Syntax - startswith() - str.startswith(search-string, start, end)

endwith() - str.endswith(search-string, start, end)

- Python String find() method - It returns the lowest index or first occurence of the substring if it is found in a given string. If it is not found, then it returns -1.

Syntax - str-obj.find(sub, start, end)

Substring that need to be searched

* List Introduction -

Python lists are just like dynamically sized arrays, declared in other language (C++, Java). In a simple language, it is a collection of things, enclosed with [] and separated by commas.

The list is a sequence data type which is used to store the collection of data. Tuples and Strings are other types of sequence data types.

Eg - Var = ["Geeks", "For", "Geeks"]
print(Var)

O/P - ['Geeks', 'For', 'Geeks']

Spiral

Date .....

- Lists need not to be homogeneous always which makes it the most powerful tool in Python.
- A single list may contain Data Types like Int, Str, as well as objects.
- Lists are mutable, and hence, they can be altered even after their creation.

* Creating a list in Python -

List can be created by just placing the sequence inside the square braces [ ]. Unlike sets, a list doesn't need a built-in function for the creation of a list.

Example -

```
List = [ ]
print (" Blank List :")
print ("List")
List = [ 10, 20, 30]
print ("\n List of numbers :")
print( List)
```

O/P -

```
Blank list :
[ ]
List of numbers :
[10, 20, 30]
```

- Accessing element from the list -

```
List = ["Geeks", "For", "Geeks"]
print (" Accessing a element:")
print ( List [0])
print ( List [2])
```

Spiral

Date .....

- Negative Indexing

- Getting the size of the list -

```
List 1 = [ ]
print ( len (List 1))
List 2 = [ 1 , 2 , 3 ]
print ( len ( list 2))
```

O/P - 0
3

- Adding Elements to a list -

Using append ( ) method -

- Only 1 element can be added at a time, for the additional of multiple elements with the append( ) method, loops are used.
- Tuples can be added to the list with the use of append method because tuples are immutable.

Example -

```
List = [ ]
print ( " Initial blank list : ")
print ( List)
List. append (1)
List. append (2)
List. append (4)
print ("\n List After Addition of three elem
                                    ents:")
print ( List)
for i in range ( 1 , 4 ):
        List. append (i)
```

Date .....

```
print ("\n List after Addition of elements from 1-3 :")
    print (List)
List. append ((5,6))
    print ("\n List after Addition of a Tuple :")
        print (List)
List 2 = ['For', 'Geeks']
    List. append (List 2)
    print ("\n List after Addition of a List :")
        print (List)
```

O/P - Initial blank List :
[ ]
List after Addition of Three elements :
[1, 2, 4]
List after Addition of elements from 1-3 :
[1, 2, 4, 1, 2, 3]
List after Addition of a Tuple :
[1, 2, 4, 1, 2, 3, (5, 6)]
List after Addition of a List :
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]

- Using insert() method -
It inserts a given element at a given index in a list using Python.

Syntax - list_name. insert (index, element)
index - at which the element has to be inserted.
element - to be inserted in the list.
Returns - Does not return any value.

Date .....

Example -

```
lis = ['Geeks', 'Geeks']
lis.insert (1, "For")
print (lis)
```

O/P - ['Geeks', 'For', 'Geeks']

- count() method - returns the count of how many times a given object occurs in a List.

Syntax - list_name. count (object)

Parameters -

- object - is the item whose count is to be returned.

Exception -

- TypeError - Raises TypeError if more than 1 parameter is passed in count() method.

Example -

```
list 2 = ['a','a','a','b','b','a','c','b']
print(list 2. count('b'))
```

O/P - 3

- del keyword - del is a keyword and remove(), pop() are in-built method.
- remove() method delete values or object from the list using value.
- the del and pop() delete values or objects from the list using an index.
- The del keyword deletes any variable, or list of values from a list.

Date .....

Syntax -

```
del list_name[index]      # To delete single valu
del list_name             # To delete whole list
```

- The remove() method removes the first matching values from the list.

  Syntax - list_name.remove(value)

- The pop() method like the del keyword deletes value at a particular index. But the pop() method returns deleted value from the list.

  Syntax - list_name.pop(index)

Example -

```
number = [1, 2, 3, 2, 3, 4, 5]
# use remove()
number.pop(3)
print(number)
# use remove()
number.pop(-1)
print(number)
number.pop(0)
print(number)
```

o/p -

```
[1, 2, 3, 3, 4, 5]
[1, 2, 3, 3, 4]
[2, 3, 3, 4]
```

- max() method used to compute the maximum of the values passed in its argument and the lexicographically largest value if strings are

Date .....

passed as arguments.

