TOKENS (16 Mark)

Token: Smallest individual unit in a program is known as token.There are five types of token in python:

1. Keyword
2. Identifier
3. Literal
4. Operators
5. Punctuators

1. **Keyword**: Reserved words in the library of a language. There are 33 keywords inpython.

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

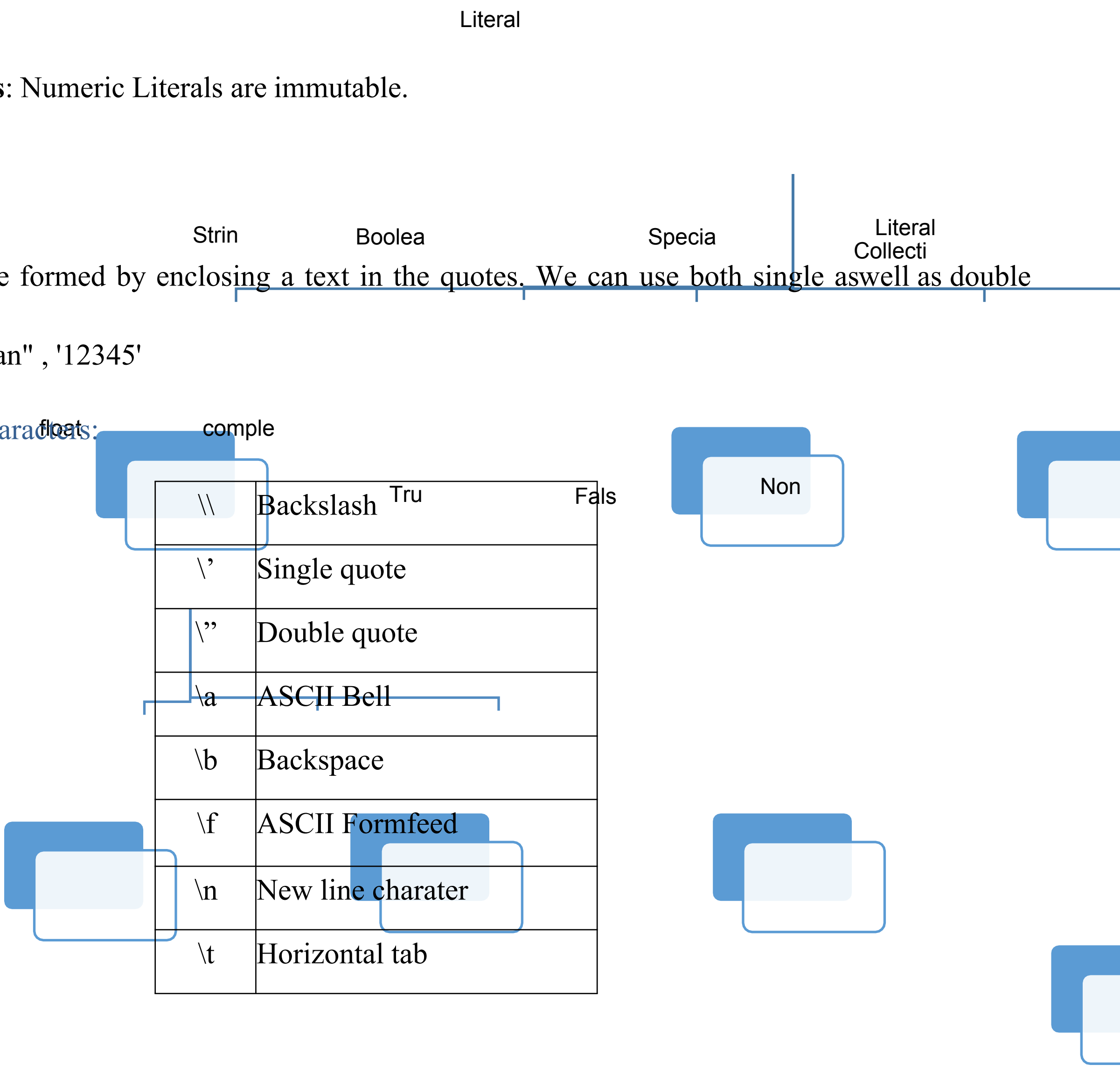
All the keywords are in lowercase except 03 keywords (True, False, None).

2. **Identifier:** The name given by the user to the entities like variable name, class-name,function-name etc.

Rules for identifiers:

- It can be a combination of letters in lowercase (a to z) or uppercase (A to Z) ordigits (0 to 9) or an underscore.
- It cannot start with a digit.
- Keywords cannot be used as an identifier.
- We cannot use special symbols like !, @, #, $, %, + etc. in identifier.
- _ (underscore) can be used in identifier.
- Commas or blank spaces are not allowed within an identifier.

3. **Literal:** Literals are the constant value. Literals can be defined as a data that is given ina variable or constant.



A. Numeric literals: Numeric Literals are immutable.

Eg.5, 6.7, 6+9j

B. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single aswell as double quotes for a String.

