

## Your first Python Program

Let's create a very simple program called Hello World. A "Hello, World!" is a simple program that outputs Hello, World! on the screen.

```
print("Hello, world!")
```

# Python Keywords and Identifiers

## Python Keywords

Keywords are the reserved words in Python.

We cannot use a keyword as a **variable** name, **function** name or any other identifier.

They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive.

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

# Python Identifiers

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

## Rules for writing identifiers

Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore `_`. Names like `myClass`, `var_1` and `print_this_to_screen`, all are valid example.

An identifier cannot start with a digit. `1variable` is invalid, but `variable1` is a valid name.

Keywords cannot be used as identifiers.

We cannot use special symbols like `!`, `@`, `#`, `$`, `%` etc. in our identifier.

# Python Statement, Indentation and Comments

## Python Statement

Instructions that a Python interpreter can execute are called statements. For example, `a = 1` is an assignment statement. `if` statement, `for` statement, `while` statement, etc.

## Multi-line statement

In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (`\`). For example:

```
a = 1 + 2 + 3 + \  
    4 + 5 + 6 + \  
    7 + 8 + 9
```

This is an explicit line continuation. In Python, line continuation is implied inside parentheses ( `)`, brackets [ `]`, and braces { `}`. For instance, we can implement the above multi-line statement as:

```
a = (1 + 2 + 3 +  
    4 + 5 + 6 +  
    7 + 8 + 9)
```

Here, the surrounding parentheses ( `)` do the line continuation implicitly. Same is the case with [ `]` and { `}`. For example:

```
colors = ['red',  
          'blue',  
          'green']
```

We can also put multiple statements in a single line using semicolons, as follows:

```
a = 1; b = 2; c = 3
```

## Python Indentation

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. Here is an example.

```
for i in range(1,11):  
    print(i)  
    if i == 5:  
        break
```

The enforcement of indentation in Python makes the code look neat and clean. This results in Python programs that look similar and consistent.

Indentation can be ignored in line continuation, but it's always a good idea to indent. It makes the code more readable. For example:

```
if True:  
    print('Hello')  
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

both are valid and do the same thing, but the former style is clearer.

**Incorrect indentation will result in IndentationError.**



## Python Comments

Comments are very important while writing a program. They describe what is going on inside a program, so that a person looking at the source code does not have a hard time figuring it out.

In Python, we use the hash (#) symbol to start writing a comment.

It extends up to the newline character. Comments are for programmers to better understand a program. Python Interpreter ignores comments.

```
#This is a comment  
#print out Hello  
print('Hello')
```

## Multi-line comments

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. Unless they are not **docstrings**, they do not generate any extra code.

```
'''This is also a  
perfect example of  
multi-line comments'''
```

## 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__)
```

# Python Variables, Constants and Literals

## Python Variables

A variable is a named location used to store data in the memory. It is helpful to think of variables as a container that holds data that can be changed later in the program. For example,

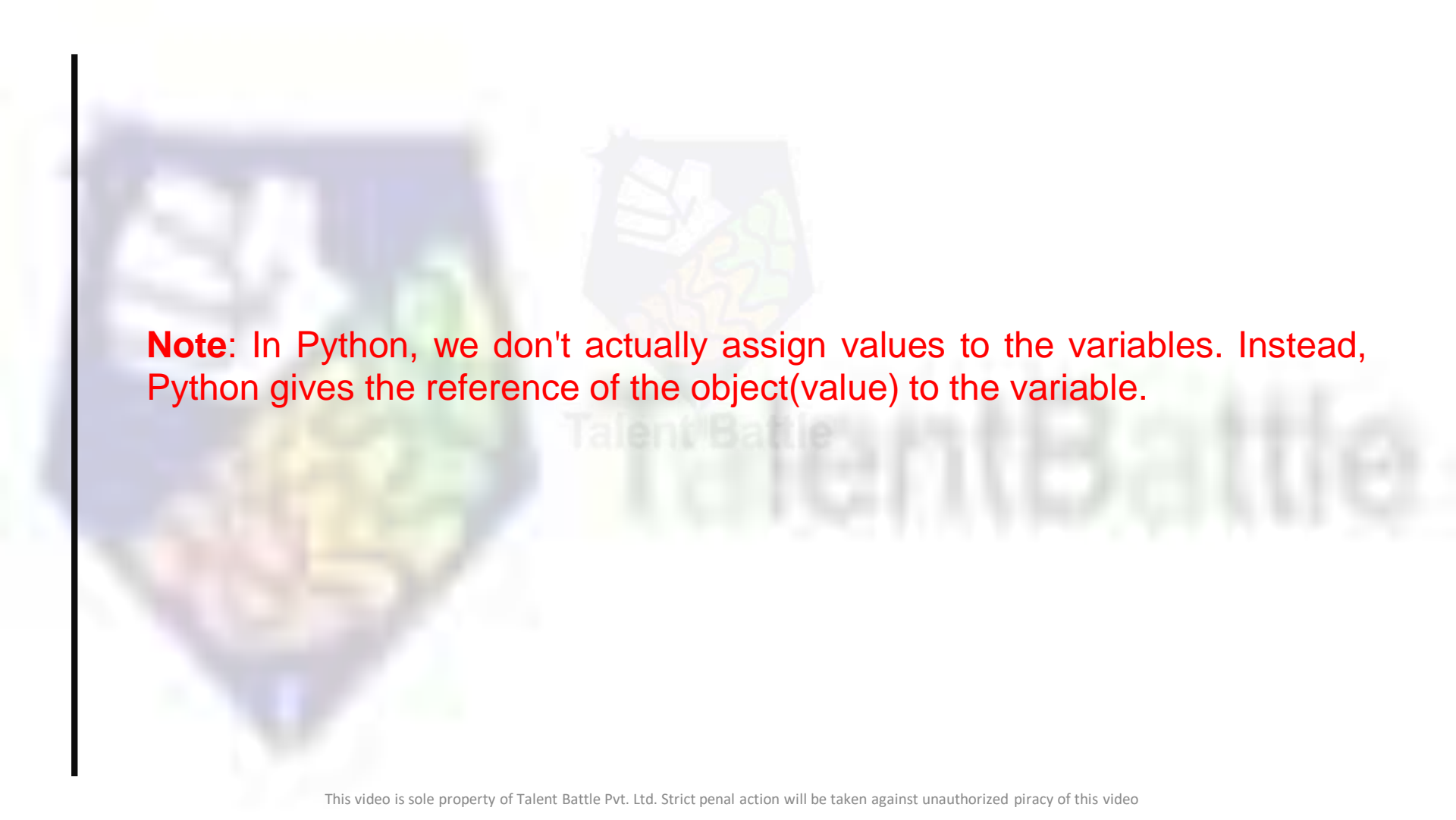
```
number = 10
```

Here, we have created a variable named number. We have assigned the value 10 to the variable.

```
number = 10
```

```
number = 1.1
```

Initially, the value of number was 10. Later, it was changed to 1.1.



**Note:** In Python, we don't actually assign values to the variables. Instead, Python gives the reference of the object(value) to the variable.

## Assigning values to Variables in Python

As you can see from the above example, you can use the assignment operator = to assign a value to a variable.

Example 1: Declaring and assigning value to a variable

```
website = "talentbattle.in"  
print(website)
```

**Note:** Python is a type-inferred language, so you don't have to explicitly define the variable type. It automatically knows that talentbattle.in is a string and declares the website variable as a string.

## **Example 2: Changing the value of a variable**

```
website = "talentbattle.in"  
print(website)
```

```
# assigning a new value to website  
website = "google.com"  
  
print(website)
```



### Example 3: Assigning multiple values to multiple variables

```
a, b, c = 5, 3.2, "Hello"
```

```
print (a)
```

```
print (b)
```

```
print (c)
```

If we want to assign the same value to multiple variables at once, we can do this as:

```
x = y = z = "same"
```

```
print (x)
```

```
print (y)
```

```
print (z)
```

The second program assigns the same string to all the three variables x, y and z.

## Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

### Rules and Naming Convention for Variables and constants

- 1.Constant and variable names should have a combination of letters in lowercase (a to z) or uppercase (**A to Z**) or digits (**0 to 9**) or an underscore (\_).
- 2.If you want to create a variable name having two words, use underscore to separate.
- 3.Use capital letters possible to declare a constant.
- 4.Never use special symbols like !, @, #, \$, %, etc.
- 5.Don't start a variable name with a digit.

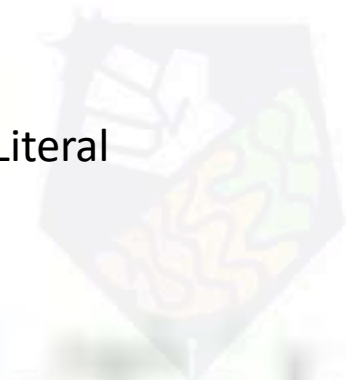
## Literals

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

### Numeric Literals

Numeric Literals are immutable (unchangeable).