Example - print ("Maximum of 4, 12, 43.3, 19 and 100 is :", end = " ")

print (max (4, 12, 43.3, 19, 100))

O/P - Maximum of 4, 12, 43.3, 19, 100 is = 100

- min() method is used to compute the minimum of the values passed in its arguments.

- sort() function - can be used to sort a List in ascending, descending, or user-defined order.

Syntax - List_name. sort(reverse = True/ False, key = myFunc)

- reverse() method - is a inbuilt method in python that reverses objects of the List in place i.e., it doesn't use any extra space but it just modifies the original list.

Syntax - list_name. reverse()

- Tuples in Python -

Python Tuple is a collection of objects separated by commas. In some ways, a tuple is similar to a list in terms of indexing, nested objects, and repetition but a tuple is immutable, unlike lists which are mutable.

To create a Tuple we will use () operators.

var = ("Geeks", "for", "Geeks")

print (var)

Date .....

- Accessing Values in Tuples -

Method - 1 Using Positive Index
Method - 2 Using Negative Index

* Sets in Python -
It is an unordered collection data type that is iterable, mutable and has no duplicate elements.
Sets are represented by {}.

Since sets are unordered, we can't access items using indexes like we do in lists.

Example - var = {"Geeks", "for", "Geeks"}
print (type(var))

o/p - <class 'set'>

- Frozen set are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. It can be done with frozenset() method.

- Methods for Sets -

- Adding elements to Sets - set.add() function
- Union operation on Sets - union() function
- Intersection operation - intersection() or & operator
- Difference - difference() or - operator
- Clearing sets - Clear() method

Date .....

* Dictionary in Python -

It is a collection of keys values, used to store data values like a map, which, unlike other data types which hold only a single value as an element.

- Dictionary holds key : value pair.

Example - Dict : {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print (Dict)

- Creating a Dictionary -

A dictionary can be created by placing a sequence of elements with curly { } braces, separated by 'comma'. Dictionary holds pairs of values, one being the key and the other corresponding pair element being its key: Value. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.

NOTE - Dictionary keys are case sensitive, the same name but different cases of key will be treated distinctly.

Example -
```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\n Dictionary with the use of Integer keys:")
print (Dict)
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\n Dictionary with the use of Mixed keys:")
print (Dict)
```

Spiral

Date .....

O/P - Dictionary with the use of Integer keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with the use of Mixed keys:
{'Name': 'Geeks', 1: [1, 2, 3, 4]}

- Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

- Nested Dictionary -

```
Dict = {1: 'Geeks', 2: 'For', 3: {'A': 'Welcome',
        'B': 'To', 'C': 'Geeks'}}

print(Dict)
```

O/P - {1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}

- Adding elements to a Dictionary -
One value can be added to a Dictionary by defining value along with the key. Eg- Dict[key] = 'Value'.

Updating an existing value in a Dictionary can be done by using the built-in update() method.
Nested key values can be also added to an existing Dictionary.

- Accessing elements to a Dictionary -
In order to access the item of a dictionary refer to its key name. key can be used inside square braces.

Date .....

- There is also a method called get() method that will be help in accessing the elements from a dictionary. This method accepts key as argument and return the value.

⇒ Dictionary methods -

1. clear() - Remove all the elements from the dictionary.
2. copy() - Return a copy of the dictionary.
3. get() - Return the value of specified key.
4. items() - Return a list containing a tuple for each key value pair.
5. keys() - Returns a list containing dictionaries keys
6. pop() - Remove the element with specified key
7. popitem() - Remove the last inserted key-value pair.
8. update() - Updates dictionary with specified key-value pairs.
9. values() - Returns a list of all the values of dictionary.

* Slicing (List, Tuple And String) -

Slicing in python is a feature that enables accessing parts of the sequence. In slicing a string, we create a substring, which is essentially a string that exists within another string. We use slicing when we require a part of the string and not the complete string.

Syntax - string [start : end : step]

- start - We provide the starting index.
- end - We provide the end index.
- step - It is an optional argument that determine

Date .....

the increment between each index for slicing.

