

UNIT - II: Computational Thinking and Introduction to Python







Simple logic building through flowcharting: Flowchart symbols, conditional and repetition blocks. Computational Thinking, Algorithm, Pseudo code, Time/Space complexity. Only Big O notation. Basic structure of a Python program, **Elements of Python programming Language:** token, literals, identifiers, keywords, expression, type conversions, Numbers, Variables, Input/output statements, basic data types. Operators and their types and precedence, expressions. Control structures in Python - conditionals and loops

Flowchart

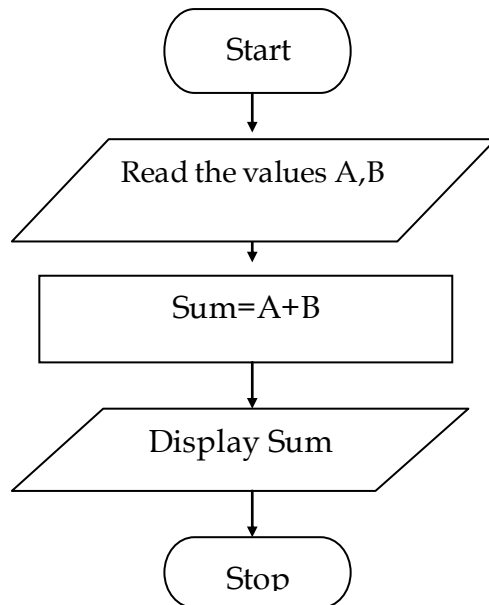
Flowchart is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart Symbols

The common symbols used in drawing flowcharts are

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.

Flow Chart for adding of two numbers



Advantages of flowchart:

1. The Flowchart is an excellent way of communicating the logic of a program.
2. It is easy and efficient to analyze problem using flowchart.
3. During program development cycle, the flowchart plays the role of a guide or a blueprint. Which makes program development process easier.
4. After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
5. It helps the programmer to write the program code.
6. It is easy to convert the flowchart into any programming language code as it does not use any specific programming language concept.

Disadvantage of flowchart

1. The flowchart can be complex when the logic of a program is quite complicated.
2. Drawing flowchart is a time-consuming task.
3. Difficult to alter the flowchart. Sometimes, the designer needs to redraw the complete flowchart to change the logic of the flowchart or to alter the flowchart.
4. In the case of a complex flowchart, other programmers might have a difficult time understanding the logic and process of the flowchart.

Algorithm Complexity

Suppose X is treated as an algorithm and N is treated as the size of input data, the time and space implemented by the Algorithm X are the two main factors which determine the efficiency of X.

Time Factor – The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.

Space Factor – The space is calculated or measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm $f(N)$ provides the running time and / or storage space needed by the algorithm with respect of N as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Space complexity $S(p)$ of any algorithm p is $S(p) = A + S_p(I)$ Where A is treated as the fixed part and $S(I)$ is treated as the variable part of the algorithm which depends on instance characteristic I . Following is a simple example that tries to explain the concept

Algorithm

```
SUM(P, Q)
Step 1 - START
Step 2 -  $R \leftarrow P + Q + 10$ 
Step 3 - Stop
```

Here we have three variables P , Q and R and one constant. Hence $S(p) = 1+3$. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Time Complexity

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function $t(N)$, where $t(N)$ can be measured as the number of steps, provided each step takes constant time.

For example, in case of addition of two n -bit integers, N steps are taken. Consequently, the total computational time is $t(N) = c*n$, where c is the time consumed for addition of two bits. Here, we observe that $t(N)$ grows linearly as input size increases.

Asymptotic Notations

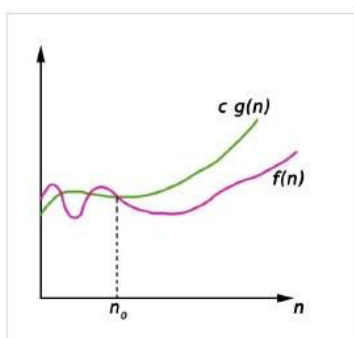
Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

Big Oh Notation

Big-Oh (O) notation gives an upper bound for a function $f(n)$ to within a constant factor.

We write $f(n) = O(g(n))$, If there are positive constants n_0 and c such that, to the right of n_0 the $f(n)$ always lies on or below $c*g(n)$.