E.g"Aman" , '12345'

Escape sequence characters:

\\	Backslash
\’	Single quote
\”	Double quote
\a	ASCII Bell
\b	Backspace
\f	ASCII Formfeed
\n	New line charater
\t	Horizontal tab

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

D. Special literals: Python contains one special literal i.e. **None**.

None is used to specify to that field that is not created. It is also used for end of lists inPython.

E. Literal Collections: Collections such as tuples, lists and Dictionary are used in Python.

4. **Operators:** An operator performs the operation on operands. Basically there are twotypes of operators in python according to number of operands:
   A. Unary Operator
   B. Binary Operator

A. Unary Operator: Performs the operation on one operand.
Example:

+	Unary plus
-	Unary minus
~	Bitwise complementnot
	Logical negation

B. **Binary Operator**: Performs operation on two operands.

5. **Separator or punctuator : , ; , ( ), { }, [ ]**

2.2 Mantissa and Exponent Form:

A real number in exponent form has two parts:

- mantissa
- exponent

Mantissa : It must be either an integer or a proper real constant.

Exponent : It must be an integer. Represented by a letter E or e followed by integer value.

Valid Exponent form	**Invalid Exponent form**
123E05	2.3E (No digit specified for exponent)
1.23E07	0.24E3.2 (Exponent cannot have fractional part)
0.123E08	23,455E03 (No comma allowed)
123.0E08	
123E+8	
1230E04	
-0.123E-3	
163.E4	
.34E-2	
4.E3	

2.3 Basic terms of a Python Programs:

A. Blocks and Indentation

B. Statements

C. Expressions

D. Comments

A. Blocks and Indentation:

- Python provides no braces to indicate blocks of code for class and function definition orflow control.
- Maximum line length should be maximum 79 characters.
- Blocks of code are denoted by line indentation, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the blockmust be indented the same amount.

  for example –if

```
True:
        print(“True”)
else:
        print(“False”)
```

B. Statements

A line which has the instructions or expressions.

C. Expressions:

A legal combination of symbols and values that produce a result. Generally it produces a value.

D. Comments: Comments are not executed. Comments explain a program and make a program understandable and readable. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

There are two types of comments in python:

i. Single line comment

ii. Multi-line comment

iii. Python Doc string

i. **Single line comment**: This type of comments start in a line and when a line ends, it is automatically ends. Single line comment starts with # symbol.

Example: if a>b: # Relational operator compare two values

ii. **Multi-Line comment**: Multiline comments can be written in more than one lines. Triple quoted ‘ ’ ’ or “ ” ”) multi-line comments may be used in python. It is also known as **docstring**.

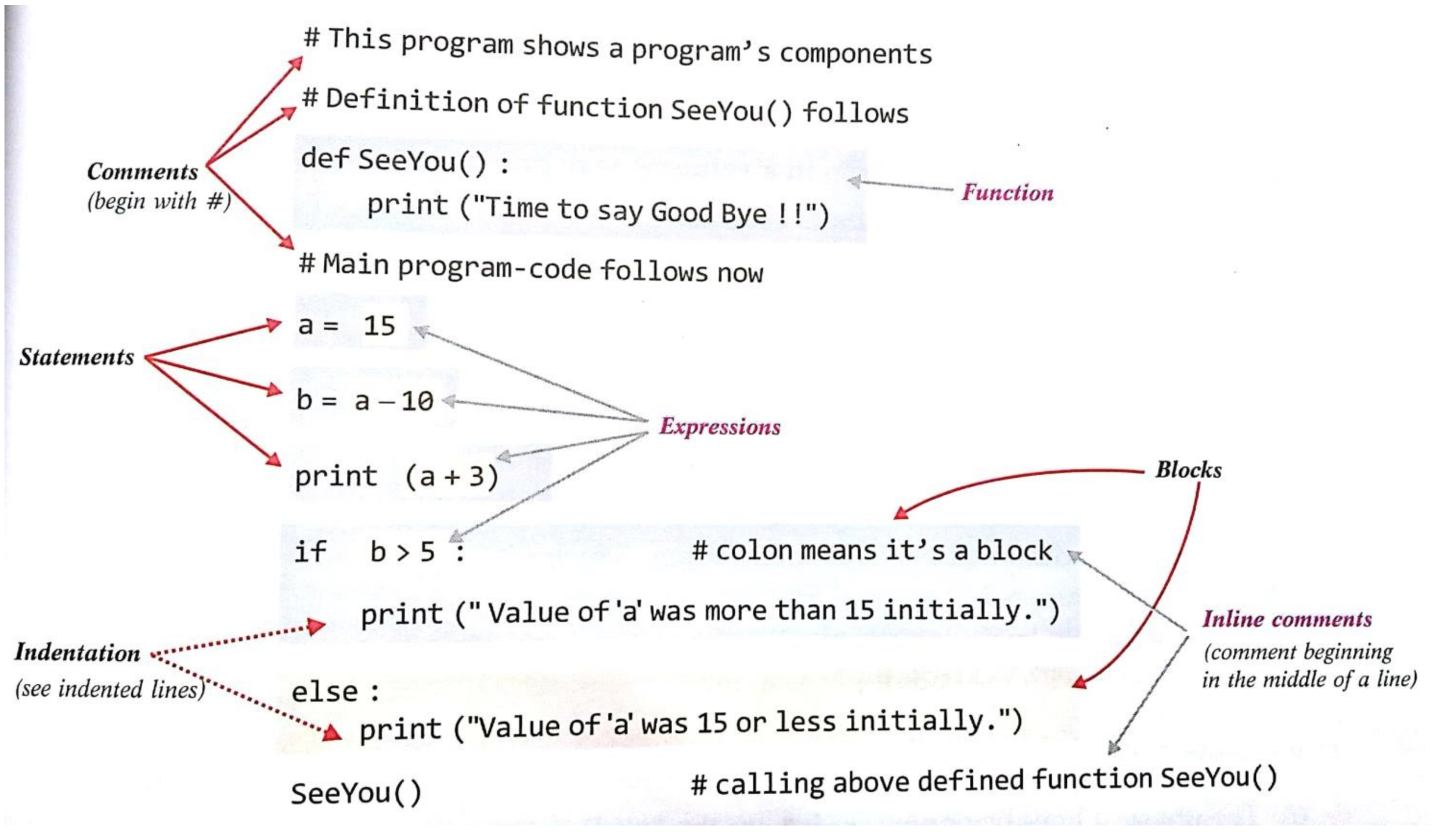
Example:

