DETAILED CONTENTS

1. Introduction (03 Periods)

- Brief History of Python
- Python Versions
- Installing Python
- Environment Variables
- Executing Python from the Command Line
- IDLE
- Editing Python Files
- Python Documentation
- Getting Help
- Dynamic Types
- Python Reserved Words
- Naming Conventions

2. Basic Python Syntax (03 Periods)

- Basic Syntax
- Comments
- String Values
- String Methods
- The format Method
- String Operators
- Numeric Data Types
- Conversion Functions
- Simple Output
- Simple Input
- The % Method

- The print Function

3. Language Components (04 Periods)

- Indenting Requirements
- The if Statement
- Relational and Logical Operators
- Bit Wise Operators
- The while Loop
- break and continue
- The for Loop

4. Collections (09 Periods)

- Introduction
- Lists
- Tuples
- Sets
- Dictionaries
- Sorting Dictionaries
- Copying Collections
- Summary

5. Functions (06 Periods)

- Introduction
- Defining Your Own Functions
- Parameters
- Function Documentation
- Keyword and Optional Parameters
- Passing Collections to a Function
- Variable Number of Arguments
- Scope
- Functions - "First Class Citizens"
- Passing Functions to a Function
- map
- filter
- Mapping Functions in a Dictionary
- Lambda
- Inner Functions
- Closures

6. Modules (03 Periods)

- Modules
- Standard Modules - sys
- Standard Modules - math
- Standard Modules - time
- The dir Function

7. Exceptions (05 Periods)

- Errors
- Runtime Errors
- The Exception Model

- Exception Hierarchy
- Handling Multiple Exceptions
- Raise
- assert

8. Input and Output (03 Periods)

- Introduction
- Data Streams
- Creating Your Own Data Streams
- Access Modes
- Writing Data to a File
- Reading Data From a File
- Additional File Methods
- Using Pipes as Data Streams
- Handling IO Exceptions

9. Classes in Python (07 Periods)

- Classes in Python
- Principles of Object Orientation
- Creating Classes
- Instance Methods
- File Organization
- Special Methods
- Class Variables
- Inheritance
- Polymorphism

10. Regular Expressions (05 Periods)

- Introduction
- Simple Character Matches
- Special Characters
- Character Classes
- Quantifiers
- The Dot Character
- Greedy Matches
- Grouping
- Matching at Beginning or End
- Match Objects
- Substituting
- Splitting a String
- Compiling Regular Expressions
- Flags

Python – Introduction

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python Features:

Python's features include −

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eye.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python- Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
#!/usr/bin/python

counter =                # An integer
100                      assignment

print
counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result −

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example −

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

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

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

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is −

```
var1 = 1
var2 = 10
```

You can delete a single object or multiple objects by using the del statement. For example −

```
del  var1[,var2[,var3[        ,varN]]]]
```

Python supports four different numerical types −

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers −

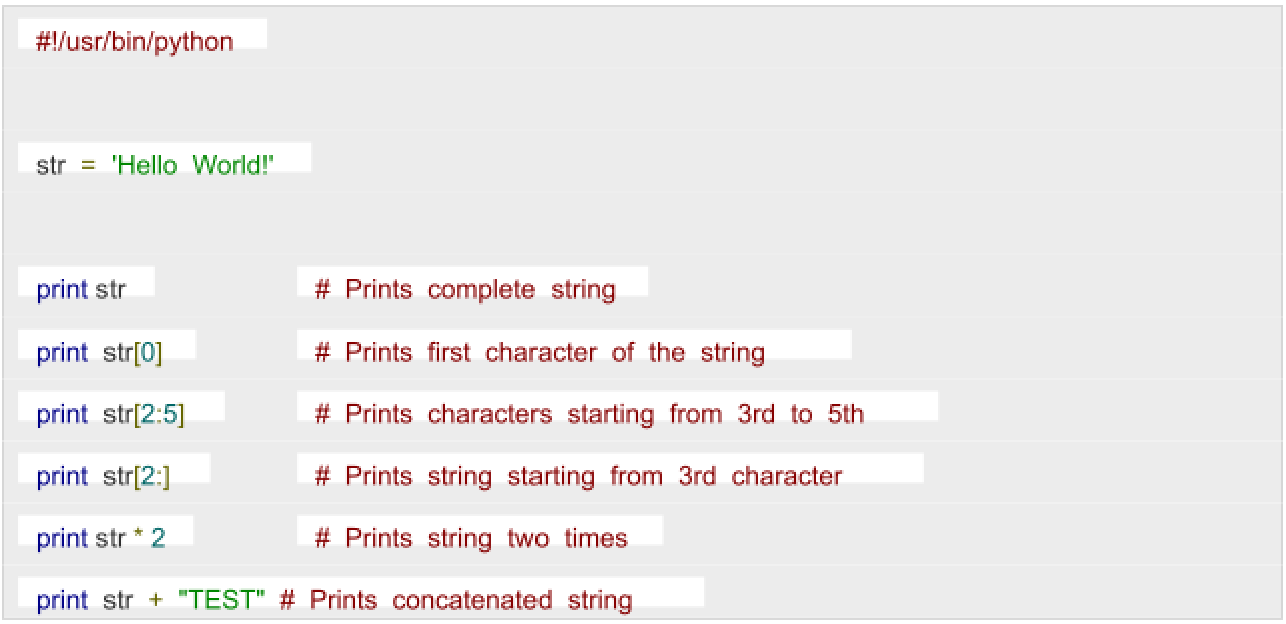
long	float	complex
51924361L	0.0	3.14j
-0x19323L	15.20	45.j
0122L	-21.9	9.322e-36j

Python Strings

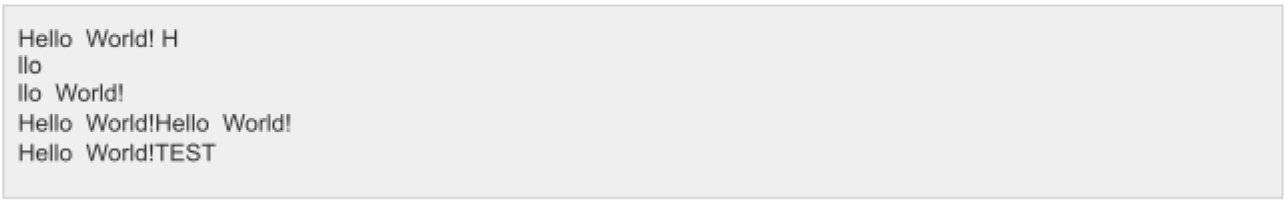
Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([
] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example −