Numeric literals can belong to 3 different numerical types: Integer, Float, and Complex.



a = 0b1010 #Binary Literals  
b = 100 #Decimal Literal  
c = 0o310 #Octal Literal  
d = 0x12c #Hexadecimal Literal

#Float Literal

float\_1 = 10.5

float\_2 = 1.5e2

print(a, b, c, d)

print(float\_1, float\_2)

In the above program,

We assigned integer literals into different variables.

Here, a is binary literal, b is a decimal literal, c is an octal literal and d is a hexadecimal literal.

When we print the variables, all the literals are converted into decimal values.  
10.5 and 1.5e2 are floating-point literals.

1.5e2 is expressed with exponential and is equivalent to  $1.5 * 10^2$ .

## String literals

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

```
strings = "This is Python Programming"  
char = "C"  
multiline_str = """This is a multiline string with more than one line code."""  
  
print(strings)  
print(char)  
print(multiline_str)
```

## Boolean literals

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

Example : How to use boolean literals in Python?

```
x = (1 == True)
```

```
y = (1 == False)
```

```
a = True + 4
```

```
b = False + 10
```

```
print("x is", x)
```

```
print("y is", y)
```

```
print("a:", a)
```

```
print("b:", b)
```

In the above program, we use boolean literal True and False.

In Python, True represents the value as 1 and False as 0. The value of x is True because 1 is equal to True. And, the value of y is False because 1 is not equal to False.

Similarly, we can use the True and False in numeric expressions as the value.

The value of a is 5 because we add True which has a value of 1 with 4. Similarly, b is 10 because we add the False having value of 0 with 10.



## Special literals

Python contains one special literal i.e. **None**. We use it to specify that the field has not been created.

Example : How to use special literals in Python?

```
drink = "Pepsi"
```

```
food = None
```

```
def menu(x):  
    if x == drink:  
        print(drink)  
    else:  
        print(food)
```

```
menu(drink)
```

```
menu(food)
```

## Literal Collections

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

Example : How to use literals collections in Python?

```
fruits = ["apple", "mango", "orange"] #list
numbers = (1, 2, 3) #tuple
alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary
vowels = {'a', 'e', 'i', 'o', 'u'} #set
```

```
print(fruits)
print(numbers)
print(alphabets)
print(vowels)
```

# Python Data Types

## Data types in Python

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

## Python Numbers

Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as int, float and complex classes in Python.

We can use the **type()** function to know which class a variable or a value belongs to. Similarly, the **isinstance()** function is used to check if an object belongs to a particular class.

```
a = 10  
print(a, "is of type", type(a))
```

```
a = 2.8  
print(a, "is of type", type(a))
```

```
a = 1+2j  
print(a, "is complex number?", isinstance(1+2j,complex))
```

Integers can be of any length, it is only limited by the memory available.

A floating-point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is an integer, 1.0 is a floating-point number.

Complex numbers are written in the form,  $x + yj$ , where  $x$  is the real part and  $y$  is the imaginary part. Here are some examples.

```
a = 1234567890123456789  
print(a)
```

```
b = 0.1234567890123456789  
print(b)
```

```
c = 1+2j  
print(c)
```

## Python List

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

```
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.**

```
a = [5,10,15,20,25,30,35,40]
```

```
# a[2] = 15  
print("a[2] = ", a[2])
```

```
# a[0:3] = [5, 10, 15]  
print("a[0:3] = ", a[0:3])
```

```
# a[5:] = [30, 35, 40]  
print("a[5:] = ", a[5:])
```

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

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

## Python Tuple

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.

It is defined within parentheses () where items are separated by commas.

```
t = (5,'programmiing', 4+5j)
```

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





```
t = (50,'programming', 5+4j)
```

```
# t[1] = 'programming'  
print("t[1] = ", t[1])
```

```
# t[0:3] = (50, 'programming', (5+4j))  
print("t[0:3] = ", t[0:3])
```

```
# Generates error  
# Tuples are immutable  
t[0] = 10
```

## Python Strings

String is sequence of Unicode characters.

We can use single quotes or double quotes to represent strings.

Multi-line strings can be denoted using triple quotes, ''' or """".

```
s = "This is a string"
```

```
print(s)
```

```
s = """A multiline
```

```
string"""
```

```
print(s)
```

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

```
s = 'Hello world!'
```

```
# s[4] = 'o'  
print("s[4] = ", s[4])
```

```
# s[6:11] = 'world'  
print("s[6:11] = ", s[6:11])
```

```
# Generates error
```

```
# Strings are immutable in Python
```

```
s[5] ='d'
```

## 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.

```
a = {5,2,3,1,4}
```

```
# printing set variable
```

```
print("a = ", a)
```

```
# data type of variable a
```

```
print(type(a))
```

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

```
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]
```

## 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.

```
d = {1:'value','key':2}  
type(d)  
<class 'dict'>
```

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



```
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])
```

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered\*, changeable and does not allow duplicates.

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict)
```



Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"])
```

Example

Duplicate values will overwrite existing values:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```

## Python Input, Output and Import

Python provides numerous built-in functions that are readily available to us at the Python prompt.

Some of the functions like **input()** and **print()** are widely used for standard input and output operations respectively.

## Python Output Using print() function

```
a = 5
```

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

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.

The sep separator is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

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

## 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('Hello {name}, {greeting}'.format(greeting = 'Good Night', name = 'TalentBattle'))
```

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'))
```

## 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
```

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
```

```
int('2+3')
```

Traceback (most recent call last):

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

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

ValueError: invalid literal for int() with base 10: '2+3'

```
eval('2+3')
```

```
5
```

## Python Import

When our program grows bigger, it is a good idea to break it into different modules.

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

# Python Operators

## What are operators in python?

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:

2+3

5

Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation.

## Arithmetic operators

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

# Output: `x + y = 19`

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

# Output: `x - y = 11`

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

# Output: `x * y = 60`

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

# Output: `x / y = 3.75`

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

# Output: `x // y = 3`

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

# Output: `x ** y = 50625`

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

## Comparison operators

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

# Output: x > y is False

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

# Output: x < y is True

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

# Output: x == y is False

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

# Output: x != y is True

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

# Output: x >= y is False

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

# Output: x <= y is True

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



Talent Battle

Talent Battle

## 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

### 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)



## Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

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	$\sim x = -11$ (1111 0101)
^	Bitwise XOR	$x \wedge y = 14$ (0000 1110)
>>	Bitwise right shift	$x >> 2 = 2$ (0000 0010)
<<	Bitwise left shift	$x << 2 = 40$ (0010 1000)

# Assignment operators

Operator	Example	Equivalent 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

## 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

**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

## Example : Identity operators in Python

```
x1 = 5
```

```
y1 = 5
```

```
x2 = 'Hello'
```

```
y2 = 'Hello'
```

```
x3 = [1,2,3]
```

```
y3 = [1,2,3]
```

```
# Output: False
```

```
print(x1 is not y1)
```

```
# Output: True
```

```
print(x2 is y2)
```

```
# Output: False
```

```
print(x3 is y3)
```

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

## 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	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

## Example : 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)
```

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.

## Python if...else Statement

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

The **if...elif...else** statement is used in Python for decision making.

Python if Statement Syntax

**if test expression:**  
    **statement(s)**

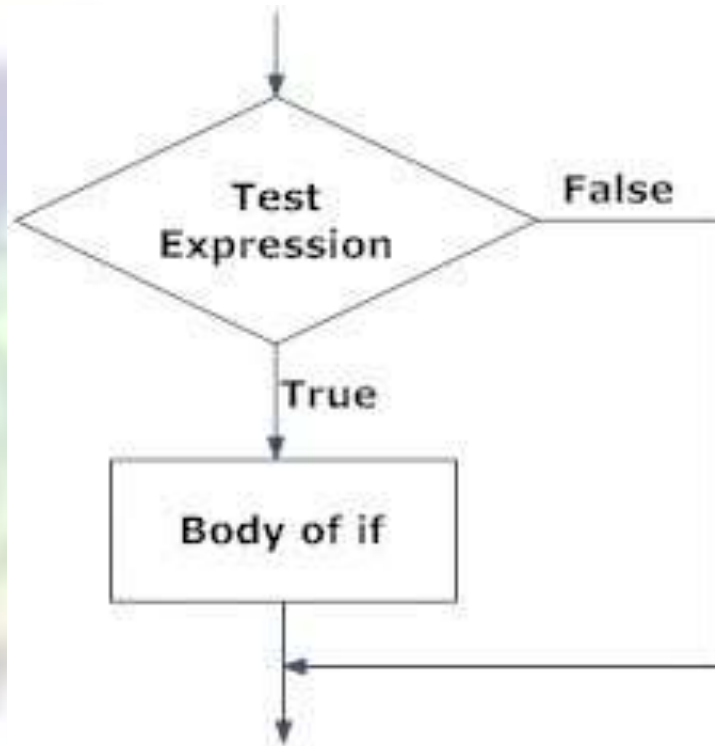
Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.