‘’’ This program will calculate the average of 10 values.

First find the sum of 10 values

and divide the sum by number of values

‘’’



```
# This program shows a program’s components
# Definition of function SeeYou() follows
def SeeYou() :
    print ("Time to say Good Bye !!")
# Main program-code follows now
a =  15
b = a – 10
print  (a + 3)
if   b > 5  :                    # colon means it’s a block
    print (" Value of 'a' was more than 15 initially.")
else :
    print ("Value of 'a' was 15 or less initially.")
SeeYou()                         # calling above defined function SeeYou()
```

Python Docstring

Python docstring is the string literals with triple quotes that are appeared right after the function. It is used to associate documentation that has been written with Python modules, functions, classes, and methods. It is added right below the functions, modules, or classes to describe what they do. In Python, the docstring is then made available via the __doc__ attribute.

Example:

```
def multiply(a, b):
    """Multiplies the value of a and b"""
    return a*b


# Print the docstring of multiply function
print(multiply.__doc__)
```

Output:
Multiplies the value of a and b

Multiple Statements on a Single Line:

The semicolon (;) allows multiple statements on the single line given that neither statementstarts a new code block.

Example:-

```
x=5; print("Value =" x)
```

2.4 Variable/Label in Python:

Definition: Named location that refers to a value and whose value can be used and processedduring program execution.

Variables in python do not have fixed locations. The location they refer to changes every timetheir values change.

Creating a variable:

A variable is created the moment you first assign a value to it.Example:

```
x = 5

y = "hello"
```

Variables do not need to be declared with any particular type and can even change type afterthey have been set. It is known as dynamic Typing.

```
x = 4 # x is of type int
x = "python" # x is now of type str
print(x)
```

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscore (A-z, 0-9,and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Python allows assign a single value to multiple variables.Example:

x = y = z = 5

You can also assign multiple values to multiple variables. For example −

x , y , z = 4, 5, “python”

4 is assigned to x, 5 is assigned to y and string “python” assigned to variable z respectively.x=12

y=14

x,y=y,x

print(x,y)

Now the result will be14

12

Python Operators (16 Mark)

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

Basic Operators in Python:

- **i.** Arithmetic Operators
- **ii.** Relational Operator
- **iii.** Logical Operators
- **iv.** Bitwise operators
- **v.** Assignment Operators
- **vi.** Other Special Operators
  - Identity Operators
  - Membership operators

i. **Arithmetic Operators**: To perform mathematical operations.

OPERATOR	**NAME**	**SYNTAX**	**RESULT (X=14, Y=4)**
+	Addition	x + y	18
_	Subtraction	x – y	10
*	Multiplication	x * y	56
/	Division (float)	x / y	3.5

//	Division (floor)	x // y	3
%	Modulus	x % y	2
**	Exponent	x**y	38416

Example:

```
>>>x= -5
>>>x**2
>>> -25
```

ii. **Relational Operators:** Relational operators compare the values. It either returns **True** or **False** according to the condition.