This will produce the following result −



```
#!/usr/bin/python


str = 'Hello World!'


print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```



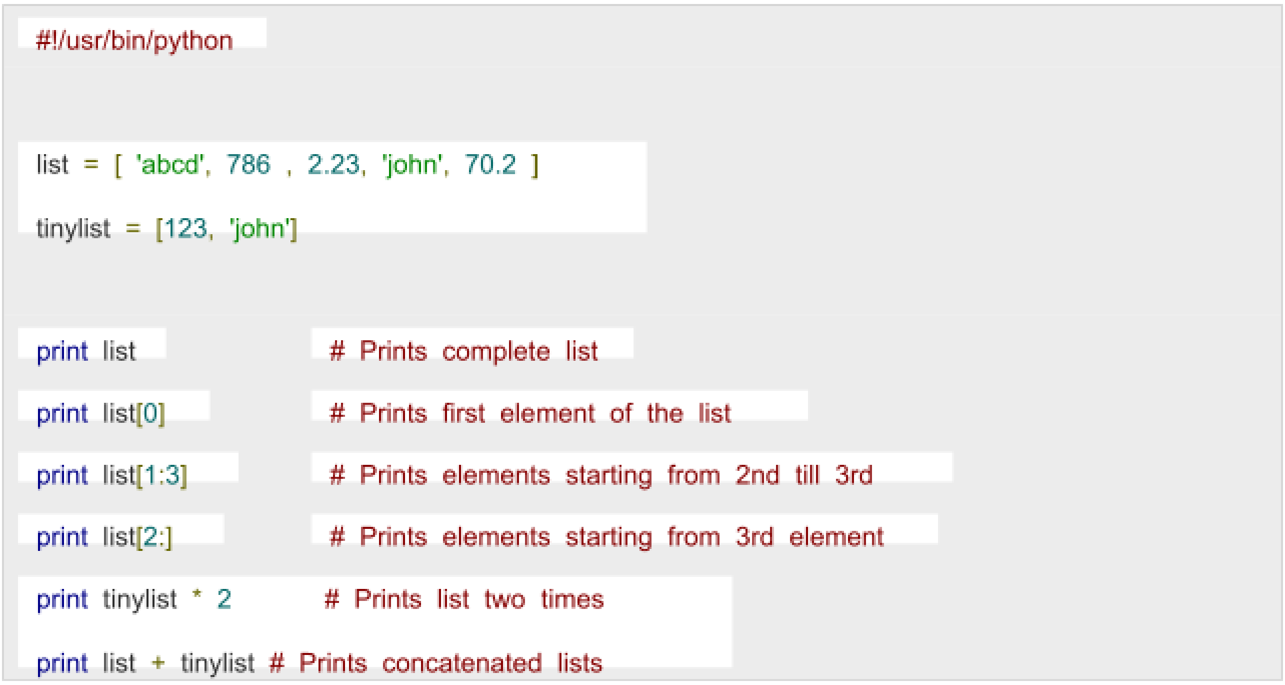
```
Hello World! H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the

asterisk (*) is the repetition operator. For example-



```
#!/usr/bin/python


list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']


print list              # Prints complete list

print list[0]           # Prints first element of the list

print list[1:3]         # Prints elements starting from 2nd till 3rd

print list[2:]          # Prints elements starting from 3rd element

print tinylist * 2      # Prints list two times

print list + tinylist # Prints concatenated lists
```

This produce the following result –

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example −

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print tuple               # Prints complete list
print tuple[0]            # Prints first element of the list
print tuple[1:3]          # Prints elements starting from 2nd till 3rd
print tuple[2:]           # Prints elements starting from 3rd element
print tinytuple * 2       # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

This produce the following result −

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result −

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'} ['dept',
'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.

Sr.No.	Function & Description
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.
10	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
11	**frozenset(s)** Converts s to a frozen set.
12	**chr(x)** Converts an integer to a character.
13	**unichr(x)** Converts an integer to a Unicode character.
14	**ord(x)** Converts a single character to its integer value.
15	**hex(x)** Converts an integer to a hexadecimal string.
16	**oct(x)** Converts an integer to an octal string.

Basic Operators

Operators are the constructs which can manipulate the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and
+ is called operator.

Types of Operator:

Python language supports the following types of operators.

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

Let us have a look at all the operators one by one.

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then −

[Show Example]

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a − b = - 10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, - 11.0//3 = -4.0

Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then −

[Show Example]

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

[Show Example]

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c /
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

----

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python

language [Show Example]

& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20

then [Show Example]

AND	true.	is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Used to reverse the logical state of its operand.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

[Show Example]

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

[Show Example]

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

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

[Show Example]

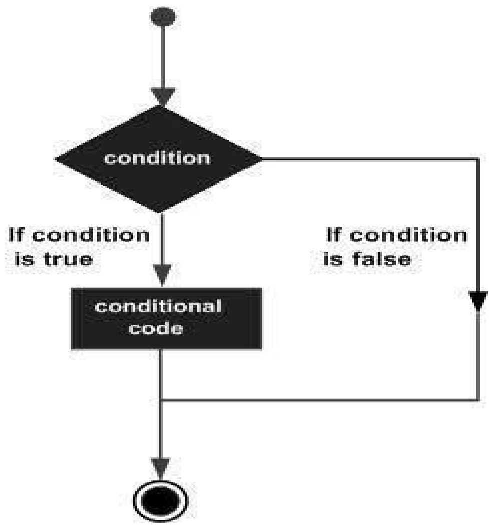
Operators	Meaning
`()`	Parentheses
`**`	Exponent

`+x, -x, ~x`	Unary plus, Unary minus, Bitwise NOT
`*, /, //, %`	Multiplication, Division, Floor division, Modulus
`+, -`	Addition, Subtraction
`<<, >>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==, !=, >, >=, <, <=, is, is not, in, not in`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

Decision Making:

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.



Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements.

Sr.No.	Statement & Description
1	if statements An if statement consists of a boolean expression followed by one or more statements.

2	if...else statements An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.
3	nested if statements You can use one if or else if statement inside another if or else if statement(s).

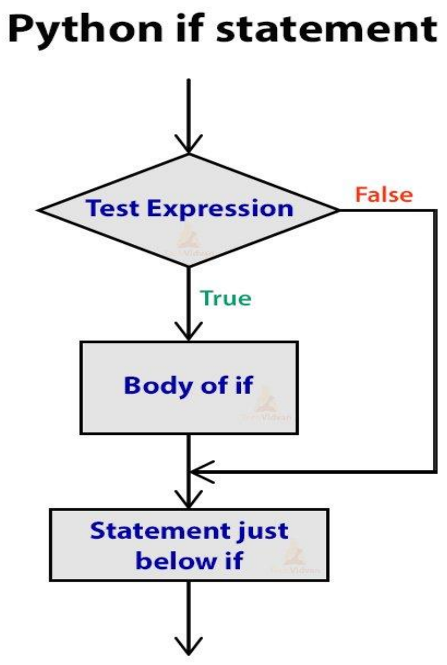
1.Python if statement:

if statement is the most simple form of decision-making statement. It takes an expression and checks if the expression evaluates to True then the block of code in if statement will be executed.

If the expression evaluates to False, then the block of code is skipped.

Syntax:

```
if ( expression ):

 Statement 1

 Statement 2

 Statement n .
```



Example 1:

```
a = 20 ; b = 20

if ( a == b ):

 print( “a and b are equal”)

print(“If block ended”)
```

Output:

```
a and b are equal

If block ended
```

Example 2:

```
num = 5

if ( num >= 10):

 print(“num is greater than 10”)

print(“if block ended”)
```

Output:

```
If block ended
```

In example 1, we see that the condition a==b evaluates to True. Therefore, the block of code inside if statement is executed.