If the test expression is False, the statement(s) is not executed.

In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end.

Python interprets non-zero values as True. None and 0 are interpreted as False.





**Fig: Operation of if statement**

# If the number is positive, we print an appropriate message

```
num = 3
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
print("This is always printed.")
```

```
num = -1
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
print("This is also always printed.")
```

## Python if...else Statement

Syntax of if...else

**if test expression:**

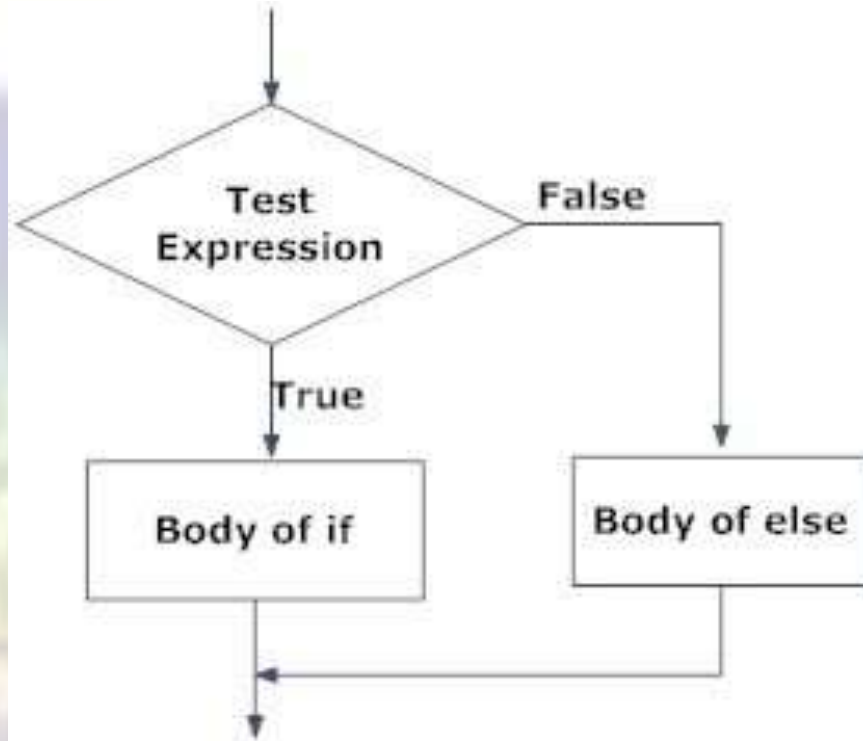
**Body of if**

**else:**

**Body of else**

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

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



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

# Program checks if the number is positive or negative  
# And displays an appropriate message

num = 3

# Try these two variations as well.

# num = -5

# num = 0

if num >= 0:

    print("Positive or Zero")

else:

    print("Negative number")

## Python if...elif...else Statement

Syntax of if...elif...else

**if test expression:**

**Body of if**

**elif test expression:**

**Body of elif**

**else:**

**Body of else**

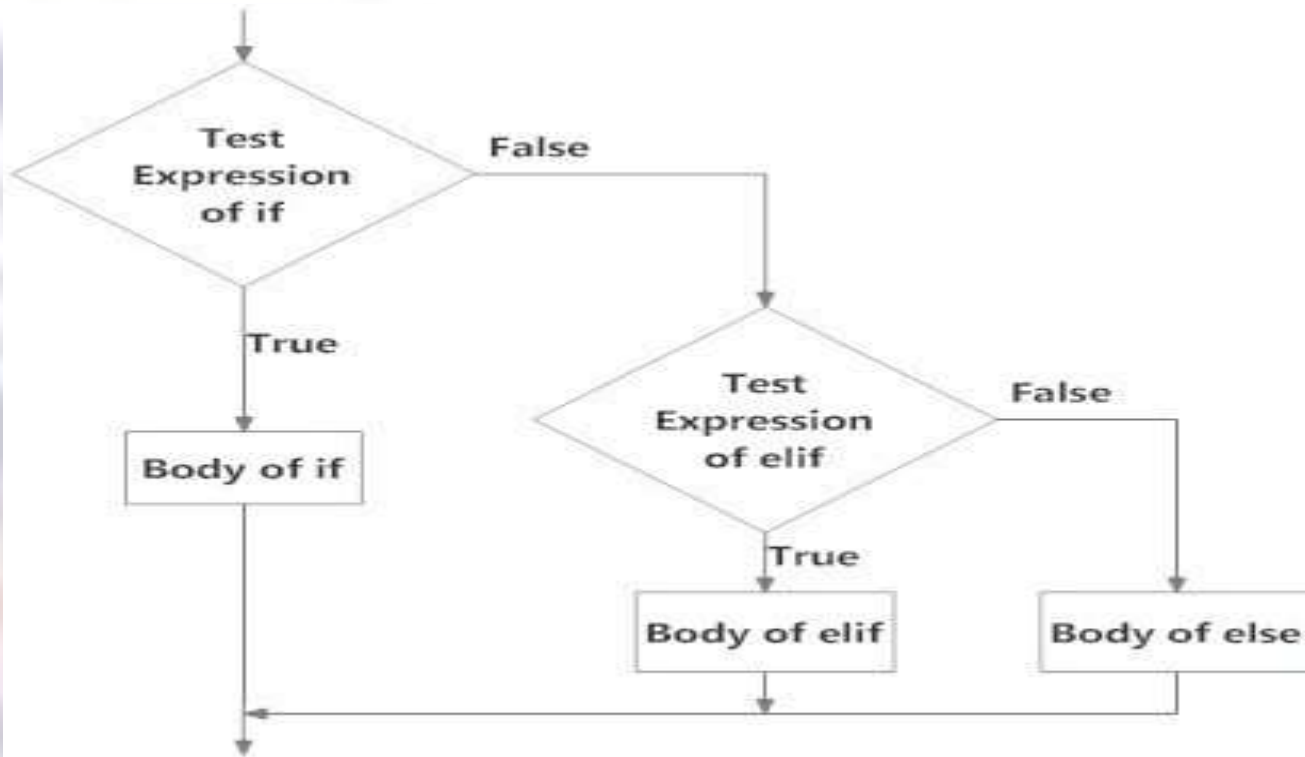
The elif is short for else if. It allows us to check for multiple expressions.

If the condition for if is False, it checks the condition of the next elif block and so on.


If all the conditions are False, the body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.



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



num = 3.4

# Try these two variations as well:

# num = 0

# num = -4.5

if num > 0:

    print("Positive number")

elif num == 0:

    print("Zero")

else:

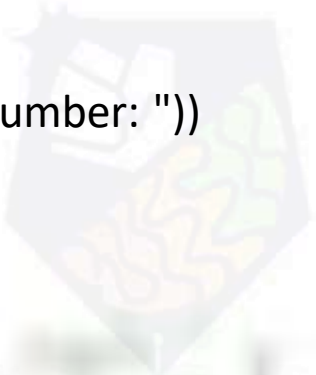
    print("Negative number")



## 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.



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

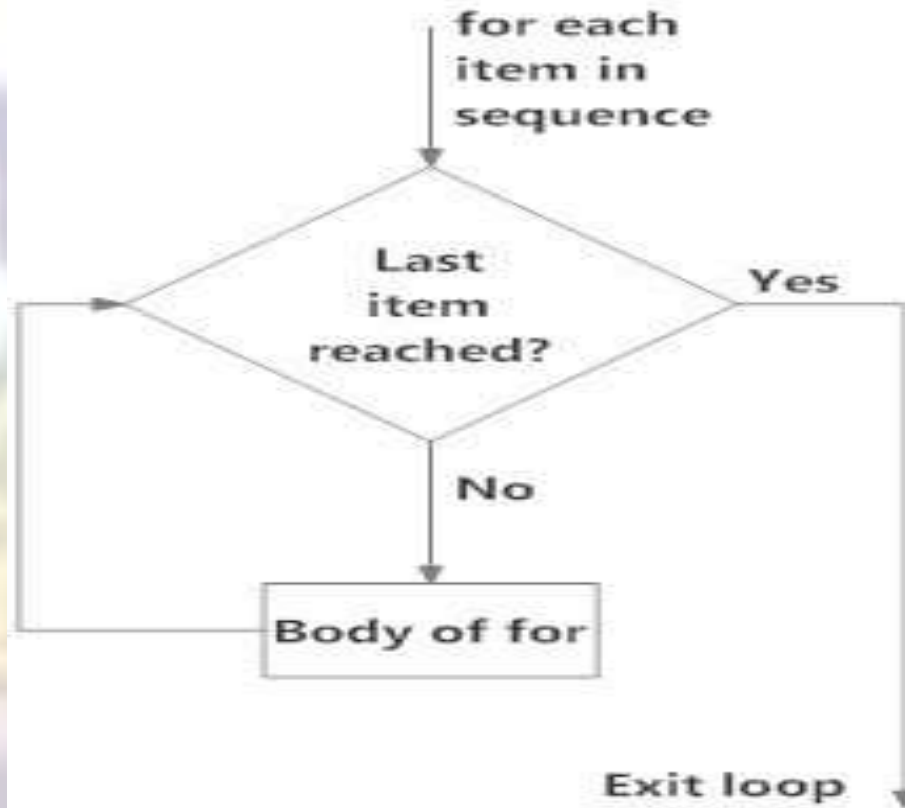
## Python for Loop

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

Syntax of for Loop  
for val in sequence:  
 Body of for

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.