OPERATOR	NAME	SYNTAX	RESULT (IF X=16, Y=42)
>	Greater than	x > y	False
<	Less than	x < y	True
==	Equal to	x == y	False
!=	Not equal to	x != y	True
>=	Greater than or equal to	x >= y	False
<=	Less than or equal to	x <= y	True

iii. **Logical operators:** Logical operators perform **Logical AND**, **Logical OR** and **LogicalNOT** operations.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

Examples of Logical Operator:

The **and** operator: The and operator works in two ways:

a. Relational expressions as operands
b. numbers or strings or lists as operands

a. Relational expressions as operands:

X	Y	X and Y
False	False	False
False	True	False
True	False	False
True	True	True

```
 >>> 5>8 and 7>3
False
>>> (4==4) and (7==7)
True
```

b. numbers or strings or lists as operands:

In an expression X and Y, if first operand has **false value**, **then return first operand X** as aresult, otherwise returns Y.

X	Y	X and Y
false	false	X
false	true	X
true	false	Y
true	true	Y

```
>>>0 and 0
0
>>>0 and 6
0
>>>'a' and 'n'
'n'
>>>6>9 and 'c'+9>5
False
```

# and operator will test the second operand only if the first operand # is true, otherwise ignores it, even if the second operand is wrong

The **or** operator: The or operator works in two ways:

a. Relational expressions as operands
b. numbers or strings or lists as operands

a. Relational expressions as operands:

X	Y	X or Y
False	False	False
False	True	True
True	False	True
True	True	True

```
 >>> 5>8 or 7>3
True
>>> (4==4) or (7==7)
True
```

b. numbers or strings or lists as operands:
In an expression X or Y, if first operand has **true value**, **then return first operand X** as aresult, otherwise returns Y.

X	**Y**	**X or Y**
false	false	Y
false	true	Y
true	false	X
true	true	X

```
>>>0 or 0
0
>>>0 or 6
6
>>>'a' or 'n'
'a'
>>>6<9 or 'c'+9>5        # or operator will test the second operand only if the first operand True        #
is false, otherwise ignores it, even if the second operand is wrong
```

The **not** operator:

```
>>>not 6
False
>>>not 0
True
>>>not -7
False
```

Chained Comparison Operators:

```
>>> 4<5>3          is equivalent to          >>> 4<5 and 5>3
True                                           True
```

iv. **Bitwise operators:** Bitwise operators acts on bits and performs bit by bit operation.

OPERATOR	**DESCRIPTION**	**SYNTAX**
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Examples:

Let

a = 10

b = 4 print(a

& b)print(a |

b) print(~a)

print(a ^ b)

print(a >> 2)

print(a << 2)

```
Output:
0
14
-11
14
2
40
```

v. **Assignment operators:** Assignment operators are used to assign values to the variables.

OPERATOR	DESCRIPTION	SYNTAX
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right side operand with left side operand andthen assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and thenassign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and thenassign to left operand	a*=b a=a*b
/=	Divide AND: Divide left operand with right operand and thenassign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands andassign result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand andthen assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value usingoperands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to leftoperand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to leftoperand	a|=b a=a|b

^=	Performs Bitwise xOR on operands and assign value to leftoperand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to leftoperand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to leftoperand	a <<=b a=a << b

vi. **Other Special operators:** There are some special type of operators like-

a. **Identity operators- is** and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equaldoes not imply that they are identical.

is	True if the operands are identical
is not	True if the operands are not identical

Example:

```
Let   a1
= 3
b1 = 3
a2 = 'PythonProgramming'b2 =
'PythonProgramming'    a3    =
[1,2,3]
b3 = [1,2,3]

print(a1 is not b1)
print(a2 is b2)              # Output is False, since lists are mutable.
print(a3 is b3)
```

Output:

```
False True

False
```

Example:

```
>>>str1= "Hello"

>>>str2=input("Enter a String :")Enter

a String : Hello

>>>str1==str2         # compares values of stringTrue

>>>str1 is str2            # checks if two address refer to the same memory addressFalse
```

b. **Membership operators-** **in** and **not in** are the membership operators; used to testwhether a value or variable is in a sequence.

in	True if value is found in the sequence
not in	True if value is not found in the sequence

Example:

Let
x = 'Digital India'y
= {3:'a',4:'b'}

```
print('D' in x)

print('digital' not in x)

print('Digital' not in x)

print(3 in y)

print('b' in y)
```

Output:

```
True

True

False

True

False
```

3.1 Operator Precedence and Associativity:

Operator Precedence: It describes the order in which operations are performed when anexpression is evaluated. Operators with higher **precedence** perform the operation first.

Operator Associativity: whenever two or more operators have the same precedence, thenassociativity defines the order of operations.

Operator	Description	Associativity	Precedence
( ), { }	Parentheses (grouping)	Left to Right	
f(args…)	Function call	Left to Right	
x[index:index]	Slicing	Left to Right	
x[index]	Subscription	Left to Right	
**	Exponent	**Right to Left**	
~x	Bitwise not	Left to Right	
+x, -x	Positive, negative	Left to Right	
*, /, %	Product, division, remainder	Left to Right	
+, –	Addition, subtraction	Left to Right	
<<, >>	Shifts left/right	Left to Right	

&	Bitwise AND	Left to Right	
^	Bitwise XOR	Left to Right	
|	Bitwise OR	Left to Right	
<=, <, >, >=	Comparisons	Left to Right	
=, %=, /=, +=	Assignment		
is, is not	Identity		
in, not in	Membership		
not	Boolean NOT	Left to Right	
and	Boolean AND	Left to Right	
or	Boolean OR	Left to Right	
lambda	Lambda expression	Left to Right	

Expressions
An expression is a combination of values, variables, operators, and calls to functions. If you type an expression at the Python prompt, the interpreter evaluates it and displays the result:

```
>>> 1 + 1=2
```

CONTROL FLOW STATEMENTS (16 MARKS)

A program's **control flow** is the order in which the program's code executes.

