

## PYTHON PROGRAMMING

### UNIT -1

**Introduction:** Introduction to Python, Program Development Cycle, Input, Processing, and Output, Displaying Output with the Print Function, Comments, Variables, Reading Input from the Keyboard, Performing Calculations, Operators. Type conversions, Expressions, More about Data Output.

**Data Types, and Expression:** Strings Assignment, and Comment, Numeric Data Types and Character Sets, Using functions and Modules.

**Decision Structures and Boolean Logic:** if, if-else, if-elif-else Statements, Nested Decision Structures, Comparing Strings, Logical Operators, Boolean Variables.

**Repetition Structures:** Introduction, while loop, for loop, Calculating a Running Total, Input Validation Loops, Nested Loops..

### Introduction to python :

Python is a general-purpose **High-Level, Interpreted, Interactive And Object-Oriented Scripting Language** It was created by Guido van Rossum during 1985- 1990.

### *Python's Benevolent Dictator For Life*

"Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered."

- Guido van Rossum



- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by **Guido Van Rossum** at CWI (Centrum Wiskunde & Informatica) in Netherland .
- Guido Van Rossum is a fan of "Monty Python's flying circus" this is a famous TV show in Netherlands.
- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Python is influenced by following programming languages:
  - ABC language.
  - Modula-3
- python is popularly used for development,scripting and software testing.

- Top IT companies like google, facebook, instagram, spotify and netflix among others,use python.
- Some Major applications :ML,AI,data science and IOT
- Major libraries: Numpy,scipy,keras tensorflow,Django & flask

### **Features in Python**

There are many features in Python, some of which are discussed below –

#### **1. Easy to code:**

Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is also a developer-friendly language.

#### **2. Free and Open Source:**

Python language is freely available at the official website and you can download it from the given download link below click on the **Download Python** keyword.

#### **Download Python**

Since it is open-source, this means that source code is also available to the public. So you can download it as, use it as well as share it.

#### **3. Object-Oriented Language:**

One of the key features of python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, objects encapsulation, etc.

#### **4. GUI Programming Support:**

Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python.

PyQt5 is the most popular option for creating graphical apps with Python.

#### **5. High-Level Language:**

Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.

#### **6. Extensible feature:**

Python is a **Extensible** language. We can write us some Python code into C or C++ language and also we can compile that code in C/C++ language.

#### **7. Python is Portable language:**

Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, **we can run this code on any platform.**

#### **8. Python is Integrated language:**

Python is also an Integrated language because we can easily integrated python with other languages like c, c++, etc.

#### **9. Interpreted Language:**

Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile python code this makes it easier to

debug our code. The source code of python is converted into an immediate form called **bytecode**.

### 10. Large Standard Library

Python has a large standard library which provides a rich set of module and functions so you do not have to write your own code for every single thing. There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.

### 11. Dynamically Typed Language:

Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

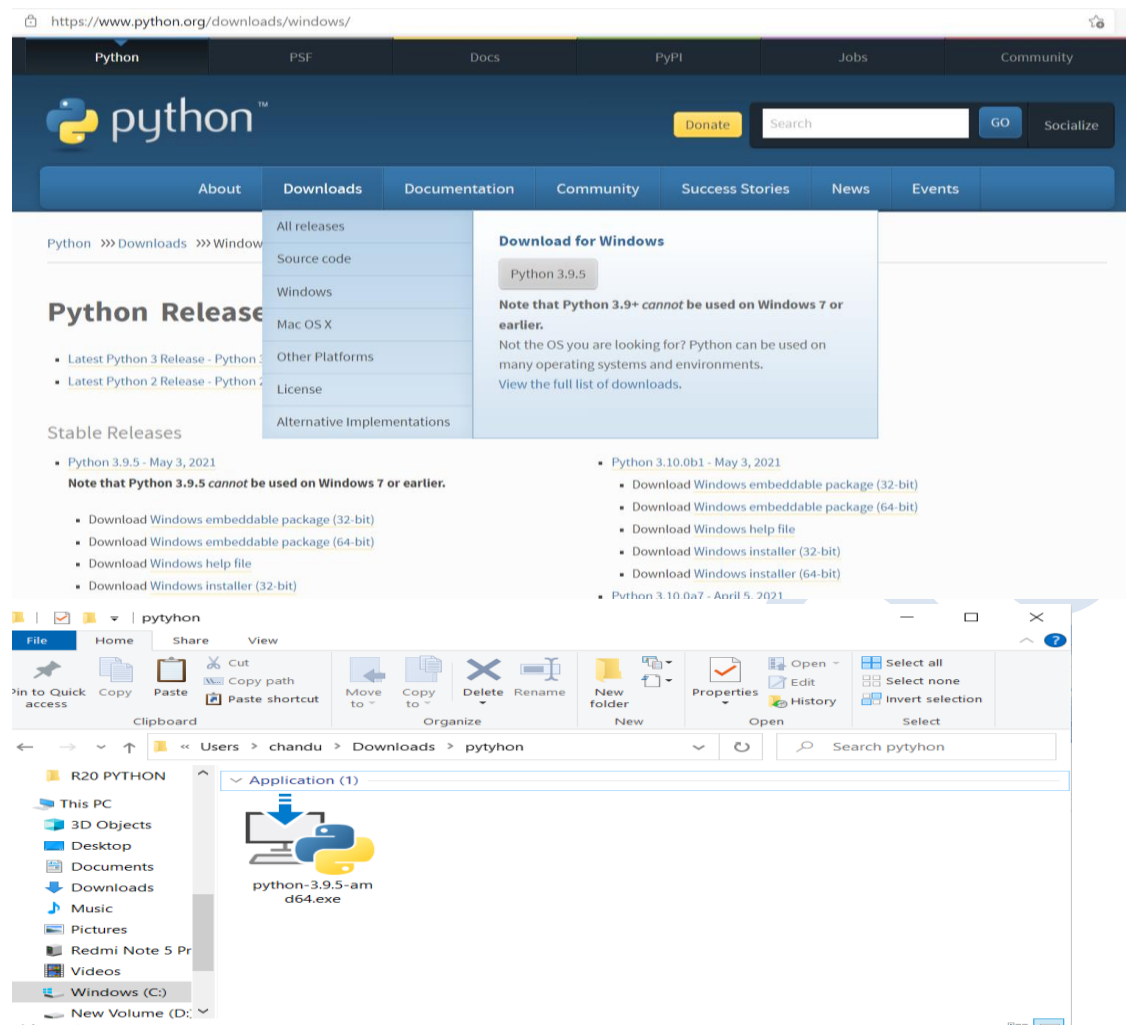


## Installing Python

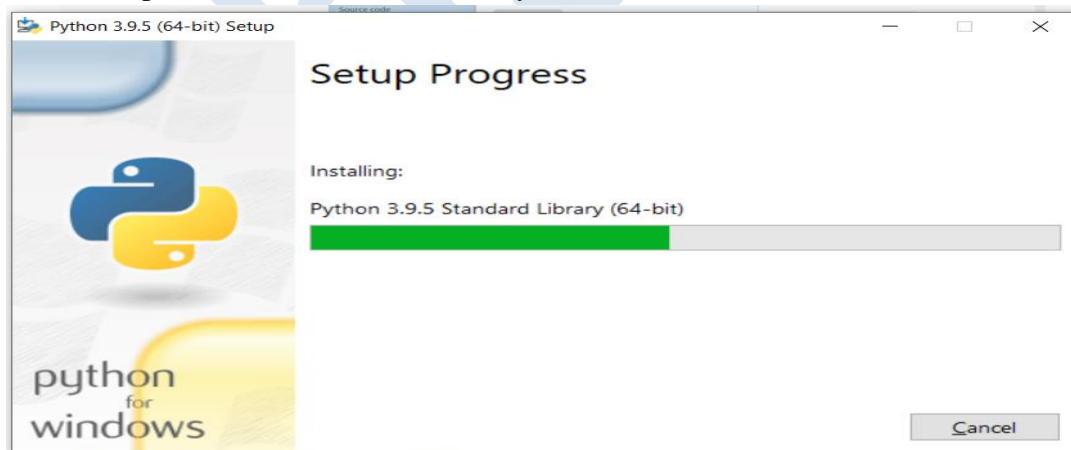
### Windows Operating System

Step1: Download the Python software from the following URL

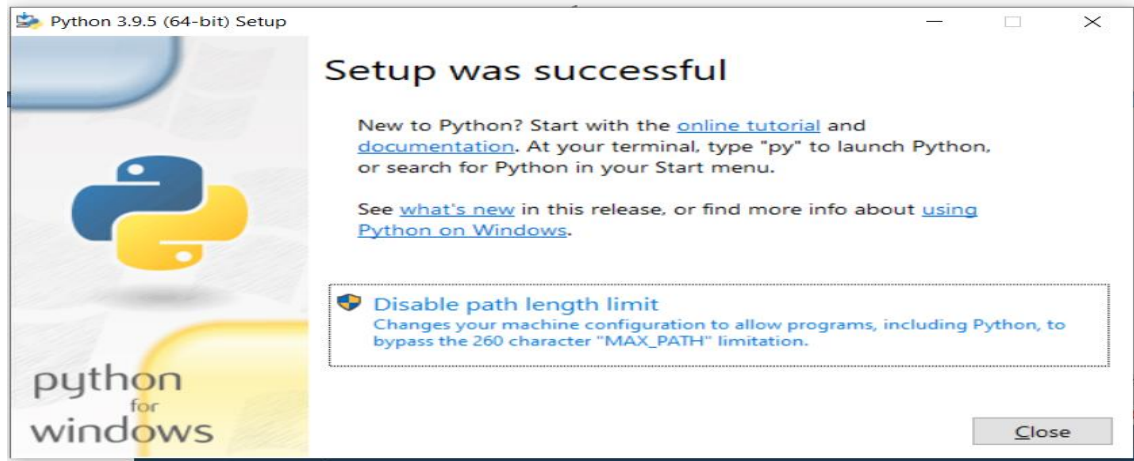
<https://www.python.org/downloads/windows/>



### Step2: Install the downloaded Python software



During installation, it will show the various components it is installing and move the progress bar towards completion. Soon, a new Python 3.9.5 (64-bit) Setup pop-up window will appear with a Setup was successfully message.



Step 3: Click the Close button.

Python should now be installed. After installing Python in Windows, you can start Python in two different modes, viz., Python (Command Line) and Python (IDLE).

