Unit-I : Introduction

History of Python:

Python is a widely used general-purpose, high-level programming language. It was initially designed by Guido van Rossum in the early 1990s at CWI(The *Centrum Wiskunde & Informatica* is a research center in the field of mathematics and theoretical computer science) in the Netherlands and later developed by Python Software Foundation.

- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.

- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.

- On October 16,2000 Python 2.0 added new features like: list comprehensions, garbage collection system.

- On December 3, 2008, Python 3.0 (also called "Py3K") was released.
  - Python 3.1 - June 27, 2009
  - Python 3.2 - February 20, 2011
  - Python 3.3 - September 29, 2012
  - Python 3.4 - March 16, 2014
  - Python 3.5 - September 13, 2015
  - Python 3.6 - December 23, 2016
  - Python 3.7 - June 27, 2018

Features of Python:

1. Python is an easy to learn, powerful programming language.

2. Python is an interpreted language, which can save you considerable time during program Development because no compilation and linking is necessary.

3. Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs

4. Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard

modules that you can use as the basis of your programs. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

5. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms
6. It has efficient high-level data structures and a simple effective approach to object-oriented programming.

7. Python is *extensible*: it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form

8. **Python is free and open source software**

Applications of Python:

Python is used in many application domains.

Web and Internet Development

Python offers many choices for web development. Its standard library supports many Internet protocols like http,FTP etc.Python supports a variety of modules to work with the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML). It also provides Frameworks such as Django, Flask etc to design and delelop web based applications.

Desktop GUI Applications

Python provides Tk GUI library to develop user interface in python based application. Some other useful toolkits wxWidgets, Kivy, pyqt that are useable on several platforms. The Kivy is popular for writing multitouch applications.

Software Development

Python is helpful for software development process. It works as a support language and can be used for build control and management, testing etc.

Scientific and Numeric

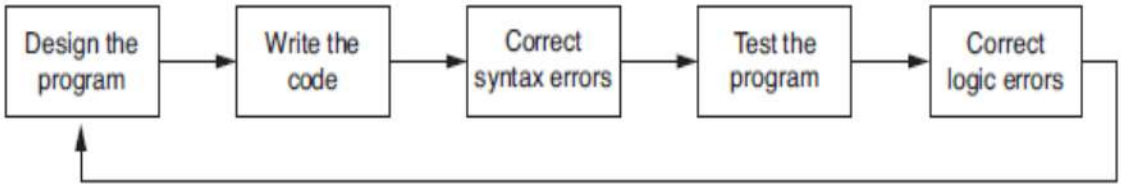
Python is popular and widely used in scientific and numeric computing. Some useful library and package are SciPy, Pandas, IPython etc. SciPy is group of packages of engineering, science and mathematics.

Business Applications

Python is also used to build ERP and e-commerce systems

The Program Development Cycle:

Programs must be carefully designed before they are written. The process of creating a program that works correctly typically requires the five phases as shown in following figure. The entire process is known as the *program development cycle*.



a) Design the program:

The process of designing a program is the most important part of the cycle. Program should be carefully designed before the code is actually written. The process of designing a program can be summarized in the following two steps:

i) Understand the task that the program is to perform.
ii) Determine the steps that must be taken to perform the task.

i) Understand the Task That the Program Is to Perform:

It is essential that you understand what a program is supposed to do. Programmer gains this understanding by working directly with the customer requirements. The programmer studies the information that was gathered from the customer and creates a list of different software requirements.
A software requirement is simply a single task that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements, is complete, the programmer can move to the next phase.

ii) Determine the Steps That Must Be Taken to Perform the Task:

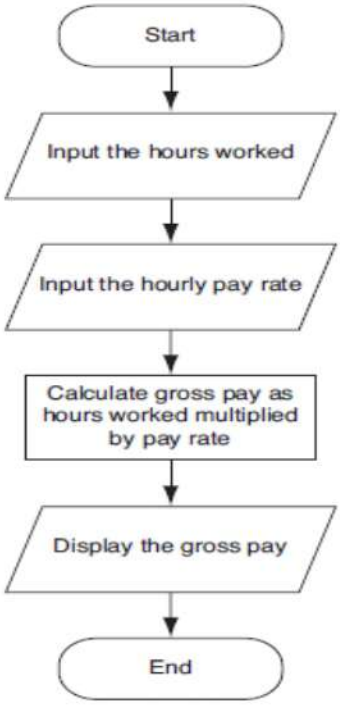
After understanding the task that the program will perform, an algorithm is created, which lists all of the logical steps that must be taken to perform the task. An Algorithm is a step-by-step procedure to perform a task. The steps of the algorithm have to be translated into code. Programmers commonly use two tools to help them accomplish this: pseudocode and flowcharts.

Ex:
suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee. The sample algorithm and flowchart are given below.

Algorithm:

1. Get the number of hours worked.
2. Get the hourly pay rate.
3. Multiply the number of hours worked by the hourly pay rate.
4. Display the result of the calculation that was performed in step 3.

Flowchart:



b) write the code:

After designing the Algorithm, the programmer begins writing code in a high-level language such as C,C++, java and Python etc.

c) Correct Syntax Errors:

Each language has its own rules, known as syntax that must be followed when writing a program. A language's syntax rules dictate things such as how key words, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.

If the program contains a syntax error, or even a simple mistake such as a misspelled key word, the compiler or interpreter will display an error message indicating what the error is.

All of the syntax errors and simple typing mistakes have to be corrected before converting to a machine level language program.

d) Test the Program:

A **correct program** produces the expected output for any legitimate input. Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A *logic error* is a mistake that does not prevent the program from running, but causes it to produce incorrect output.

e) Correct Logic Errors :

If the program produces incorrect results, the programmer *debugs* the code. This means that the programmer finds and corrects logic errors in the program. Sometimes during this process, the programmer discovers that the program's original design must be changed. In this event, the program development cycle starts over, and continues until no errors can be found.