The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Python has *three* types of control structures:

- **Sequential** - default mode
- **Selection** - used for decisions and branching
- **Repetition** - used for looping, i.e., repeating a piece of code multiple times.

1. Sequential

Sequential statements are a set of statements whose execution process happens in a sequence. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

```
# This is a Sequential statement

a=20

b=10

c=a-b

print("Subtraction is : ",c)
```

2. Selection/Decision control statements

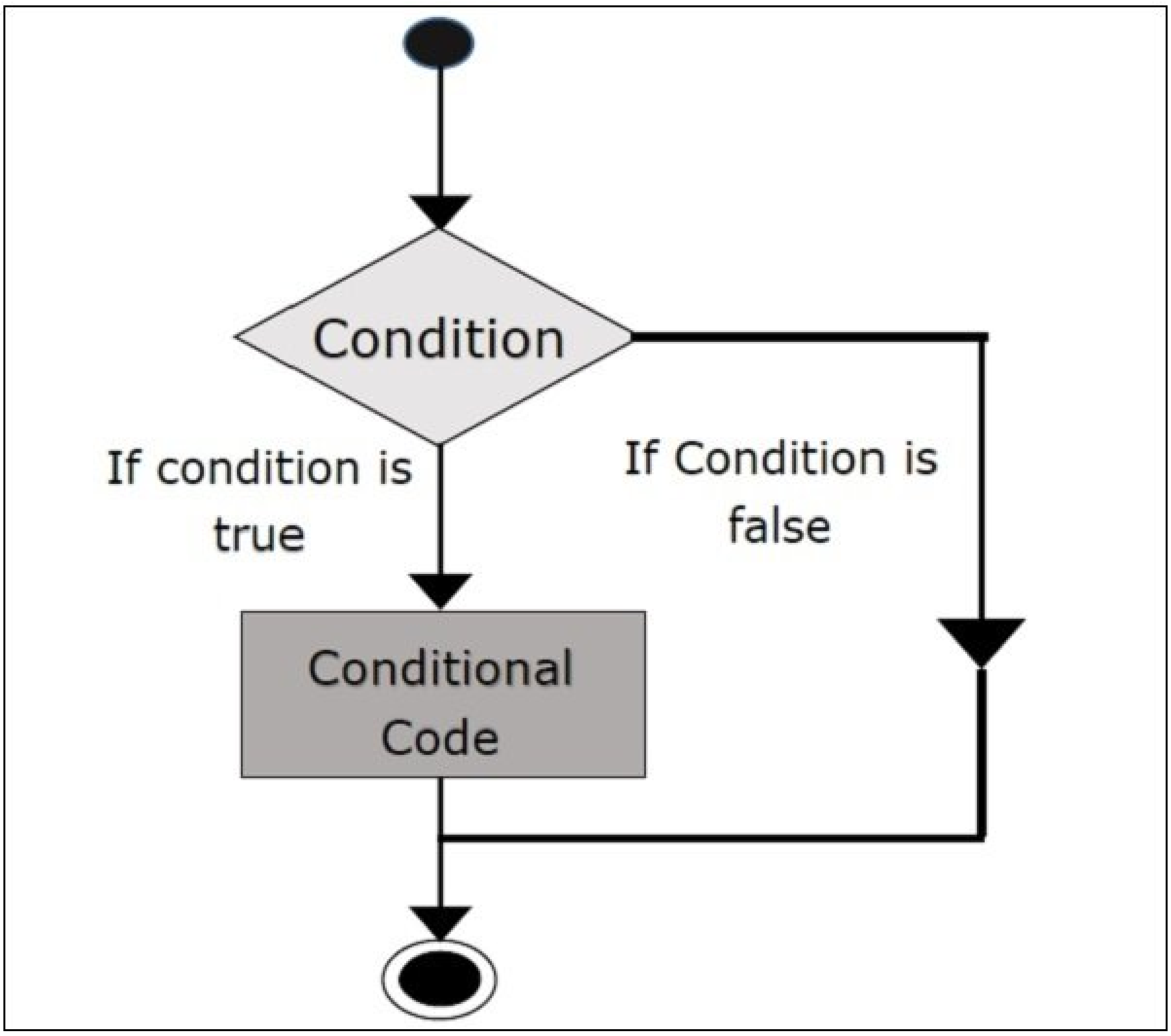
In Python, the selection statements are also known as *Decision control statements* or *branching statements*.