In example 2, the condition evaluates to False, therefore, the print statement was not executed and the only statement that got executed was because it was outside the if block.

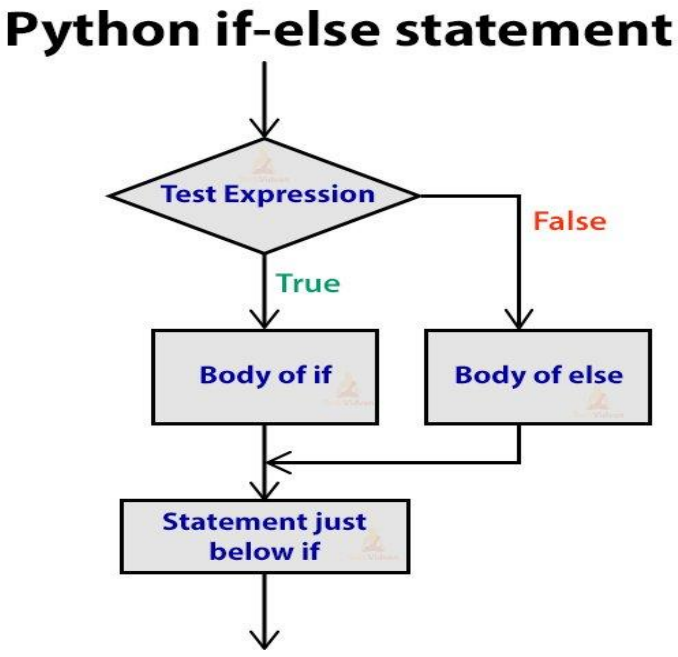
Note: Don’t forget to add a colon(:) after if statement and indent the statements properly that are executed when a condition is True.

2.Python if-else statement:

From the name itself, we get the clue that the if-else statement checks the expression and executes the if block when the expression is True otherwise it will execute the else block of code. The else block should be right after if block and it is executed when the expression is False.

Syntax:

```
if( expression ):

 Statement

else:

 Statement
```



Example 1:

```
number 1 = 20 ; number2 = 30

if(number1 >= number2 ):

 print("number 1 is greater than number 2

else:

 print("number 2 is greater than number 1
```

Output:

```
number 2 is greater than number 1
```

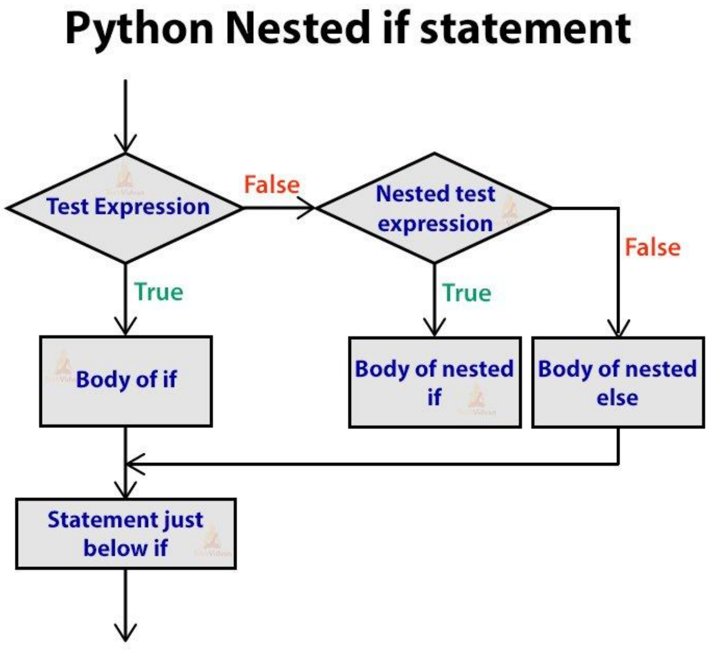
Note: Only one else statement is followed by an if statement. If you use two else

3.Python Nested if statement

In very simple words, Nested if statements is an if statement inside another if statement. Python allows us to stack any number of if statements inside the block of another if statements. They are useful when we need to make a series of decisions.

Syntax:

```
if (expression):

 if(expression):

   Statement of nested if

 else:

   Statement of nested if else

 Statement of outer if

Statement outside if block
```



Example :

```
num1 = int( input())

num2 = int( input())

if( num1>= num2):

   if(num1 == num2):

       print(f'{num1} and {num2} are equal')

   else:

       print(f'{num1} is greater than {num2}')

else:

   print(f'{num1} is smaller than {num2}')
```

Output 1:

```
10

20

10 is smaller than 20
```

Output 2:

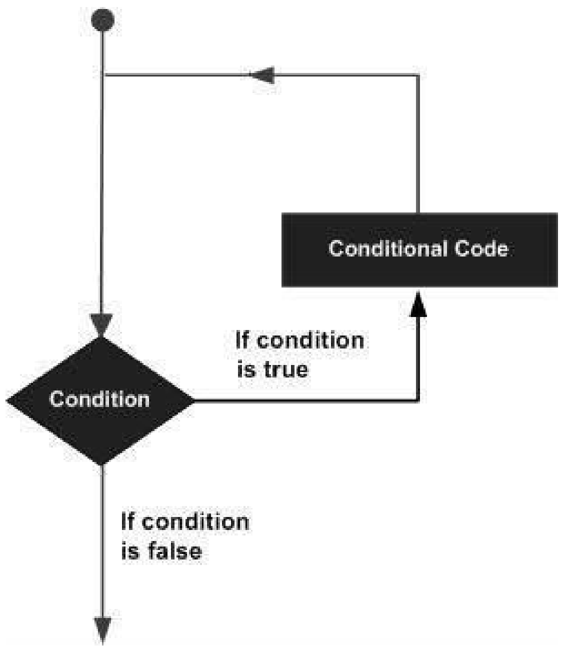
```
5

5

5 and 5 are equal
```

Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several times.

- A loop statement allows us to execute a statement or group of statements multiple times. They are pretty useful and can be applied to various use cases.

The following diagram illustrates a loop statement −



1. Python For in loop

For loop in Python is used to iterate over a sequence of items like list, tuple, set, dictionary, string or any other iterable objects.