$O(g(n)) = \{ f(n) : \text{There exist positive constant } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n), \text{ for all } n \leq n_0 \}$



conditional and repetition blocks: Check the flowcharts in the topic “Control structures in Python - conditionals and loops”

Computational Thinking:

It follows 4 steps

1. **Decomposition:** Divide the problem into smaller problems, usually problems that we have solved before.
2. **Pattern Recognition:** Find similar repeated patterns, identify the differences. Note patterns may come from other domains that you have dealt with.
3. **Abstraction:** Define the bigger problem as combination of solution of smaller problems, each of which is solved the same way. Hide all the details and focus on the bigger picture.
4. **Algorithm Design:** Write step by step procedure

Algorithm:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Characteristic or features of an algorithm :

1. **Input**, means it has zero or more inputs, i.e., an algorithm can run without taking any input.
2. **Output**, means it has one or more outputs, i.e., an algorithm must produce atleast one output.
3. **Finiteness**, means it must always terminate after a finite number of steps.
4. **Definiteness**, means each step must be precisely defined and clear.
5. **Effectiveness**, means it is also generally expected to be effective.

An algorithm should be and **unambiguous** and independent of any programming code, i.e., **language independent**.

Algorithm to add two numbers entered by the user

Step 1: Start

Step 2: Read values num1 and num2.

Step 3: Add num1 and num2 and assign the result to sum.

sum ← num1 + num2

Step 4: Display sum

Step 5: Stop

Algorithm to find greatest of three numbers

```
Step 1: Start
Step 2: Read variables a,b and c.
Step 3: If a >= b and a>=c
    Assign a to the largest.
    Elif b>=a and b>=c
    Assign b to the largest.
    Else
    Assign c to the largest.
Step 4: Display largest
Step 5: Stop
```

Pseudo code:

- 1)Pseudo code is one of the method that is used to represent an algorithm
- 2)It is not written using specific rules, so it will not execute in computer
- 3)There are lots of formats for writing pseudo code and most of them follow languages as C,FORTRAN
- 4)Pseudo code can be read ,understood by programmers who are familiar with different programming languages.

Pseudo code for adding of two numbers

```
BEGIN
NUMBER s1, s2, sum
OUTPUT("Input number1:")
INPUT s1
OUTPUT("Input number2:")
INPUT s2
sum=s1+s2
OUTPUT sum
END
```

Difference between Algorithm and pseudo code

Algorithm	Pseudocode
1)Well defined sequence of steps that provide solution for a problem	1)One of the method that can be used to represent algorithm
2)It is written in natural language.	2)It is written in informal way,closely related to programming language
3)Any user can understand	3)Only programmer familiar with language can understand

Elements of a programming Language Python:

Token:

Tokens are the smallest unit of the program.

They are **Reserved words or Keywords,Identifiers,Literals,Operators**

Literals:

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

- 1) Numeric Literals
- 2) String literals
- 3) Boolean literals
- 4) Literal Collections

Numeric Literals

Binary literals
Decimal Literals
Octal Literal
Hexadecimal Literal
Float Literal
Complex Literal

program to demonstrate Numeric Literals

```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal
print(a, b, c, d)
```

#Float Literal

```
float_1 = 10.5
float_2 = 1.5
print(float_1, float_2)
```

#Complex Literal

```
x = 2+3.14j
print(x, x.imag,x.real)
```

String literals

A string literal is a sequence of characters surrounded by quotes. We can use both single, double, or triple quotes for a string.

```
S1='This is Python'
S2="This is Python"
S3=""""This is Python"""
print(S1)
print(S2)
print(S3)
```

Character Literals: Character literal is a single character surrounded by single or double quotes .

```
c1='a'
c2="a"
c3=""""a"""
print(c1)
```

```
print(c2)
print(c3)
```

Boolean literals:

A Boolean literal can have any of the two values: `True` or `False`.

```
x = True
y = False
a = True + 4
b = False + 10
print("x is", x)
print("y is", y)
print("a:", a)
print("b:", b)
```

Literal Collections

There are four different literal collections List literals, Tuple literals, Dict literals, and Set literals.

```
L=[1,2,3]
T=(1,2,3)
S={1,2,3}
d={'1':'v1','2':'v2'}
print("List:",L)
print("Tuple:",T)
print("Set:",S)
print("Dictionary:",d)
```

Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

Rules for writing identifiers

- ✓ 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 _.
- ✓ An identifier cannot start with a digit.
- ✓ Keywords cannot be used as identifiers
- ✓ We cannot use special symbols like `!`, `@`, `#`, `$`, `%` etc. in our identifier

Keywords :

- ✓ Keywords are reserved words in python
- ✓ We cannot use keyword as variable names, function names and any other identifier.
- ✓ They are used to define syntax and structure of python language.
- ✓ Keywords are case sensitive
- ✓ There are 33 keywords in python 3.7

Following are keywords available in Python:

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

Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1)Implicit Type Conversion

2)Explicit Type Conversion

1)Implicit Type Conversion

In this method Python automatically converts one data type(lower) to another data type(higher). This process doesn't need any user involvement.

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

#Program to demonstrate Implicit Conversion

```
a=10
b=5.5
c=a+b
print("The sum of a and b is ",c)
print("The datatype of c is",type(c))
```

2)Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type.