**Fig: operation of for loop**

# 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]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

sum = sum+val

print("The sum is", sum)


## The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start, stop, step\_size). step\_size defaults to 1 if not provided.

However, it is not an iterator since it supports **in**, **len** and **\_\_getitem\_\_** operations.

To force this function to output all the items, we can use the function list().




`print(range(10))`

`print(list(range(10)))`

`print(list(range(2, 8)))`

`print(list(range(2, 20, 3)))`



# Program to iterate through a list using indexing

random = ['abc', 'xyz', 'lmn']

# iterate over the list using index

for i in range(len(random)):

print("I like", random[i])




## **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.



digits = [0, 1, 5]

for i in digits:

    print(i)

else:

    print("No items left.")

```
# program to display student's marks from record  
student_name = 'Talent'
```

```
marks = {'abc': 90, 'xyz': 55, 'lmn': 77}
```

```
for student in marks:
```

```
    if student == student_name:
```

```
        print(marks[student])
```

```
        break
```

```
else:
```

```
    print('No entry with that name found.')
```

## Python while Loop

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

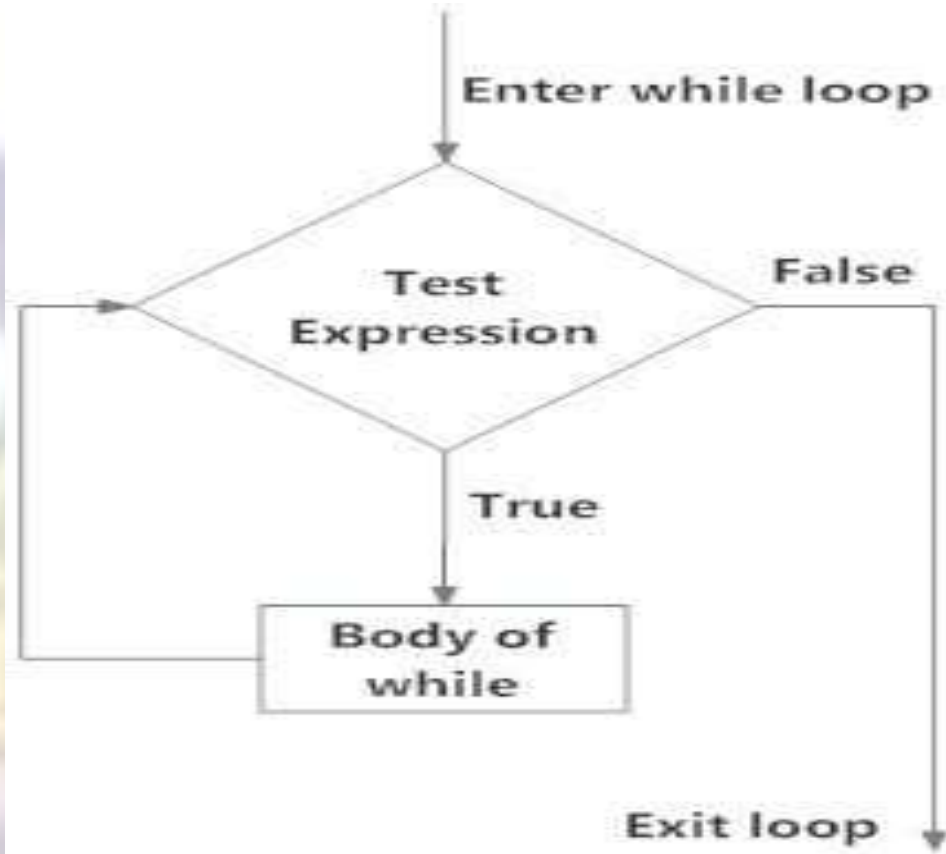
We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax of while Loop in Python

```
while test_expression:
```

```
    Body of while
```

In the while loop, test expression is checked first. The body of the loop is entered only if the test\_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test\_expression evaluates to False.



**Fig: operation of while loop**

```
# Program to add natural  
# numbers up to  
# sum = 1+2+3+...+n
```

```
# To take input from the user,  
# n = int(input("Enter n: "))  
n = 10  
# initialize sum and counter  
sum = 0  
i = 1
```

```
while i <= n:  
    sum = sum + i  
    i = i+1    # update counter
```

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

## While loop with else

Same as with for loops, while loops can also have an optional else block.

The else part is executed if the condition in the while loop evaluates to False.

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

#Illustration

counter = 0

while counter < 3:

    print("Inside loop")

    counter = counter + 1

else:

    print("Inside else")



# Python break and continue

## What is the use of break and continue in Python?

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

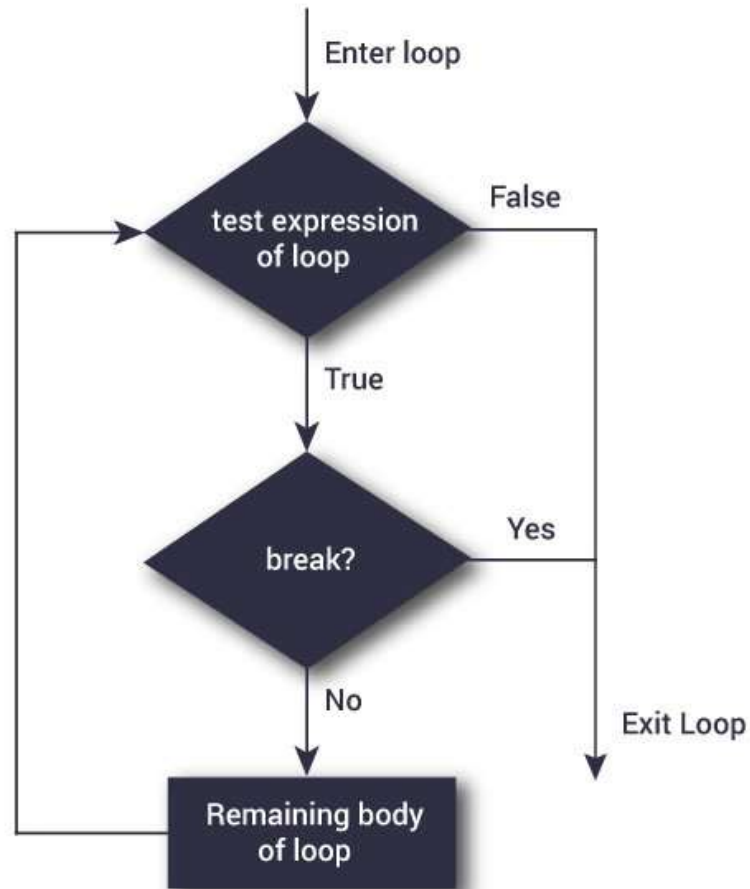
The break and continue statements are used in these cases.

## Python break statement

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

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

Syntax of break  
break



**for var in sequence:**

# codes inside for loop

**if condition:**

**break**

# codes inside for loop

→ # codes outside for loop

---

**while test expression:**

# codes inside while loop

**if condition:**

**break**

# codes inside while loop

→ # codes outside while loop



# Use of break statement inside the loop

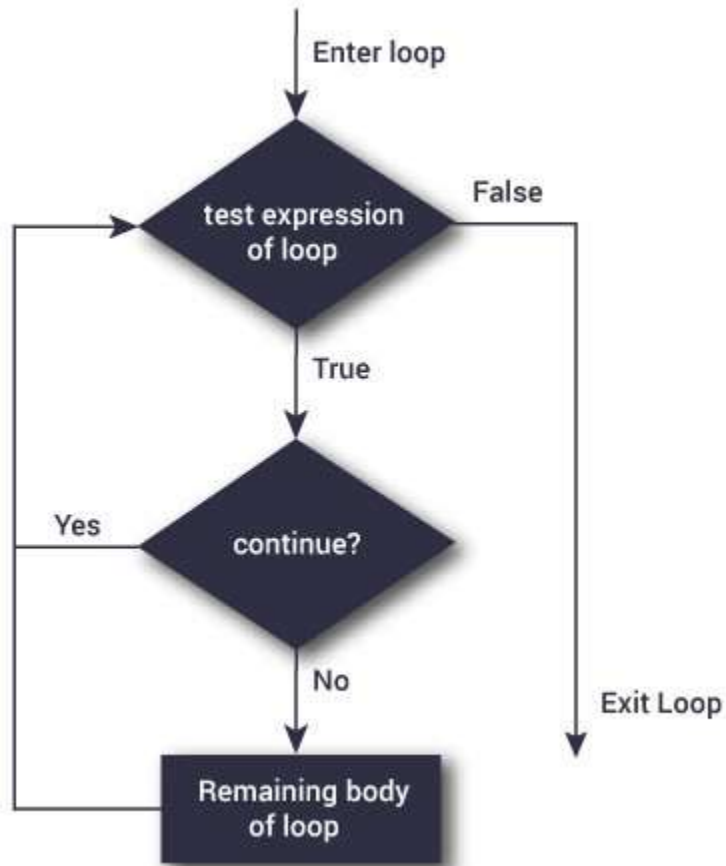
```
for val in "string":  
    if val == "i":  
        break  
    print(val)  
  
print("The end")
```

In this program, we iterate through the "string" sequence. We check if the letter is i, upon which we break from the loop. Hence, we see in our output that all the letters up till i gets printed. After that, the loop terminates.