- Slicing in List -

```
lst = [50, 70, 30, 20, 90, 10, 50]
print(lst[1:5])
```

O/P - [70, 30, 20, 90]

- Slicing in Tuple -

```
tup = (22, 3, 45, 4, 2.4, 2, 56, 890, 1)
print(tup[1:4])
```

O/P - (3, 45, 4)

* Comprehension in Python -

A python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python List.

- Python List comprehension provides a much more short syntax for creating a new list based on the values of an existing list.

Syntax -

```
newList = [expression(element) for element in
                oldList if condition]
```

Date .....

- List Comprehensions Vs For Loop -

There are various ways to iterate through a list. However, the most common approach is to use the for loop.

```
List = []
for character in 'Geeks 4 Geeks!':
    List.append(character)
print(List)
```

O/P - ['G','e','e','k','s','4','G','e','e','k','s','!']

- Python Dictionary Comprehension -

Here we have two lists named keys and values and we are iterating over them with the help of zip() function.

```
keys = ['a','b','c','d','e']
values = [1,2,3,4,5]
myDict = {k:v for (k,v) in zip(keys, value)}
print(myDict)
```

O/P - {'a':1, 'b':2, 'c':3, 'd':4, 'e':5}

Date .....

* Python OOPs Concepts and Class -

OOPs is a programming paradigm that uses object and classes in programming. It aims to implement real-world entities like, inheritance, polymorphism, encapsulations, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

Main concepts of OOPs -

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

- Class - A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

- Classes are created by keyword class.
- Attributes are the variables that belongs to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg - Myclass.Myattribute

Syntax -
```
class ClassName :
    # Statement-1
    :
    # Statement-N
```

Date .....

- Objects - Object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

An object consists of -

- State - It is represented by the attribute of an object. It also reflects the properties of an object.
- Behavior - It is represented by the method of an object. It also reflects the response of an object to other objects.
- Identity - It gives a unique name to an object & enables one object to interact with other objects.

```
obj = Dog()
```

Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

① The self

- Class method must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
- If we have a method that takes no arguments, then we still have to have one arguments.

Spiral

Date .....

- This is similar to this pointer in C++. When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2).

② The __init__ method -

It is similar to constructions in C++ & Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your objects.

- Encapsulation in Python -

It is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
In this, the variables of a class is hidden from other classes, and can be accessed only through the methods of their current class.

```
class Base:                          # Base Class
    def __init__(self):
        self.__a = 2                 # Protected member
class Derived(Base):                 # Derived class
    def __init__(self):
# Calling constructor of Base class
        Base.__init__(self)
    print("Calling protected member of base class:
                                self.__a)
# Modify the protected variable:
        self.__a = 3
```

Spiral

Date .....

```
print ("Calling modified protected member outside
        class :", self._a)
    obj1 = Derived()
    obj2 = Base()
print ("Accessing protected member of obj1 :",
        obj1._a)
print ("Accessing protected member of obj2 :",
        obj2._a)
```

- Class Instance Attribute -

Class attributes belong to the class itself they will be shared by all the instances. Such attributes are defined in the class body parts usually at the top, for legibility.

```
class Sampleclass:
    count = 0
    def increase (self):
    Sampleclass.count += 1
        s1 = sampleclass()
        s1.increase()
        print (s1.count)
        s2 = sampleclass()
        s2.increase()
        print (s2.count)
        print (sampleclass.count)
```

O/P - 1
2
2

Spiral

Date .....

Unlike class attributes, instance attributes are not shared by objects. Every object has its own copy of the instance attribute.

To list the attributes of an instance/ object, we have two functions -

① vars() - This function is displays the attribute of an instance in the form of dictionary.

② dir() - This function displays more attributes than vars functions, as it is not limited to instance. It displays the class attributes as well. It also displays the attributes of its ancestor classes.

* Class Member Access -

Attribute of a class are function objects that defines corresponding methods of its instance. They are used to implement access controls of the classes.

Built-in functions -

① getattr() - used to access the attribute of object.
② hasattr() - used to check if an attribute exist or not.
③ setattr() - used to set an attribute.
④ delattr() - used to delete an attribute.

Date .....

* Class method vs static method -

Class method is a method that is bound to the class and not the object of the class.
They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
It can modify a class state that would apply across all the instances of the class. For eg, it can modify a class variable that will be applicable to all the instances.
Static method doesn't receive an implicit first argument.

Syntax - class C(object):
@staticmethod
def fun(arg1, arg2, ...):

Returns : a static method for function fun.