We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called **typecasting** because the user casts (changes) the data type of the objects.

In Type Casting, loss of data may occur as we enforce the object to a specific data type.

Syntax :

```
<required_datatype>(expression)
```

Write a program to demonstrate explicit conversion

```
a=5.5
b=10
c=a+b
print("Before explicit conversion",c)
print(type(c))
```



```
c=int(c)
print("After explicit conversion",c)
print(type(c))
```

Output:

Before explicit conversion 15.5

<class 'float'>

After explicit conversion 15

<class 'int'>

Variables

Variables are just names of memory locations. Variables are used to store and retrieve values from memory. Type of the variable is based on the value we assign to the variable.

Variables are containers for storing the data value.

But there are a few rules to follow when choosing name of variable

- ✓ Names must start with a letter or underscore and can be followed by any number of letters, digits, or underscores.
- ✓ Cannot start with a number
- ✓ It contain alpha-numeric characters & underscore.
- ✓ Variable names are case sensitive(age, Age, AGE are different).
- ✓ Names cannot be reserved words or keywords

Syntax :

Variablename = value

Eg:

- ✓ x = 10
 - ✓ x='python'
 - ✓ z=6.1
- Assign value to multiple variable
- ✓ x,y,z=10,20,30
print(x)
print(y)
print(z)
 - ✓ x,y,z=10.0,'Ayaan',6
print(x)
print(y)
print(z)

Identify the following are valid or invalid???

A=10 ---- **valid**

_a=11---- **valid**

1a=12---- **Invalid**

a@=13 ---- **Invalid**

a1_2=14---- **valid**

Input/Output Statements:

input(): The `input()` function allows user to give input.

Syntax

`input(prompt)`

prompt- It can be empty or A String.

Eg:

- ❖ `a=int(input())`
- ❖ `b=float(input())`
- ❖ `c=input()`
- ❖ `a=int(input("enter the value of a"))`
- ❖ `print('Enter your name:')`
`x = input()`
`print('Hello, ' + x)`
- ❖ `x = input('Enter your name:')`
`print('Hello, ' + x)`

Multiple input at a time:

```
a,b=input("enter firstname"),input("enter lastname")
```

```
print(a)
```

```
print(b)
```

Output(): This function is used to output the data or print the result on monitor.

Eg:

- ❖ `print('Welcome')`
- ❖ `print("Python")`
- ❖ `print("""Hello""")`
- ❖ `print(1,2,3,4)`
- ❖ `print(1,2,3,sep='*')`
- ❖ `print(1,2,3,4,sep='*',end='&')`
- ❖ **Output formatting:**

Sometimes we would like to format our output to make it look attractive.

```
x = 5
```

```
y = 10
print('The value of x is {} and y is {}'.format(x,y))
```

Output:

The value of x is 5 and y is 10

Basic Data Types(Numbers also included)

Every value in Python has a datatype.

Python has the following data types/ objects built-in by default, in these categories:

<u>Text Type:</u>	<code>str</code>
<u>Numeric Types:</u>	<code>int</code> , <code>float</code> , <code>complex</code>
<u>Sequence Types:</u>	<code>list</code> , <code>tuple</code> , <code>range</code>
<u>Mapping Type:</u>	<code>dict</code>
<u>Set Types:</u>	<code>set</code> , <code>frozenset</code>
<u>Boolean Type:</u>	<code>bool</code>

type(): To know the data type of any object by using the `type()` function

```
✓ x=5
  print(type(x))
✓ x=10.0
  print(type(x))
✓ x='hello'
  print(type(x))
```

Python Numbers:

They are categorized into mainly three types. They are int,float and complex.

Int: It consists of positive and negative numbers without decimals.They are of unlimited length.

```
✓ x=1
  print(x)
  print(type(x))
✓ y=123456789012345678901234567890
  print(y)
  print(type(y))
✓ z=-234
  print(z)
  print(type(z))
```

float: It consists of positive and negative numbers containing decimals numbers

```
✓ x=1.10
  print(x)
  print(type(x))
✓ y=0.123
  print(y)
  print(type(y))
✓ z=-0.123
  print(z)
  print(type(z))
```

complex numbers: Complex numbers in python are in the form of $a + bj$

```
✓ x=3+5j
  print(x)
  print(type(x))
✓ y=5j
  print(y)
  print(type(y))
✓ z=-5j
  print(z)
  print(type(z))
```

Booleans:

Python provides Boolean datatype called bool with values True & False. True represents integer 1 and False represents integer 0. bool is a sub class of integer class. Following are examples on Booleans:

Eg:

```
✓ print(True+1)
✓ print(False*99)
✓ print(True+22)
✓ print(False*2)
✓ a=True
  print(type(a))
✓ b=False
  print(type(b))
✓ c=true
  print(type(c))
```

Strings:

A string is a sequence of characters.