Input, Processing and output:

Computer programs typically perform the following three-step process:

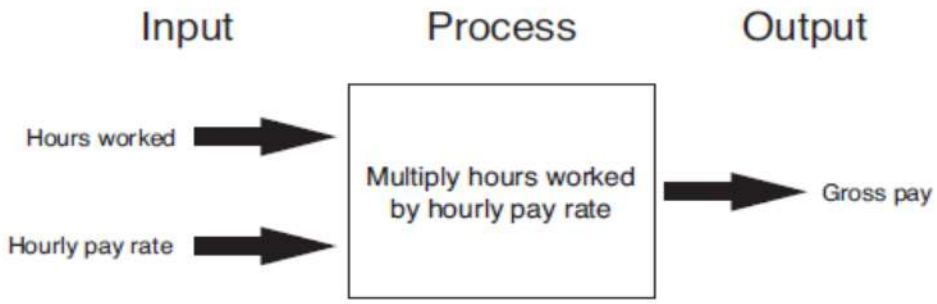
1. Input is received.
2. Some process is performed on the input.
3. Output is produced.

Input is any data that the program receives while it is running. One common form of input is data that is typed on the keyboard. Once input is received, some process,

such as a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output.

For the pay calculating problem, the number of hours worked and the hourly pay rate are provided as input. The program processes this data by multiplying the hours worked by the hourly pay rate. The results of the calculation are then displayed on the screen as output.

The following figure illustrates the input, processing and output for pay calculating program.



Displaying Output with the print Function:

In python, We use the print() function to output data to the standard output device (screen) or to a file.

Syntax:

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

This function Prints the values to a stream, or to sys.stdout by default.

Ex1:
```
>>> print("sviet",",Nandamuru")
```

```
sviet ,Nandamuru
```

Ex2:
```
>>> print("sviet" ,10,22.5,+2j)
```

```
sviet 10 22.5 2j
```

Here there are following **Optional keyword arguments**:

i) sep ii) end iii) file iv) flush

sep is the string inserted between values, default a space.(optional)

end is the string appended after the last value. It defaults into a new line(optional)

The file is the object where the values are printed and its default value is sys.stdout(screen).(optional)

flush is Boolean, specifying if the output is flushed (True) or buffered (False). Default is False(optional)

Ex:

```
>>> print('G', 'F', 'G', sep ='')
GFG
>>> print('G', 'F', 'G', sep =',')
G,F,G
>>>  print('G', 'F', 'G', sep =',',end = '@')
G,F,G@
```

Comments and docstrings:

Comments are descriptions that help programmers to better understand the functionality of the program. Comments make the program more readable which helps us remember why certain blocks of code were written. Comments are part of the program, but the Python interpreter ignores them.

In Python, we use the hash (#) symbol to start writing a comment. Programmers commonly write end-of-line comments in their code. End-of-line comments can begin with the # symbol and extend to the end of a line. When the Python interpreter sees a end-of-line comments, it ignores everything from # character to the end of the line.

Example program:

program	output

#The following statement prints Hello
print('Hello')
	Hello

Multiline Python Comment:

We must use the hash(#) at the beginning of every line of code to apply the multiline Python comment.

Program	output

# Variable a holds value 5 	
# Variable b holds value 10 	
# Variable c holds sum of a and b 	
# Print the result 	
a = 5
b = 10
c = a+b
print("The sum is:", c)
	The sum is: 15

We can also use the triple quotes (either single quotes or double quotes) for multiline comment.

By convention, a sequence of characters enclosed in triple quotation marks that Python uses to document program components such as modules, classes, methods, and functions is called as a **docstring(documentation string).**

Program	output
''' Variable a holds value 5 Variable b holds value 10 Variable c holds sum of a and b Print the result ''' a = 5 b = 10 c = a+b print("The sum is:", c)	The sum is: 15

Keywords:

Keywords are a list of reserved words that have predefined meaning. They cannot be used as identifiers for variables, classes, functions etc. Attempting to use a keyword as an identifier name will cause an error. There are 35 keywords in python 3.7.The list of keywords in python are

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

Identifiers:

An identifier is a name given to a variable, function, class or module. The rules to name an identifier are given below.

- The first character of the Identifier must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore or digit (0-9).
- Identifier names must not contain any white-space, or special character (!, @, #, %, ^, &, * etc.).
- Keywords cannot be used as identifiers
- Identifier names are case sensitive for example average, and Average is not the same.

Variables:

Variable is a name which is used to refer memory location. Variable can hold any type of data which the program can use to assign and modify during its execution.

Creating Variables with Assignment Statements

We can use an assignment statement to create a variable and make it reference a piece of data. To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it as shown below.

<variable name> = <expression>

Python uses assignment operator(=) to assign values to variables. Here expression is a value, or any piece of code that results in a value.

The python interpreter first evaluates the expression on the right hand side of the assignment operator and then binds the variable name on the left side to this value.

```
Ex:
>>>a = 2
>>>b = 9223372036854775807
>>>pi = 3.14
>>>c = 'A'
>>>name = 'John Doe'
>>>q = True
```

Note 1:

There's no need to declare a variable in advance (or to assign a data type to it), assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.

Note 2:

Variables can reference different values while a program is running. A variable in Python can refer to values of any type. After a variable has been assigned a value of one type, it can be reassigned any value of a different type.

When you assign a value to a variable, the variable will reference that value until you assign it a different value.

When a value in memory is no longer referenced by a variable, the Python interpreter automatically removes it from memory through a process known as garbage collection.

Ex:

```
>>> x=4
>>> print(x)
4
>>> x="sviet"
>>> print(x)
sviet
```

Note 3:
You cannot use a variable until you have assigned a value to it. An error will occur if you try to perform an operation on a variable, such as printing it, before it has been assigned a value.

