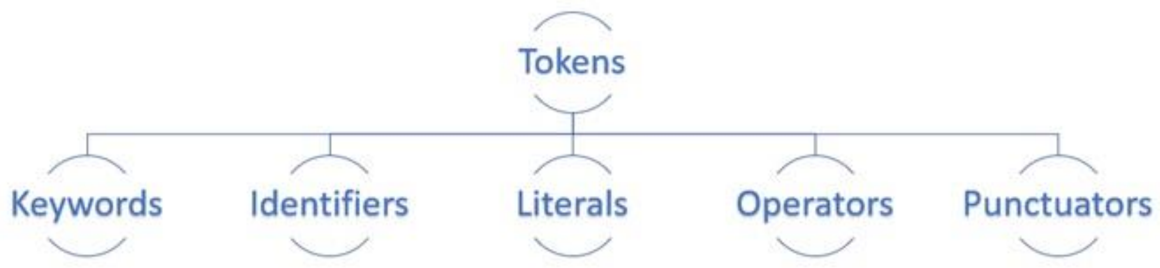
Computational thinking using python (CSE1500)

Python Character set
Any program written in Python contains words or statements which follow a sequence of characters. When these characters are submitted to the Python interpreter, they are interpreted or uniquely identified in various contexts, such as characters, identifiers, names or constants. Python uses the following character set:

- Letters: Upper case and lower case letters
- Digits: 0,1,2,3,4,5,6,7,8,9
- Special Symbols: Underscore (_), (,), [,], {,}, +, -, *, &, ^, %, $, #, !, Single quote('), Double quotes("), Back slash(\), Colon(:), and Semi Colon (;)
- White Spaces: ('\t\n\x0b\x0c\r'), Space, Tab.

=======================

TOKENS :
A program in Python contains a sequence of instructions. Python breaks each statement into a sequence of lexical components known as tokens. Each token corresponds to a substring of a statement. Python contains various types of tokens. Figure 2.1 shows the list of tokens supported by Python.



=====================

Literals:
A *literal* is a constant value that is stored into a variable in a program.
Observe the following statement:
(or)

Literals are numbers or strings or characters that appear directly in a program. A list of some literals in Python is as follows:

```
a = 15
```

Here, 'a' is the variable into which the constant value '15' is stored. Hence, the value 15 is called 'literal'. Since 15 indicates integer value, it is called 'integer literal'.

The following are different types of literals in Python:
- ❑ Numeric literals
- ❑ Boolean literals
- ❑ String literals

Example
- ➢ 78 #Integer Literal
- ➢ 2l.98 #Floating Point Literal
- ➢ 'Q' #Character Literal
- ➢ "Hello" #String Literal

Numeric Literals
These literals represent numbers. Please observe the different types of numeric literals available in Python, as shown in Table 3.1:

Table 3.1: Numeric Literals

Examples	Literal name
450, -15	Integer literal
3.14286, -10.6, 1.25E4	Float literal
0x5A1C	Hexadecimal literal
0557	Octal literal
0B110101	Binary literal
18+3J	Complex literal

Boolean Literals
Boolean literals are the True or False values stored into a bool type variable.

String Literals
A group of characters is called a string literal. These string literals are enclosed in single quotes (') or double quotes (") or triple quotes (''' or """). In Python, there is no difference between single quoted strings and double quoted strings. Single or double quoted strings should end in the same line as:

```
s1 = 'I am a student of Presidency University'
s2 = "Computational thinking using Python CSE1500"
```

When a string literal extends beyond a single line, we should go for triple quotes as:

```
s1 = '''I am a Engineering student
        studying first semester CSE
        Python is very interesting course'''
```

```
s2 = """ I am a Engineering student
        studying third semester CSE AIML
        Python is very interesting course """
```

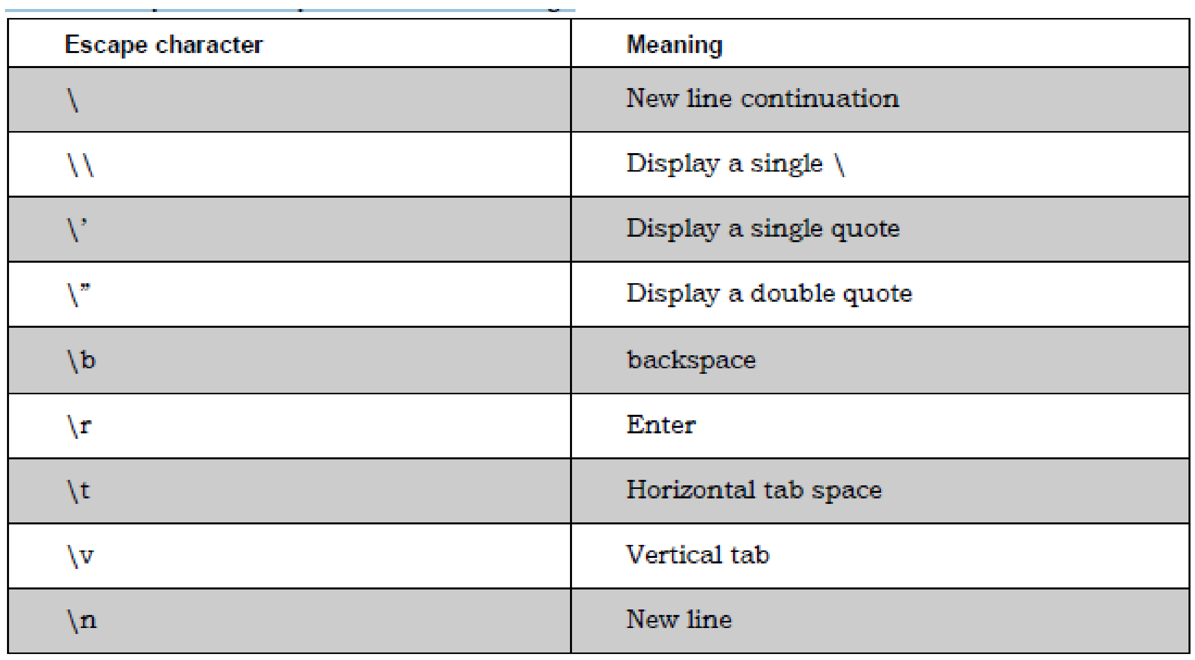
Note: We can use escape characters like \n inside a string literal.