Strings are used to store textual information. It is represented by single quote or double quote.

Eg:

- ✓ 'B.V.Prasanthi'
- ✓ "A"
- ✓ "123"
- ✓ a='python'
print(a)
print(type(a))
- ✓ b='12.345'
print(b)
print(type(b))
- ✓ c=str(input("enter c"))
print(c)
print(type(c))
- ✓ d=input()
print(d)
print(type(d))

Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

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

Python Arithmetic Operators

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

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Examples:

#Arithmetic Operators

```
x=int(input("Enter the value of x"))
y=int(input("Enter the values of y"))
print("Sum=",x+y)
print("Sub=",x-y)
print("Product=",x*y)
print("Division",x/y)
print("Modulus=",x%y)
print("Exponentiation=",x**y)
print("Floor division=",x//y)
```

```
guest-gUh7cf@slave03:~$ gedit arithmetic.py
guest-gUh7cf@slave03:~$ python3 arithmetic.py
Enter the value of x3
enter the values of y2
Sum= 5
Sub= 1
Product= 6
Division 1.5
Modulus= 1
Exponentiation= 9
Floor division= 1
```

Python Comparison Operators

Comparison operators are used to compare two values:

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

#Comparison Operators

```
x=int(input("Enter the values of x:"))
y=int(input("Enter the values of y:"))
```

```

print(x==y)
print(x!=y)
print(x>y)
print(x<y)
print(x>=y)
print(x<=y)

```

```

guest-gUh7cf@slave03:~$ gedit comparison.py
guest-gUh7cf@slave03:~$ python3 comparison.py
Enter the values of x:3
Enter the values of y:6
False
True
False
True
False
True

```

Python Logical Operators

Logical operators are used to combine conditional statements:

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

p	q	$p \wedge q$	p	q	$p \vee q$
T	T	T	T	T	T
T	F	F	T	F	T
F	T	F	F	T	T
F	F	F	F	F	F

#Logical operators

```

x=int(input("Enter the value of x:"))
print(x<5 and x<10)
print(x<5 or x<4)
print(not(x>10))

```

```

guest-gUh7cf@slave03:~$ gedit logical.py

```

```
guest-gUh7cf@slave03:~$ python3 logical.py
```

```
Enter the value of x:6
```

```
False
```

```
False
```

```
True
```

Python Identity Operators

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

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

#identity operators

```
x=int(input("Enter the value of x:"))
```

```
y=int(input("Enter the value of y:"))
```

```
z=x
```

```
print(x is y)
```

```
print(x is not y)
```

```
print(x is z)
```

```
guest-gUh7cf@slave03:~$ gedit identity.py
```

```
guest-gUh7cf@slave03:~$ python3 identity.py
```

```
Enter the value of x:5
```

```
Enter the value of y:9
```

```
False
```

```
True
```

```
True
```

Python Membership Operators

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

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

#Membership operators


```
x=[1,2,3,4,5]
y=int(input("enter the value of y:"))
print(y in x)
print(y not in x)
```