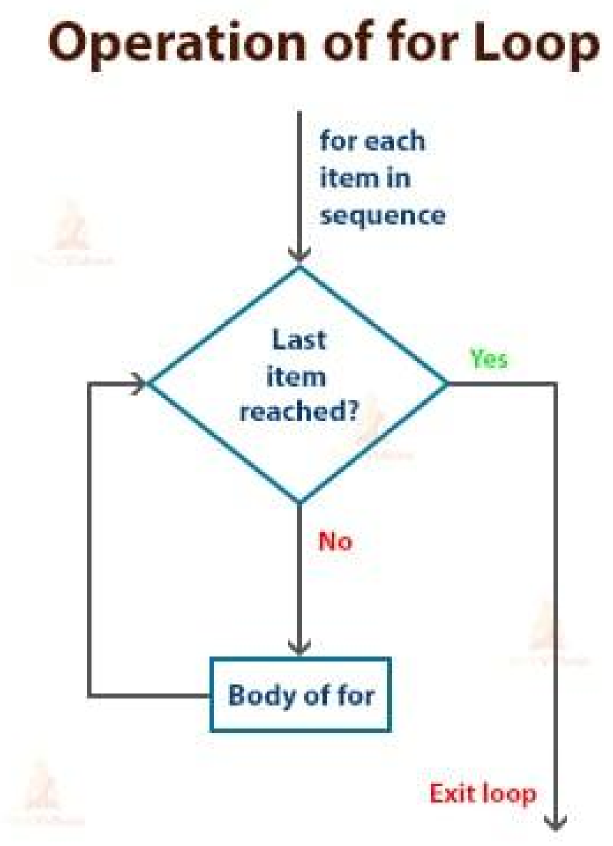
Syntax:

```
for item in sequence:

body of for loop
```

The Python for loop doesn’t need indexing unlike other programming languages (C/C++ or Java). It works like an iterator and the item variable will contain an item from the sequence at each iteration.

The for loop continues until we reach the end of the sequence.

Flowchart of for loop in Python



Code:

```
for item in [1,2,3,4]:

print( item)
```

Output:

```
1
2
3
4
```

a. The range() function

When using for loops in Python, the range() function is pretty useful to specify the number of times the loop is executed. It yields a sequence of numbers within a specified range.

Syntax:

```
range(start, stop, step)
```

- The first argument is the starting value. It is zero by default.
- The second argument is the ending value of the range.
- The third argument is the number of steps to take after each yield.

#converting range to list

Code:

```
list(range(10))

list(range(4,10))

list(range(2,10,2))
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],

[4, 5, 6, 7, 8, 9],

[2, 4, 6, 8]
```

b. Iterating over range object

You can use the for loop to iterate over the range of objects.

Code:

```
for i in range(2,20,2):

 print(i)
```

Output:

```
2

4

6

8

10

12

14

16

18
```

Similarly, we can iterate the same way in tuples, lists, sets, and strings.

Code:

```
for char in “Hello”:

 print(char)
```

Output:

```
H

e

l

l
```

o

c. Using else in for loop

In Python programming, the loops can also have an else part which will be executed once when the loop terminates.

Code:

```
for i in [1, 2, 3, 4]:

 print(i)

else:

 print(“The loop ended”)
```

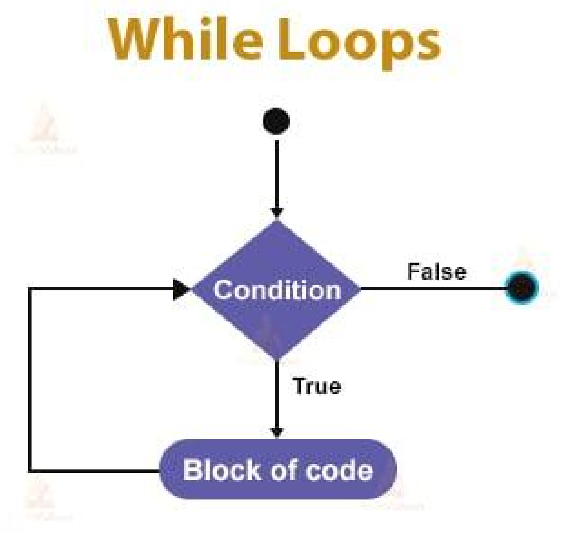
Output:

```
1

2

3

4

The loop ended
```

2. Python While loop

The while loop in Python executes a block of code until the specified condition becomes False.

Flowchart of while loop in Python



Syntax:

```
while( condition):

 Body of while

 Inside while block
```

Code:

```
count = 0

while( count< 10):

 print(count)

 count = count + 2
```

Output:

```
0

2

4

6

8
```

In the example, the while statement checks if count is less than 10.

Initially, count is zero so the statement is true and it executes the body of while. Then the count gets incremented by 2. Again we check the condition and this goes on till the condition becomes false.

Here, when our code checks 10<10, the statement returns False and so the code in while block is not executed.

a. Infinite loop

A loop is called an infinite loop when the loop will never reach its end.

Usually, when a condition is always True in a while loop, the loop will become an infinite loop. So we should be careful when writing conditions and while updating variables used in the loop.

In Python shell, we can program on an infinite loop by using CTRL + C. Sometimes, we need to implement an infinite terminate loop for example, when reading frames from a webcam.

Code:

```
while (True):

   print("Infinite Loop")
```

This code will keep on printing the "Infinite Loop" statement. We terminate the loop by pressing CTRL + C.

b. Using else in while loop

The while loop may also have an else part after the loop. It is executed only when the condition of while loop becomes false. But if we break out of the loop before the condition has not reached false, then the else block does not get executed.

Code:

```
num = 0

while( num<10 ):

   print(num)

   num += 1

   if( num==5):

       break

else:
      print('Loop ended')
```

Output:

```
0
1
2
3
4
```

Here the else statement didn’t get executed because the break statement ends the loop execution and the while condition therefore never becomes false.

3. Python Nested Loops

We can nest a loop inside another loop which simply means that a loop within a loop. Let’s see this with an example.

Code:

```
for num1 in range(3):

   for num2 in range(5, 8):

       print(num1, ",", num2)
```

Output:

```
0 , 5
0 , 6
0 , 7
1 , 5
1 , 6
1 , 7
2 , 5
2 , 6
2 , 7
```

From this example, you can observe that the first iteration of the outer loop will run the whole inner loop and then in the next iteration of the outer loop, the inner loop gets executed again. This process is repeated until we reach the end of the outer loop.

4. Loop Control Statements in Python

Python allows us to control the flow of the execution of the program in a certain manner. For this we use the continue, break and pass keywords.

a. break

The break statement inside a loop is used to exit out of the loop. Sometimes in a program, we need to exit the loop when a certain condition is fulfilled.

Code:

```
num = 0

while( num <10 ):

   num +=1

   if(num==5): break

   print( num )

print("Loop ended")
```

Output:

```
1

2

3

4
```

```
Loop ended
```

b. continue

The continue statement is used to skip the next statements in the loop.

When the program reaches the continue statement, the program skips the statements after continue and the flow reaches the next iteration of the loop.

Let’s take the same example –

Code:

```
num = 0

while( num <10 ):

   num +=1

   if(num==5): continue

   print( num )

print("Loop ended")
```

Output:

```
1

2

3

4

6

7

8

9

10

Loop ended
```

Here, we see that when the num variable is equal to 5, the continue statement is executed. It then doesn’t execute the lines after the continue statement and the control is sent to the next iteration.

c. pass

The pass is a null statement and the Python interpreter returns a no-operation (NOP) after reading the pass statement. Nothing happens when the pass statement is executed. It is used as a placeholder when implementing new methods or we can use them in exception handling.

Code:

```
nums = [1,2,3,4]

for num in nums:

 pass

print(nums)
```

Python Functions

Python is object-oriented but also supports functional programming.

In this tutorial, you will learn about Python functions and will learn to create them and call them.

Also, you will get to understand the parameters of functions in Python, the return statement, anonymous functions in Python, and lastly, you will see what is recursion in Python.

What are Functions in Python?