For example,

```
str = "This is \nPython"
print(str)
This is
Python
```

Escape characters escape the normal meaning and are useful to perform a different task.
When a \n is written, it will throw the cursor to the new line. Table 3.2 gives some important escape characters:

Table 3.2: Important Escape Characters in Strings



Escape character	Meaning
\	New line continuation
\\	Display a single \
\'	Display a single quote
\"	Display a double quote
\b	backspace
\r	Enter
\t	Horizontal tab space
\v	Vertical tab
\n	New line

Display Literals in Interactive Mode
Let us consider a simple example. Print the message “Hello World” as a string literal in Python interactive mode.

Example

```
>>> ‘Hello World’
‘Hello World’
```

As shown above, type Hello World in interactive mode and press enter. Immediately after pressing enter you will see the required message.

==================

Determining the Datatype of a Variable (or) Value and Type on Literals :

To know the datatype of a variable or object, Python offers an in-built method called type. The syntax to know the type of any value is type (value)

For example:

type(a) displays the datatype of the variable ‘a’.

```
a = 15
print(type(a))
<class 'int'>
```

To know the exact type of any value,

Example

```
>>> type(‘Hello Presidency’)
<class ‘str’>
>>> type(123)
<class ‘int’>
```

Note: When the above examples are executed in Python interactive mode, return type of value is passed to the in-built function type().

===================

Keywords or Reserved words in Python :

Reserved words are the words that are already reserved for some particular purpose in the Python language. The names of these reserved words should not be used as identifiers. The following are the reserved words available in Python:

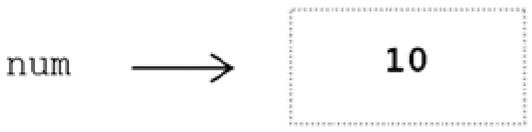
FALSE	None	TRUE	and
as	assert	async	await
break	class	continue	def
elif	else	except	finally
for	from	global	if
import	in	is	lambda
nonlocal	not	or	pass
raise	return	try	while
with	yield	del	

To retrieve the keywords in Python, write the following code at the prompt. All keywords except True, False, and None are in the lowercase letter.

```
import keyword
# printing all keywords at once using "kwlist()"
print("The list of keywords is : ")
print(keyword.kwlist)
```

==============

What Is a Variable?
A variable is a name (identifier) that is associated with a value, as for variable num depicted in Figure



A variable can be assigned different values during a program's execution—hence, the name "variable."
Wherever a variable appears in a program (except on the left-hand side of an assignment statement), it is the value associated with the variable that is used, and not the variable's name
For Example: num + 1 → 10+ 1 → 11
Variables are assigned values by use of the assignment operator.
Example:

```
num=10

num= num+1

tax = salary*0.05
```

===================

Datatypes in Python

A datatype represents the type of data stored into a variable or memory. The datatypes which are already available in Python language are called *Built-in* datatypes. The datatypes which can be created by the programmers are called *User-defined* datatypes.

Built-in datatypes

The built-in datatypes are of five types:

- None Type
- Numeric types
- Sequences
- Sets

- Mappings

The None Type

In Python, the 'None' datatype represents an object that does not contain any value. In languages like Java, it is called 'null' object. But in Python, it is called 'None' object. In a Python program, maximum of only one 'None' object is provided. One of the uses of 'None' is that it is used inside a function as a default value of the arguments. When calling the function, if no value is passed, then the default value will be taken as 'None'. If some value is passed to the function, then that value is used by the function. In Boolean expressions, 'None' datatype represents 'False'.

Numeric Types

The numeric types represent numbers. There are three sub types:

- int
- float
- complex

int Datatype

The int datatype represents an integer number. An integer number is a number without any decimal point or fraction part. For example, 200, -50, 0, 9888998700, etc. are treated as integer numbers. Now, let's store an integer number -57 into a variable 'a'.
a = -57
Here, 'a' is called int type variable since it is storing -57 which is an integer value. In Python, there is no limit for the size of an int datatype. It can store very large integer numbers conveniently.

float Datatype

The float datatype represents floating point numbers. A floating point number is a number that contains a decimal point. For example, 0.5, -3.4567, 290.08, 0.001 etc. are called floating point numbers. Let's store a float number into a variable 'num' as:
num = 55.67998
Here num is called float type variable since it is storing floating point value. Floating point numbers can also be written in scientific notation where we use 'e' or 'E' to represent the power of 10. Here 'e' or 'E' represents 'exponentiation'. For example, the number 2.5 × 104 is written as 2.5E4. Such numbers are also treated as floating point numbers. For example,
x = 22.55e3
Here, the float value 22.55 × 103 is stored into the variable 'x'. The type of the variable 'x' will be internally taken as float type. The convenience in scientific notation is that it is possible to represent very big numbers using less memory.