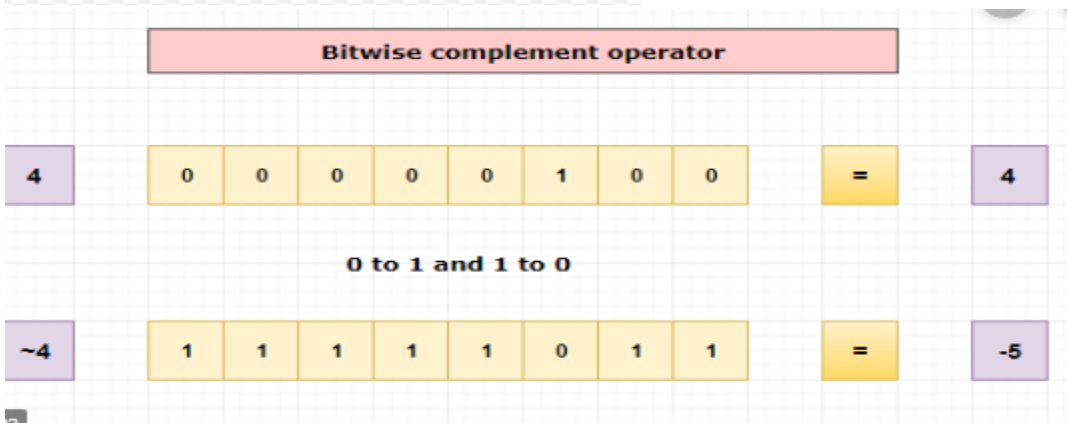
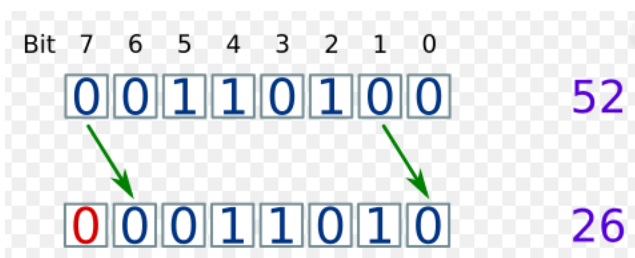
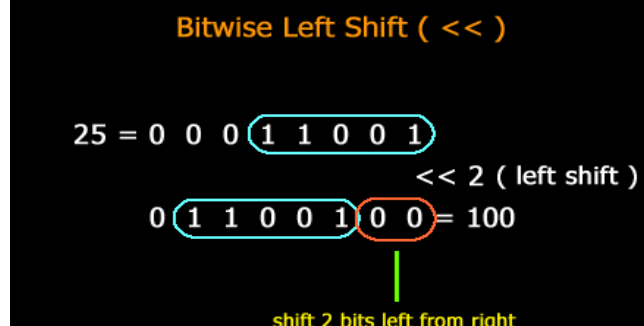
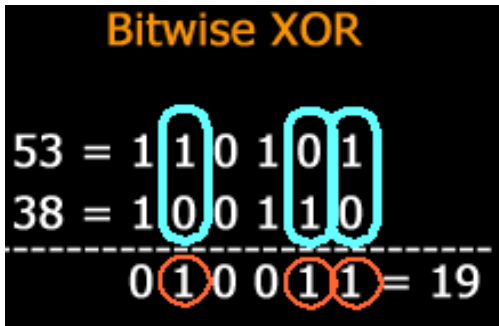
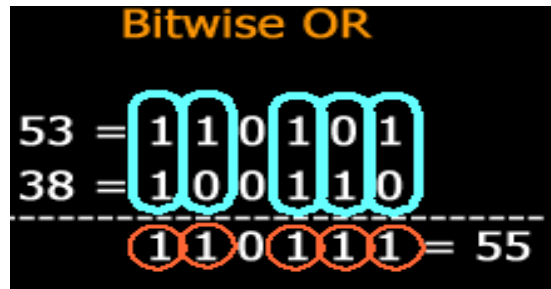
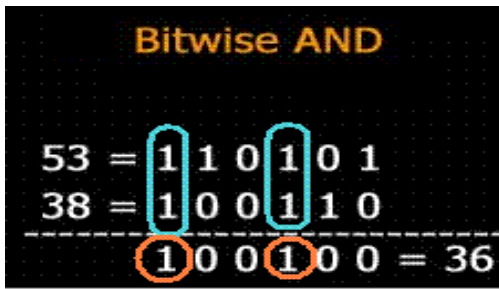
```
guest-gUh7cf@slave03:~$ gedit membership.py
guest-gUh7cf@slave03:~$ python3 membership.py
enter the value of y:5
True
False
```

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

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

a	b	a&b	a b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0



#program on Bitwise Operator

```
x=int(input("enter the value of x:"))
y=int(input("enter the value of y:"))
print(x&y)
print(x | y)
print(x ^ y)
print(~x)
print(x<<1)
print(x>>1)
```

```
guest-gUh7cf@slave03:~$ gedit bitwise.py
guest-gUh7cf@slave03:~$ python3 bitwise.py
enter the value of x:5
enter the value of y:3
1
7
```

6
-6
10
2

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

write a program to demonstrate assignment operators

```
x=int(input("enter the value of x:"))
x += 3          #same as x=x+3
print("x=",x)
x -= 3          #same as x=x-3
print("x=",x)
x *= 3          #same as x=x*3
print("x=",x)
x /= 3          #same as x=x/3
print("x=",x)
x %= 3          #same as x=x%3
print("x=",x)
x //= 3         #same as x=x//3
print("x=",x)
x **= 3         #same as x=x**3
print("x=",x)
```

guest-gUh7cf@slave03:~\$ gedit assignment.py

```
guest-gUh7cf@slave03:~$ python3 assignment.py
```

```
enter the value of x:5
```

```
x= 8
```

```
x= 5
```

```
x= 15
```

```
x= 5.0
```

```
x= 2.0
```