A function is a block of code that has a name and you can call it. Instead of writing something 100 times, you can create a function and then call it 100 times. You can call it anywhere and anytime in your program. This adds reusability and modularity to your code.Functions can take arguments and return values.

Types of Python Functions

Functions can be of two types – built-in or user-defined.

- Build-in functions:

  The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use.

  The `dict()` method creates a dictionary

  The `dir()` method returns a list of valid attributes of the specified object.

- User defined Functions:

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

Python has many useful built-in functions like print() and type() but in this article, we will focus on user-defined functions.

- Creating Functions in Python:

To define a function, we follow this syntax:

Syntax:

```
def func(parameters):

 code

 return
```

- To define a function, you use the def keyword. You give the function a name and it can take some parameters if you want. Then you specify a block of code that will execute when you call the function.

- There are no curly braces in Python and so you need to indent this code block otherwise it won't work. You can make the function return a value.

Let's take an example for you.

Code:

```
def add(a,b):

 print("I will add two numbers")

 return a+b
```

This function prints "I will add two numbers" and returns the sum of two numbers.

Code:

```
>>> add(2,4)
```

Output:

```
I will add two numbers

6
```

- Calling Functions in Python

To call a function, use the function name followed by parenthesis:

Example:

```
def my_function():

  print("Hello from a function"

my_function()
```

Output:

```
Hello from a function
```

Closures in Python:

Like nested loops, we can also nest functions. That said, Python gives us the power to define functions within functions.

- Python Closures are these inner functions that are enclosed within the outer function. Closures can access variables present in the outer function scope. It can access these variables even after the outer function has completed its execution.

Python simple closure example

The following is a simple example of a Python closure.

simple_closure.py

```
#!/usr/bin/python

def make_printer(msg):

    msg = "hi there"

    def printer():

        print(msg)

    return printer

myprinter = make_printer("Hello there")

myprinter()

myprinter()

myprinter()
```

In the example, we have a make_printer function, which creates and returns a function. The nested printer function is the closure.

```
myprinter = make_printer("Hello there")
```

The make_printer function returns a printer function and assigns it to the myprinter variable. At this moment, it has finished its execution. However, the printer closure still has access to the msg variable.

```
$ ./simple_closure.py

hi there

hi there

hi there
```

The *globals()* and *locals()* Functions

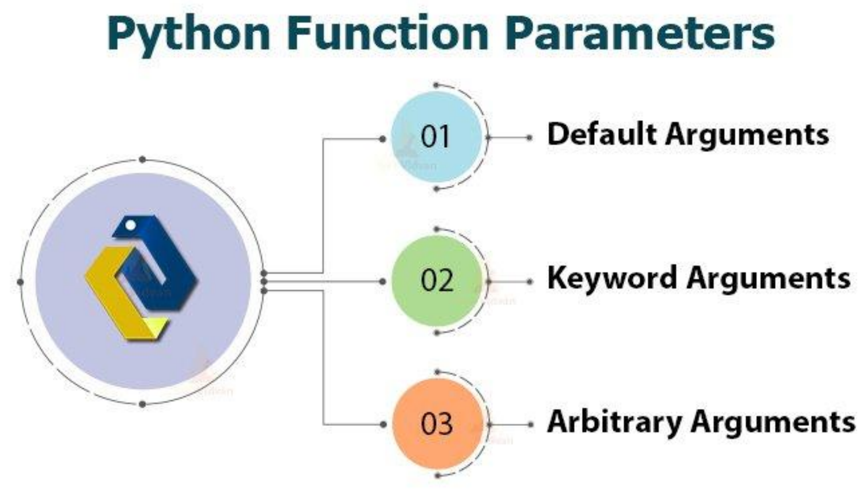
- The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

- If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

- If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

- The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

Python Function Parameters



Functions can take values and operate on them.

In the above example, the add() function takes parameters a and b.

Python has 3 types of parameters or arguments other than positional/required arguments – default, keyword, arbitrary.

1. Default Arguments

You can specify a default value for arguments. If the user calls the function, they can skip providing a value for that argument. The default value is used.

Code:

```
def add(a, b=4):

 return a+b
```

Now you can call it with one or two values.

Code:

```
>>> add(2,4)
```

Output:

```
6
```

Code:

```
>>> add(2)
```

Output:

```
6
```

Here b is 4 by default so it returns 2+4, which is 6.

Functions can have any number of default arguments.

2. Keyword Arguments

If you pass keyword arguments to a function, you don't need to remember the order of the parameters.

Code:

```
def add(a, b):

 return a+b
```

Code:

```
>>> add(b=4,a=2)
```

Output:

```
6
```

3. Arbitrary Arguments

If you don’t know how many arguments your function will get at runtime, you can use the arbitrary arguments *args and **kwargs.

*args is a variable number of arguments and **kwargs is a variable number of keyword arguments. You can call them anything.

Code:

```
def add(*args,**kwargs):

 result=0

 for arg in args:

   result+=arg

 print(result)

 for key,value in kwargs.items():

   print(key,value)
```

Code:

```
>>> add(2,4,6,a=8,b=10)
```

Output:

```
12

a 8

b 10
```

2, 4, 6, are in *args and a=8 and b=10 are in **kwargs.

Result is 12.

You cannot pass positional arguments after keyword arguments.

Code:

```
>>> add(2,4,6,a=8,7,b=10)
```

Output:

```
SyntaxError: positional argument follows keyword argument
```

a.The Return Statement in Python

It is not mandatory for a function to return a value but it can. For this, you use the return keyword.

Code:

```
def show_names(names):

 return names
```

Code:

```
>>> show_names(['Jack', 'Paris', 'Nicole'])
```

Output:

```
[‘Jack’, ‘Paris’, ‘Nicole’]
```

This function returns a list.

b.Anonymous Functions in Python

If you want a function only once or it has only one line of code, you can avoid giving it a name. They are called lambda expressions.

Code:

```
>>> lambda a=2,b=4:a+b

<function <lambda> at 0x00000198A3692168>
```

Now to call this:

Code:

```
>>> (lambda a=2,b=4:a+b)()
```

Output:

```
6
```

Let’s try more possibilities.

Code:

```
>>> (lambda a,b:a+b)(2,4)
```

Output:

```
6
```

c.Recursion in Python

When a function calls itself, it is recursion. In other words, when a function body has calls to the function itself, it is recursion.

For recursion, you should specify a base condition otherwise the function can execute infinitely.

Let’s take the example of calculating factorials.

Code:

```
def factorial(num):

 if num==1: return 1

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

Code:

```
>>> factorial(4)
```

Output:

```
24
```

factorial(4) returns 4*factorial(3). factorial(3) returns 3*factorial(2). factorial(2) returns 2*factorial(1). factorial(1) returns 1.

So we have 4*3*2*1. This is equal to 24. So it returns the value 24.

Modules in Python

Modules provide us with a way to share reusable functions. A module is simply a “Python file” which contains code we can reuse in multiple Python programs. A module may contain functions, classes, lists, etc.

Modules in Python can be of two types:

- Built-in Modules.
- User-defined Modules.

1. Built-in Modules in Python:

One of the many superpowers of Python is that it comes with a “rich standard library”. This rich standard library contains lots of built-in modules. Hence, it provides a lot of reusable code.

To name a few, Python contains modules like “os”, “sys”, “datetime”, “random”.

You can import and use any of the built-in modules whenever you like in your program.

2. User-Defined Modules in Python:

Another superpower of Python is that it lets you take things in your own hands. You can create your own functions and classes, put them inside modules and voila! You can now include hundreds of lines of code into any program just by writing a simple import statement.

To create a module, just put the code inside a .py file. Let’s create one.