## Python continue statement

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

Syntax of Continue  
continue



**for var in sequence:**




```
# codes inside for loop  
if condition:  
    continue  
# codes inside for loop
```

```
# codes outside for loop
```

---

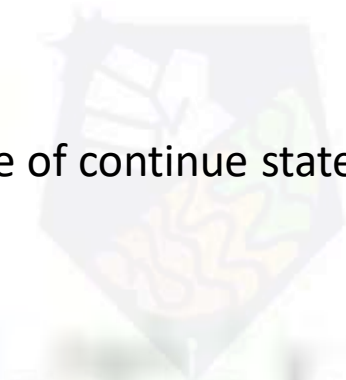
**while test expression:**



```
# codes inside while loop  
if condition:  
    continue  
# codes inside while loop
```

```
# codes outside while loop
```





# Program to show the use of continue statement inside loops

```
for val in "string":  
    if val == "i":  
        continue  
    print(val)  
  
print("The end")
```

We continue with the loop, if the string is i, not executing the rest of the block. Hence, we see in our output that all the letters except i gets printed.

# Python pass statement

## What is pass statement in Python?

In Python programming, the pass statement is a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.

However, nothing happens when the pass is executed. It results in no operation (NOP).

Syntax of pass

Pass

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would give an error. So, we use the pass statement to construct a body that does nothing.

Example: pass Statement

#pass is just a placeholder for functionality to be added later.

```
sequence = {'p', 'a', 's', 's'}
```

```
for val in sequence:
```

```
    pass
```


We can do the same thing in an empty function or class as well.

```
def function(args):
```

```
    pass
```

```
class Example:
```

```
    pass
```



```
a = 10
```

```
b = 20
```

```
if(a<b):
```

```
    pass
```

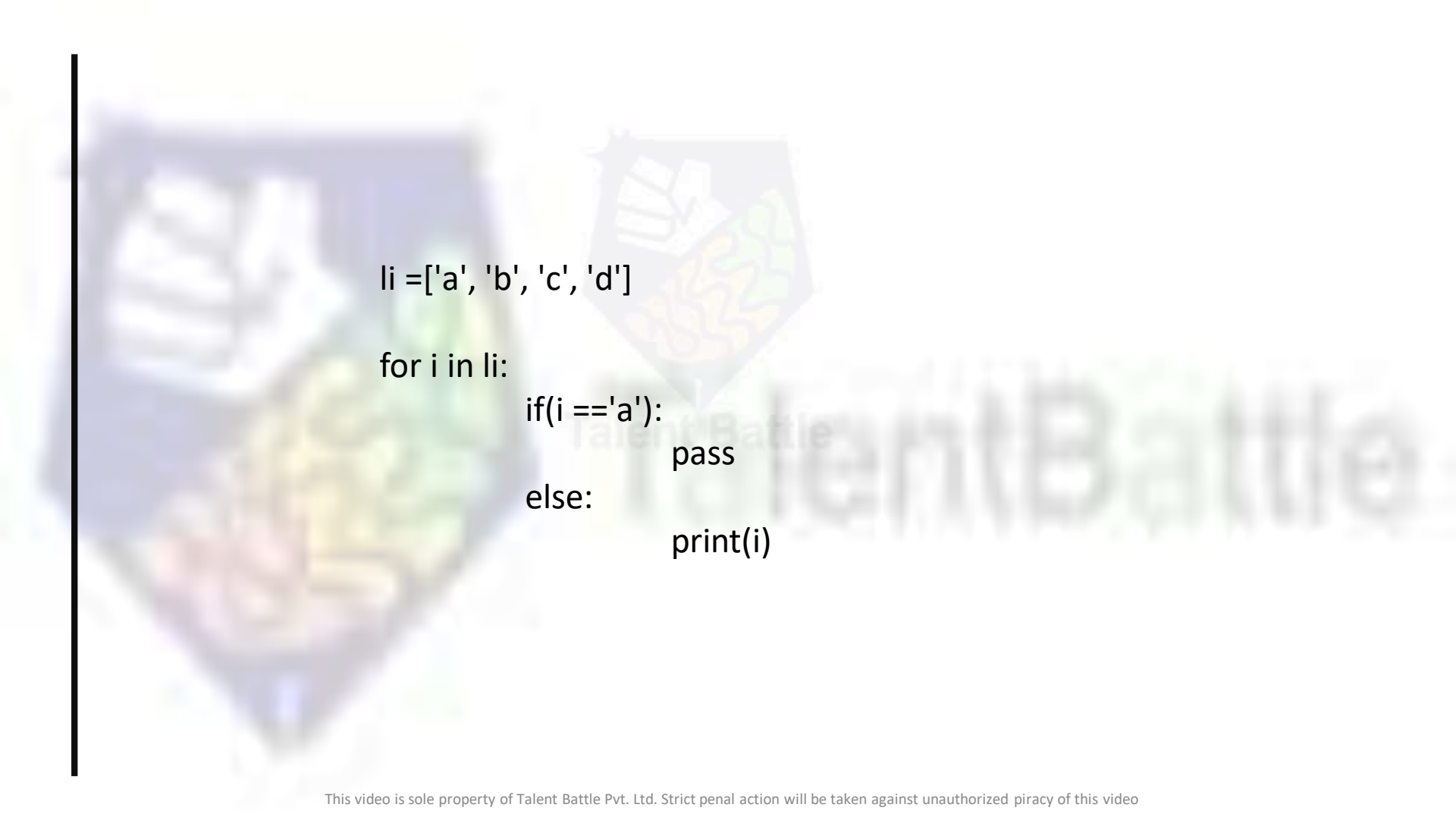
```
else:
```

```
    print("b<a")
```



Talent Battle

TalentBattle



```
li=['a', 'b', 'c', 'd']
```

```
for i in li:
```

```
    if(i=='a'):
```

```
        pass
```

```
    else:
```

```
        print(i)
```



Talent Battle

TalentBattle

# Python Functions

## What is a function in Python?

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks.

As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

## Syntax of Function

```
def function_name(parameters):  
    """docstring"""  
    statement(s)
```

Above shown is a function definition that consists of the following components.

- Keyword def that marks the start of the function header.
- A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
- An optional return statement to return a value from the function.





```
def daily(name):
```

```
    print("Hello, " + name + ". Good morning!")
```

```
daily('Talent Battle')
```

```
def absolute_value(num):
```

```
    if num >= 0:  
        return num
```

```
    else:  
        return -num
```

```
print(absolute_value(2))
```

```
print(absolute_value(-4))
```



Talent Battle

# TalentBattle

```
def functionName():
```

```
    ... ..
```

```
    ... ..
```

```
    ... ..
```

```
    ... ..
```

```
functionName();
```

```
    ... ..
```

```
    ... ..
```

## Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized.

Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.

The lifetime of a variable is the period throughout which the variable exists in the memory.

The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

```
def my_func():  
    x = 10  
    print("Value inside function:",x)  
  
x = 20  
my_func()  
print("Value outside function:",x)
```

Here, we can see that the value of x is 20 initially. Even though the function my\_func() changed the value of x to 10, it did not affect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

## Types of Functions

Basically, we can divide functions into the following two types:

- **Built-in functions** - Functions that are built into Python.
- **User-defined functions** - Functions defined by the users themselves.

## Python User-defined Functions

Functions that we define ourselves to do certain specific task are referred as user-defined functions.


Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions.