#### **interactive shell:**

In Python, there are two options/methods for running code:

Interactive mode

Script mode

#### **Interactive Mode :**

Interactive mode is a command line shell. Typically the interactive mode is used to test the features of the python, or to run a smaller script that may not be reusable.

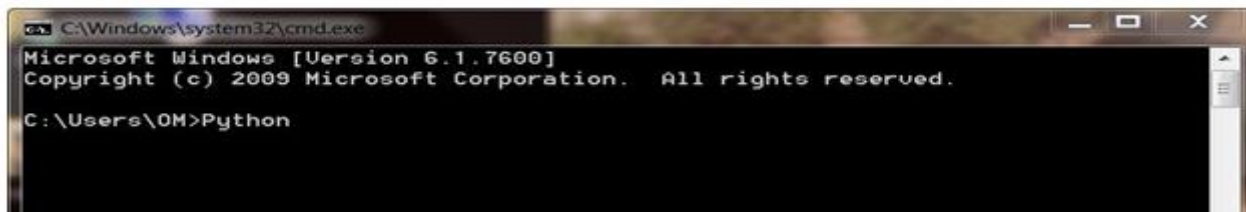
#### **Batch Mode or script mode :**

Batch mode is mainly used to develop business applications. In batch mode, we can write a group of python statements in any one of the following editors or IDEs

Editors : Notepad, Notepad++, editPlus, nano, gedit, IDLE, Etc..

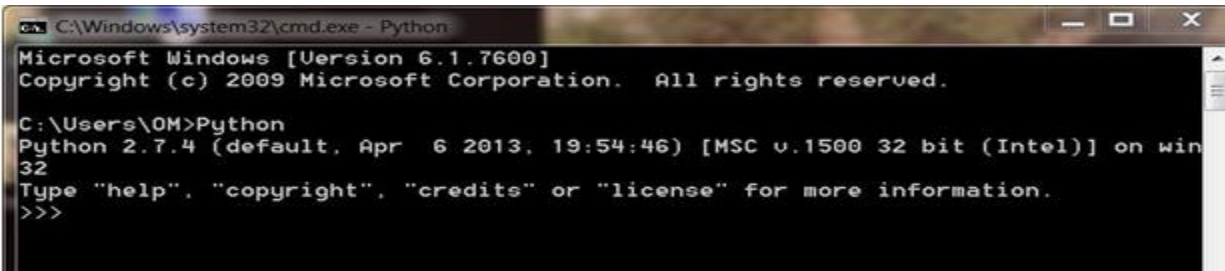
IDEs : pycharm, Eric, Eclipse, Netbeans Etc..

1) **Interactive Mode:** Python provides Interactive Shell to execute code immediately and produce output instantly. To get into this shell, write python in the command prompt and start working with Python.



Press Enter key and the Command Prompt will appear like:

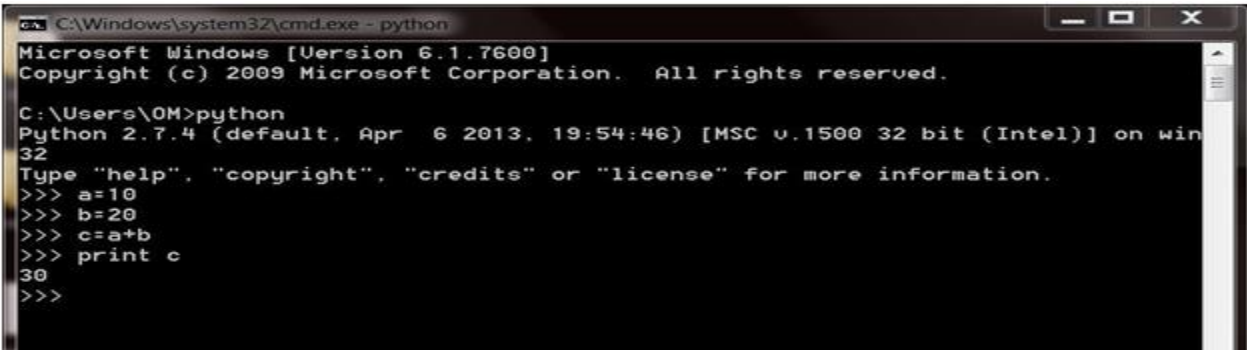




```
C:\Windows\system32\cmd.exe - Python
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\OM>Python
Python 2.7.4 (default, Apr 6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now we can execute our Python commands.Eg:



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

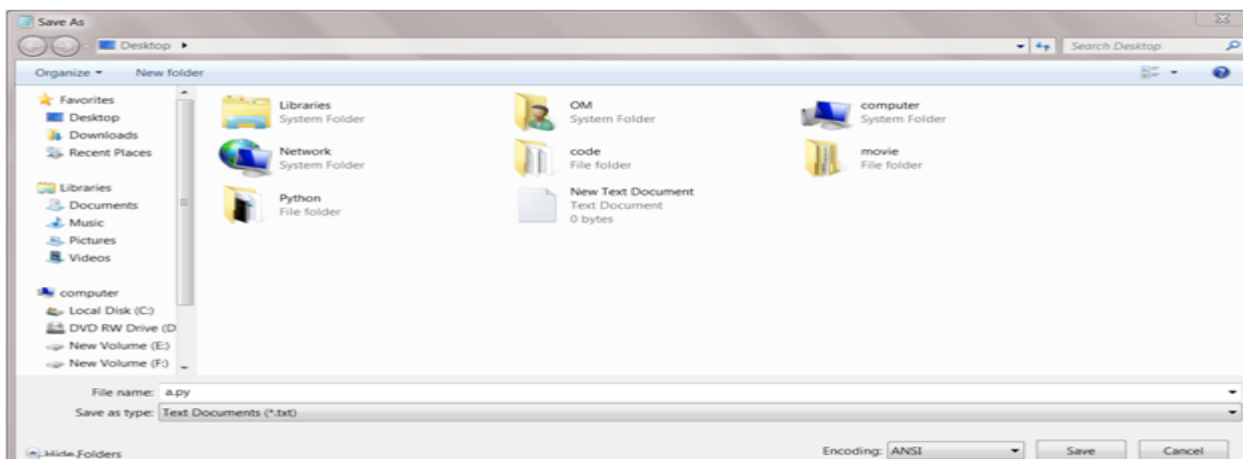
C:\Users\OM>python
Python 2.7.4 (default, Apr 6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> a=10
>>> b=20
>>> c=a+b
>>> print c
30
>>>
```

**2.Script Mode:** Using Script Mode, we can write our Python code in a separate file of any editor in our Operating System.

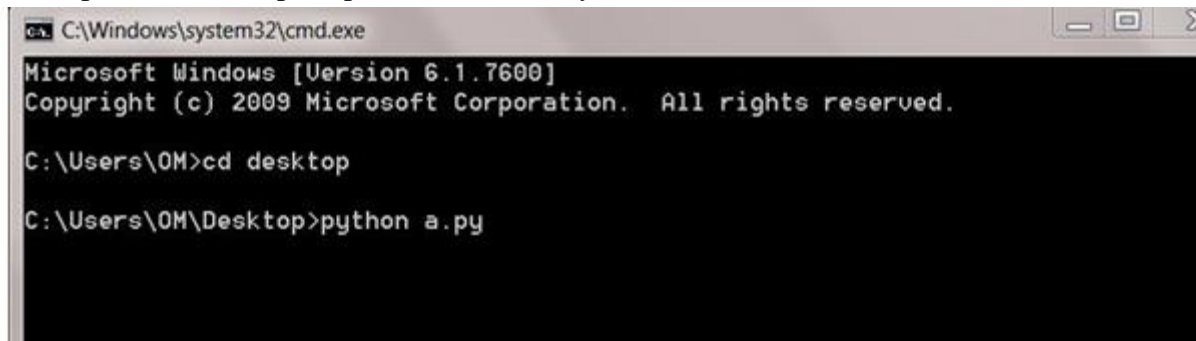


```
File Edit Format View Help
a=10
b=20
c=a+b
print c
```

Save it by .py extension.



Now open Command prompt and execute it by :



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\OM>cd desktop
C:\Users\OM\Desktop>python a.py
```

NOTE: Path in the command prompt should be location of saved file. where you have saved your file. In the above case file should be saved at desktop.

### 3 Using IDE (Integrated Development Environment)

We can execute our Python code using a Graphical User Interface (GUI). We can use both Interactive as well as Script mode in IDE.

#### 1) Interactive Mode

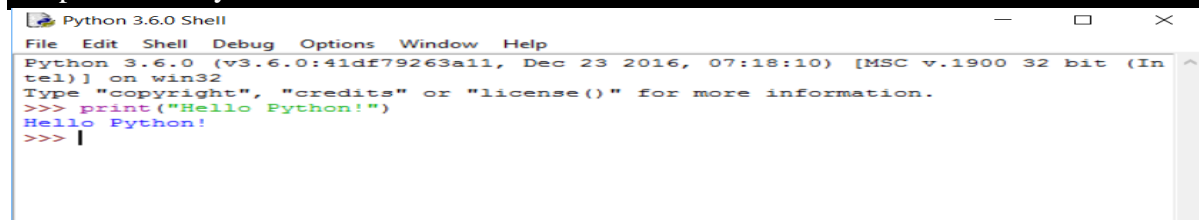
We type Python expression / statement / command after the prompt (>>>) and Python immediately responds with the output of it.

Executing first program on Python

1) Let's start with typing print "Hello Python!" after the prompt.

```
>>>print("Hello Python!")
```

Output: Hello Python!