Note 4:
You can assign multiple values to multiple variables in one line. Note that there must be the same number of arguments on the right and left sides of the = operator:

Ex:

```
>>>a, b, c = 1, 2, 3
>>>print(a, b, c)
```

Output:
1 2 3

Ex:

```
a, b, c = 1, 2
```

Output:

```
=> Traceback (most recent call
last):
=> File "name.py", line N, in
<module>
=> a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack
```

Note 5:
You can also assign a single value to several variables simultaneously.

Ex:

```
>>>a = b = c = 1
>>>print(a, b, c)
```

Output:
1 1 1

Note 6:

consider the following statement

```
>>>a = b = c = 1
```

When using such cascading assignment, it is important to note that all three variables a, b and c refer to the same object in memory, an int object with the value of 1.

In other words, a, b and c are three different names given to the same int object. Assigning a different object to one of them afterwards doesn't change the others, just as expected:

Ex:

```
>>>a = b = c = 1   all three names a,
b and c refer to same int object with
value 1
>>>print(a, b, c)

 Output: 1 1 1

>>>b = 2     # b now refers to another
int object, one with a value of 2
>>>print(a, b, c)

 Output: 1 2 1
```

Reading Input from Keyboard:

In python, we have the input() function to take the input from the user. **The input() function takes the user input as a string.**

syntax :

input([prompt])

where prompt is the string we wish to display on the screen. It is optional. If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised.

Consider the following statement

name = input('What is your name? ')

When this statement executes, the following things happen:

- The string 'What is your name? ' is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard and then to press the Enter key.
- When the Enter key is pressed, the data that was typed is returned as a string and assigned to the name variable

Program	output

#program illustrating input() function
firstname=input("enter your first name:")
lastname=input("enter your last name:")
print("hello,",firstname,lastname,",welcome to python")
	enter your first name:pavan
	enter your last name:kumar
	hello, pavan kumar ,welcome to python

Reading Numbers with the input Function:

The input function always returns the user's input as a string, even if the user enters numeric data. Python has built-in functions that you can use to convert a string to a numeric type.

Ex:

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> print(num)
'10'
```

In the above example, the entered value 10 is a string, not a number.

To convert this into a number we can use built-in-functions **int( ) or float( )** functions as given below.

```
>>>num = int(input('Enter a number: '))
Enter a number: 10
>>> type(num)
<class 'int'>
```

Note:
The int() and float() functions work only if the item that is being converted contains a valid numeric value. If the argument cannot be converted to the specified data type, an error known as ValueError arises.

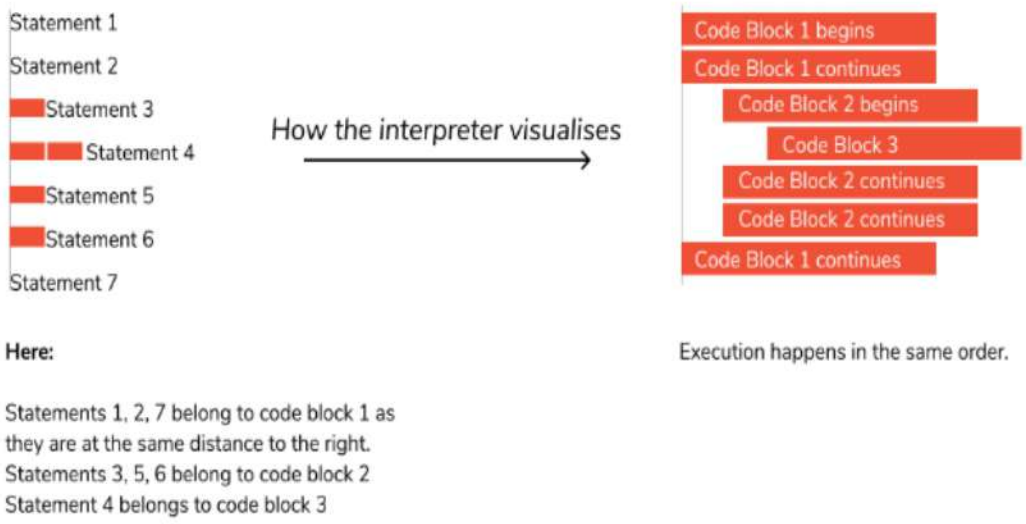
Program	output

# Get the student's name, age, and 	
#percentage.
name = input('What is your name? ')
age = int(input('What is your age? '))
percentage = float(input('What is your
percentage? '))
 	
# Display the data. 	
print('Here is the data you entered:')
print('Name:', name)
print('Age:', age)
print('percentage:', percentage)
	What is your name? pavan
	What is your age? 40
	What is your percentage? 75.8
	Here is the data you entered:
	Name: pavan
	Age: 40
	percentage: 75.8

Indentation:

Python indentation is a way of telling the Python interpreter that a series of statements belong to a particular block of code.

Leading whitespaces at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the block of statements.



Most of the programming languages like C, C++, and Java use braces { } to define a block of code. Python, however, uses indentation.

A code block (body of a function, loop, etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

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

Incorrect Indentation will result into IndentationError.

Performing Calculations:

Operators:

Operators are special symbols in Python that carry out arithmetic or logical computation.

Types of Operators:

Python language supports the following types of operators −

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators:

Python has numerous operators that can be used to perform mathematical Calculations. Programmers use the **arithmetic operators** to create math expressions. A *math expression* performs a calculation and gives a value.

Arithmetic operators are used perform arithmetic operations like addition, substraction , multiplication, division etc. The various arithmetic operators are

Operator	Meaning	Example
+	Add two operands or unary plus	>>>x,y=20,10 >>>x+y 30
-	Subtract right operand from the left or unary minus	>>>x,y=20,10 >>>x-y 10
*	Multiply two operands	>>>x,y=20,10 >>>x*y 200

/	Divide left operand by the right one (always results into float)	>>> x,y=20,10
		>>> x/y
		2.0

%	Modulus - remainder of the division of left operand by the right	>>> x,y=20,10
		>>> x%y
		0

//	Floor division - Divides one number by another and gives the result as an integer.	>>> x,y=20,10
		>>> x//y
		2

**	Exponent - left operand raised to the power of right	>>> x,y=20,10
		>>> x**y
		10240000000000

Floating-Point and Integer Division:

Python has two different division operators. The / operator performs floating-point division, and the // operator performs integer division.

Both operators divide one number by another. The difference between them is that the / operator gives the result as a floating-point value, and the // operator gives the result as an integer.

Ex:

```
>>> 5 / 2
2.5
```

Ex:

```
>>> 5 // 2
2
```

The // operator works like this:

• When the result is positive, it is *truncated*, which means that its fractional part is thrown away.

• When the result is negative, it is rounded *away from zero* to the nearest integer.

Ex:

```
>>> -5 // 2
-3
```

Operator Precedence:

Operator precedence determines which operator is evaluated first when an expression has more than one operator. First, operations that are enclosed in parentheses are performed. Then, when two operators share an operand, the operator with the higher *precedence* is applied first.

The precedence of the arithmetic(or math) operators, from highest to lowest, is:

1. Exponentiation: **

2. Multiplication, division, and remainder: * / // %

3. Addition and subtraction: + -

Notice that the multiplication (*), floating-point division (/), integer division (//), and remainder (%) operators have the same precedence. The addition (+) and subtraction (-) operators also have the same precedence.

When two operators with the same precedence share an operand, the operators execute from left to right(except for exponentiation operations which are evaluated right to left).

Note:

Mixed-Type Expressions and **Data Type Conversion**:

An expression that uses operands of different data types is called a *mixed-type expression*. When an operation is performed on an int and a float, the int value will be temporarily converted to a float and the result of the operation will be a float.

Ex:Consider the statement

x= 2*10 + 6/3

= 20+6/3

= 20+2.0

=22.0

Comparison (Relational) Operators:

Comparison operators are used to compare values. It either returns True or False according to the condition.

Operator	Meaning	Example

>	Greater than - True if left operand is greater than the right	>>> x,y=20,10
		>>> x>y
		True

<	Less that - True if left operand is less than the right	>>> x,y=20,10
		>>> x<y
		False

==	Equal to - True if both operands are equal	>>> x,y=20,10
		>>> x==y
		False

!=	Not equal to - True if operands are not equal	>>> x,y=20,10
		>>> x!=y
		True

>=	Greater than or equal to - True if left operand is greater than or equal to the right	>>> x,y=20,10
		>>> x>=y
		True

<=	Less than or equal to - True if left operand is less than or equal to the right	>>> x,y=20,10
		>>> x<=y
		False

Assignment operators:

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

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Logical operators:

Logical operators are the and, or, not operators which can be used to create complex Boolean expressions.

Operator	Meaning	Example

and	True if both the operands are true	>>> x=True
		>>> y=False
		>>> x and y
		False

or	True if either of the operands is true	>>> x=True
		>>> y=False
		>>> x or y
		True

not	True if operand is false (complements the operand)	>>> y=False
		>>> not y
		True

Bitwise operators:

Bitwise operator works on bits and performs bit-by-bit operation.
Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x& y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

Membership operators:

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).In a dictionary we can only test for presence of key, not the value.

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	>>> 'py' in 'python' True
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	>>> 'py' not in 'python' False

Identity operators:

is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Description	Example

is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	>>> x=3
		>>> y=3
		>>> id(x)
		1631445808
		>>> id(y)
		1631445808
		>>> x is y
		True

is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	>>> x=[1,2,3,4]
		>>> y=[1,2,3,4]
		>>> id(x)
		38243464
		>>> id(y)
		38279328
		>>> x is not y
		True

Operator precedence: (python 3.7.4)

The following table summarizes the operator precedence in Python, from **lowest precedence (least binding) to highest precedence (most binding).** Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left).

Operator	Description
lambda	Lambda expression
if – else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+x, -x, ~x	Positive, negative, bitwise NOT

Operator	Description
**	Exponentiation
await x	Await expression
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...), [expressions...], {key: value...},{expressions...}	Binding or tuple display, list display, dictionary display, set display

Data types:

There are various data types in python. Some of the important types are

1. Numeric types 2.strings 3.booleans

Numeric types:

There are 3 distinct numeric types –

a) Integers
b) floating point numbers and
c) Complex numbers.

Integers:

Integers are whole numbers which can be either positive or negative **without any decimal point** of unlimited length. No commas and blank spaces are allowed.

The magnitude of a Python integer is much larger and is limited only by the memory of your computer.

Ex: 23, 34 ,0 ,-345 etc.

Floating point numbers:

Floating point numbers are the numbers which can be either positive or negative that **contain a decimal point**. Floating point numbers can also be expressed in scientific form with an 'e' or 'E' to indicate the power of '10'

Computer's memory limits not only the range but also the precision that can be represented for real numbers.

Ex: 0.0, 2.3,-4.5, 3.5e9 etc.

Complex numbers:

Complex number is a number that can be expressed in a form a+bj where a is real part and b is imaginary part and 'j' or 'J' as an imaginary unit.

Ex: 3+4j 3.14j 10.j 10j .001j 1e100j 3.14e-10j

Booleans:

Python has explicit Boolean data type called bool with the values True and False available as preassigned built-in names. Internally, the names True and False are instances of bool, which in turn a subclass of built-in integer type int.