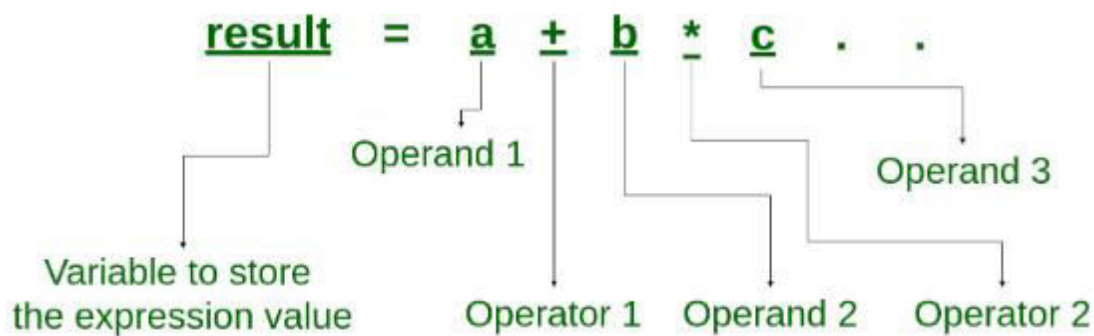
```
x= 0.0
```

```
x= 0.0
```

Expressions and order of evaluations(precedence)

Expression: An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

What is an Expression?



Eg:

✓ `10 - 4 * 2`

✓ `a*b/c`

✓ `a*b+2`

Python has well-defined rules for specifying the **order** in which the operators in an **expression** are **evaluated** when the **expression** has several operators.

The operator precedence in Python are listed in the following table. It is in descending order, upper group has higher precedence than the lower ones.

Operators	Meaning
()	Parentheses
**	Exponent

+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Operator precedence rule in Python

Eg:

- ✓ `print(10 - 4 * 2)`
- ✓ `print((10 - 4) * 2)`
- ✓ `print(3+2-1)`

Associativity of Python Operators

We can see in the above table that more than one operator exists in the same group. These operators have the same precedence.

When two operators have the same precedence, associativity helps to determine which the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. Almost all the operators have left-to-right associativity.

For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, left one is evaluates first.

- ✓ `print(5 * 2 // 3)` L to R

Exponent operator `**` has right-to-left associativity in Python.

- ✓ # Right-left associativity of ** exponent operator
Output: 512
print(2 ** 3 ** 2)
- ✓ # Shows the right-left associativity of **
Output: 64
print((2 ** 3) ** 2)

Control structures in Python - conditionals and loops

Conditionals:

Decision Making Statements/Selection/Conditional Statements

- ❖ if statement
- ❖ if else statements
- ❖ elif ladder
- ❖ Nested if statements

if statement

Decision making is required when we want to execute a code only if a certain condition is satisfied.

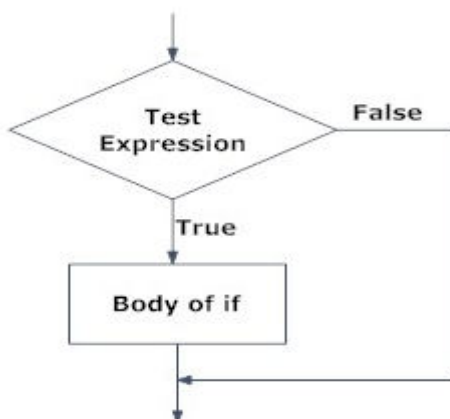
Python if Statement Syntax

```
If testexpression:  
    statement(s)
```

Here, 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 if Statement Flowchart



Eg

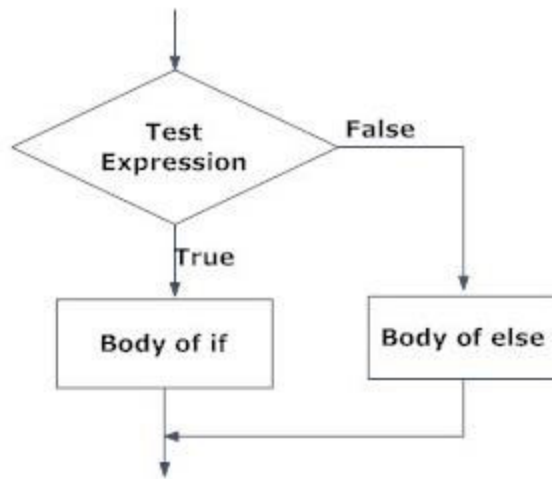
```
num = int(input("Enter the value of num"))  
if num > 0:  
    print(num, "is a positive number.")
```

if...else Statement

Python if..else Flowchart

Syntax of if...else

```
if test expression:  
    Body of if  
else:  
    Body of else
```



The if..else statement evaluates **test expression** and will execute body of if only when test condition is True.

If the condition is False, body of else is executed. Indentation is used to separate the blocks.