```
Python 3.6.0 Shell
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello Python!")
Hello Python!
>>> |
```

### Editing, saving, and running a script (Script mode)

#### 2) Script Mode( Using Integrated Development Environment)

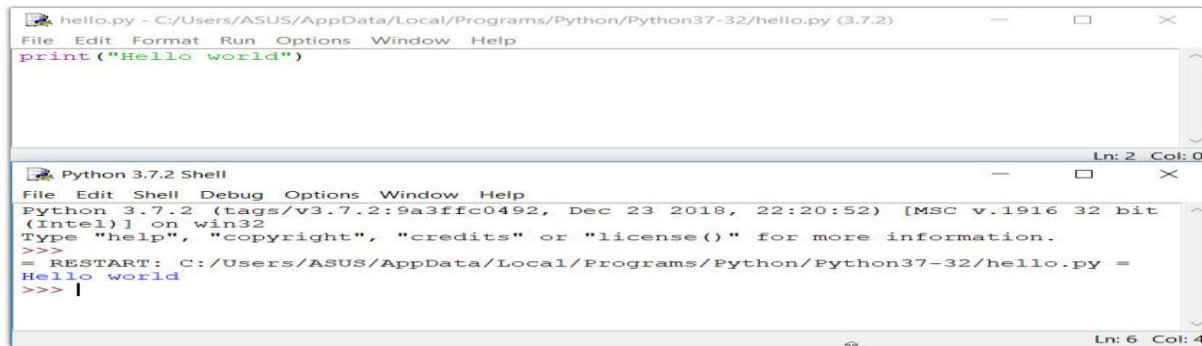
Running Python programs from a script file is known as running Python in script mode. You can write a sequence of instructions in one file and execute them.

Step 1: In Python IDLE's – Shell window, click on File and then on New File or just press CTRL + N.

Step 2: A series of instructions can be written in this window and saved.

Step 3: A Python program is executed only after it is saved with a specific file name. You can give any name to the file. However, the file name should end with .py

Step 4: To run the Python program, click on Run and then Run Module. Alternatively, just press CTRL + F5 to run the program.



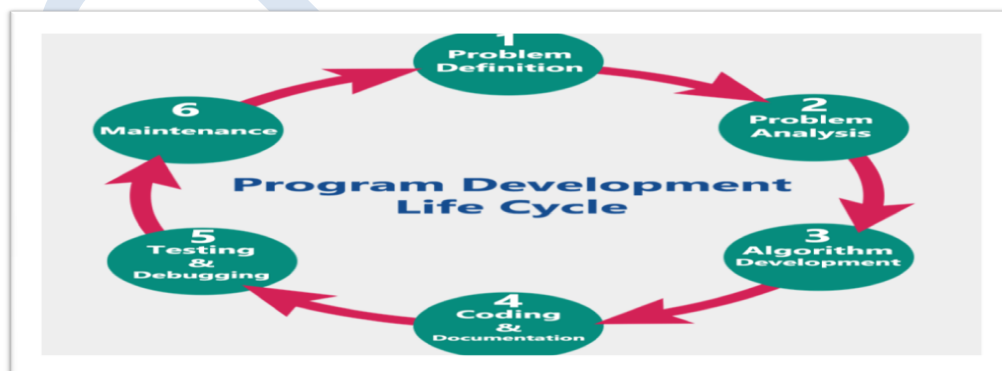
Running a Python program in IDLE

### Program Development Cycle

When we want to develop a program using any programming language, we follow a sequence of steps. These steps are called phases in program development. The program development life cycle is a set of steps or phases that are used to develop a program in any programming language.

Generally, the program development life cycle contains 6 phases, they are as follows....

- Problem Definition
- Problem Analysis
- Algorithm Development
- Coding & Documentation
- Testing & Debugging
- Maintenance



#### 1. Problem Definition

In this phase, we define the problem statement and we decide the boundaries of the problem. In this phase we need to understand the problem statement, what is our requirement, what should be



the output of the problem solution. These are defined in this first phase of the program development life cycle.

## 2. Problem Analysis

In phase 2, we determine the requirements like variables, functions, etc. to solve the problem. That means we gather the required resources to solve the problem defined in the problem definition phase. We also determine the bounds of the solution.

## 3. Algorithm Development

During this phase, we develop a step by step procedure to solve the problem using the specification given in the previous phase. This phase is very important for program development. That means we write the solution in step by step statements.

Adding two integer numbers	
Algorithm	Flowchart
i. Declare variables a,b,c ii. Read a, and b from keyboard iii. Add a, b and store result in c iv. Display result	<pre> graph TD     Start([Start]) --&gt; Declare[Declare variables num1, num2 and sum]     Declare --&gt; Read[/Read num1 and num2/]     Read --&gt; Sum[sum=a+b]     Sum --&gt; Display[/Display sum/]     Display --&gt; Stop([Stop])           </pre>

## 4. Coding & Documentation

This phase uses a programming language to write or implement the actual programming instructions for the steps defined in the previous phase. In this phase, we construct the actual program. That means we write the program to solve the given problem using programming languages like C, C++, Java, PYTHON etc.,

<i>adding two integers numbers in Python</i>
<pre> a=int(input('Enter          a')) b=int(input('Enter b')) c=a+b print('The sum is',c)           </pre>

## 5. Testing & Debugging

During this phase, we check whether the code written in the previous step is solving the specified problem or not. That means we test the program whether it is solving the problem for various input data values or not. We also test whether it is providing the desired output or not.

## 6. Maintenance

During this phase, the program is actively used by the users. If any enhancements found in this phase, all the phases are to be repeated to make the enhancements. That means in this phase, the solution (program) is used by the end-user. If the user encounters any problem or wants any

enhancement, then we need to repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

### **Input, Processing, and Output**

input .... is something the program receives, usually from the user or from a file.

Processing...is what happens when the program does something with the input.

Output.... is something the program produces, usually to a screen, to a file, or to both.

The input function is used in all latest version of the Python. It takes the input from the user and then evaluates the expression. The [Python](#) interpreter automatically identifies the whether a user input a string, a number, or a list. Let's understand the following example.

#### **Example –1**

```
>>> name=input("enter your name:")
enter your name:sai
>>> print(name)
sai
>>>
```

#### **Python Input**

To allow flexibility, we might want to **take the input from the user**. In Python, we have the `input()` function to allow this. The syntax for `input()` is:

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

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

Here, we can see that the entered value 10 is **a string, not a number**. To convert this into a number we can use `int()` or `float()` functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

#### **Example input() with int, float, string:**

```
>>> name=input("enter your name:")
enter your name:sai
>>> age=int(input("enter your age:"))
enter your age:19
>>> marks=float(input('enter your marks'))
enter your marks:90
```

```
>>> print(marks)
90.0
>>> print(age)
19
>>> print(name)
sai
```

### Displaying Output with the Print Function

We use the print() function to output data to the standard output device (screen). .

An example of its use is given below.

```
print('This sentence is output to the screen')
```

#### Output

```
This sentence is output to the screen
```

Another example is given below:

```
a = 5
print('The value of a is', a)
```

#### Output

```
The value of a is 5
```

In the second print() statement, we can notice that space was added between the [string](#) and the value of variable a. This is by default, but we can change it.

The actual syntax of the print() function is:

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

Here, objects is the value(s) to be printed.

- sep='separator': (Optional) Specify how to separate the objects, if there is more than one. Default: ' '
- end='end': (Optional) Specify what to print at the end. Default : '\n'
- file: (Optional) An object with a write method. Default :sys.stdout
- flush: (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

Python automatically flushes the files when closing them. But you may want to flush the data before closing any file.

```
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4, sep='#', end='&')
```

#### Output

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

### Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the **str.format()** method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>> print("The value of x is {} and y is {}".format(x,y))
The value of x is 5 and y is 10
```

Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('bread','butter'))
print('I love {1} and {0}'.format('bread','butter'))
```

### Output

```
I love bread and butter
I love butter and bread
```

We can even use keyword arguments to format the string.

```
>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))
Hello John, Goodmorning
```

We can also format strings like the old printf() style used in [C programming language](#). We use the % operator to accomplish this.

```
>>> x = 12.3456789
>>> print("The value of x is %3.2f" %x)
The value of x is 12.35
>>> print("The value of x is %3.4f" %x)
The value of x is 12.3457
```

### Comments in Python Programming:

Comments **are the way to improve the readability of a code, by explaining what we have done in code in simple english.**

A comment is text that doesn't affect the outcome of a code, it is just a **piece of text** to let someone know what you have done in a program or what is being done in a block of code. , **by**

reading a comment you can understand the purpose of code much faster than by just going through the actual code.

```
.  
# This is just a text, it won't be executed.
```

```
print("Python comment example")
```

output: Python comment example

## Types of Comments in Python

There are two types of comments in Python.

1. Single line comment
2. Multiple line comment

### Single line comment

In python we use # special character to start the comment. Lets take few examples to understand the usage.

```
# This is just a comment. Anything written here is ignored by Python
```

### Multi-line comment:

We can have comments that extend up to multiple lines. One way is to use the hash(#) symbol at the beginning of each line. For example:

```
#This is a long comment  
#and it extends  
#to multiple lines
```

Another way of doing this is to use triple quotes, either ''' or """".

**These triple quotes are generally used for multi-line strings. But they can be used as a multi-line comment as well.**

```
'''  
This is a  
multi-line  
comment  
'''
```

### Python Comments Example

In this Python program we are seeing three types of comments. Single line comment, multi-line comment and the comment that is starting in the same line after the code.

```
'''  
We are writing a simple program here
```



First print statement.

This is a multiple line comment.

```
'''
```

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

```
# Second print statement
```

```
print("How are You all?")
```

```
print("Welcome to BeginnersBook") # Third print statement
```

**Output:**

Hello Guys

How are You all?

Welcome to BeginnersBook

**# character inside quotes**

When # character is encountered inside quotes, it is not considered as comment. For example:

```
print("#this is not a comment")
```

**Output:**

```
#this is not a comment
```

## Docstrings in Python

A docstring is short for documentation string.

Python docstrings (documentation strings) are the [string](#) literals that appear right after the definition of a function, method, class, or module.

Triple quotes are used while writing docstrings. For example:

```
def double(num):  
    """Function to double the value"""  
    return 2*num
```

Docstrings appear right after the definition of a function, class, or a module. This separates docstrings from multiline comments using triple quotes.

The docstrings are associated with the object as their `__doc__` attribute.

So, we can access the docstrings of the above function with the following lines of code:

```
def double(num):  
    """Function to double the value"""  
    return 2*num  
print(double.__doc__)
```

[Run Code](#)

**Output**

### Function to double the value

#### Tokens:

- The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
- The token is the smallest unit inside the given program.

There are following tokens in Python:

- Keywords.
- Identifiers.
- Literals.
- Operators.

#### Python Keywords (Reserved Words )

- Keywords are the reserved words in Python.
- We cannot use a keyword as a variable name, function name or any other identifier. .
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.7. This number can vary slightly over the course of time. There are 35 keywords in Python 3.8

All the keywords except **True, False and None** are in lowercase and they must be written as they are.

Python Keywords			
False	def	if	raise
None	del	import	return
True	elif	in	try
And	else	is	while
As	except	lambda	with
Assert	finally	nonlocal	yield
Break	for	not	async
Class	from	or	await
Continue	global	pass	

#### Python Variables

- **Variable** is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.
- In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.
- The first character of the variable must be an alphabet or underscore ( \_ )..

- It is recommended to use lowercase letters for the variable name. Rahul and rahul both are two different variables.
- Variables in Python are not subject to this restriction. **In Python, a variable may be assigned a value of one type and then later re-assigned a value of a different type:**