Complex Datatype

A complex number is a number that is written in the form of a + bj or a + bJ. Here, 'a' represents the real part of the number and 'b' represents the imaginary part of the number. The suffix 'j' or 'J' after 'b' indicates the square root value of -1. The parts 'a' and 'b' may contain integers or floats. For example, 3+5j, -1-5.5J, 0.2+10.5J are all complex numbers. See the following statement:

c1 = -1-5.5J

Here, the complex number -1-5.5J is assigned to the variable 'c1'. Hence, Python interpreter takes the datatype of the variable 'c1' as complex type.

bool Datatype

The bool datatype in Python represents boolean values. There are only two Boolean values True or False that can be represented by this datatype. Python internally represents True as 1 and False as 0. A blank string like "" is also represented as False. Conditions will be evaluated internally to either True or False.

Example 1:

```
a = 10
b = 20
if(a<b): print("Hello") # displays Hello.
```

Example 2:

```
a = 10>5 # here 'a' is treated as bool type variable
print(a) # displays True
a = 5>10
print(a) # displays False
```

NOTE:

True+True will display 2 # True is 1 and false is 0

True-False will display 1

(Remaining data types will be discussed later)

Operators in Python:

An operator is a symbol that performs an operation. An operator acts on some variables called *operands*. For example, if we write a + b, the operator ' + ' is acting on two operands 'a' and 'b'. If an operator acts on a single variable, it is called *unary* operator. If an operator acts on two variables, it is called *binary* operator. If an operator acts on three variables, then it is called *ternary* operator. This is one type of classification.

We can classify the operators depending upon their nature, as shown below:

- ❑ Arithmetic operators
- ❑ Assignment operators

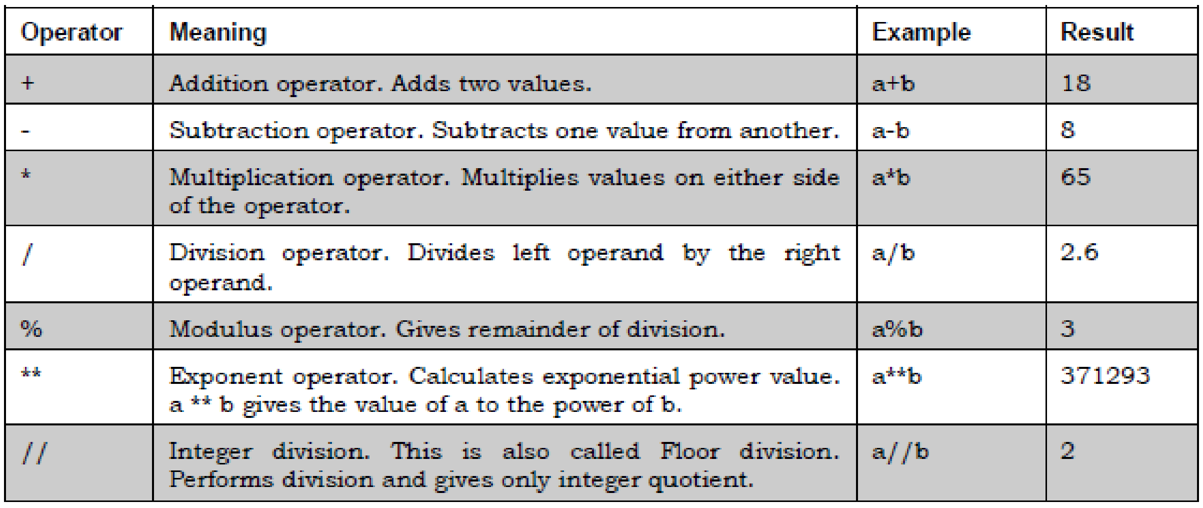
- Unary minus operator
- Relational operators
- Logical operators
- Boolean operators
- Bitwise operators
- Membership operators
- Identity operators

Arithmetic Operators

These operators are used to perform basic arithmetic operations like addition, subtraction, division, etc.

There are seven arithmetic operators available in Python. Since these operators act on two operands, they are called 'binary operators' also. Let's assume a = 13 and b = 5 and see the effect of various arithmetic operators in the Table 4.1:

Table 4.1: Arithmetic Operators in Python



Operator	Meaning	Example	Result
+	Addition operator. Adds two values.	a+b	18
-	Subtraction operator. Subtracts one value from another.	a-b	8
*	Multiplication operator. Multiplies values on either side of the operator.	a*b	65
/	Division operator. Divides left operand by the right operand.	a/b	2.6
%	Modulus operator. Gives remainder of division.	a%b	3
**	Exponent operator. Calculates exponential power value. a ** b gives the value of a to the power of b.	a**b	371293
//	Integer division. This is also called Floor division. Performs division and gives only integer quotient.	a//b	2

Assignment Operators

These operators are useful to store the right side value into a left side variable. They can also be used to perform simple arithmetic operations like addition, subtraction, etc., and then store the result into a variable. These operators are shown in Table 4.2.