The selection statement allows a program to test several conditions and execute instructions based on which condition is true.

Some Decision Control Statements are:

- Simple if
- if-else
- nested if
- if-elif-else

Simple if: *If statements* are control flow statements that help us to run a particular code, but only when a certain condition is met or satisfied. A *simple if* only has one condition to check.



Syntax:
if (<condition-1>):
    statement-1
    statement-2
statement-p

Example:

```
n = 10

if n % 2 == 0:

  print("n is an even number")
```

if-else: The *if-else statement* evaluates the condition and will execute the body of if if the test condition is True, but if the condition is False, then the body of else is executed.

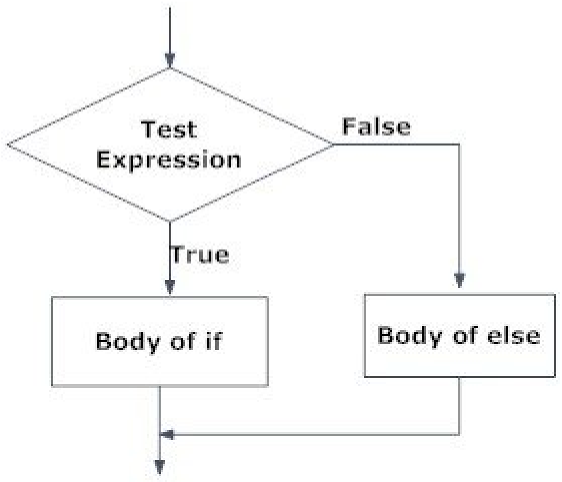


Fig: Operation of if...else statement

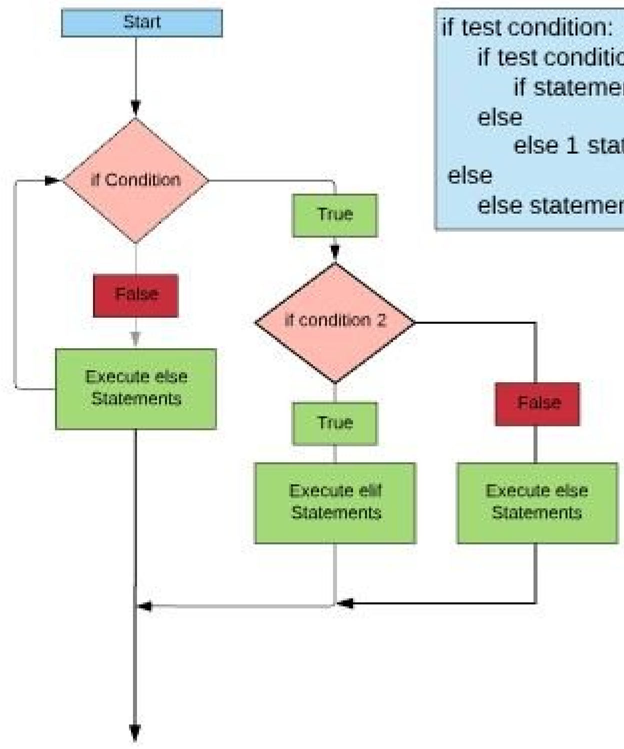
Syntax:

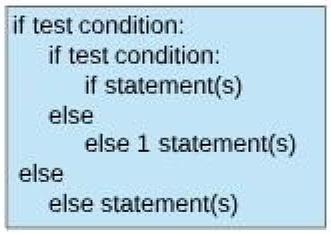
```
if(<condition-1>):
    statement-1
    statement-2
else:
    statement-3
    statement-4
```

Example:

```
n = 5

if n % 2 == 0:

   print("n is even")

else:

   print("n is odd")
```

nested if: *Nested if statements* are an if statement inside another if statement.





```
if test condition:
    if test condition:
        if statement(s)
    else
        else 1 statement(s)
else
    else statement(s)
```

Syntax:

```
f(<condition-1>):
    statement-block-1
elif(<condition-2>):
    statement-block-2
elif(<condition-3>):
    statement-block-3
.....................
.....................
elif(<condition-n>):
    statement-block-n
[else:
    statement-block-p]
```

Example:

```
a = 5

b = 10

c = 15

if a > b:

  if a > c:

    print("a value is big")

  else:

     print("c value is big")
```

```
elif b > c:
   print("b value is big")
else:
    print("c is big")
```

if-elif-else: The *if-elif-else statement* is used to conditionally execute a statement or a block of statements.