```
>>> var = 23.5
>>> print(var)
23.5
>>> var = "Now I'm a string"
>>> print(var)
Now I'm a string
```

### Python Identifiers

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

#### Rules for writing identifiers

- The first character of the variable 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).
- An identifier can be of any length.
- Multiple words can be separated using an underscore, like this\_is\_a\_long\_variable
- Identifier name must not contain special character (!, @, #, %, ^, &, \*).
- Keywords cannot be used as identifiers
- Examples of valid identifiers: a123, \_n, n\_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

```
global = 1
```

Output

```
File "<interactive input>", line 1
```

```
global = 1  SyntaxError: invalid syntax
```

### Declaring Variable and Assigning Values:

In Python, to create a variable, you just assign it a value and then start using it. Assignment is done with a single equals sign (=)

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

### Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments..

#### 1. Assigning single value to multiple variables

Eg:

```
x=y=z=50
print(x)
```

```
print(y)
print(z)
```

**Output:**

```
50
50
50
```

**2. Assigning multiple values to multiple variables:****Eg:**

```
a,b,c=5,10,15
print a
print b
print c
```

**Output:**

```
5
10
15
```

The values will be assigned in the order in which variables appear.

**Plus and concatenation operation on the variables**

```
x = 10
y = 20
print(x + y)
```

```
p = "Hello"
q = "World"
print(p + " " + q)
```

**Output:**

```
30
Hello World
```

**Python Indentation**

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

Indentation in Python refers to the (spaces and tabs) that are used at the beginning of a **statement**. The statements with the same indentation belong to the same group called a suite. Consider the example of a correctly indented Python code statement mentioned below.

Ex:

```
if a==1:
    print(a)
    if b==2:
```

```
print(b)
print('end')
```

In the above code, the first and last line of the statement is related to the same suite because there is no indentation in front of them. So after executing first "if statement", the Python interpreter will go into the next statement. If the condition is not true, it will execute the last line of the statement

### Reading Input from the Keyboard

Python provides us with two inbuilt functions to read the input from the keyboard.

- **input ( prompt )**
- **raw\_input ( prompt )**

**input ( ) :** This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For example –

```
# Python program showing
# a use of input()
val = input("Enter your value: ")
print(val)
```

#### Output:

```
Enter your value: 123
123
>>>
```

### How the input function works in Python :

- When input() function executes program flow will be stopped until the user has given an input.
- The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
- Whatever you enter as input, input function convert it into a string. if you enter an integer value still input() function convert it into a string. You need to explicitly convert it into an integer in your code using [typecasting](#).

#### Code:

```
# Program to check input
# type in Python
```



```
num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)
# Printing type of input value
print ("type of number", type(num))
print ("type of name", type(name1))
```

**Output :**

```
Enter number :123
123
Enter name : geeksforgeeks
geeksforgeeks
type of number <class 'str'>
type of name <class 'str'>
>>> |
```

**raw\_input ( ) :** This function works in older version (like Python 2.x). This function takes exactly what is typed from the keyboard, convert it to string and then return it to the variable in which we want to store. For example –

```
# Python program showing
# a use of raw_input()
g = raw_input("Enter your name : ")
print g
```

**Output :**

```
Enter your name : geeksforgeeks
geeksforgeeks
>>> |
```

Here, *g* is a variable which will get the string value, typed by user during the execution of program. Typing of data for the `raw_input()` function is terminated by enter key. We can use `raw_input()` to enter numeric data also. In that case we use typecasting. For more details on typecasting refer [this](#).

**Performing Calculations**

Performing calculations involving both integers and floating-point

numbers is called mixed-mode arithmetic. For instance, if a circle has radius 3, we compute the area as follows:

```
>>> 3.14*3*3 28.259999999999998
```

In the binary operation the less general type (int) will be automatically converted into more general type (float) before operation is performed. For example:

```
>>> 9*5.0 45.0
```

Here, 9 is integer, and 5.0 is float, then the less general type that is int will be converted into more general type that is float and the entire expression will result in float value.

The eval () function

We can even use eval () function to perform calculation at the interpreter. The expression is written inside the single quotes. For example:

```
>>> eval('45/9*2') 10.0
```

The eval() method parses the expression passed to this method and runs python expression (code) within the program.

Example

```
number = 9

# eval performs the multiplication passed as argument
square_number = eval('number * number')

print(square_number)

# Output: 81
```

### eval() Syntax

The syntax of eval() is:

```
eval(expression, globals=None, locals=None)
```

### eval() Parameters

The eval() function takes three parameters:

- **expression** - the string parsed and evaluated as a Python expression
- **globals** (optional) - a dictionary
- **locals** (optional)- a mapping object. Dictionary is the standard and commonly used mapping type in Python.

### eval() Return Value

The eval() method returns the result evaluated from the expression.

Example 1: How eval() works in Python

```
x = 1
print(eval('x + 1'))
```

## Output

2

Here, the `eval()` function evaluates the expression `x + 1` and `print` is used to display this value.

### Python Operators:

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language.

A sequence of operands and operators, like `a + b - 5`, is called an **Expression**. Python supports many operators for combining data objects into expressions.

Python provides a variety of operators, which are described as follows.

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

**Arithmetic Operators:** Arithmetic operators are used to perform arithmetic operations between two operands. It includes `+` (addition), `-` (subtraction), `*` (multiplication), `/` (divide), `%` (remainder), `//` (floor division), and exponent `(**)` operators.

Operator	Meaning	Example
<code>+</code>	Add two operands or unary plus	<code>x + y + 2</code>
<code>-</code>	Subtract right operand from the left or unary minus	<code>x - y - 2</code>
<code>*</code>	Multiply two operands	<code>x * y</code>
<code>/</code>	Divide left operand by the right one (always results into float)	<code>x / y</code>
<code>%</code>	Modulus - remainder of the division of left operand by the right	<code>x % y</code> (remainder of x/y)
<code>//</code>	Floor division - division that results into whole number	<code>x // y</code>

	adjusted to the left in the number line	
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

### Example 1: Arithmetic operators in Python

```
x = 15
y = 4
print('x + y =',x+y)
print('x - y =',x-y)
print('x * y =',x*y)
print('x / y =',x/y)
print('x // y =',x//y)
print('x ** y =',x**y)
```

#### Output

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

### Comparison operators:

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table..

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y

<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y
----	---	--------

**Example 2: Comparison operators in Python**

```
x = 10
```

```
y = 12
```

```
print('x > y is',x>y)
```

```
print('x < y is',x<y)
```

```
print('x == y is',x==y)
```

```
print('x != y is',x!=y)
```

```
print('x >= y is',x>=y)
```

```
# Output: x <= y is True
```

```
print('x <= y is',x<=y)
```

**Output**

```
x > y is False
```

```
x < y is True
```

```
x == y is False
```

```
x != y is True
```

```
x >= y is False
```

```
x <= y is True
```

**Logical operators**

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Meaning	Example
And	True if both the operands are true	x and y
Or	True if either of the operands is true	x or y
Not	True if operand is false (complements the operand)	not x

Truth table for and ,or

A	B	A or B	A and B
True	True	True	True
True	False	True	False



False	True	True	False	Truth tabel for not	
False	False	False	False	A	not A
				True	False
				False	True

### Example 3: Logical Operators in Python

**x = True**

**y = False**

**print('x and y is',x and y)**

**print('x or y is',x or y)**

**print('not x is',not x)**

#### Output

x and y is False

x or y is True

not x is False

### Bitwise operators

The bitwise operators perform bit by bit operation on the values of the two operands

For example, 2 is 10 in binary and 7 is 111.

**In the table below:** 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)

#### Example:

```
a = 60      # 60 = 0011 1100
b = 13      # 13 = 0000 1101
c = 0
c = a & b;   # 12 = 0000 1100
print ("Line 1 - Value of c is ", c)
```

```
c = a | b;   # 61 = 0011 1101
print ("Line 2 - Value of c is ", c)
```

```
c = a ^ b;   # 49 = 0011 0001
print ("Line 3 - Value of c is ", c)
```

```
c = ~a;      # -61 = 1100 0011
print ("Line 4 - Value of c is ", c)
```

```
c = a << 2;   # 240 = 1111 0000
print ("Line 5 - Value of c is ", c)
```

```
c = a >> 2;   # 15 = 0000 1111
print ("Line 6 - Value of c is ", c)
```

**output:**

Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15

**Assignment operators**

Assignment operators are used in Python to assign values to variables. The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

`a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left. There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Operator	Example	Equivalent to
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 5</code>	<code>x = x + 5</code>

-=	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

Example:

```

a = 21
b = 10
c = 0
c = a + b
print ("Line 1 - Value of c is ", c)
c += a
print ("Line 2 - Value of c is ", c)
c *= a
print( "Line 3 - Value of c is ", c)
c **= a
print ("Line 4 - Value of c is ", c)
c //= a
print ("Line 5 - Value of c is ", c)
output:

```

Line 1 - Value of c is 31

Line 2 - Value of c is 52

Line 3 - Value of c is 1092

Line 4 - Value of c is 2097152

Line 5 - Value of c is 99864

### Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

#### Identity operators:

- The identity operators are used to decide whether an element certain class or type.
- 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	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

#### Example1:

```
a = 20
```

```
b = 20
```

```
if ( a is b ):
    print ("Line 1 - a and b have same identity")
else:
    print ("Line 1 - a and b do not have same identity")

if ( id(a) == id(b) ):
    print ("Line 2 - a and b have same identity")
else:
    print ("Line 2 - a and b do not have same identity")
```

#### output:

```
Line 1 - a and b have same identity
```

```
Line 2 - a and b have same identity
```

### Membership operators

Python membership operators are used to check the membership of value inside a Python data structure (**string**, **list**, **tuple**, **set** and **dictionary**). If the value is present in the data structure, then the resulting value is true otherwise it returns false..

**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	Meaning	Example
In	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).	5 in x
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).	5 not in x

#### Example #5: Membership operators in Python

```
x = 'Hello world'
```

```
y = {1:'a',2:'b'}
```

```
# Output: True
```

```
print('H' in x)
```

```
# Output: True
```

```
print('hello' not in x)
```

```
# Output: True
```

```
print(1 in y)
```

```
# Output: False
```

```
print('a' in y)
```

#### Output

```
True
```

```
True
```

```
True
```

```
False
```

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.



## Type conversions

**Data conversion** in Python can happen in two ways: either you tell the compiler to convert a data type to some other type explicitly, or the compiler understands this by itself and does it for you

There are two-types of type-conversions in Python:

**Explicit Conversion:** In explicit conversion, users convert the data type in to their required type using `int()`, `float()`, `str()`, etc.

The general form of an explicit data type conversion is as follows:

`(required_data_type)(expression)`

**Implicit Conversion:** In implicit conversion, the python interpreter itself converts the lower data type to greater data type

example:

```
a_int = 1
b_float = 1.0
c_sum = a_int + b_float
print(c_sum)
print(type(c_sum))
```

## Python String to Int Conversion

It is crucial for a developer to know the different [Python Data types](#) and conversions between them. Let's take Python String to Int conversion itself, whenever we get the input from the user we typically use the **input() function**

The **input() function** reads a data entered by the user and converts it into a **string** and returns it. Even though the user has entered a **numeric value**, it will be automatically converted to **String** by Python, which cannot be used directly for any manipulation.

For example, if we want to **divide** the number entered by the user by **2**.