All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

## Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmers working on large project can divide the workload by making different functions.





def add\_numbers(x,y):  
 sum = x + y  
 return sum

num1 = 5  
num2 = 6

print("The sum is", add\_numbers(num1, num2))

## Python Function Arguments

In Python, you can define a function that takes variable number of arguments.

```
def greet(name, msg):  
    """This function greets to  
    the person with the provided message"""  
    print("Hello", name + ', ' + msg)  
  
greet("Monica", "Good morning!")
```

## Python Default Arguments

Function arguments can have default values in Python. We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
def greet(name, msg="Good morning!"):

```

```
    print("Hello", name + ', ' + msg)
```

```
greet("Kate")

```

```
greet("Bruce", "How do you do?")
```

In this function, the parameter name does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter msg has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

**SyntaxError: non-default argument follows default argument**

## Python Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

For example, in the above function `greet()`, when we called it as `greet("Bruce", "How do you do?")`, the value "Bruce" gets assigned to the argument name and similarly "How do you do?" to `msg`.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.

# 2 keyword arguments

```
greet(name = "Bruce",msg = "How do you do?")
```

# 2 keyword arguments (out of order)

```
greet(msg = "How do you do?",name = "Bruce")
```

#1 positional, 1 keyword argument

```
greet("Bruce", msg = "How do you do?")
```

As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

Having a positional argument after keyword arguments will result in errors.

## Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition, we use an asterisk (\*) before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):
```

```
    # names is a tuple with arguments
```

```
    for name in names:
```

```
        print("Hello", name)
```

```
greet("Monica", "Luke", "Steve", "John")
```

# Python Recursion

## What is recursion?

Recursion is the process of defining something in terms of itself.

## Python Recursive Function

In Python, we know that a **function** can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

```
def recurse():  
    ...  
    recurse()  
    ...  
  
recurse()
```

recursive call



Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is  $1*2*3*4*5*6 = 720$ .

### **#Example of a recursive function**

```
def factorial(x):
```

```
    if x == 1:
```

```
        return 1
```

```
    else:
```

```
        return (x * factorial(x-1))
```

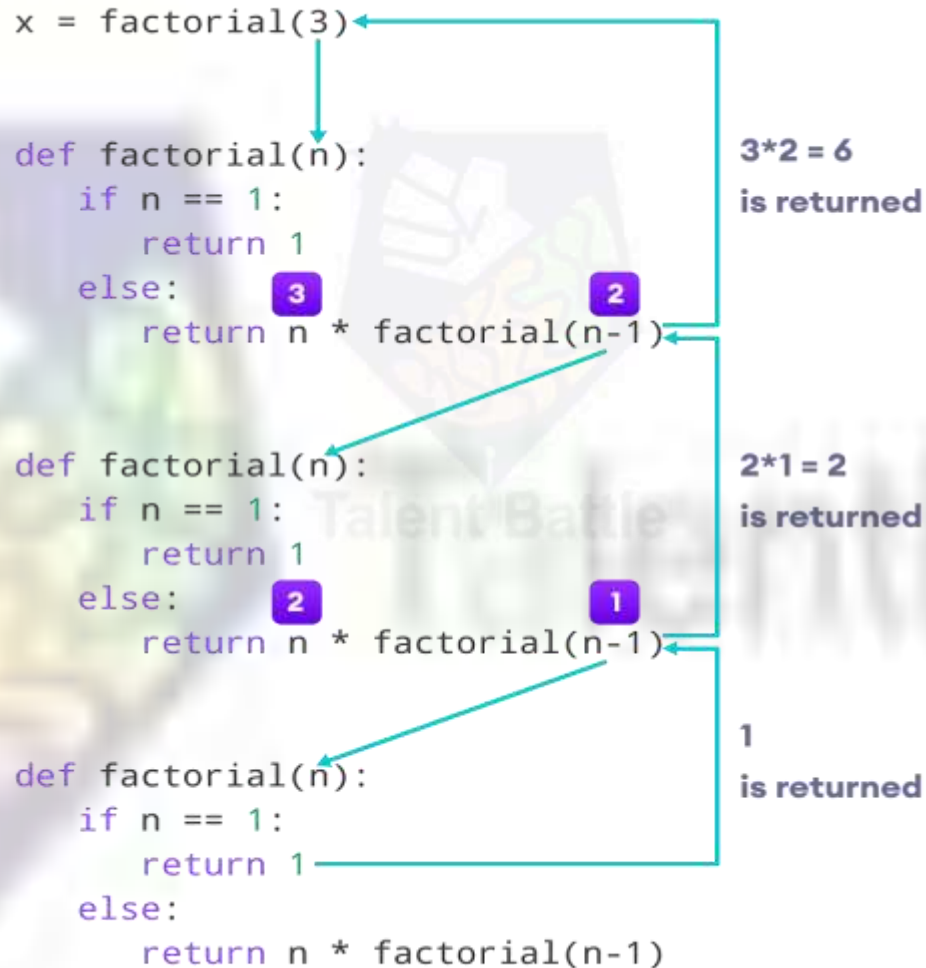
```
num = 3
```

```
print("The factorial of", num, "is", factorial(num))
```

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)      # 1st call with 3
3 * factorial(2)  # 2nd call with 2
3 * 2 * factorial(1) # 3rd call with 1
3 * 2 * 1        # return from 3rd call as number=1
3 * 2            # return from 2nd call
6                # return from 1st call
```



Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

**By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError.**

```
def recursor():  
    recursor()  
recursor()
```

## **Advantages of Recursion**

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

## **Disadvantages of Recursion**

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

# Python Anonymous/Lambda Function

## What are lambda functions in Python?

In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the `def` keyword in Python, anonymous functions are defined using the **lambda** keyword.

Hence, anonymous functions are also called lambda functions.

## **How to use lambda Functions in Python?**

A lambda function in python has the following syntax.

### **Syntax of Lambda Function in python**

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

## Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions
```

```
double = lambda x: x * 2
```

```
print(double(5))
```

### Output

10



In the above program, `lambda x: x * 2` is the lambda function. Here `x` is the argument and `x * 2` is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier `double`. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as:

```
def double(x):  
    return x * 2
```

## Example use with filter()

The filter() function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of filter() function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
```

```
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
```

```
print(new_list)
```

## Example use with map()

The map() function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of map() function to double all the items in a list.

# Program to double each item in a list using map()

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
```

```
new_list = list(map(lambda x: x * 2 , my_list))
```

```
print(new_list)
```

## Python import statement

We can import a module using the import statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example  
# to import standard module math
```

```
import math  
print("The value of pi is", math.pi)
```

When you run the program, the output will be:

The value of pi is 3.141592653589793

## Import with renaming

We can import a module by renaming it as follows:

```
# import module by renaming it
```

```
import math as m  
print("The value of pi is", m.pi)
```

We have renamed the math module as m. This can save us typing time in some cases.

Note that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is the correct implementation.

## Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

```
# import only pi from math module
```

```
from math import pi  
print("The value of pi is", pi)
```

Here, we imported only the pi attribute from the math module.

In such cases, we don't use the dot operator.

## Import all names

We can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math
```

```
from math import *  
print("The value of pi is", pi)
```

Here, we have imported all the definitions from the math module.

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

## The dir() built-in function

We can use the dir() function to find out names that are defined inside a module.

```
#type below in shell  
import math  
dir(math)
```



## Python Package

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.

A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

## Importing module from a package

We can import modules from packages using the dot (.) operator.

```
#check below code  
import math  
math.factorial(5)
```

# Python Strings

## What is String in Python?

A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

## How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
my_string = 'Hello'  
print(my_string)
```

```
my_string = "Hello"  
print(my_string)
```

```
my_string = """Hello"""  
print(my_string)
```

## How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an `IndexError`. The index must be an integer. We can't use floats or other types, this will result into `TypeError`.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator `:`(colon).

#Accessing string characters in Python

```
str = 'talent battle'
```

```
print('str = ', str)
```

#first character

```
print('str[0] = ', str[0])
```

#last character

```
print('str[-1] = ', str[-1])
```

#slicing 2nd to 5th character

```
print('str[1:5] = ', str[1:5])
```

#slicing 6th to 2nd last character

```
print('str[5:-2] = ', str[5:-2])
```

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned.

We can simply reassign different strings to the same name.

```
>>> my_string = 'talent battle'
```

```
>>> my_string[5] = 'a'
```

```
...
```

```
TypeError: 'str' object does not support item assignment
```

```
>>> my_string = 'Python'
```

```
>>> my_string
```

```
'Python'
```

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

## Concatenation of Two or More Strings

Joining of two or more strings into a single one is called concatenation.

The + operator does this in Python. Simply writing two string literals together also concatenates them.

The \* operator can be used to repeat the string for a given number of times.

# Python String Operations

```
str1 = 'Hello'
```

```
str2 = 'World!'
```

```
# using +
```

```
print('str1 + str2 = ', str1 + str2)
```

```
# using *
```

```
print('str1 * 3 =', str1 * 3)
```



## Iterating Through a string

We can iterate through a string using a for loop. Here is an example to count the number of 'l's in a string.