In Table 4.2, let's assume the values x = 20, y = 10 and z = 5:

Table 4.2: Assignment Operators

Operator	Example	Meaning	Result
=	z = x+y	Assignment operator. Stores right side value into left side variable, i.e. x+y is stored into z.	z = 30
+=	z+=x	Addition assignment operator. Adds right operand to the left operand and stores the result into left operand, i.e. z = z+x.	z = 25
-=	z-=x	Subtraction assignment operator. Subtracts right operand from left operand and stores the result into left operand, i.e. z = z-x.	z = -15
*=	z*=x	Multiplication assignment operator. Multiplies right operand with left operand and stores the result into left operand, i.e. z = z *x.	z = 100
/=	z/=x	Division assignment operator. Divides left operand with right operand and stores the result into left operand, i.e. z = z/x.	z = 0.25
%=	z%=x	Modulus assignment operator. Divides left operand with right operand and stores the remainder into left operand, i.e. z = z%x.	z = 5
**=	z**=y	Exponentiation assignment operator. Performs power value and then stores the result into left operand, i.e. z = z**y.	z= 9765625
//=	z//=y	Floor division assignment operator. Performs floor division and then stores the result into left operand, i.e. z = z// y.	z = 0

Unary Minus Operator

The unary minus operator is denoted by the symbol minus (-). When this operator is used before a variable, its value is negated. That means if the variable value is positive, it will be converted into negative and vice versa. For example, consider the following statements:

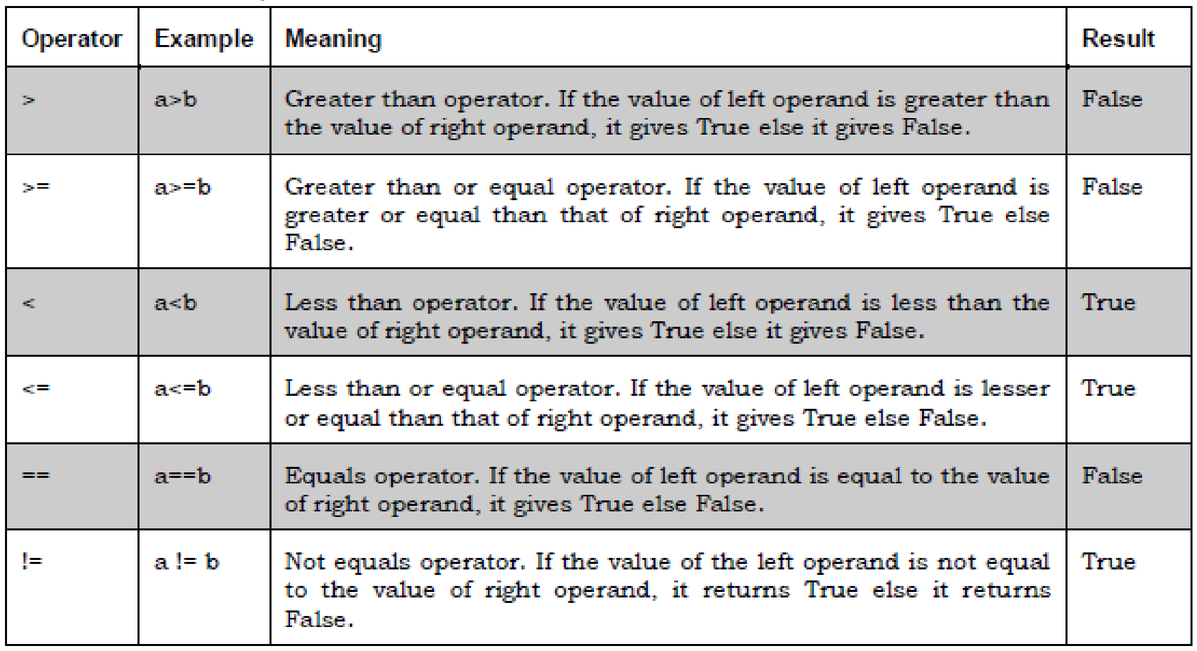
```
n = 10
print(-n) # displays -10
num = -10
num = - num
print(num) # displays 10
```

Relational Operators :

Relational operators are used to compare two quantities. We can understand whether two values are same or which one is bigger or which one is lesser, etc. using these operators.

These operators will result in True or False depending on the values compared, as shown in Table 4.3. In this table, we are assuming a =1 and b = 2.

Table 4.3: Relational Operators



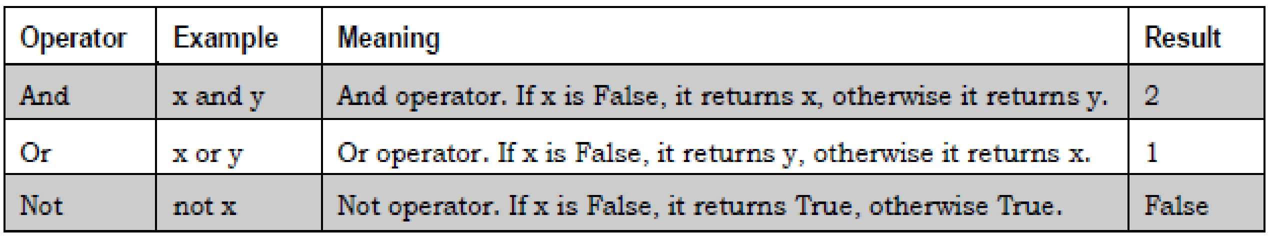
Operator	Example	Meaning	Result
>	a>b	Greater than operator. If the value of left operand is greater than the value of right operand, it gives True else it gives False.	False
>=	a>=b	Greater than or equal operator. If the value of left operand is greater or equal than that of right operand, it gives True else False.	False
<	a<b	Less than operator. If the value of left operand is less than the value of right operand, it gives True else it gives False.	True
<=	a<=b	Less than or equal operator. If the value of left operand is lesser or equal than that of right operand, it gives True else False.	True
==	a==b	Equals operator. If the value of left operand is equal to the value of right operand, it gives True else False.	False
!=	a != b	Not equals operator. If the value of the left operand is not equal to the value of right operand, it returns True else it returns False.	True

Logical Operators

Logical operators are useful to construct compound conditions. A compound condition is a combination of more than one simple condition. Each of the simple condition is evaluated to True or False and then the decision is taken to know whether the total condition is True or False. We should keep in mind that in case of logical operators, False indicates 0 and True indicates any other number. There are 3 logical operators as shown in Table 4.4.