```
>>> num = input("Enter a number : ")
```

```
Enter a number : 10
```

```
>>> num
```

```
'10'
```

The value stored inside **num** is not an **integer 10** but the **string '10'**. Let's check the type of **num** for a confirmation.

```
>>> type(num)
```

```
<class 'str'>
```

If we try to divide the **num** by **2**, we will get an **Unsupported operand type error**.

```
>>> print( num /2)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#4>", line 1, in
```

```
print( num /2)
```

TypeError: unsupported operand type(s) for /: 'str' and 'int'

We cannot perform **division operation** on a **String**. So we need to perform the type conversion into the corresponding data type. Let's limit it to **int** for now.

### Using int() function to convert Python String to Int

For **integer** related operation, Python provides us with **int class**. We can use the **int()** **function** present in the **int class** to convert Python String to Int.

The **int()** function comes in two flavors

- **int(x)** – Returns the integer objects constructed from the argument passed, it returns **0** if no argument is passed. It will create an integer object with the default **base 10 (decimal)**.
- **int (x, base)** – This method also returns the **integer object** with the corresponding **base** passed as the argument.

Let's try to fix the issue which has happened in the above code. All we need to do is to just pass the **num** variable to the **int()** function to get it converted to an integer.

```
>>> num = int(num)
```

```
>>> type(num)
```

```
<class 'int'>
```

```
>>> print(num /2)
```

```
5.0
```

### Python Int to String Conversion

Converting a **Python Int to String** is pretty straight forward, we just need to pass the **integer** to the **str() function**. We don't have to worry about the **type of number** or its **bases**.

```
>>> num = 13
```

```
>>> strNum = str(num)
```

```
>>> strNum
```

```
'13'
```

### Converting to different Bases

We are able to successfully convert a string to an integer using the **int()** function, which is of **base 10**.

Let's try to convert the String to **different bases** other than decimal such as **binary (base 2)**, **octal (base 8)** and **hexadecimal (base 16)**

```
>>> val = '1101'
```

```
>>> base2int = int(val, 2)
```

```
>>> base2int
```

```
13
```

```
>>> val = '13'
```

```
>>> base8int = int(val, 8)
```

```
>>> base8int
```

```
11
```

```
>>> val = '1A'
>>> base16int = int(val, 16)
>>> base16int
26
```

### Conversion program by using predefined functions

Python built-in functions `bin()`, `oct()`, or `hex()` can be used to convert an integer to a binary, octal, or hexadecimal string respectively.

```
dec = int(input("Enter a decimal number: "))
```

```
print(bin(dec),"in binary.")
print(oct(dec),"in octal.")
print(hex(dec),"in hexadecimal.")
```

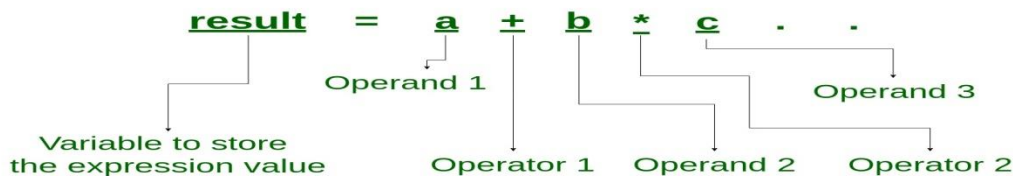
Output:

```
Enter a decimal number: 10
0b1010 in binary.
0o12 in octal.
0xa in hexadecimal.
```

### Python Expressions:

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

### What is an Expression?



### Example:

```
a+b
c
s-1/7*f
.
.
etc
```

Expressions are representations of value. They are different from statement in the fact that statements do something while expressions are representation of value. For example any string is also an expressions since it represents the value of the string as well.

Python has some advanced constructs through which you can represent values and hence these constructs are also called expressions.

Python expressions only contain identifiers, literals, and operators. So, what are these?

**Identifiers:** Any name that is used to define a class, function, variable module, or object is an identifier.

**Literals:** These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

### **Following are a few types of python expressions:List comprehension**

The syntax for list comprehension is shown below:

```
[ compute(var) for var in iterable ]
```

For example, the following code will get all the number within 10 and put them in a list.

```
>>> [x for x in range(10)]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### **Dictionary comprehension**

This is the same as list comprehension but will use curly braces:

```
{ k, v for k in iterable }
```

For example, the following code will get all the numbers within 5 as the keys and will keep the corresponding squares of those numbers as the values.

```
>>> {x:x**2 for x in range(5)}  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

### **Generator expression**

The syntax for generator expression is shown below:

```
( compute(var) for var in iterable )
```

For example, the following code will initialize a generator object that returns the values within 10 when the object is called.

```
>>> (x for x in range(10))
<generator object <genexpr> at 0x7fec47aee870>
>>> list(x for x in range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### Conditional Expressions

You can use the following construct for one-liner conditions:

```
true_value if Condition else false_value
```

Example:

```
>>> x = "1" if True else "2"
>>> x
'1'
```

### Python Operators Precedence :

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'

<code>^  </code>	Bitwise exclusive `OR' and regular `OR'
<code>&lt;= &lt; &gt; &gt;=</code>	Comparison operators
<code>&lt;&gt; == !=</code>	Equality operators
<code>= %= /= //=- += *= **=</code>	Assignment operators
<code>is is not</code>	Identity operators
<code>in not in</code>	Membership operators
<code>not or and</code>	Logical operators

For example, `x = 7 + 3 * 2`; here, x is assigned 13, not 20 because operator `*` has higher precedence than `+`, so it first multiplies `3*2` and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

### Example

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

```
e = (a + b) * c / d    #( 30 * 15 ) / 5
```

```
print( "Value of (a + b) * c / d is ", e)
```

```
e = ((a + b) * c) / d  # (30 * 15) / 5
```

```
print( "Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);  # (30) * (15/5)
```

```
print( "Value of (a + b) * (c / d) is ", e)
```

```
e = a + (b * c) / d;    # 20 + (150/5)
```

```
print( "Value of a + (b * c) / d is ", e)
```

### result :

```
Value of (a + b) * c / d is 90
```

```
Value of ((a + b) * c) / d is 90
```

```
Value of (a + b) * (c / d) is 90
```

```
Value of a + (b * c) / d is 50
```

**More about Data Output.**

There are several ways to present the output of a program, data can be printed to the console.

The print function will print everything as strings.

**Syntax:** `print(value(s), sep= ' ', end = '\n', file=file, flush=flush)`

Parameters:

**value(s):** Any value, and as many as you like. Will be converted to string before printed.

**sep='separator':** (Optional) Specify how to separate the objects, if there is more than one. Default : ' '

**end='end':** (Optional) Specify what to print at the end. Default : '\n'

**file: (Optional)** An object with a write method. Default : sys.stdout

**flush: (Optional)** A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

Example:

```
>>>print(10,20,30,40,sep='-',end='&')
```

**Output:**

```
10-20-30-40&
```

```
>>>print('apple',1,'mango',2,'orange',3,sep='@',end='#')
```

**Output:**

```
apple@1@mango@2@orange@3#
```

Even though there are different ways to print values in Python, we discuss two major string formats which are used inside the print() function to display the contents onto the console.

- str.format() method
- f-strings

**str.format()** –this function is used to insert value of a variable into another string and display it as a single string.

**Syntax:** str.format(p0,p1,..k0=val1,k1=val1..), where p0,p1 are called **positional**, and k0,k1 are called **keyword arguments**.

Positional arguments are accessed using the index, and keyword arguments are accessed using the name of the argument.

**f-strings** -Formatted strings or f-strings were introduced in Python

3.6. A f-string is a string literal that is prefixed with “f”.

**Using str.format() with positional arguments**

dataoutput.py	Output
#more about data output country=input('Enter your country') print('I Love my { }'.format(country))	Enter your country india I Love my india

Where { } are called **placeholder**. The value of the variable is placed inside the placeholder according to the position. The arguments are placed according to their position.

dataout.py	Output
branch=input('Enter branch name')  year=int(input('Enter the year of study'))  print('The branch name is {0} and the year is {1}'.format(branch,year))	Enter branch name CSE Enter the year of study 2 The branch name is CSE and the year is 2

**Using str.format() with keyword arguments**

It may be difficult for us to remember the order or positions of arguments. Keyword arguments are suitable when you are not sure of position, but know the names of the arguments.



**dataout1.py**

```
branch=input('Enter branch name')  
  
year=int(input('Enter the year of study'))  
  
print('The branch is {b} and year is {y}'.format(b=branch,y=year))
```

**Output:**

```
Enter branch name cse Enter  
  
the year of study 2  
  
The branch is cse and year is 2
```

**Using f-string**

Formatted strings or f-strings were introduced in Python 3.6. A f- string is a string literal that is prefixed with “f”. These strings may contain replacement fields, which are expressions enclosed within curly braces { }. The expressions are replaced with their values. An f at the beginning of the string tells Python to allow any valid variable names within the string.