```
>>> type(True)
<class 'bool'>

>>>x=True

>>> isinstance(x,int)
True
```

Strings:

A string is a sequence of characters enclosed in single quotation marks('...') or double quotation marks("...").
Ex:

```
a = "Hey"
b = "Hey there!"
c = "742 Evergreen Terrace"
d = "1234"
e = "'How long is a piece of string?' he asked"
f = "'!$*#@ you!' she replied"
```

note:
Double-quoted strings are used for composing strings that contain single quotation marks or apostrophes and use single-quote marks to enclose a string literal that contains double quotes as part of the string

Ex:

```
>>> print("I'm using a single quote in this string!")
I'm using a single quote in this string!


>>> print('Your assignment is to read "Hamlet" by tomorrow')
Your assignment is to read "Hamlet" by tomorrow
```

Access characters in a string:

The individual characters of a string can be accessed using indexing and a range of characters can be accessed using slicing. The index must be an integer and Index starts from 0. Python also allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

Trying to access a character out of index range will raise an IndexError. We can't use float or other types for specifying index value. This will result into TypeError.

Ex:

```
>>>str = 'python programming'
>>>print('str = ', str)
str =  python programming

>>>str = 'python programming'
>>>print('str[0] = ', str[0])
str[0] =  p

>>>str = 'python programming'
>>>print('str[-1] = ', str[-1])
str[-1] =  g

>>>str = 'python programming'
>>>print('str[1:5] = ', str[1:5])
str[1:5] =  ytho
```

```
>>>str = 'python programming'
>>>print('str[5:-2] = ', str[5:-2])
str[5:-2] =  n programmi
```

Strings are immutable. This means that elements of a string cannot be changed once it has been assigned. We can simply reassign different strings to the same name.

```
>>> my_string='python'
>>> my_string[5]='a'          # results in TypeError: 'str' object does not support
                                item assignment
```

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

```
>>> my_string='python'
>>> print(my_string)
python

>>> del my_string[1]     #results in TypeError: 'str' object doesn't support item
                           deletion

>>> del my_string    # deletes the entire string
```

Operations on strings:

The following are some of the operations that can be performed over strings

Operator	Description	Example

+	The + operator can be used to concatenate two strings.
	Simply writing two string literals together also concatenates them.
		>>> str1='py'
		>>> str2='thon'
		>>> str3=str1+str2
		>>> print(str3)
		python
		
		Ex2:
		>>> 'py'+'thon'
		'python'
		
		Ex3:
		>>> 'py' 'thon'
		'python'

*	The * operator can be used to repeat the string for a given number of times.	>>> str1='python'
		>>> str1*3
		'pythonpythonpython'

[ ]	Returns the character at the given index	>>>str = 'python programming' >>>print(str[0]) p
[:]	Fetches the characters in the range specified by two index operands separated by the : symbol	>>>str = 'python programming' >>>print(str[1:5]) ytho
in	Returns True if a sub string exists within a string else returns False	>>> 'a' in 'program' True
not in	Returns True if a sub string does not exists within a string else returns False	>>> 'at' not in 'battle' False

len(string)	returns the length of the string	Ex:
		>>> string="this is a good example"
		>>> len(string)
		22

More about Data Output:

Escape Characters:

An *escape character* is a special character that is preceded with a **backslash (\),** appearing inside a string literal. When a string literal that contains escape characters is printed, the escape characters are treated as special commands that are embedded in the string.

Some of the escape characters are given below.

Escape Character	Effect
\n	Causes output to be advanced to the next line.
\t	Causes output to skip over to the next horizontal tab position.
\'	Causes a single quote mark to be printed.
\"	Causes a double quote mark to be printed.
\\	Causes a backslash character to be printed.

Ex:

Program	Output
>>>print('One\nTwo\nThree')	One Two Three

Ex:

Program	Output
>>> print('One\tTwo\tThree')	One Two Three

String formatting:

Python uses C-style string formatting to create new, formatted strings. The string format operator "%" is used to perform string formatting.

To format a sequence of data values, construct a format string that includes a format specifier for each datum and place the data values in a tuple following the % operator.

The form of this operation is as follows:

<format string> % (<datum–1>, ..., <datum–*n*>)

The format specifiers like %s ,%d etc, can be used

```
Ex1:
>>> name = "John"
>>> print("Hello, %s!" % name)
Hello, John!
```

```
Ex2:
>>> name = "John"
>>> age = 23
>>> print("%s is %d years old." % (name, age))
John is 23 years old.
```

```
Ex3:
>>> print("my name is %s and my age is %d"%('pavan',21))
my name is pavan and my age is 21
```

Note:
We can also specify field widths like **%<field width>s** in the format string. When the field width is positive, the data is right-justified and when the field width is negative, the data is left-justified. If the field width is less than or equal to the datum's print length in characters, no justification is added. The % operator works with this information to build and return a formatted string

```
Ex4:
>>> name = "John"
>>> print("Hello, %10s!" % name)
Hello,       John!
```

```
Ex5:
>>> name = "John"
>>> print("Hello, %-10s!" % name)
Hello, John      !
```

The list of complete set of specifiers or symbols which can be used along with % are given below.

Format Symbol	Conversion
%c	character
%s	string conversion via str prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer *lowercaseletters*
%X	hexadecimal integer *UPPERcaseletters*
%e	exponential notation *withlowercase'e'*
%E	exponential notation *withUPPERcase'E'*
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

String formatting using format( ) :

The format() method can handle complex string formatting more efficiently. The general syntax of the format() method is as follows:

string.format(str1, str2,...)

The string itself contains placeholders {}, in which the values of variables are successively inserted.