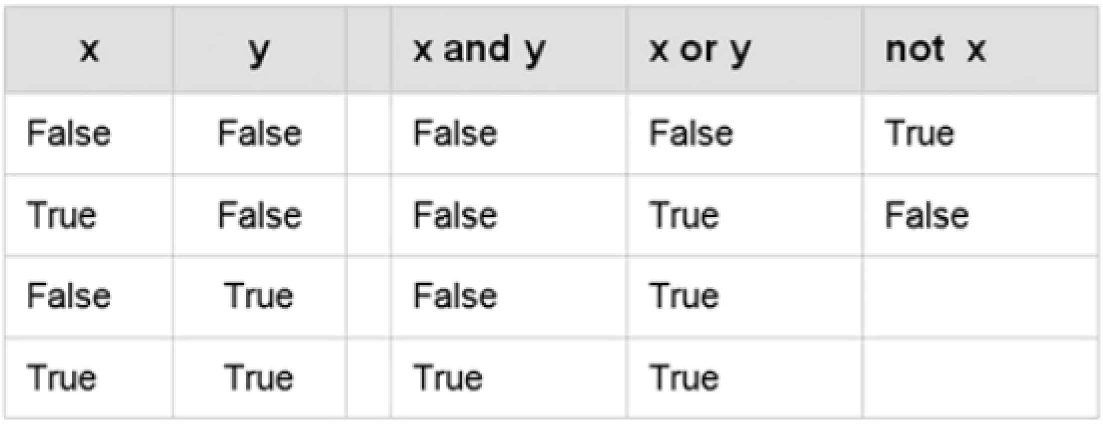
Let's take x = 1 and y=2 in this table.

Table 4.4: Logical Operators



Operator	Example	Meaning	Result
And	x and y	And operator. If x is False, it returns x, otherwise it returns y.	2
Or	x or y	Or operator. If x is False, it returns y, otherwise it returns x.	1
Not	not x	Not operator. If x is False, it returns True, otherwise True.	False

Truth Table of Logical operators (below table)



x	y		x and y	x or y	not x
False	False		False	False	True
True	False		False	True	False
False	True		False	True	
True	True		True	True	

Bitwise Operators

These operators act on individual bits (0 and 1) of the operands. We can use bitwise operators directly on binary numbers or on integers also. When we use these operators on integers, these numbers are converted into bits (binary number system) and then bitwise operators act upon those bits. The results given by these operators are always in the form of integers.

There are 6 types of bitwise operators as shown below:

- ❑ Bitwise Complement operator (~)
- ❑ Bitwise AND operator (&)
- ❑ Bitwise OR operator (|)
- ❑ Bitwise XOR operator (^)
- ❑ Bitwise Left shift operator (<<)
- ❑ Bitwise Right shift operator (>>)

Membership Operators

The membership operators are useful to test for membership in a sequence such as strings, lists, tuples or dictionaries. For example, if an element is found in the sequence or not can be asserted using these operators. There are two membership operators as shown here:

- ❑ in
- ❑ not in

The in Operator

This operator returns True if an element is found in the specified sequence. If the element is not found in the sequence, then it returns False.

The not in Operator

This works in reverse manner for 'in' operator. This operator returns True if an element is not found in the sequence. If the element is found, then it returns False.

Identity Operators

These operators compare the memory locations of two objects. Hence, it is possible to know whether the two objects are same or not. The memory location of an object can be seen using the id() function. This function returns an integer number, called the identity number that internally represents the memory location of the object.

For example, id(a) gives the identity number of the object referred by the name 'a'. See the following statements:

```
a = 25
b = 25

id(a)
1670954952
id(b)
1670954952
```

There are two identity operators:
- is
- is not

The is Operator

The 'is' operator is useful to compare whether two objects are same or not. It will internally compare the identity number of the objects. If the identity numbers of the objects are same, it will return True; otherwise, it returns False.

```
a = 25
b = 25
if(a is b):
        print("a and b have same identity")
else:
        print("a and b do not have same identity")
```

Output:

```
a and b have same identity
```

The is not Operator

This is not operator returns True, if the identity numbers of two objects being compared are not same. If they are same, then it will return False.
The 'is' and 'is not' operators do not compare the values of the objects. They compare the identity numbers or memory locations of the objects. If we want to compare the value of the objects, we should use equality operator (==).