**Dataout2.py**

```
branch=input('Enter branch name') year=int(input('Enter the year of  
study'))  
  
print( f 'The branch name is {branch} and the year is {year}')
```

**Output**

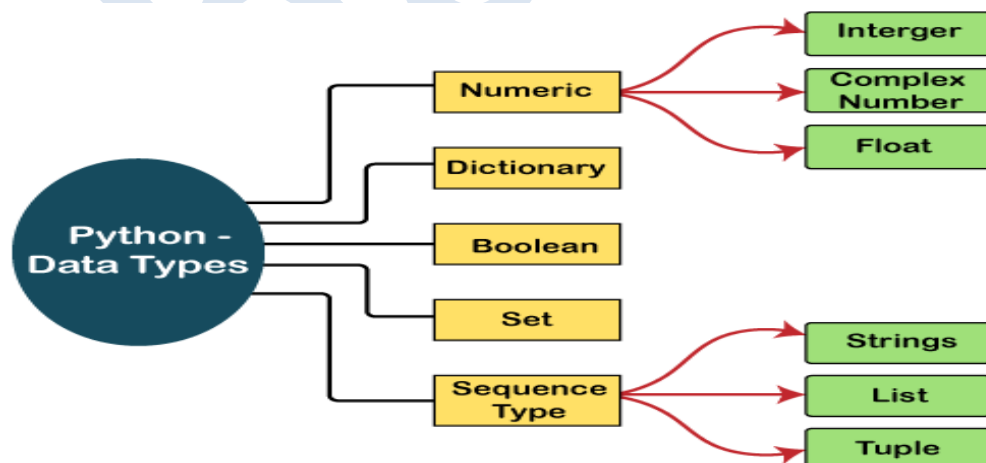
```
Enter branch name cse Enter the  
  
year of study 2  
  
The branch name is cse and the year is 2
```

## I. Data types, and expressions

- **Data type** defines the type of the variable, whether it is an integer variable, string variable, tuple, dictionary, list etc
- Data types represent a kind of value which determines what operations can be performed on that data. Numeric, non-numeric and Boolean (true/false) data are the most used data types.
- Number values, strings, and tuple are **immutable**, which means their contents can't be altered after creation.
- On the other hand, collection of items in a List or Dictionary object can be modified. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are **mutable objects**.
- You can get the data type of any object by using the `type()` function

the data types defined in Python are given below.

1. Numbers
2. Sequence Type
3. Boolean
4. Set
5. Dictionary



### String(immutable variables)

- The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

- String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.
- In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.
- The operator \* is known as a repetition operator as the operation *"Python" \*2* returns *'Python Python'*.

**Example - 1**

```
str = "string using double quotes"
print(str)
s = """A multiline
string"""
print(s)
```

**Output:**

string using double quotes

A multiline

String

Just like a list and tuple, the slicing operator [ ] can be used with strings. Strings, however, are immutable.

**Example:**

```
s = 'Hello world!'
print("s[4] = ", s[4])
print("s[6:11] = ", s[6:11])
s[5] = 'd'
```

**Output:**

s[4] = o

s[6:11] = world

Traceback (most recent call last):

File "<string>", line 11, in <module>

TypeError: 'str' object does not support item assignment

**1. String literals:**

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

**Example:**

```
"Aman" , '12345'
```

**Types of Strings:**

There are two types of Strings supported in Python:

**a) Single-line String-** Strings that are terminated within a single-line are known as Single line Strings.

**Example:**

```
text1='hello'
```

**b) Multi-line String** - A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

**1) Adding black slash at the end of each line.**

**Example:**

```
text1='hello\  
user'  
print(text1)  
'hellouser'
```

**2) Using triple quotation marks:-**

**Example:**

```
str2="""welcome  
to  
SSSIT"""  
print str2
```

**Output:**

```
welcome  
to  
SSSIT
```

### EscapeSequence

The newline character `\n` is called as *escape sequence*. Escape sequences are the way Python expresses special characters, such as the tab, the newline, and the backspace (delete key), as literals.

<i>ESCAPE SEQUENCE</i>	<i>MEANING</i>
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\t</code>	Horizontal tab
<code>\\</code>	The <code>\</code> Character
<code>\'</code>	Single Quotation mark
<code>\''</code>	Double quotation mark

### Comment

A comment is a piece of program text that the interpreter ignores but that provides useful documentation to programmers. At the very least, the author of

a program can include his or her name and a brief statement about the purpose of the program at the beginning of the program file. This type of comment, called a ***docstring***, is a multi-line string. This can be written inside Triple double quotes or Triple single quotes. In addition to docstrings, ***end-of-line*** comments can document a program. These comments begin with the # symbol and extend to the end of a line.

<b><i>Docstring</i></b>	<b><i>End-of-line</i></b>
<pre> """ Program: VowelTest.py Author : KSR Purpose:    Testing    whether    a given word contains any vowels or not """ </pre>	<pre> # read word from keyboard </pre>

## Numeric Data Types

- Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type.
- Python provides the **type()** function to know the data-type of the variable.
- Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python supports three types of numeric data.

1. **Int** - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float** - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
3. **complex** - A complex number contains an ordered pair, i.e.,  $x + iy$  where  $x$  and  $y$  denote the real and imaginary parts, respectively. The complex numbers like  $2.14j$ ,  $2.0 + 2.3j$ , etc

For example;

```
a = 5
```

```
print("The type of a", type(a))
```

```
print(" a is a integer number", isinstance(a,int))
```

```
b = 40.5
```

```
print("The type of b", type(b))
```

```
c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

**Output:**

```
The type of a <class 'int'>
a is a integer number: True
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

In Python we can print decimal equivalent of binary, octal and hexadecimal numbers using the prefixes.

0b(zero + 'b') and 0B(zero + 'B') – Binary Number

0o(zero + 'o') and 0O(zero + 'O') – Octal Number

0x(zero + 'x') and 0X(zero + 'X') – Hexadecimal Number

```
# integer equivalent of binary number 101
```

```
num = 0b101
```

```
print(num)
```

```
# integer equivalent of Octal number 32
```

```
num2 = 0o32
```

```
print(num2)
```

```
# integer equivalent of Hexadecimal number FF
```

```
num3 = 0xFF
```

```
print(num3)
```

**output:**

```
5
```

```
26
```

```
255
```

## 2.2 Python List

- Python Lists are similar to arrays in C.
- **List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible.** However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].
- We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (\*) works with the list in the same way as they were working with the strings.

```
a = [1, 2.2, 'python']
```

We can use the slicing operator [ ] to extract an item or a range of items from a list. The index starts from 0 in Python.

**Example:**

```
a = [5,10,15,20,25,30,35,40]
print("a[2] = ", a[2])
print("a[0:3] = ", a[0:3])
print("a[5:] = ", a[5:])
```

**Output**

```
a[2] = 15
a[0:3] = [5, 10, 15]
a[5:] = [30, 35, 40]
```

**Lists are mutable, meaning, the value of elements of a list can be altered.**

**Example:**

```
a = [1, 2, 3]
a[2] = 4
print(a)
```

**Output**

```
[1, 2, 4]
```

**2.3 Python Tuple(immutable variables)**

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

Tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically.

**Syntax:**

```
tuple = (value1, value2, value3,...valueN)
t = (5,'program', 1+3j)
```

We can use the slicing operator [] to extract items but we cannot change its value.

**Example:**

```
t = (5,'program', 1+3j)
print("t[1] = ", t[1])
print("t[1:] = ", t[1:])
print("t[0:3] = ", t[0:3])
# Tuple concatenation using + operator
print (t + t)
# Tuple repetition using * operator
print (t * 3)
t[0] = 10
```

**Output**

```
t[1] = program
t[1:] = ('program', (1+3j))
t[0:3] = (5, 'program', (1+3j))
(5, 'program', (1+3j), 5, 'program', (1+3j))
(5, 'program', (1+3j), 5, 'program', (1+3j), 5, 'program', (1+3j))
Traceback (most recent call last):
  t[0] = 10
```

TypeError: 'tuple' object does not support item assignment

**3. Python Dictionary**

- Dictionary is an unordered collection of key-value pairs.
- It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.
- In Python, dictionaries are defined within braces { } with each item being a pair in the form key:value. Key and value can be of any type.

**Syntax:**

```
dict = { key1:value1, key2:value2,...keyN:valueN }
```

```
>>> d = { 1:'value','key':2}
```

```
>>> type(d)
```

```
<class 'dict'>
```

We use key to retrieve the respective value. But not the other way around.

**Example:**

```
d = { 1:'value','key':2}
```

```
print(type(d))
```

```
print("d[1] = ", d[1]);
```

```
print("d['key'] = ", d['key']);
```

```
# Generates error
```

```
print("d[2] = ", d[2]);
```

**Output**

```
<class 'dict'>
```

```
d[1] = value
```

```
d['key'] = 2
```

Traceback (most recent call last):

```
File "<string>", line 9, in <module>
```

KeyError: 2

**4. Boolean**



Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'.

**Example:**

```
# Python program to check the boolean type
print(type(True))
print(type(False))
print(false)
```

**Output:**

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

**5.Python Set**

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

**Example:**

```
a = {5,2,3,1,4}
# printing set variable
print("a = ", a)
# data type of variable a
print(type(a))
```

**Output**

```
a = {1, 2, 3, 4, 5}
<class 'set'>
```

We can perform set operations like union, intersection on two sets. Sets have unique values. They eliminate duplicates.

**Example:**

```
a = {1,2,2,3,3,3}
print(a)
```

**Output**

```
{1, 2, 3}
```

Since, set are unordered collection, indexing has no meaning. Hence, the slicing operator [] does not work.

```
>>> a = {1,2,3}
>>> a[1]
```

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

TypeError: 'set' object does not support indexing

## Python Character Set

Character set is the set of valid characters that a language can recognize. A character represents any letter, digit or any other symbol. Python has the following character sets:

Letters – A to Z, a to z

Digits – 0 to 9

Special Symbols - + - \* / etc.

Whitespaces – Blank Space, tab, carriage return, newline, formfeed

Other characters – Python can process all ASCII and Unicode characters as part of data or literals.

## Using functions and Modules.

A function is a chunk of code that can be called by name to perform a task. Functions often require arguments, that is, specific data values, to perform their tasks.

Arguments are also known as parameters

The process of sending a result back to another part of a program is known as returning a value.

For example, the argument in the function call `round(6.6)` is the value 6.6, and the value returned is 7. When an argument is an expression, it is first evaluated, and then its value is passed to the function for further processing. For instance, the function call `abs(4 - 5)` first evaluates the expression `4 - 5` and then passes the result, -1, to `abs`. Finally, `abs` returns 1.

Functions and other resources are placed in components called modules. Functions like `abs()` and `round()` from the `__builtin__` module are always available for use, whereas the programmer must explicitly import other functions from the modules where they are defined. The `math` module includes several functions that perform basic mathematical operations

## Python Import

A module is a file containing Python definitions and statements. [Python modules](#) have a filename and end with the extension `.py`.