```
Ex:
>>>name="Bill"
>>>age=25
>>>"My name is {}  and I am {}years old.". format (name, age)
'My name is Bill and I am 25 years old.'
```

```
Ex:
>>>name="Bill"
>>>age=25
>>>myStr = "My name is {} and I am {} years old."
>>>myStr.format(name, age)
'my name is Bill and I am 25 years old.'
```

Formatting Numbers:

We can also use the built in format function for formatting numbers.

We can call the built-in format function, using two arguments to the function:

i) a numeric value, and
ii) a format specifier.

Here the *format specifier* is a **string** that contains special characters specifying how the numeric value should be formatted.

The format function returns a string containing the formatted number.

Ex: consider format(12345.6789, '.2f')

Here the first argument, which is the floating-point number 12345.6789, is the number that we want to format.

The second argument, which is the string '.2f', is the format specifier. The meaning of format specifier is

• The .2 specifies the precision. It indicates that we want to round the number to two decimal places.

• The f specifies that the data type of the number we are formatting is a floating-point number.

Suppose we want the number to be formatted with comma separators (a comma will be placed between all thousands.), we can insert a comma into the format specifier, as shown here:

```
>>> print(format(12345.6789, ',.2f'))
12,345.68

>>> print(format(123456789.456, ',.2f'))
123,456,789.46
```

The format specifier can also include a minimum field width, which is the minimum number of spaces that should be used to display the value.

For example, the following statement prints the number in a field that is 12 spaces wide

```
>>>print('The number is', format(12345.6789, '12,.2f'))
The number is    12,345.68
```

To format integers, the format specifier to be used is 'd' and we cannot specify the precision.

```
>>> print(format(123456123456, ',d'))     #number with comma separator
123,456,123,456

>>> print(format(123456, '10d'))     #number with  field width 10
   123456

>>> print(format(123456, '10,d'))   #number with field width 10 and comma
  123,456
```

Character sets:

All data and instructions in a program are translated to binary numbers before being run on a real computer. To support this translation, **the characters in a string each map to an integer value**. This mapping is defined in character sets, among them the **ASCII set (**ASCII stands for American Standard Code for Information Interchange).and the **Unicode set**.

In the 1960s, the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 127. As new function keys and some international characters were added to keyboards, the ASCII set doubled in size to 256 distinct values in the mid-1980s.

Then, when characters and symbols were added from languages other than English, the Unicode set was created to support 65,536 values in the early 1990s. Unicode supports more than 128,000 values at the present time.

Python 3 uses Unicode, this means Python 3 is capable of interpreting practically every character from most world languages

Python's ord() function is to return the Unicode code point for a one-character string.

```
>>> ord('A')
65

>>> ord('Z')
90

>>> ord('a')
97
>>> ord('z')
122
```

Python's chr() return a Unicode string of one character with
Ordinal i; 0 <= i <= 0x10ffff

```
>>> chr(65)
'A'
>>> chr(90)
'Z'
```

Using Functions and Modules:

Function:
A function is block of re-usable code to perform specific task.

In python, functions are classified into two types. They are
a) Built-in functions
b) User defined functions

a) **Built-in functions:**

The Python interpreter has a number of functions built into it that are always available. These functions are called Built-in-functions. If you want to use one of these built-in functions in a program, you simply call the function. The built-in-functions available in python3 are given below.

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Many of the functions in the standard library, however, are stored in files that are known as *modules*. *A module is a file with .py extension containing Python definitions and statements. so any Python file can be referenced as a module. A module can define functions, classes and variables that can then be utilized in other Python programs*

In order to call a function that is stored in a module, you have to write an import statement at the top of your program. An import statement tells the interpreter the name of the module that contains the function. For example, one of the Python standard modules is named math.

The math module contains various mathematical functions that work with floating-point numbers. If you want to use any of the math module's functions in a program, you should write the following import statement at the top of the program:

import math

This statement causes the interpreter to load the contents of the math module into memory and makes all the functions in the math module available to the program.

To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource.

Ex1:
To use the value of pi from the math module, we can write the following code: math.pi.

```
>>> math.pi
3.141592653589793
```

Ex2:
The function say 'factorial' in the math module can be used as
math.factorial()

```
>>> math.factorial(5)
120
```

Control Statements:

In Python program, statements are normally executed sequentially in the order in which they appear. But sometimes, we have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are satisfied.

For this, Python language provides Control statements(or decision making statements) which alter the flow of execution and provide better control to the programmer on the flow of execution. They are two types. They are

1. Conditional statements
2. Iterative statements

Conditional (Selection) statements:

Conditional statements are used to execute a statement or a group of statement based on certain conditions.

1.**Simple if statement:**

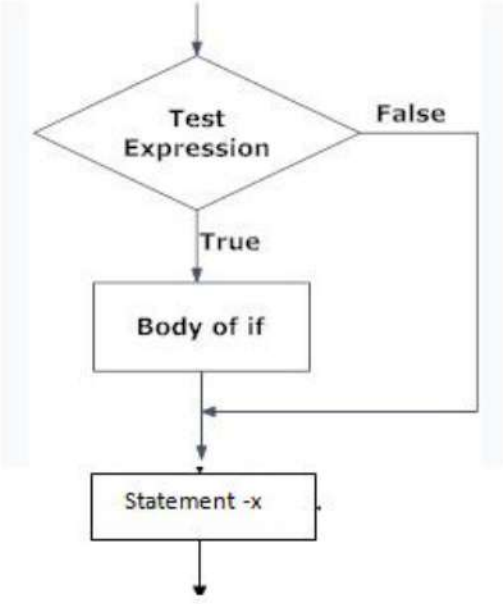
The syntax of simple if statement is