```
# Iterating through a string
count = 0
for letter in 'Hello World':
    if(letter == 'l'):
        count += 1
print(count)
```

## Built-in functions to Work with Python

Various built-in functions that work with sequence work with strings as well.

Some of the commonly used ones are **enumerate()** and **len()**. The `enumerate()` function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, `len()` returns the length (number of characters) of the string.

```
str = 'talent'

# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str) = ', list_enumerate)

#character count
print('len(str) = ', len(str))
```

## **The format() Method for Formatting Strings**

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contain curly braces {} as placeholders or replacement fields which get replaced.

We can use positional arguments or keyword arguments to specify the order.

## # Python string format() method

### # default(implicit) order

```
default_order = "{}, {} and {}".format('Ram','Sham','Meera')  
print('\n--- Default Order ---')  
print(default_order)
```

### # order using positional argument

```
positional_order = "{1}, {0} and {2}".format('Ram','Sham','Meera')  
print('\n--- Positional Order ---')  
print(positional_order)
```

### # order using keyword argument

```
keyword_order = "{m}, {s} and {r}".format(r='Ram',s='Sham',m='Meera')  
print('\n--- Keyword Order ---')  
print(keyword_order)
```

## Common Python String Methods

There are numerous methods available with the string object.

Some of the commonly used methods are `lower()`, `upper()`, `join()`, `split()`, `find()`, `replace()` etc.

```
>>> "TalEntBaTTle".lower()
```

```
>>> "talEntBattLe".upper()
```

```
>>> "This will split all words into a list".split()
```

```
>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a', 'string'])
```

```
>>> 'Happy New Year'.replace('Happy','Brilliant')
```



# Python OOP

## Object Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects.

This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

A parrot is an object, as it has the following properties:

- name, age, color as attributes
- singing, dancing as behavior



## Class

A class is a blueprint for the object.

The example for class of parrot can be :

```
class Parrot:  
    pass
```

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances.

An instance is a specific object created from a particular class.

## Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, obj is an object of class Parrot.

## The `__init__()` Function

To understand the meaning of classes we have to understand the built-in `__init__()` function.

All classes have a function called `__init__()`, which is always executed when the class is being initiated.

Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created.

**Note:** The `__init__()` function is called automatically every time the class is being used to create a new object.

## Example

Create a class named Person, use the `__init__()` function to assign values for name and age:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
p1 = Person("Talent", 20)  
  
print(p1.name)  
print(p1.age)
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("Python", 3)
p1.myfunc()
```

## The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

```
class Person:  
    def __init__(xyz, name, age):  
        xyz.name = name  
        xyz.age = age  
  
    def myfunc(abc):  
        print("Hello my name is " + abc.name)  
        print("My age is =",abc.age)
```

```
p1 = Person("Python", 3)  
p1.myfunc()
```

class Parrot:

```
# class attribute  
species = "bird"
```

```
# instance attribute  
def __init__(self, name, age):  
    self.name = name  
    self.age = age
```

```
# instantiate the Parrot class
```

```
blu = Parrot("Blu", 10)  
woo = Parrot("Woo", 15)
```

```
# access the class attributes
```

```
print("Blu is a {}".format(blu.__class__.species))  
print("Woo is also a {}".format(woo.__class__.species))
```

```
# access the instance attributes
```

```
print("{} is {} years old".format( blu.name, blu.age))  
print("{} is {} years old".format( woo.name, woo.age))
```

## Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.



class Parrot:

```
# instance attributes
```

```
def __init__(self, name, age):
```

```
    self.name = name
```

```
    self.age = age
```

```
# instance method
```

```
def sing(self, song):
```

```
    return "{} sings {}".format(self.name, song)
```

```
def dance(self):
```

```
    return "{} is now dancing".format(self.name)
```

```
# instantiate the object
```

```
blu = Parrot("Blu", 10)
```

```
# call our instance methods
```

```
print(blu.sing("Happy"))
```

```
print(blu.dance())
```

## Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it.

The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

```
# parent class
class Bird:
    def __init__(self):
        print("Bird is ready")
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```



Talent Battle

Talent Battle

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method.

Again, the child class modified the behavior of the parent class. We can see this from the whoisThis() method. Furthermore, we extend the functions of the parent class, by creating a new run() method.

Additionally, we use the super() function inside the \_\_init\_\_() method. This allows us to run the \_\_init\_\_() method of the parent class inside the child class.

## Encapsulation

Using OOP in Python, we can restrict access to methods and variables.

This prevents data from direct modification which is called encapsulation.

In Python, we denote private attributes using underscore as the prefix i.e single `_` or double `__`.

class Computer:

```
def __init__(self):  
    self.__maxprice = 900
```

```
def sell(self):  
    print("Selling Price: {}".format(self.__maxprice))
```

```
def setMaxPrice(self, price):  
    self.__maxprice = price
```

```
c = Computer()  
c.sell()
```

```
# change the price  
c.__maxprice = 1000  
c.sell()
```

```
# using setter function  
c.setMaxPrice(1000)  
c.sell()
```

In the above program, we defined a Computer class.

We used `__init__()` method to store the maximum selling price of Computer. We tried to modify the price. However, we can't change it because Python treats the `__maxprice` as private attributes.

As shown, to change the value, we have to use a setter function i.e `setMaxPrice()` which takes price as a parameter.

## Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle).

However we could use the same method to color any shape. This concept is called Polymorphism.



```
class Parrot:
```

```
    def fly(self):  
        print("Parrot can fly")
```

```
    def swim(self):  
        print("Parrot can't swim")
```

```
class Penguin:
```

```
    def fly(self):  
        print("Penguin can't fly")
```

```
    def swim(self):  
        print("Penguin can swim")
```

```
# common interface  
def flying_test(bird):  
    bird.fly()
```

```
#instantiate objects  
blu = Parrot()  
peggy = Penguin()
```

```
# passing the object  
flying_test(blu)  
flying_test(peggy)
```



Talent Battle

Talent Battle

In the above program, we defined two classes Parrot and Penguin. Each of them have a common fly() method. However, their functions are different.

To use polymorphism, we created a common interface i.e flying\_test() function that takes any object and calls the object's fly() method.

Thus, when we passed the blu and peggy objects in the flying\_test() function, it ran effectively.

## Key Points to Remember:

- Object-Oriented Programming makes the program easy to understand as well as efficient.
- Since the class is sharable, the code can be reused.
- Data is safe and secure with data abstraction.
- Polymorphism allows the same interface for different objects, so programmers can write efficient code.

## Python Objects and Classes

Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.

An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

An object is also called an instance of a class and the process of creating this object is called **instantiation**.

## Defining a Class in Python

Like function definitions begin with the `def` keyword in Python, class definitions begin with a **class** keyword.

The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

```
class MyNewClass:  
    """This is a docstring. I have created a new class"""  
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores \_\_. For example, \_\_doc\_\_ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:  
    "This is a person class"  
    age = 10
```

```
def greet(self):  
    print('Hello')
```

```
# Output: 10  
print(Person.age)
```

```
# Output: <function Person.greet>  
print(Person.greet)
```

```
# Output: "This is a person class"  
print(Person.__doc__)
```

## Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a **function** call.



```
class Person:  
    "This is a person class"  
    age = 10  
  
    def greet(self):  
        print('Hello')
```

```
# create a new object of Person class  
harry = Person()
```

```
# Output: <function Person.greet>  
print(Person.greet)
```

```
# Output: <bound method Person.greet of <__main__.Person object>>  
print(harry.greet)
```

```
# Calling object's greet() method  
# Output: Hello  
harry.greet()
```

You may have noticed the self parameter in function definition inside the class but we called the method simply as harry.greet() without any arguments. It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument.

## Constructors in Python

Class functions that begin with double underscore `__` are called special functions as they have special meaning.

Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')

num1 = ComplexNumber(2, 3)

num1.get_data()

num2 = ComplexNumber(5)
num2.attr = 10
print((num2.real, num2.imag, num2.attr))
print(num1.attr)
```

In the above example, we defined a new class to represent complex numbers. It has two functions, `__init__()` to initialize the variables (defaults to zero) and `get_data()` to display the number properly.

We created a new attribute `attr` for object `num2` and read it as well. But this does not create that attribute for object `num1`.

When we do `c1 = ComplexNumber(1,3)`, a new instance object is created in memory and the name `c1` binds with it.

On the command `del c1`, this binding is removed and the name `c1` is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.

This automatic destruction of unreferenced objects in Python is also called **garbage collection**.

# Python Inheritance

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

## Python Inheritance Syntax

```
class BaseClass:
```

```
    Body of base class
```

```
class DerivedClass(BaseClass):
```

```
    Body of derived class
```

Derived class inherits features from the base class where new features can be added to it. This results in **re-usability** of code.