```python
# my Python module
def greeting(x):
   print("Hello,", x)
```

- Importing Modules in Python

We use the import keyword to import both built-in and user-defined modules in Python.

Let’s import our user-defined module from the previous section into our Python shell:

```
>>> import mypymodule
```

```
To call the greeting function of mypymodule, we simply need to use the dot notation:
```

```
>>> mypymodule.greeting("Techvidvan")
```

Output

```
Hello, Techvidvan
```

Using import...as statement (Renaming a module)

This lets you give a shorter name to a module while using it in your program.

```
>>> import random as r

>>> r.randint(20, 100)
```

Output

```
54
```

The dir( ) Function:

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example –

Code:

```
#!/usr/bin/python

# Import built-in module math
import math

content = dir(math)
print content
```

When the above code is executed, it produces the following result –

Output:

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
 'sqrt', 'tan', 'tanh']
```

1.sys Module

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment.

sys.argv

sys.argv returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

sys.exit

This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

2.Math Module

Some of the most popular mathematical functions are defined in the math module. These include trigonometric functions, representation functions, logarithmic functions, angle conversion functions, etc. In addition, two mathematical constants are also defined in this module.

math.log()

The math.log() method returns the natural logarithm of a given number. The natural logarithm is calculated to the base e.

```
>>> import math

>>>math.log(10)

2.302585092994046
```

Exceptions:

What are Exceptions?

Python is an interpreted language. When you are coding in the Python interpreter, you often have to deal with runtime errors. These are runtime exceptions.

When there is an exception, all execution stops and it displays a red error message on the screen. But if we can handle it, the program will not crash. So an exception is when something goes wrong and your program cannot run anymore.

- Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

- In Python, we can throw an exception in the try block and catch it in except block.

Python has built-in exceptions but you can also define your own exceptions. This code raises an exception:

Code:

```
>>> print(1/0)
```

Output:

```
Traceback (most recent call last):

  File "<pyshell#213>", line 1, in <module>

    print(1/0)

ZeroDivisionError: division by zero
```

It is not possible to divide 1 by 0, so the program has to stop, it cannot execute anymore. So it raises a ZeroDivisionError. This is an in-built exception in Python.

Why use Exception

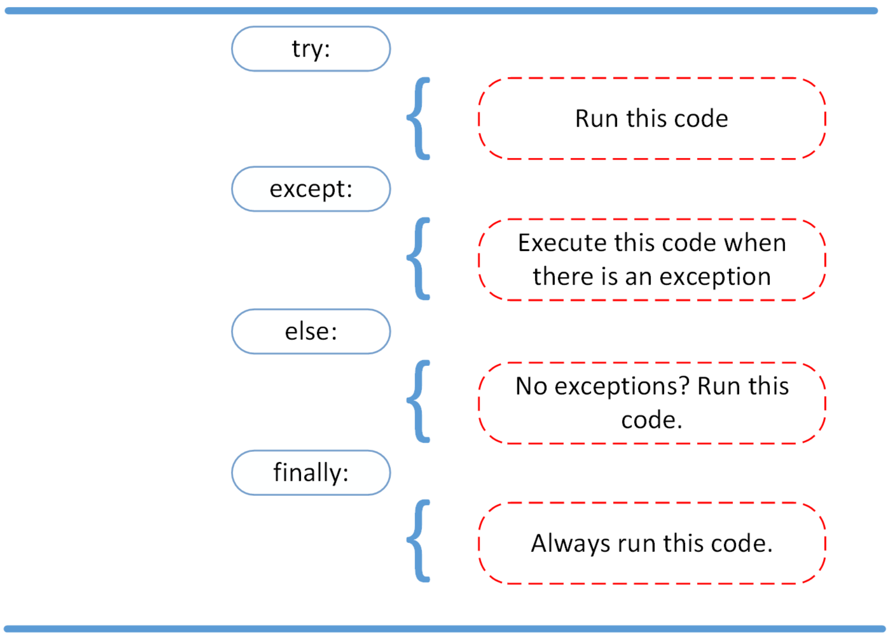
- Standardized error handling: Using built-in exceptions or creating a custom exception with a more precise name and description, you can adequately define the error event, which helps you debug the error event.
- Cleaner code: Exceptions separate the error-handling code from regular code, which helps us to maintain large code easily.
- Robust application: With the help of exceptions, we can develop a solid application, which can handle error event efficiently
- Exceptions propagation: By default, the exception propagates the call stack if you don’t catch it. For example, if any error event occurred in a nested function, you do not have to explicitly catch-and-forward it; automatically, it gets forwarded to the calling function where you can handle it.
- Different error types: Either you can use built-in exception or create your custom exception and group them by their generalized parent class, or Differentiate errors by their actual class.

What are Errors?

An **error** is an action that is incorrect or inaccurate. For example, syntax error. Due to which the program fails to execute.

What is exception handling?

Exception handling is the process of responding to unwanted or unexpected events when a computer program runs. Exception handling deals with these events to avoid the program or system crashing, and without this process, exceptions would disrupt the normal operation of a program.

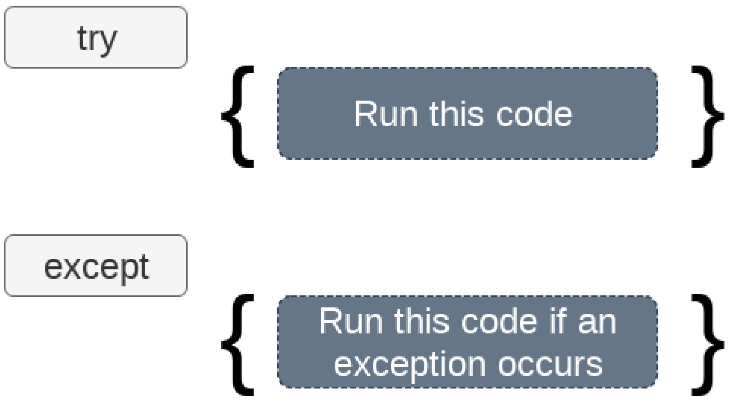

try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

● The try-expect statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the try block. The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.

Syntax:

```
try:

    #block of code


except Exception1:

    #block of code
```



Consider the following example.