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the `import` keyword to do this.

For example, we can import the `math` module by typing the following line:

```
import math
```

We can use the module in the following ways:

```
import math
print(math.pi)
```

## Output

```
3.141592653589793
```

Now all the definitions inside math module are available in our scope. We can also import some specific attributes and functions only, using the from keyword. For example:

```
>>> from math import pi
>>> pi
3.141592653589793
```

## Decision Structures and Boolean Logic:

### Python Decision making statements or Control statements :

- Decision making is the most important aspect of almost all the programming languages. The decisions are made on the validity of the particular conditions.
- Condition checking is the backbone of decision making.
- Conditional Statement in Python perform different actions depending on whether a specific Boolean constraint or condition evaluates to true or false.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
If-elif-else	The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

### Indentation in Python

For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to **declare a block**. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. .

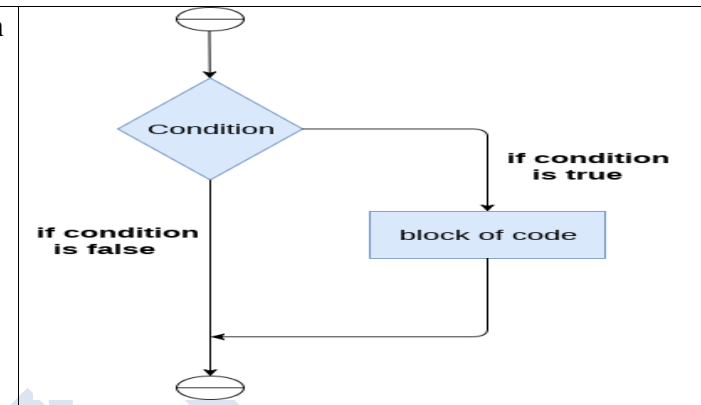
### The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block.

The condition of if statement can be any valid logical expression which can be either evaluated to true or false.

The syntax of the if-statement is given below.

```
if expression:
    statement
```



### Example 1

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
```

**Output:**

```
enter the number?10
Number is even
```

### Example 2 : Program to print the largest of the three numbers.

```
a = int(input("Enter a? "));
b = int(input("Enter b? "));
c = int(input("Enter c? "));
if a>b and a>c:
    print("a is largest");
if b>a and b>c:
    print("b is largest");
if c>a and c>b:
    print("c is largest");
```

**Output:**

```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

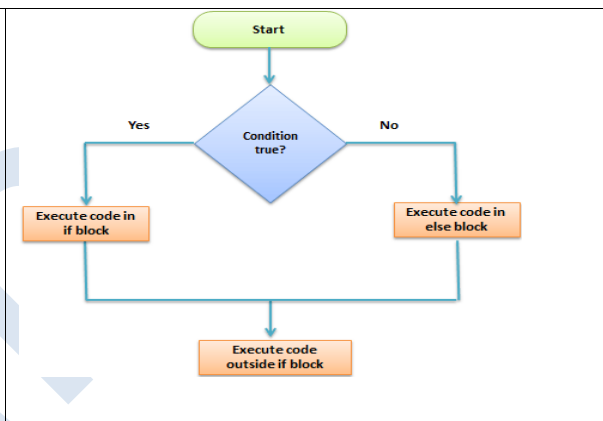
**The if-else statement**

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

The syntax of the if-else statement is given below.

```
if condition:
    #block of statements
else:
    #another block of statements (else-
    block)
```



**Example 1 :** Program to check whether a person is eligible to vote or not.

```
age = int(input("Enter your age? "))
if age >= 18:
    print("You are eligible to vote !!")
else:
    print("Sorry! you have to wait !!")
```

**Output:**

```
Enter your age? 90
You are eligible to vote !!
```

**Example 2:** Program to check whether a number is even or not.

```
num = int(input("enter the number?"))
if num % 2 == 0:
    print("Number is even...")
else:
    print("Number is odd...")
```

**Output:**

```
enter the number?10
Number is even
```

**The elif statement**

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax and flowchart of the elif statement is given below.

```
if[boolean expression]:
    [statements]
elif [boolean expression]:
    [statements]
elif [boolean expression]:
    [statements]
else:
    [statements]
```

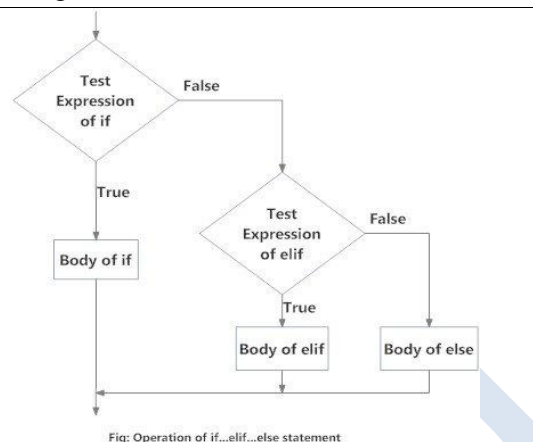


Fig: Operation of if...elif...else statement

### Example 1

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

### Example 2

```
marks = int(input("Enter the marks? "))
if marks > 85 and marks <= 100:
    print("Congrats ! you scored grade A ...")
elif marks > 60 and marks <= 85:
    print("You scored grade B + ...")
elif marks > 40 and marks <= 60:
    print("You scored grade B ...")
elif (marks > 30 and marks <= 40):
    print("You scored grade C ...")
else:
    print("Sorry you are fail ?")
```

### Python Nested if statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

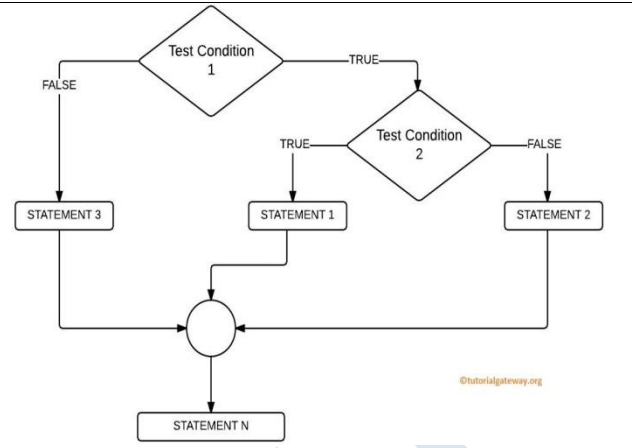
Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

Syntax:

```
if ( test condition 1):
    if ( test condition 2):
        Test condition 2 True statements
    else:
        Test condition 2 False statements
```

Flowchart:

else:  
Test condition 1 False statements



**Example:**input a number check if the number is positive or negative or zero and display an appropriate message .

```

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

#### Output 1

Enter a number: 5

Positive number

#### Output 2

Enter a number: -1

Negative number

#### Output 3

Enter a number: 0

Zero

### Comparing Strings :

To compare two strings, we mean that we want to identify whether the two strings are equivalent to each other or not, or perhaps which string should be greater or smaller than the other.

This is done using the following operators:

- **==**: This checks whether two strings are equal
- **!=**: This checks if two strings are not equal
- **<**: This checks if the string on its left is smaller than that on its right
- **<=**: This checks if the string on its left is smaller than or equal to that on its right
- **>**: This checks if the string on its left is greater than that on its right
- **>=**: This checks if the string on its left is greater than or equal to that on its right

### How to execute the comparison

```
>>> "january" == "jane" False
```

String equality is compared using == (double equal sign). This comparison process is carried out as follow:

- First two characters (j and j) from the both the strings are compared using the ASCII values.
- Since both ASCII values are same then next characters (a and a) are compared. Here they are also equal, and hence next characters (n and n) from both strings are compared.
- This comparison also returns True, and comparison is continued with next characters (u and e). Since the ASCII value of the 'u' is greater than the ASCII value of 'e' this time it returns False. Finally, the comparison operation returns False

**Example1:**

```
print("Geek" == "Geek")  
print("Geek" < "geek")  
print("Geek" > "geek")  
print("Geek" != "Geek")
```

Output:

```
True  
True  
False  
False
```

**Boolean Variables**

The variables that store Boolean value either True or False called Boolean variables.

If the expression is returning a Boolean value then it is called Boolean expression.

The Boolean data type can be one of two values, either **True** or **False**. Named for the mathematician George Boole, the word Boolean always begins with a capitalized B. The values True and False will also always be with a capital T and F respectively.

Boolean logic is implemented with Comparison operators and logical operators

**BOOLEAN WITH COMPARISION OPERATORS:** Comparison operators are used to compare values and evaluate down to a single Boolean value of either True or False.



Conditional Operator	Meaning of Condition
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

### EXAMPLE FOR BOOLEAN WITH COMPARISON OPERATORS:

```
x = 5
y = 8
print("x == y:", x == y)
print("x != y:", x != y)
print("x < y:", x < y)
print("x > y:", x > y)
print("x <= y:", x <= y)
print("x >= y:", x >= y)
```

#### Output:

```
x == y: False
x != y: True
x < y: True
x > y: False
x <= y: True
x >= y: False
```

### BOOLEAN WITH LOGICAL OPERATORS:

The logical operators and, or and not are also referred to as boolean operators.

There are three types of Boolean Logical operators:

The AND operator (&& or “and”)

The OR operator (|| or “or”)

The NOT operator (not)

#### AND Boolean Operator in Python

The **AND Boolean operator** is similar to the bitwise **AND operator** where the operator analyzes the expressions written on both sides and returns the output.

- True and True = True
- True and False = False
- False and True = False
- False and False = False

**Program:**

```
a = 30
b = 45
if(a > 30 and b == 45):
    print("True")
else:
    print("False")
```

**output:** False

**Or Boolean Operator in Python**

The **OR operator** is similar to the **OR bitwise operator**. In the bitwise OR, we were focussing on either of the bit being 1. Here, we take into account if either of the expression is true or not. If at least one expression is true, consequently, the result is true.

- True or True = True
- True or False = True
- False or True = True
- False or False = False

**Program:**

```
a = 25
b = 30
if(a > 30 or b < 45):
    print("True")
else:
    print("False")
```

**output:** True

**Not Boolean Operator in Python**

The **NOT operator** reverses the result of the boolean expression that follows the operator. It is important to note that the NOT operator will only reverse the final result of the expression that **immediately follows**. Moreover, the NOT operator is denoted by the keyword “**not**”.

- not(True) = False
- not(False) = True