```
if test-expression:
        Statement-1
        Statement-2
        Statement-3
        .....
        .....
        Statement-n
Statement-x
```

Here, the test-expression can be any valid python expression.

First test-expression will be evaluated. **If the expression is true** the statements that appear in the block following the if clause are executed and then control transfers to the next immediate statement of simple if i.e., statement-x.

If the expression is false then control transfers to execute the next immediate statement of simple if i.e., statement-x by skipping the statements in the if- block. This is illustrated in the following flowchart.



Example program:

Program	output
#program to illustrate simple if statement #program to print the absolute value of a number num=eval(input("enter a number")) if(num<0): num=-num print("absolute value of a number is",num)	Output1: enter a number5 absolute value of a number is 5 Output2: enter a number-45 absolute value of a number is 45

2. if…else statement:

The syntax of simple if…else statement is

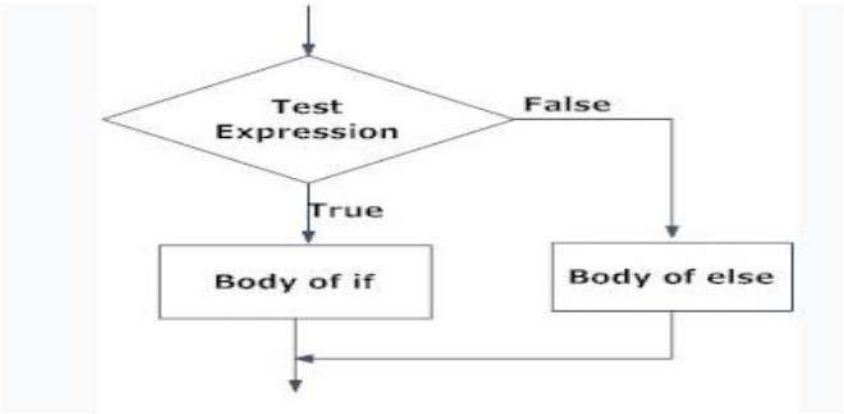
```
if test-expression:
        Statement-1
        Statement-2
        Statement-3
        ..............
        ..............
        Statement-n
else:
        Statement-1
        Statement-2
        Statement-3
        ..............
        ..............
        Statement-n

Statement-x
```

Here, the test-expression can be any valid python expression.

First test-expression will be evaluated. **If the expression is true** the statements that appear in the block following the if clause are executed and then control transfers to the next immediate statement of if..else i.e., statement-x.

If the expression is false the statements that appear in the block following the else clause are executed and then control transfers to the next

immediate statement of if..else i.e., statement-x This is illustrated in the following flowchart.



Example program:

Program	output

#program to illustrate if..else statement
#program to determine whether given integer is even or odd
 	
num=eval(input("enter an integer"))
if(num%2)==0 :
  print(num,"is an even number")
else:
  print(num,"is not an even number")
	Output1:
	enter an integer46
	46 is an even number
	
	Output2:
	enter an integer1001
	1001 is not an even number

3.if..elif…else statement:

The syntax of if..elif…else statement is

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
statement-x
```

When the statement executes, expression1 is tested. If expression1is true, the block of statements that immediately follow is executed, up to the elif clause. The rest of the structure is ignored.

If expression1 is false, the program jumps to the very next elif clause and tests expression2. If it is true, the block of statements that immediately follows is executed, up to the next elif clause. The rest of the structure is then ignored.

This process continues until a condition is found to be true, or no more elif clauses are left. If no expression is true, the block of statements following the else clause is executed.
There can be zero or more elif parts, and the else part is optional.

Program	output

#program illustrating if-elif-else statement
 	
var = eval(input("enter an integer"))
if var == 200:
  print("1 - Got a true expression value")
  print("var is",var)
 	
elif var == 150:
  print ("2 - Got a true expression value")
  print("var is",var)
 	
elif var == 100:
  print ("3 - Got a true expression value")
  print("var is",var)
 	
else:
  print ("4 - Got a false expression value")
  print("var is",var)
 	
print ("Good bye!")
	Output1:
	
	enter an integer200
	1 - Got a true expression value
	var is 200
	Good bye!
	
	Output2:
	
	enter an integer100
	3 - Got a true expression value
	var is 100
	Good bye!
	
	Output3:
	
	enter an integer45
	4 - Got a false expression value
	var is 45
	Good bye!

Repetition Structures or Loops :

While loop:

The syntax of while loop is

```
while expression:
        Block1 of statements
else:
        Block2 of statements
Statement-x
```

Here expression is any valid expression. Block of statements can contain one or more statements with uniform indentation. the else clause in while loop is optional. The while statement repeatedly execute block1 of statements as long as the expression is true. If the expression is false then the block2 of statements in the else clause (if present) gets executed and loop terminates by transferring the control to the next immediate statement of while loop i.e., statement –x.

While loop can be terminated with a break statement. In such case, the else part is also ignored. Hence a while loop's else clause runs if no break occurs and condition is false.

Program	output

#demo of while with else clause
count=1
while(count<=3):
  print("python programming")
  count =count+1
else:
  print("else clause statement is executed")
print("end of program")
	python programming
	python programming
	python programming
	else clause statement is executed
	end of program

Program	output

#demo of while with else clause and break
count=1
while(count<=3):
  print("python programming")
  count =count+1
  if count==3:
  break
else:
  print("else clause statement is executed")
print("end of program")
	python programming
	python programming
	end of program

for loop:

The for loop is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object.

```
for iterating_var in sequence:
        Block1 statements
else:
        Block2 statements
Statement-x
```

Block of statements can contain one or more statements with uniform indentation. the else clause in for loop is optional. If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the block1 statements gets executed. Each item in the list is assigned to *iterating_var*, and the block1 statements gets executed until the entire sequence is exhausted.