Program checks if the number is positive or negative

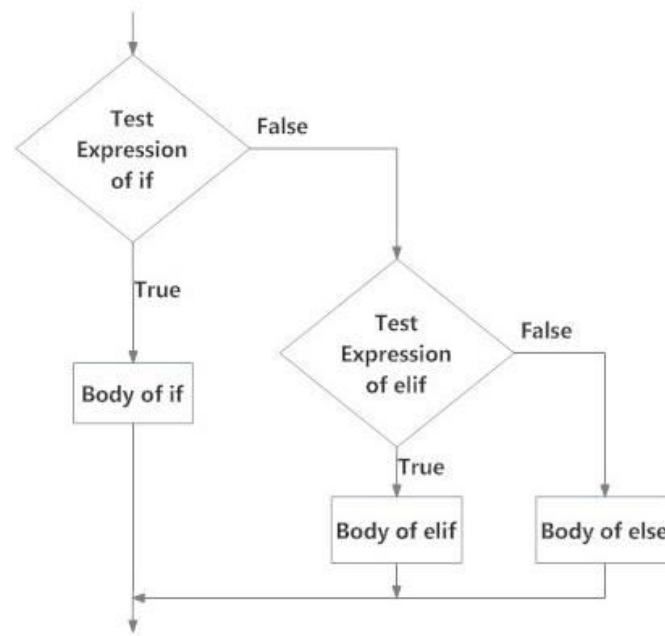
```
num = int(input("Enter the value of num"))  
if num >= 0:  
    print("Positive or Zero")  
else:  
    print("Negative number")
```

Elif ladder

Flowchart of if...elif...else

Syntax of if...elif...else

```
if test expression:  
    Body of if  
elif test expression:  
    Body of elif  
else:  
    Body of else
```



The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed.

#program on elif ladder

```
num = float(input("Enter the value of num"))  
if num > 0:  
    print("Positive number")  
elif num == 0:  
    print("Zero")  
else:  
    print("Negative number")
```

Python program to find the largest number among the three input numbers

```
a = int(input("Enter the value of a"))  
b = int(input("Enter the value of b"))  
c = int(input("Enter the value of c"))  
if (a >= b) and (a >= c):  
    largest = a  
elif (b >= a) and (b >= c):  
    largest = b  
else:  
    largest = c  
print("The largest number is",largest)
```

Nested if Statement

If statement in another if is called as nested if statements.

Syntax:

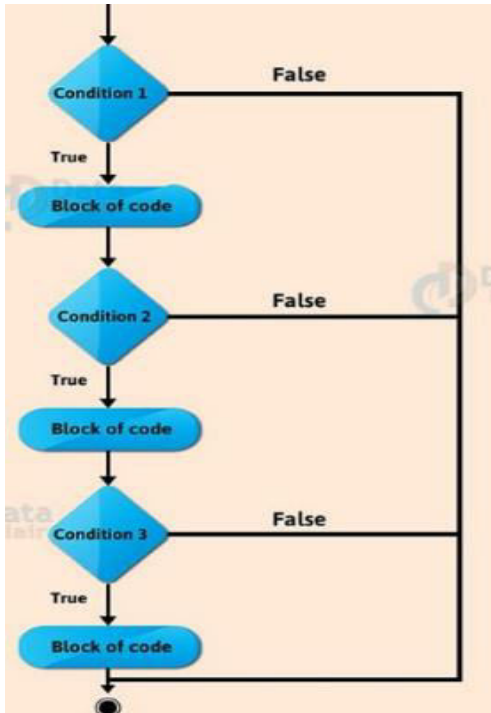
```
if cond1:  
    if cond2:  
        if cond3:  
            statements
```

or


```

if cond1:
    if cond2:
        statements
    else:
        statements
else:
    if cond3:
        statements
    else:
        statements

```



#program on Nested if

```

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")

```

Repetition Statements(loops):

- ❖ for loop
- ❖ while loop

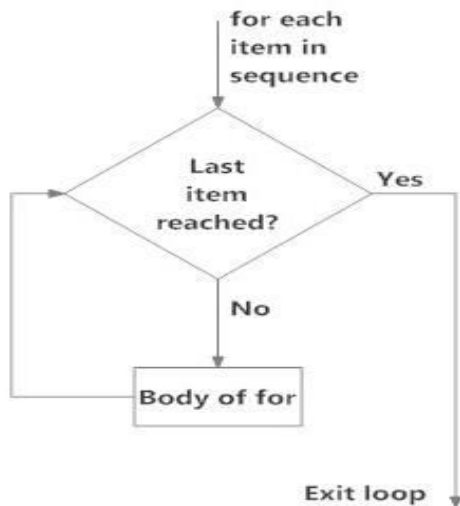
for loop

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 of for Loop

```
for val in sequence:  
    Body of for
```

Flow chart



Here, `val` 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. The body of for loop is separated from the rest of the code using indentation.