Example 1

```
  a = int(input("Enter a:"))

    b = int(input("Enter b:"))

    c = a/b

except:

    print("Can't divide with zero")
```

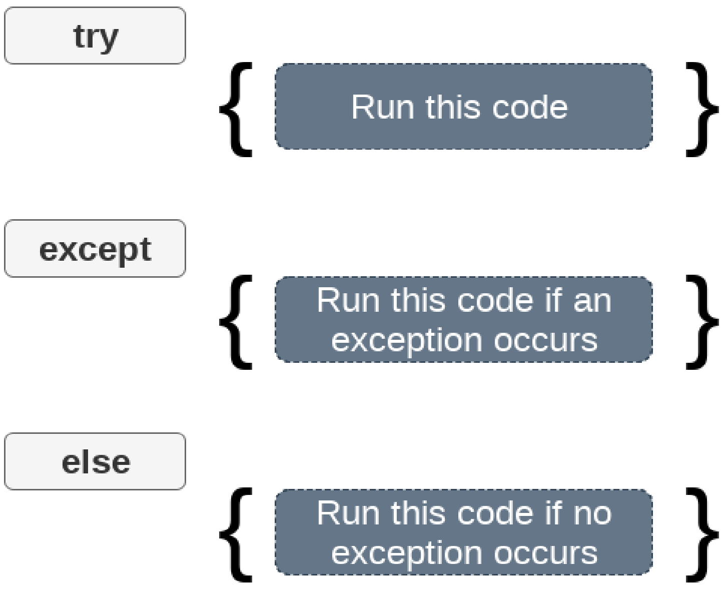
Output:

```
Enter a:10
Enter b:0
Can't divide with zero
```

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

- The syntax to use the else statement with the try-except statement is given below.

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs

Consider the following program.

Example 1:

```
try:

    a = int(input("Enter a:"))

    b = int(input("Enter b:"))

    c = a/b

    print("a/b = %d"%c)

# Using Exception with except
statement. If we print(Exception) it will return exception class

except Exception:

    print("can't divide by zero")

    print(Exception)

else:

    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
```

- ## The except statement with no exception

Python provides the flexibility not to specify the name of exception with the exception statement.

- Consider the following example.

```
Example:

try:

    a = int(input("Enter a:"))

    b = int(input("Enter b:"))

    c = a/b;

    print("a/b = %d"%c)

except:
```

- ## The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception.

Consider the following example:

```
try:

    a = int(input("Enter a:"))

    b = int(input("Enter b:"))
    c = a/b

    print("a/b = %d"%c)

    # Using exception object with the except statement

except Exception as e:

    print("can't divide by zero")

    print(e)

else:

    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
division by zero
```

Points to remember:

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

● Declaring Multiple Exceptions

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

Syntax:

```
try:
    #block of code


except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code


else:
    #block of code
```

Consider the following example.

Example:

```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

Output:

```
Arithmetic Exception
```

● The try...finally block

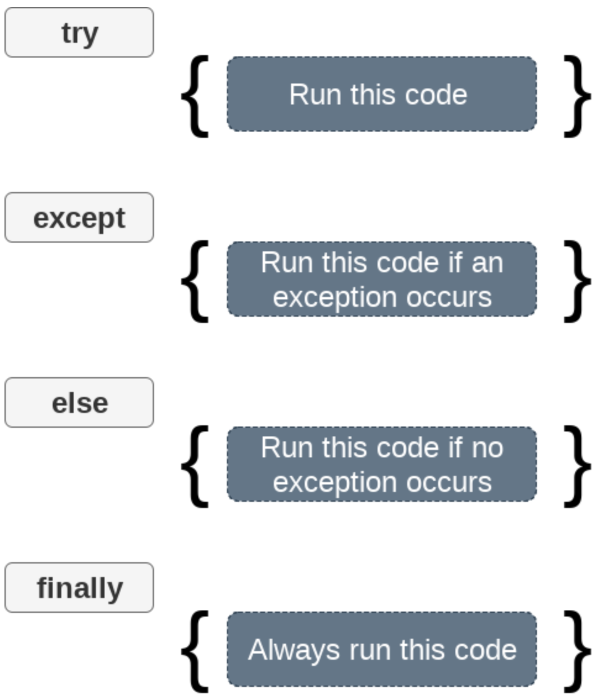
Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

Syntax:

```
try:
    # block of code
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
```

Example:

```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code

Output:

```
file closed
Error
```

Example:

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

```
try:

    print('try block')

    x=int(input('Enter a number: '))

    y=int(input('Enter another number: '))

    z=x/y

except ZeroDivisionError:

    print("except ZeroDivisionError block")

    print("Division by 0 not accepted")

else:

    print("else block")

    print("Division = ", z)

finally:

    print("finally block")

    x=0

    y=0

print ("Out of try, except, else and finally blocks." )
```

Output:

- The first run is a normal case. The out of the else and finally blocks is displayed because the try block is error-free.

Output

```
try block Enter a number: 10 Enter another number: 2 else block Division = 5.0 finally block Out of try, except, else and finally blocks.
```

- The second run is a case of division by zero, hence, the except block and the finally block are executed, but the else block is not executed.

Output

```
try block Enter a number: 10 Enter another number: 0 except ZeroDivisionError block Division by 0 not accepted finally block Out of try, except, else and finally blocks.
```

- In the third run case, an uncaught exception occurs. The finally block is still executed but the program terminates and does not execute the program after the finally block.

Output

```
try block Enter a number: 10 Enter another number: xyz finally block Traceback (most recent call last): File "C:\python36\codes\test.py", line 3, in <module> y=int(input('Enter another number: ')) ValueError: invalid literal for int() with base 10: 'xyz'
```

- Typically the finally clause is the ideal place for cleaning up the operations in a process. For example closing a file irrespective of the errors in read/write operations.

- Raising exceptions

An exception can be raised forcefully by using the **raise** clause in Python. It is useful in that scenario where we need to raise an exception to stop the execution of the program.

For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

The syntax to use the raise statement is given below.

Syntax:

```
raise Exception_class,<value>
```

Points to remember:

1. To raise an exception, the raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.
4. We can pass the value to an exception to specify the exception type.

Example:

```
try:

    age = int(input("Enter the age:"))

    if(age<18):

        raise ValueError

    else:

        print("the age is valid")

except ValueError:

    print("The age is not valid")
```

Output:

```
Enter the age:17
The age is not valid
```

- The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError exception.*

The syntax for assert is –

```
assert Expression[, Arguments]
```

Input-Output in Python

Python provides some built-in functions to perform both input and output operations.

1.Output Operation

In order to print the output, python provides us with a built-in function called print().

Example:

```
 Print("Hello Python")
```

Output:

```
Hello Python
```

2.Reading Input from the keyboard (Input Operation)

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

- raw_input()
- input()

raw_input(): This function reads only one line from the standard input and returns it as a String.

Note: This function is decommissioned in Python 3.

Example:

```
value = raw_input("Please enter the value: ");
print("Input received from the user is: ", value)
```

Output:

```
Please enter the value: Hello Python
```

Input received from the user is:

Hello Python

input(): The input() function first takes the input from the user and then evaluates the expression, which means python automatically identifies whether we entered a string or a number or list.

But in Python 3 the raw_input() function was removed and renamed to input().