===================

Input and Output operations in Python:

The purpose of a computer is to process data and return results. It means that first of all, we should provide data to the computer. The data given to the computer is called input. The results returned by the computer are called output. So, we can say that a computer takes input, processes that input and produces the output, as shown in Figure 5.1:

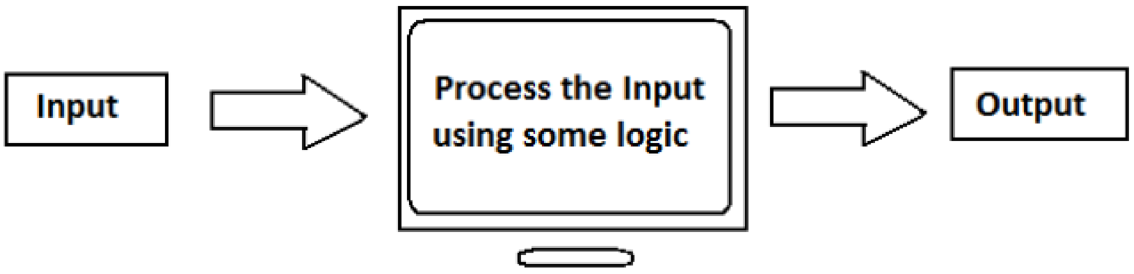


Figure 5.1: Processing Input by the Computer

To provide input to a computer, Python provides some statements which are called Input statements. Similarly, to display the output, there are Output statements available in Python.

Input Statements

To accept input from keyboard, Python provides the input() function. This function takes a value from the keyboard and returns it as a string. For example,
str = input() # this will wait till we enter a string
Raj kumar # enter this string
print(str)
Raj kumar
It is a better idea to display a message to the user so that the user understands what to enter. This can be done by writing a message inside the input() function as:

```
str = input('Enter your name: ')
Enter your name: Raj kumar
print(str)
Raj kumar
```

Once the value comes into the variable 'str', it can be converted into 'int' or 'float' etc.

This is useful to accept numbers as:

```
str = input('Enter a number: ')
Enter a number: 125
x = int(str) # str is converted into int
print(x)
125
```

We can use the int() function before the input() function to accept an integer from the keyboard as:

```
x = int(input('Enter a number: '))
Enter a number: 125
print(x)
125
```

Similarly, to accept a float value from the keyboard, we can use the float() function along with the input() function as:

```
x = float(input('Enter a number: '))
Enter a number: 12.345
print(x)
12.345
```

Output statements :

To display output or results, Python provides the print() function. This function can be used in different formats which are discussed hereunder.

1) The print("string") Statement

A string represents a group of characters. When a string is passed to the print() function, the string is displayed as it is. See the example:

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

print('Hello Dear')
Hello Dear
```

Note : double quotes and single quotes have the same meaning and hence can be used interchangeably.

```
print("This is the \n first line")
This is the
first line
print("This is the \t first line")
This is the 	 first line
```

We can use repetition operator (*) to repeat the strings in the output as:

print(3*'Hai')

HaiHaiHai

When we use '+' on strings, it will join one string with another string. Hence '+' is called concatenation (joining) operator when used on strings.

print("City name="+"Hyderabad")

City name=Hyderabad

We can also write the preceding statement by separating the strings using ', ' as:

print("City name=","Hyderabad")

City name= Hyderabad

2) The print(variables list) Statement

We can also display the values of variables using the print() function. A list of variables can be supplied to the print() function as:

a=2

b=4

print(a, b)

2 4

Observe that the values in the output are separated by a space by default. To separate the output with a comma, we should use **'sep'** attribute as shown below. 'sep' represents separator. The format is sep="characters" which are used to separate the values in the output.

print(a, b, sep=",")

2,4

print(a, b, sep=':')

2:4

print(a, b, sep='----')

2----4

Example :

print("Presidency")

print("University")

print('Bengaluru')

Output:

Presidency

University

Bengaluru

Note : Each print() function throws the cursor into the next line after displaying the output. This is the reason we got the output in 3 lines. We can ask the print() function not to throw the cursor into the next line but display the output in the same line. This is done using 'end' attribute. The way one can use it is end="characters" which indicates the ending characters for the line. Suppose, we write end=", then the ending character for each line will be " (nothing) and hence it will display the next output in the same line. See the example:

Example :

print("Presidency", end=' ')

print("University", end=' ')

print('Bengaluru', end=' ')

Output:

Presidency University Bengaluru

Example :

print("Presidency", end=' \t')

print("University", end=' \t ')

print('Bengaluru', end=' \t ')

Output:

```
Presidency            University            Bengaluru
```

3) The print("string", variables list) Statement

The most common use of the print() function is to use strings along with variables inside the print() function.

Example :

```
a=2
print(a, "is even number")
2 is even number
print('You typed ', a, 'as input')
You typed 2 as input
```

Example)

```
num=123.456789
print('The value is: %f' % num)
The value is: 123.456789
print('The value is: %8.2f' %num)

The value is: 123.46 # observe 2 spaces before the value
print('The value is: %.2f' %num)
The value is: 123.46
```

4) **print with formatted string**

Inside the formatted string, we can use replacement field which is denoted by a pair of curly braces { }. We can mention names or indexes in these replacement fields. These names or indexes represent the order of the values. After the formatted string, we should write member operator and then format() method where we should mention the values to be displayed. Consider the general format given below:

print('format string with replacement fields'.format(values))

Example:

```
n1, n2, n3 = 1, 2, 3
print('number1={0}, number2={1}, number3={2}'.format(n1, n2, n3))
number1=1, number2=2, number3=3
```

In the above statement, {0}, {1}, {2} represent the values of n1, n2 and n3, respectively. Hence, if we change the order of these fields, we will have order of the values to be changed in the output as:

```
print('number1={1}, number2={0}, number3={2}'.format(n1, n2, n3))
number1=2, number2=1, number3=3
```

As an alternate, we can also use names in the replacement fields. But the values for these names should be provided in the format() method.

Consider the following example:

```
print('number1={two}, number2={one}, number3={three}'.format(one=n1,
```

two=n2, three=n3))

number1=2, number2=1, number3=3

We can also use the curly braces without mentioning indexes or names. In this case, those braces will assume the sequence of values as they are given. Consider the following statements:

print('number1={}, number2={}, number3={}'. format(n1, n2, n3))

number1=1, number2=2, number3=3

All the four examples given below will display the same output:

name, salary = 'Ravi', 12500.75

print('Hello {0}, your salary is {1}'.format(name, salary))

Hello Ravi, your salary is 12500.75

print('Hello {n}, your salary is {s}'.format(n=name, s=salary))

Hello Ravi, your salary is 12500.75

print('Hello {:s}, your salary is {:.2f}'.format(name, salary))

Hello Ravi, your salary is 12500.75

print('Hello %s, your salary is %.2f' % (name, salary))

Hello Ravi, your salary is 12500.75

========================

Indentation in Python

In Python, Indentation is used to define blocks of code. It tells the Python interpreter that a group of statements belongs to a specific block. All statements with the same level of indentation are considered part of the same block. Indentation is achieved using whitespace (spaces or tabs) at the beginning of each line. The most common convention is to use 4 spaces or a tab, per level of indentation.

Python Comments:

Python comments are programmer-readable explanation or annotations in the Python source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter. Comments enhance the readability of the code and help the programmers to understand the code very carefully.

{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{000000000000000}}}}}}}}}}}}}}}}}}}}}}}}}}}}}

How to write comment lines in python program :

In **Python**, there are two types of comments:

1. **Single-line comments:** These start with a **#** symbol, and everything after it on the same line is treated as a comment.

   Example:

   # this is a comment line

   # this program is written by Anagha first sem student

2. **Multi-line comments:**

   Python does not provide the option for multiline comments. However, there are different ways through which we can write multiline comments in python.
   Python ignores the string literals that are not assigned to a variable. So, we can use these string literals as Python Comments**.**

Note: Python allows the use of both **"""***triple-double-quotes***"""** or **'''***triple-single-quotes***'''** to create docstrings. However, the Python coding conventions specification recommends us to use **"""***triple-double-quotes***"""** for consistency.

The docstring should always begin with a capital letter and end with a period (.)

Example :

```
"""
This is a comment
written in
more than just one line.
"""

'''
This is a comment
written in another way
more than just one line.
'''
```

Python Identifiers

A Python identifier is the name given to a variable, function, class, module or other object. An identifier can begin with an alphabet (A – Z or a – z), or an underscore (_) and can include any number of letters, digits, or underscores. Spaces are not allowed.

- ❑ Python will not accept @, $ and % as identifiers.
- ❑ Python is a case-sensitive language. Thus, Hello and hello both are different identifiers.
- ❑ Keywords cannot be used as identifiers.

Expressions :

An expression is a combination of operands (variable / value / constant) and operators.

Operators are symbols that represent particular actions (such as addition, subtraction, comparison etc.,)

Any expression evaluates to a single value, which becomes the value of the expression.

Example 1)

```
a=11

b=22

sum=a+b   # a+b is an arithmetic expression, because operator used is arithmetic operator

print(sum)
```

Example 2)

```
n+1    # here n and 1 are operands
```

Example 3)

```
 a>b   # a>b is a relational expression, because operator used is relational operator

age>=18    # here age and 18 are operands
```

Example 4)

x > 3 and x < 10 # logical expression, because logical operator **and** is used

Note: An expression may also include call to functions and objects.

Control Statements:

When we write a program, the statements in the program are normally executed one by one. This type of execution is called 'sequential execution'.
To develop critical programs, the programmer should be able to change the flow of execution as needed by him. For example, the programmer may wish to repeat a group of statements several times or he may want to directly jump from one statement to another statement in the program. For this purpose, we need control statements. In this chapter, you will learn more about control statements.

They are divided into

- Decision Making Statements / Branching Statements / Conditional Statements
- Iterative / Looping Statements

Decision Making Statements / Branching Statements / Conditional Statements

Conditional statements in Python are used to execute certain blocks of code based on specific conditions.

Conditional / branching statements are statements which control or change the flow of execution. The following are the control statements available in Python:

- if statement
- if … else statement
- if … elif … else statement

if statement :

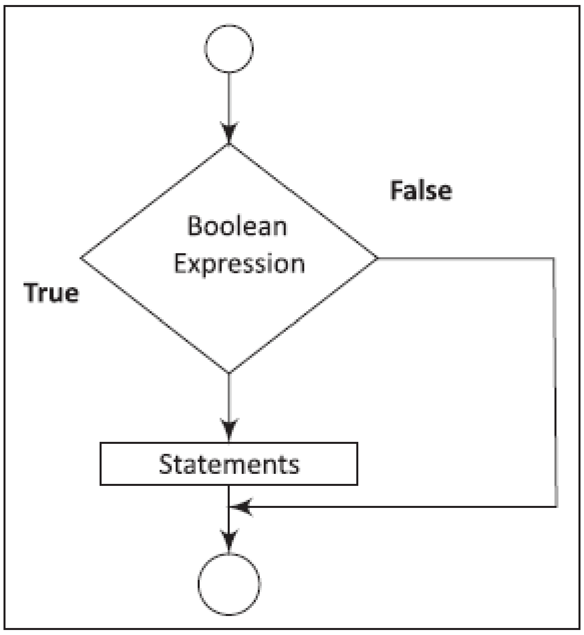
The if statement is known as the decision-making statement in programming languages.
If statement is used to execute one or more statement depending on whether a condition is True or not. The syntax or correct format of if statement is given below:

```
if condition:
     statements
```

(or)

```
if condition:
     block
```

Flow chart of if statement:



First, the condition is tested. If the condition is True, then the statements given after colon (:) are executed. We can write one or more statements after colon (:). If the condition is False, then the statements mentioned after colon are not executed.

Example:

```
if radius>0:
        area=22/7*radius*radius
        print(" Area of Circle is = ",area)
        circumference=2*22/7*radius
        print("Circumference of Circle is = ",circumference)
```

Note: Observe that every statement mentioned after colon is starting after 4 spaces only (or we can give tab also).
When we write the statements with same indentation, then those statements are considered as a block or suite (or belonging to same group). In above example, the four lines or statements written after the colon form a block or suite.

Note: If we don't provide spaces or tab for the block, then code does not run and displays an error called *indentation error*.

Short Hand if :

```
age = 22
if age >= 18:
   print("Eligible to vote.")
```

the above statement can also be written using short hand if as follows

```
age = 22
if age >= 18: print("Eligible to vote.")
```

----
if … else Statement
The if-else statement takes care of both True and False conditions. It has two blocks. One block is for if and it may contain one or more than one statements. The block is executed when the condition is true. The other block is for else. The else block may also have one or more than one statements. It is executed when the condition is false. A colon (:) must always be followed by the condition. The keyword else should also be followed by a colon (:)
Its syntax is

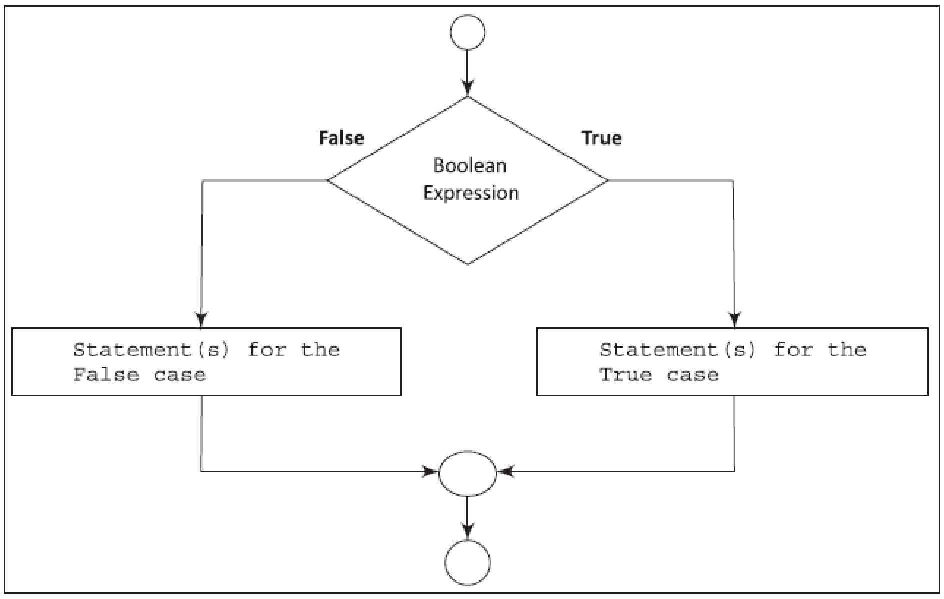
```
if condition:
    statement(s)
else:
    statement(s)
```

```
if condition:
    if_Block
else:
   else_Block
```

Example)

```
salary=float(input("enter employee salary"))
if salary<=50000:
    tax=salary*0.05    # if block
    da=salary*0.06     # if block
else:
    tax=salary*0.07   # else block
    da=salary*0.08    # else block
print("salary is",salary)    # next statement
print("tax is",tax)
print("da is",da)
```

The flow chart for if-else statement is given in the following figure.


False
True
Boolean Expression
Statement(s) for the False case
Statement(s) for the True case

Short Hand if-else:

The short-hand if-else statement allows us to write a single-line if-else statement.

Example 1)

```
a=int(input("enter first number"))
b=int(input("enter second number"))
if a>b:
  big=a
else:
  big=b
print("biggest is",big)
```

the above statement can also be written using short hand if as follows

```
a=int(input("enter first number"))
b=int(input("enter second number"))
big=a if a>b else b
print("biggest is",big)
```

Note: Many programming languages, such as Java, C have a ‘ **? :** ’, i.e. ternary operator, which uses three operands. The ternary operator works like if-else.

But, Python does not have a ternary operator. It uses a conditional expression. (like short hand if-else)

----

if … elif … else Statement (or) Multi-way if-elif-else Statements

Sometimes, the programmer has to test multiple conditions and execute statements depending on those conditions. if … elif … else statement is useful in such situations.

Its syntax is

```
if condition 1:
      statements1
elif condition 2:
      statements2
elif condition 3:
      statements3
      ------
      ------
      ------
elif condition n:
      statements n

else:
      statement(s)
```

In this kind of statements, the number of conditions, i.e. Boolean expressions are checked from top to bottom. When a true condition is true / found, the statement associated with it is executed and the rest of the conditional statements are skipped. If none of the conditions are found true then the last else statement is executed.
Note: else part is always optional. In this case if all the conditions are false and if the final else statement is not present then no action takes place.

Example:

```
a=int(input("enter first number"))
b=int(input("enter second number"))
c=int(input("enter third number"))
if a>b and a>c:
   print(a,"is biggest")
elif b>c:
   print(b,"is biggest")
else:
   print(c,"is biggest")
print("end of the program")
```