After the entire sequence is exhausted, the block2 of statements in the else clause (if present) gets executed and loop terminates by transferring the control to the next immediate statement of for loop i.e., Statement –x

for loop can be terminated with a break statement. In such case, the else part is also ignored. Hence a for loop's else clause runs if no break occurs and sequence is exhausted.

program

```
numbers = [11,33,55,39,55,75,36,21,23,41,13]

for num in numbers:
   if num%2 == 0:
      print ('the list contains an even number',num)
      break
else:
   print ('the list doesnot contain even number')

print("exit")
```

Output:

```
the list contains an even number 36
exit
```

Program:

```
#program that prints out the decimal equivalents of 1/2, 1/3, 1/4, . . . ,1/10
for  i  in range(2,11):
   print("decimal equivalent of 1/",i, "is" ,1/i)
```

Output:

```
decimal equivalent of 1/ 2 is 0.5
decimal equivalent of 1/ 3 is 0.3333333333333333
decimal equivalent of 1/ 4 is 0.25
decimal equivalent of 1/ 5 is 0.2
decimal equivalent of 1/ 6 is 0.16666666666666666
decimal equivalent of 1/ 7 is 0.14285714285714285
decimal equivalent of 1/ 8 is 0.125
decimal equivalent of 1/ 9 is 0.1111111111111111
decimal equivalent of 1/ 10 is 0.1
```

```
# program using a for loop that loops over a sequence.
str="cse"
print("the characters in the string are")
for i in str:
   print(i)
list1=["civil","eee","mech","ece","cse"]
print("the items in the list are")
for i in list1:
   print(i)
```

Output:

```
the characters in the string are
c
s
e
the items in the list are
civil
eee
mech
ece
cse
```

break statement:

The break statement breaks out of the innermost enclosing for or while loop. When the loop is terminated by a break statement, the else clause of the loop (if present) is not executed.

Program	output

#program to illustrate break statement
 	
for i in range(1,11):
 	
  if (i==5):
 	
  break
 	
  print(i,end=" ")
 	
print("\ndone")
	1 2 3 4
	
	done

continue statement:

The continue statement is used to end the current iteration in a for loop (or a while loop) and continues with the next iteration of the loop

. Program	output

#program to illustrate continue statement
 	
for i in range(1,11):
 	
  if (i==5):
 	
  continue
 	
  print(i,end=" ")
 	
print("\ndone")
	1 2 3 4 6 7 8 9 10
	
	done

pass statement:

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

Program	output

#program to illustrate pass statement
 	
for i in range(1,11):
  if (i==5):
  pass
  print(i,end=" ")
print("\ndone")
	1 2 3 4 5 6 7 8 9 10
	done

Calculating a Running Total:

In Many programming tasks, calculating the total of a series of numbers is required. The program would read the series of numbers as input and calculate the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator. When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop.

```
# This program calculates the sum of a series of numbers entered by the user.

total = 0.0
max=int(input("enter the number of elements:"))
print('This program calculates the sum of')
print(max, 'numbers you will enter.')
for counter in range(max):
    number = int(input('Enter a number: '))
    total = total + number
print('The total is', total)
```

Output:

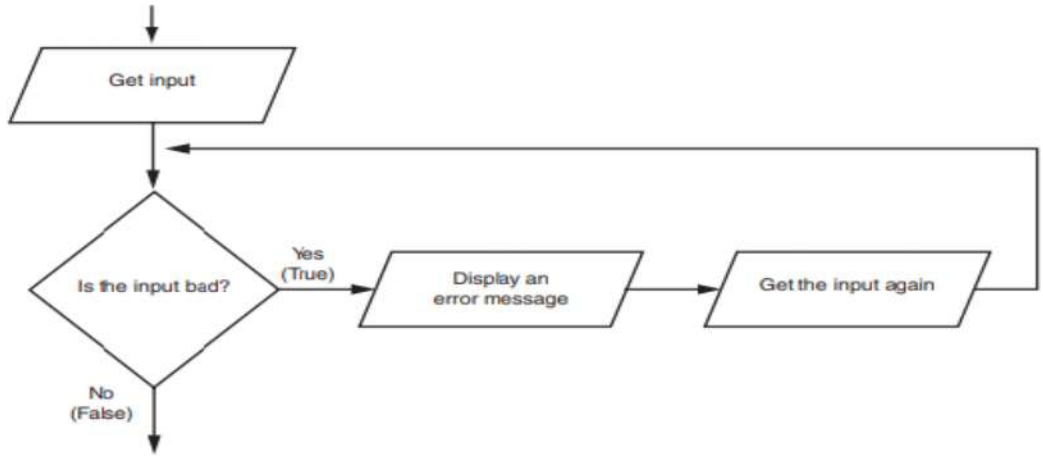
```
enter the number of elements:5
This program calculates the sum of
5 numbers you will enter.
Enter a number: 5
Enter a number: 6
Enter a number: 7
Enter a number: 8
Enter a number: 4
The total is 30.0
```

Input Validation Loops:

When input is given to a program, it should be inspected before it is processed. If the input is invalid, the program should discard it and prompt the user to enter the correct data. This process is known as input validation.

So Input validation is the process of inspecting data that has been input to a program, to make sure it is valid before it is used in a computation. Input validation is commonly done with a loop that iterates as long as an input variable references bad data.

The Logic containing an input validation loop is illustrated in the following figure.



In this technique, the input is read, and then a loop is executed. If the input data is bad, the loop executes its block of statements displaying an error message so the user will know that the input was invalid, and then it reads the new input. The loop will continue to iterate until the user enters a valid input.

Ex:
Suppose we were asked to write a program that reads a test score which should not be less than 0. The following code shows how you can use an input validation loop to reject any input value that is less than 0.

```
# Get a test score.
score = int(input('Enter a test score: '))
while score < 0:       #validating the input using loop
   print('ERROR: The score cannot be negative.')
   score = int(input('Enter the correct score: '))
```

Nested Loops:

A loop that is inside another loop is called a nested loop.

The syntax of nested for loop is given as follows.

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

The syntax for nested while loop is given as follows.

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

Also , any type of loop can be inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Ex:

```
for row in range(8):
   for col in range(6):
      print('*', end='')
   print( )
```

Output:

```
******
******
******
******
******
******
******
******
```

Ex:

```
# This program displays a triangle pattern.
BASE_SIZE = 8
for r in range(BASE_SIZE):
    for c in range(r + 1):
        print('*', end='')
    print( )
```

Output:

```
*
**
***
****
*****
******
*******
********
```