A static method is also a method that is bound to the class and not the object of the class.
A static method can't access or modify the class state.
It is present in a class because it makes sense for the method to be present in class.

- Class method vs Static method -

1. A class method takes cls as the first parameter while a static method needs no specific parameters.
2. A class method can access or modify the class state while a static method can't access or modify it.
3. In general, static methods know nothing about the class state. They are utility-type methods that take some

Date .....

parameters and work upon those parameters. On the other hand class methods must have class as a parameter.

* Inheritance -

It is the capability of one class to derive or inherit the properties from another class.

Advantages -

- It represents real-world relationship well.
- It provides reusuability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another Class A, then all the subclasses of B would automatically inherit from class A.

```
class Person(object):
    # Constructor
    def_init_(self, name):
        self.name = name
    # To get name
        def getName(self):
            return self.name
    # To check if this person is an employee
        def isEmployee(self):
            return False
    # Inherited or Subclass
        class Employee(Person):
    # Here we return true
        def isEmployee(self):
            return True
```

Date .....

```
# Driver code
emp = Person ("Geek 1")
print (emp. get Name(), emp. isEmployee())
emp = Employee ("Geek 2")
print (emp. getName(), emp. isEmployee())
```

O/P - ('Geek 1', False)
('Geek 2', True)

- What is object class ?
Like Java object class, in python, object is a root of all classes.

Example - class subclass_name (superclass_name):

```
class Person (object):
    def_init_ (self, name, idnumber):
        self.name = name
        self. idnumber = idnumber
    def display (self):
        print (self. name)
        print ( self. idnumber)
# child class
class Employee( Person):
    def_init_ (self, name, idnumber, salary, post):
        self. salary = salary
        self. post = post
Person._init_ (self, name, idnumber)
a = Employee ('Rahul', 8853, 200000, "Intern")
a. display ()
```

Spiral

Date .....

- Different forms of Inheritance -

1. Single Inheritance - When a child class inherits from only one parent class, it is called single inheritance.

2. Multiple Inheritance - When a child class inherits from multiple parent classes, it is called multiple inheritance.

```
class Base1(object):
    def __init__(self):
        self.str1 = "Geek1"
        print("Base1")
class Base2(object):
    def __init__(self):
        self.str2 = "Geek2"
        print("Base 2")
class Derived(Base1, Base2):
    def __init__(self):
# Calling constructor of Base1 and Base2 classes
        Base1.__init__(self)
        Base2.__init__(self)
        print("Derived")
    def printStrs(self):
        print(self.str1, self.str2)
ob = Derived()
ob.printStrs()
```

O/P -

```
Base1
Base2
Derived
```

```
Base1
Base2
Derived
Geek1 Geek2
```

Date .....

③ Multi-level Inheritance -
When we have a child and grandchild relationship.

④ Hierarchical Inheritance -
More than one derived classes are created from a single base.

⑤ Hybrid Inheritance - This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

Private members of parent class -
We don't always want the instance variables of the parent class to be inherited by the child class i.e., we can make some of the instance variables of the parent class private, which won't be available to the child class.

* Polymorphism -
It means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

Example - Inbuilt polymorphism function -

```
print(len("geeks"))
print(len([10, 20, 30]))
```

O/P - 5
3

Spiral

Date .....

Example of user-defined polymorphism function -

```
def add (x, y, z = 0):
    return x+y+z
print (add (2, 3))
print (add (2, 3, 4))
```

O/P - 5
9

- Polymorphism with class method -

```
class India():
    def capital (self):
print("New Delhi is a capital of India")
    def language (self):
print("Hindi is the most widely spoken language in 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 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()
```

Date .....

- Polymorphism with Inheritance -

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 bird 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()
```

Date .....

```
obj-spr.intro()
obj-spr.flight()
obj-ost.intro()
obj-ost.flight()
```

* **Abstract Class** -

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are ~~declass~~ designing large functional units we use an abstract class.

- By defining an abstract base class, you can define a common Application Program Interface (API) for a set of subclasses.

Example -

```
class Polygon(ABC):
    @abstractmethod
    def noofsides(self):
        pass
class Triangle(Polygon):
    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")
```

Date .....

```
class Pentagon(Polygon):
    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")
class Hexagon(Polygon):
    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")
# Driver code
R = Triangle()
R.noofsides()
K = Quadrilateral()
K.noofsides()
R = Pentagon()
R.noofsides()
K = Hexagon()
K.noofsides()
```

O/P - I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides

x ———————————————— x