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```



```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

```
>>> t = Triangle()
```

```
>>> t.inputSides()
```

```
Enter side 1 : 3
```

```
Enter side 2 : 5
```

```
Enter side 3 : 4
```

```
>>> t.dispSides()
```

```
Side 1 is 3.0
```

```
Side 2 is 5.0
```

```
Side 3 is 4.0
```

```
>>> t.findArea()
```

```
The area of the triangle is 6.00
```

We can see that even though we did not define methods like `inputSides()` or `dispSides()` for class `Triangle` separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

## Python Access Modifiers

In most of the object-oriented languages access modifiers are used to limit the access to the variables and functions of a class. Most of the languages use three types of access modifiers, they are - private, public and protected.

Just like any other object oriented programming language, access to variables or functions can also be limited in python using the access modifiers. Python makes the use of underscores to specify the access modifier for a specific data member and member function in a class.

## **Python: Types of Access Modifiers**

There are 3 types of access modifiers for a class in Python. These access modifiers define how the members of the class can be accessed. Of course, any member of a class is accessible inside any member function of that same class. Moving ahead to the type of access modifiers, they are:

### **Access Modifier: Public**

The members declared as Public are accessible from outside the Class through an object of the class.

### **Access Modifier: Protected**

The members declared as Protected are accessible from outside the class but only in a class derived from it that is in the child or subclass.

### **Access Modifier: Private**

These members are only accessible from within the class. No outside Access is allowed.

## public Access Modifier

By default, all the variables and member functions of a class are public in a python program.

# defining a class Employee

```
class Employee:
```

```
    # constructor
```

```
    def __init__(self, name, sal):
```

```
        self.name = name;
```

```
        self.sal = sal;
```

All the member variables of the class in the above code will be by default public, hence we can access them as follows:

```
>>> emp = Employee("Ironman", 999000);
```

```
>>> emp.sal;
```

```
999000
```

## protected Access Modifier

According to Python convention adding a prefix `_` (single underscore) to a variable name makes it protected.

# defining a class Employee

```
class Employee:
```

```
    # constructor
```

```
    def __init__(self, name, sal):
```

```
        self._name = name; # protected attribute
```

```
        self._sal = sal;   # protected attribute
```

In the code above we have made the class variables `name` and `sal` protected by adding an `_` (underscore) as a prefix, so now we can access them as follows:

```
>>> emp = Employee("Captain", 10000);
```

```
>>> emp._sal;
```

```
10000
```

Similarly if there is a child class extending the class `Employee` then it can also access the protected member variables of the class `Employee`.

```
# defining a child class
class HR(Employee):
```

```
    # member function task
    def task(self):
        print ("We manage Employees")
```

Now let's try to access protected member variable of class Employee from the class HR:

```
>>> hrEmp = HR("Captain", 10000);
>>> hrEmp._sal;
10000
>>> hrEmp.task();
We manage Employees
```

## private Access Modifier

While the addition of prefix `__`(double underscore) results in a member variable or function becoming private.

# defining class Employee

```
class Employee:
```

```
    def __init__(self, name, sal):
```

```
        self.__name = name;    # private attribute
```

```
        self.__sal = sal;      # private attribute
```

If we want to access the private member variable, we will get an error.

```
>>> emp = Employee("Bill", 10000);
```

```
>>> emp.__sal;
```



```
# define parent class Company
class Company:
    # constructor
    def __init__(self, name, proj):
        self.name = name    # name(name of company) is public
        self._proj = proj   # proj(current project) is protected

    # public function to show the details
    def show(self):
        print("The code of the company is = ",self.ccode)

# define child class Emp
class Emp(Company):
    # constructor
    def __init__(self, eName, sal, cName, proj):
        # calling parent class constructor
        Company.__init__(self, cName, proj)
        self.name = eName # public member variable
        self.__sal = sal  # private member variable

    # public function to show salary details
    def show_sal(self):
        print("The salary of ",self.name," is ",self.__sal,)

# creating instance of Company class
c = Company("Stark Industries", "Mark 4")
# creating instance of Employee class
e = Emp("Steve", 9999999, c.name, c._proj)

print("Welcome to ", c.name)
print("Here ", e.name," is working on ",e._proj)

# only the instance itself can change the __sal variable
# and to show the value we have created a public function show_sal()
e.show_sal()
```

# Method Overriding in Python

Method overriding is a concept of **object oriented programming** that allows us to change the implementation of a function in the **child class** that is defined in the **parent class**.

It is the ability of a child class to change the implementation of any method which is already provided by one of its parent class(ancestors).

Following conditions must be met for overriding a function:

**1.Inheritance** should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.

2.The function that is redefined in the child class should have the same signature as in the parent class i.e. same **number of parameters**.

```
# parent class
class Parent:
    # some random function
    def anything(self):
        print('Function defined in parent class!')
```

```
# child class
class Child(Parent):
    # empty class definition
    pass
```

```
obj2 = Child()
obj2.anything()
```

While the child class can access the parent class methods, it can also provide a new implementation to the parent class methods, which is called **method overriding**.

class Animal:

    # properties

        multicellular = True

        # Eukaryotic means Cells with Nucleus

        eukaryotic = True

    # function breathe

    def breathe(self):

        print("I breathe oxygen.")

    # function feed

        def feed(self):

            print("I eat food.")

class Herbivorous(Animal):

    # function feed

        def feed(self):

            print("I eat only plants. I am vegetarian.")

herbi = Herbivorous()

herbi.feed()

# calling some other function

herbi.breathe()

## Python Polymorphism Example

```
class Square:  
    side = 5  
    def calculate_area(self):  
        return self.side * self.side
```

```
class Triangle:  
    base = 5  
    height = 4  
    def calculate_area(self):  
        return 0.5 * self.base * self.height
```

```
sq = Square()  
tri = Triangle()  
print("Area of square: ", sq.calculate_area())  
print("Area of triangle: ", tri.calculate_area())
```

## Python Multiple Inheritance

A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

Example

```
class Base1:
```

```
    pass
```

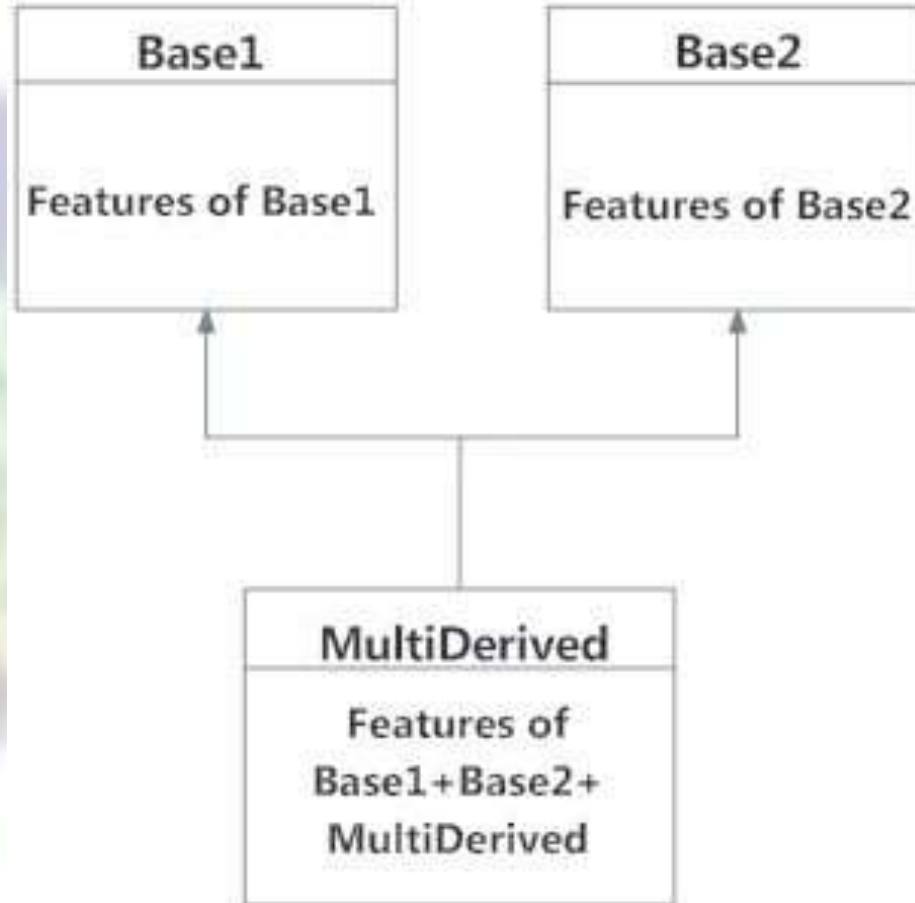
```
class Base2:
```

```
    pass
```

```
class MultiDerived(Base1, Base2):
```

```
    pass
```

Here, the MultiDerived class is derived from Base1 and Base2 classes.



## Python Multilevel Inheritance

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.

An example with corresponding visualization is given below.

```
class Base:  
    pass  
class Derived1(Base):  
    pass  
class Derived2(Derived1):  
    pass
```

Here, the Derived1 class is derived from the Base class, and the Derived2 class is derived from the Derived1 class.