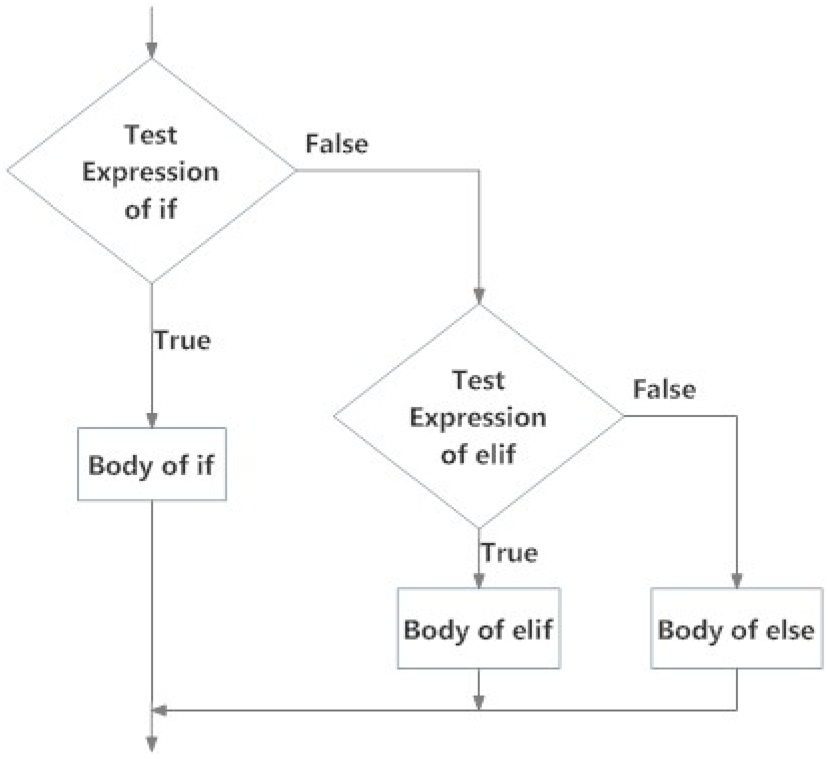


Fig: Operation of if...elif...else statement

Example:

```
x = 15
y = 12
if x == y:
  print("Both are Equal")
elif x > y:
   print("x is greater than y")
else:
   print("x is smaller than y")
```

3. Repetition or Iteration statements

A **repetition statement** is used to repeat a group(block) of programming instructions.

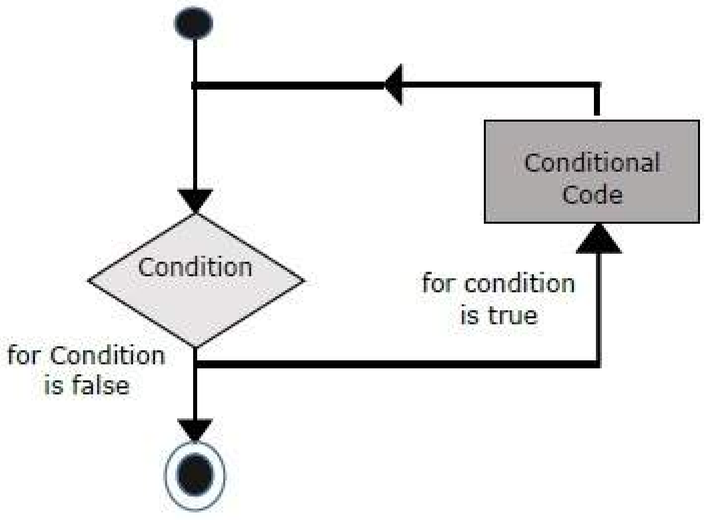
In Python, we generally have two loops/repetitive statements:

- for loop
- while loop

for loop: A *for loop* is used to iterate over a sequence that is either a list, tuple, dictionary, or a set. We can execute a set of statements once for each item in a list, tuple, or dictionary.