Eg:

```
for i in range(3):  
    print(i)
```

Syntax for range:

```
range(end)           #0 to end-1  
range(start,end)     #start to end-1  
range(start,end,step)
```

Few more example on for loop:

```
for i in range(0,3):  
    print(i)
```

```
for i in range(1,3):  
    print(i)
```

```
#print odd numbers  
for i in range(1,10,2):  
    print(i)
```

```
#print even numbers  
for i in range(0,11,2):  
    print(i)
```

Else in For Loop

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

```
else:  
    print("Finally finished!")
```

Looping Through a String

```
for i in "ayaan":  
    print(i)
```

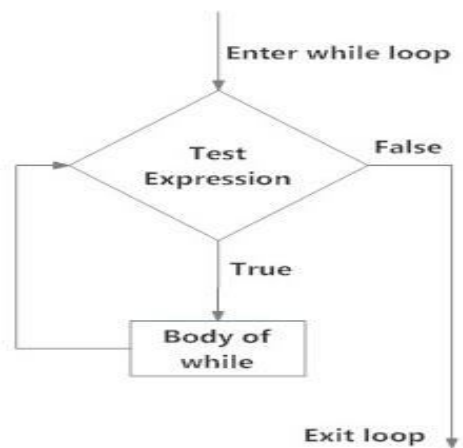
while Loop

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

Syntax of while Loop

```
while test_expression:  
    Body of while
```

Flow chart



Eg:

```
#print numbers from 1 to 5
```

```
i = 1
```

```
while i < 6:
```

```
    print(i)
```

```
    i += 1
```

```
#print odd numbers 1,3,5,7,9
```

```
i=1
```

```
while i<10:
```

```
    print(i)
```

```
    i=i+2
```

```
#print even numbers 0,2,4,6,8,10
```

```
i=0
```

```
while i<=10:
```

```
    print(i)
```

```
    i=i+2
```

Break and continue:

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 checking test expression.

The break and continue statements are used in these cases.

Python break statement

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

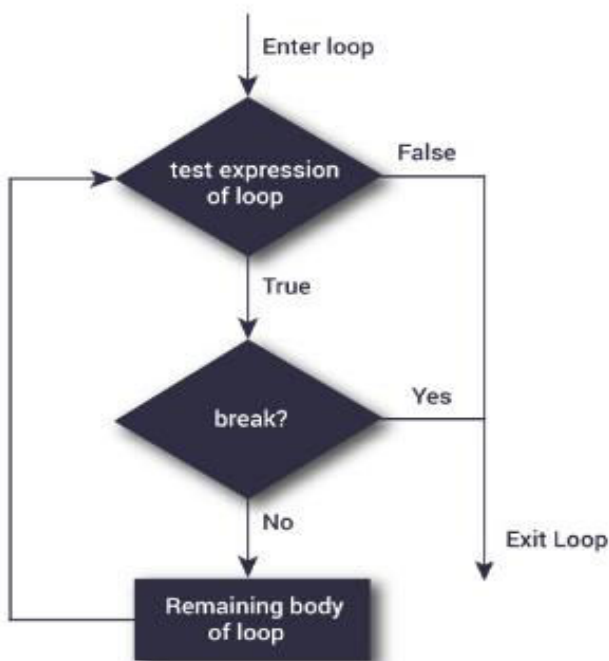
If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Syntax of break

break

```
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
```



#using of break in for loop(output: 1 2)

```
for i in range(1,5):
```

```
    if i==3:
```

```
        break
```

```
    print(i)
```

#using break in while loop (output: 1 2)

i=1

while(i<5):

if i==3:

break

print(i)

i=i+1

Python continue statement

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

Syntax of Continue

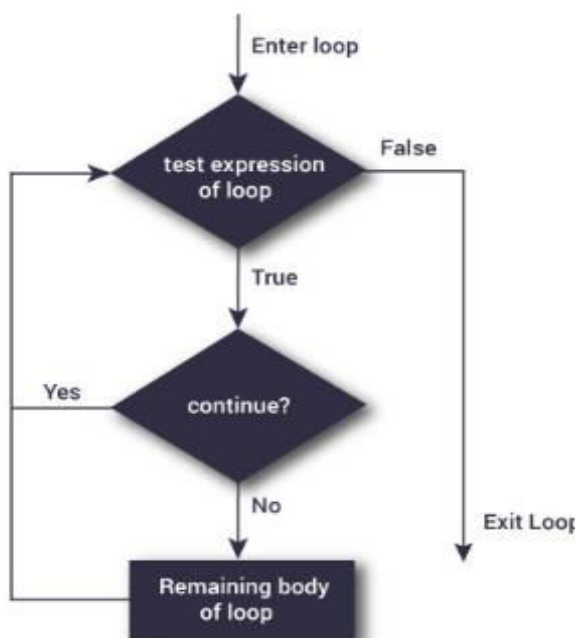
```
continue
```

```
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
```



#using continue in for loop (output:1 2 4 5)

for i in range(1,5):

if i==3:

continue

print(i)

```
#using continue in while loop (output:1 2 4 5)
i = 0
while i < 4:
    i=i+1
    if i == 3:
        continue
    print(i)
```