```
a = 2
b = 2
if(not(a == b)):
    print("If Executed")
else:
    print("Else Executed")
```

**ouput:** Else Executed

**NOTE:** You can directly use the word “**and**” instead of “&&” to denote the “**and**” boolean operator

**bool() function:**

The bool() function allows you to evaluate any value, and give you True or False in return,

Example:

```
print(bool("sai"))- True
```

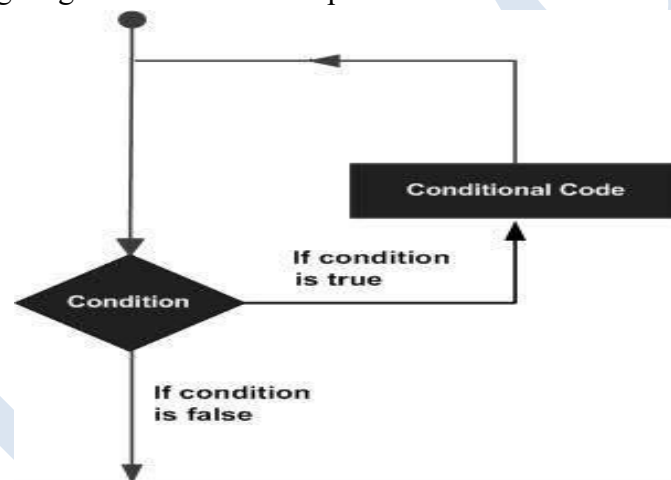
```
print(bool(9))- True
```

## Repetition Structures

### Python loops:

- In python ,loops statements are used to execute the block of code repeatedly for a specified number of times or until it meets a specified condition.
- Loops are used in programming to repeat a specific block of code.
- A loop is a used for iterating over a set of statements repeatedly.
- A loop statement allows us to execute a statement or group of statements multiple times.

The following diagram illustrates a loop statement –



Python programming language provides following types of loops :

1.while loop

2.for loop

### While loop:

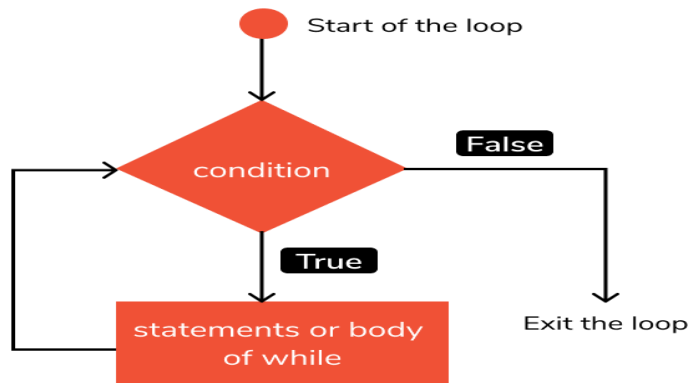
In python, while loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.

We use while loop when we don't know the number of times to iterate.

## Syntax of while Loop in Python

while expression:

Body of while



- In Python, the body of the while loop is determined through indentation.
- The body starts with indentation and the first unindented line marks the end.
- Python interprets any non-zero value as True. None and 0 are interpreted as False.

**Example1:**

# Python program to illustrate

# while loop

count = 0

while (count &lt; 3):

count = count + 1

print("Hello")

output:

Hello

Hello

Hello

**Example2:**

# Program to add 10 natural

n = 10

# initialize sum and counter

sum = 0

i = 1

while i &lt;= n:

sum = sum + i

i = i+1   # update counter

# print the sum

print("The sum is", sum)

output :

Enter n: 10

The sum is 55

Example	WHILE
<pre> i=1 while i &lt; 4:     print(i)     i+=1 print('END') 1 2 3 END </pre>	<pre> i=1 while i &lt; 4:     print(i)     i+=1     print('END') 1 END 2 END 3 END </pre>

**Infinite while loop:**

If the condition is given in the while loop never becomes false, then the while loop will never terminate, and it turns into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition, whereas zero indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

**Example1:**

**This will print the word 'hello' indefinitely because the condition will always be true.**

```
while True:
```

```
    print("hello")
```

**Example 2:**

```
num = 1
```

```
while num<5:
```

```
    print(num)
```

This will print '1' indefinitely because inside loop we are not updating the value of num, so the value of num will always remain 1 and the condition  $\text{num} < 5$  will always return true.

**Using else Statement with While Loop**

Python supports to have an else statement associated with a loop statement.

If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

Example:

```
count = 0
```

```
while count < 5:
```

```
    print count, " is less than 5"
```

```
    count = count + 1
```

```
else:
```

```
    print count, " is not less than 5"
```

**result:**

```
0 is less than 5
```

```
1 is less than 5
```

```
2 is less than 5
```

```
3 is less than 5
```

```
4 is less than 5
```

```
5 is not less than 5
```

**Single statement while block:** Just like the if block, if the while block consists of a single statement the we can declare the entire loop in a single line as shown below:

```
# Python program to illustrate
```

```
# Single statement while block
```

```
count = 0
```

```
while (count == 0): print("Hello Geek")
```

Note: It is suggested not to use this type of loops as it is a never ending infinite loop where the condition is always true and you have to forcefully terminate the compiler.

### For loop:

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

We use for loop when we know the number of times to iterate.

Syntax of for Loop

**for val in sequence:**

**Body of for**

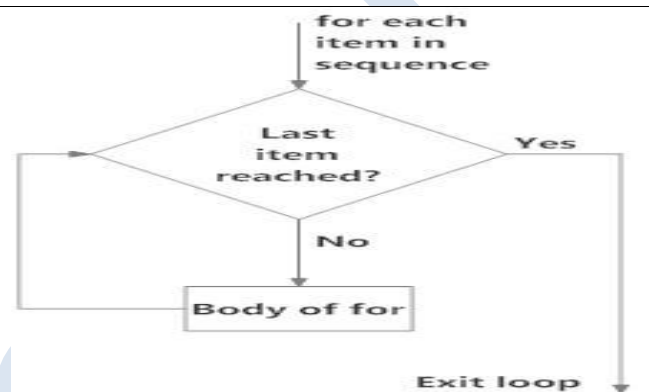


Fig: operation of for loop

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

### Example:

**# Program to find the sum of all numbers stored in a list**

**# List of numbers**

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
```

```
sum = 0
```

**# iterate over the list**

```
for val in numbers:
```

```
    sum = sum+val
```

```
print("The sum is", sum)
```

### output:

The sum is 48

### Function range()

In the above example, we have iterated over a list using for loop. However we can also use a range() function in for loop to iterate over numbers defined by range().

**range(n):** generates a set of whole numbers starting from 0 to (n-1).

For example: range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

**range(start, stop):** generates a set of whole numbers starting from start to stop-1.

For example: range(5, 9) is equivalent to [5, 6, 7, 8]

**range(start, stop, step\_size):** The default step\_size is 1 which is why when we didn't specify the step\_size, the numbers generated are having difference of 1. However by specifying step\_size we can generate numbers having the difference of step\_size.

For example: range(1, 10, 2) is equivalent to [1, 3, 5, 7, 9]

ex: we are using range() function to calculate and display the sum of first 5 natural numbers.

```
# Program to print the sum of first 5 natural numbers
```

```
sum = 0
```

```
# iterating over natural numbers using range()
```

```
for val in range(1, 6):
```

```
    sum = sum + val
```

```
# displaying sum of first 5 natural numbers
```

```
print(sum)
```

**Output: 15**

**for loop with else**

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

The break keyword can be used to stop a for loop. In such cases, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Example:

```
digits = [0, 1, 5]
```

```
for i in digits:
```

```
    print(i)
```

```
else:
```

```
    print("No items left.")
```

output :

0

1

5

No items left.

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.

### **Calculating a Running Total**

We can calculate the sum of input numbers while entering from the keyboard as demonstrated in the following example.

```
n=int(input('Enter n:'))
sum=0
for i in range(n):
    data=float(input('Enter value'))
    sum=sum+data
#display sum
print('Sum is:',sum)
```

**Output:**

```
Enter n:4 Enter
value12 Enter
value13 Enter
value21 Enter
value22 Sum
is: 68.0
```

**Input Validation Loops**

Loops can be used to validate user input. For instance, a program may require the user to enter a positive integer. Many of us have seen a “yes/no” prompt at some point, although probably in the form of a dialog box with buttons rather than text.

When we accept user input we need to check that it is valid. This checks to see that it is the sort of data we were expecting.

**Use a flag variable.** This will initially be set to False. If we establish that we have the correct input then we set the flag to True. We can now use the flag to determine what we do next (for instance, we might repeat some code, or use the flag in an if statement).

Syntax

**Using a flag**

```
flagName = False
while not flagName:
    if [Do check here]:
        flagName = True
    else:
        print('error message')
```

**Example: A length check using a flag**



```

isLongEnough = False
while not isLongEnough:
    password = input('Enter password at least 5 characters: ')
    if len(password) >= 5:
        isLongEnough = True
    else:
        print('Password entered is too short')

```

```
print('Your password entered is: ' + password)
```

### Output

```

Enter password at least 5 characters: asdf
Password entered is too short
Enter password at least 5 characters: 1234
Password entered is too short
Enter password at least 5 characters: ad4fgj
Your password entered is: ad4fgj

```

**Nested Loops:** Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

**Syntax for a nested for loop statement**

**Syntax for a nested while loop statement**

)

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

<pre> for iterator_var in sequence:     for iterator_var in sequence:         statements(s)         statements(s) </pre>	<pre> while expression:     while expression:         statement(s)         statement(s) </pre>
<p><b>Nested For loop in Python</b></p> <p>When a for loop is present inside another for loop then it is called a nested for loop. Lets take an example of nested for loop.</p> <pre> for num1 in range(3):     for num2 in range(10, 14):         print(num1, ", ", num2) </pre> <p>Output:</p> <pre> 0 , 10 0 , 11 0 , 12 0 , 13 1 , 10 </pre>	<p><b>Nested while loop in Python</b></p> <p>When a while loop is present inside another while loop then it is called nested while loop. Lets take an example to understand this concept.</p> <pre> i = 1 j = 5 while i &lt; 4:     while j &lt; 8:         print(i, ", ", j)         j = j + 1         i = i + 1 </pre> <p>Output:</p>

1 , 11	1 , 5
1 , 12	2 , 6
1 , 13	3 , 7
2 , 10	
2 , 11	
2 , 12	
2 , 13	

-----\*\*\*\*\*-----