Syntax

```
for iterator_var in sequence:
    statements(s)
```



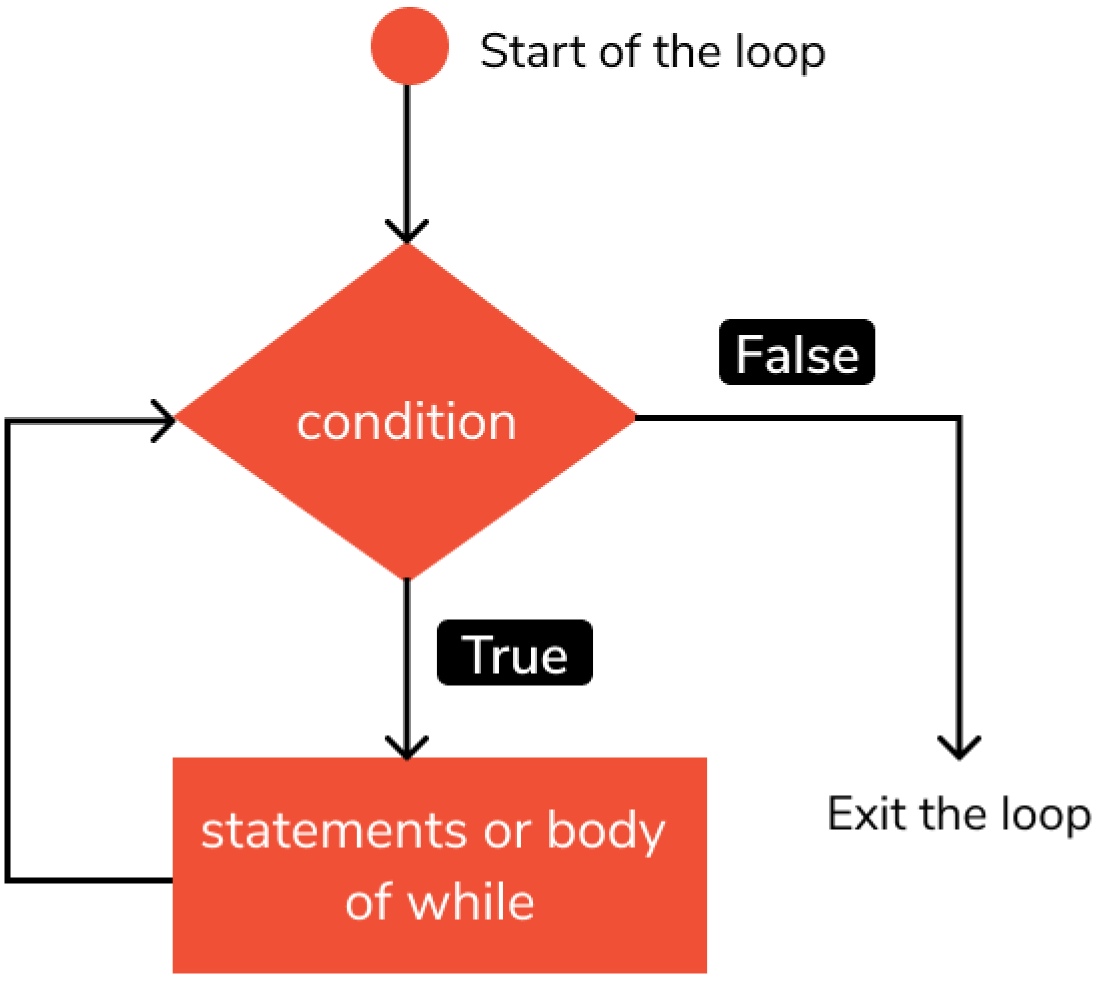
Example

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

for i in range(len(lst)):

     print(lst[i], end = " ")



for j in range(0,10):

    print(j, end = " ")
```

while loop: In Python, *while loops* are used to execute a block of statements repeatedly until a given condition is satisfied. Then, the expression is checked again and, if it is still true, the body is executed again. This continues until the expression becomes false.

Syntax :

```
while expression:
    statement(s)
```



Example

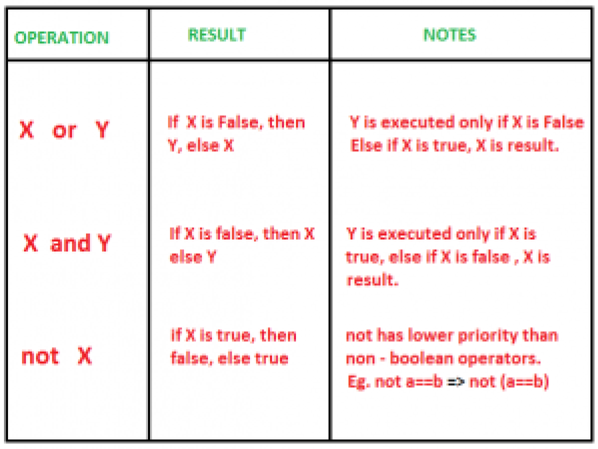
```
m = 5
i = 0
while i < m:
    print(i, end = " ")
    i = i + 1
print("End")
```

Short Circuiting Techniques in Python (6 or 8 Mark)

By short-circuiting, we mean the **stoppage of execution of boolean operation if the truth value of expression has been determined already**. The evaluation of expression takes place from **left to right**. In python, short-circuiting is supported by various boolean operators and functions.

Short Circuiting in Boolean Operators

The chart given below gives an insight into the short-circuiting of in case of boolean expressions. Boolean operators are ordered by ascending priority.



OPERATION	RESULT	NOTES
X or Y	If X is False, then Y, else X	Y is executed only if X is False Else if X is true, X is result.
X and Y	If X is false, then X else Y	Y is executed only if X is true, else if X is false , X is result.
not X	if X is true, then false, else true	not has lower priority than non - boolean operators. Eg. not a==b => not (a==b)

or: When the Python interpreter scans **or** expression, it takes the first statement and checks to see if it is true. If the first statement is true, then Python returns that object's value without checking the second statement. The program does not bother with the second statement. If the first value is false, only then Python checks the second value and then the result is based on the second half.

and: For an **and** expression, Python uses a short circuit technique to check if the first statement is false then the whole statement must be false, so it returns that value. Only if the first value is true, it checks the second statement and return the value.
An expression containing **and** and **or** stops execution when the truth value of expression has been achieved. Evaluation takes place from left to right.

Example Explanation

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
```

When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right. Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called short-circuiting the evaluation.