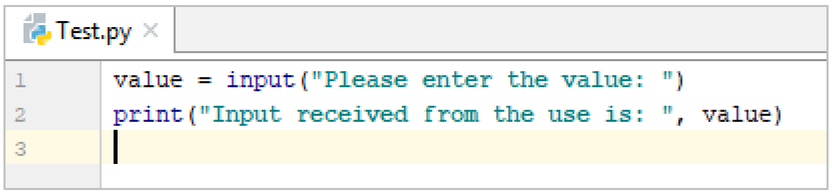
Example:

```
value = input("Please enter the value: ");

print("Input received from the user is: ", value)
```

Output:

Please enter the value: [10, 20, 30]

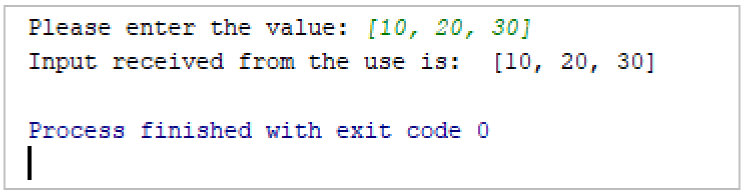
Input received from the user is: [10, 20, 30]



Test.py

```
value = input("Please enter the value: ")
print("Input received from the use is: ", value)
```

Output:



```
Please enter the value: [10, 20, 30]
Input received from the use is:  [10, 20, 30]

Process finished with exit code 0
```

Files in Python

A file is a named location on the disk which is used to store the data permanently.

Here are some of the operations which you can perform on files:

- open a file
- read file
- write file
- close file

1.Open a File

Python provides a built-in function called open() to open a file, and this function returns a file object called the handle and it is used to read or modify the file.

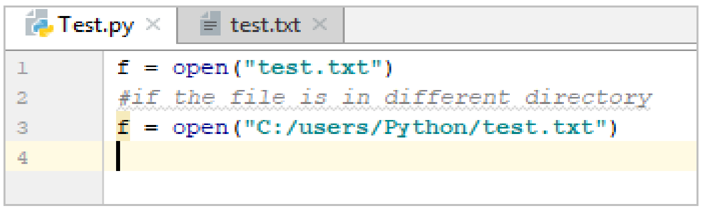
Syntax:

file_object = open(filename)

Example:

I have a file called test.txt in my disk and I want to open it. This can be achieved by:

```
#if the file is in the same directory

f = open("test.txt")

#if the file is in a different directory

f = open("C:/users/Python/test.txt")
```



```
f = open("test.txt")
#if the file is in different directory
f = open("C:/users/Python/test.txt")
```

We can even specify the mode while opening the file as if we want to read, write or append etc.

If you don't specify any mode by default, then it will be in reading mode.

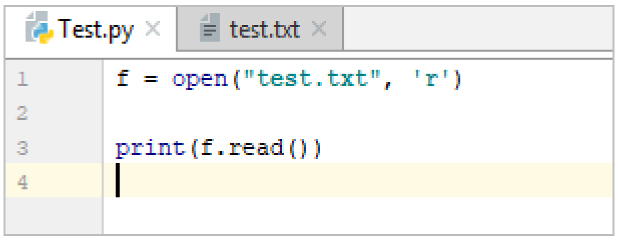
2. Reading Data from the File

In order to read the file, first, we need to open the file in reading mode.

Example:

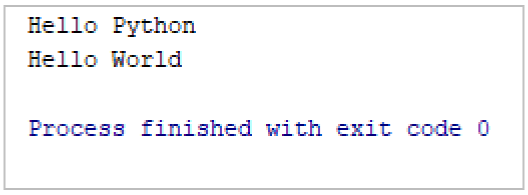
```
f = open("test.txt", 'r')


#To print the content of the whole file

print(f.read())


#To read only one line

print(f.readline())
```

Example: 1



```
f = open("test.txt", 'r')

print(f.read())
```

Output:



```
Hello Python
Hello World

Process finished with exit code 0
```

#3) Writing Data to File

In order to write the data into a file, we need to open the file in write mode.

Example:

```
f = open("test.txt", 'w')

f.write("Hello Python \n")


#in the above code '\n' is next line which means in the text file it will
write Hello Python and point the cursor to the next line

f.write("Hello World")
```

Output:

Now if we open the test.txt file, we can see the content as:

Hello Python

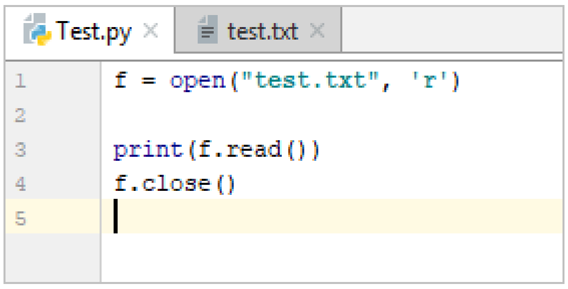
Hello World

4.Close a File

Every time when we open the file, as a good practice we need to ensure to close the file, In python, we can use close() function to close the file.

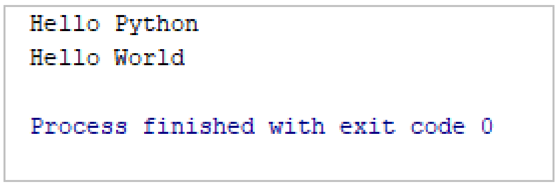
When we close the file, it will free up the resources that were tied with the file.

Input:



```
f = open("test.txt", 'r')

print(f.read())
f.close()
```

Output:



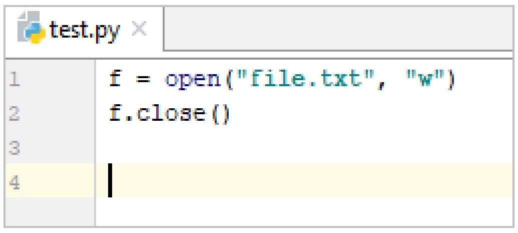
```
Hello Python
Hello World

Process finished with exit code 0
```

#5) Create & Delete a File

In python, we can create a new file using the open method.

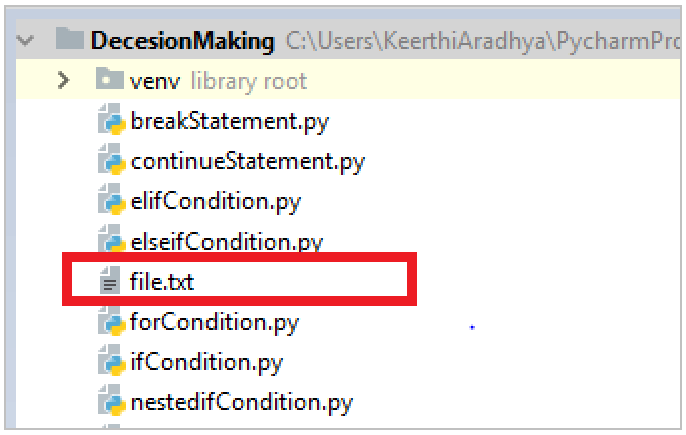
Example:

```
f = open("file.txt", "w")

f.close()
```



```
f = open("file.txt", "w")
f.close()
```

Output:



```
DecesionMaking C:\Users\KeerthiAradhya\PycharmPro
  venv library root
  breakStatement.py
  continueStatement.py
  elifCondition.py
  elseifCondition.py
  file.txt
  forCondition.py
  ifCondition.py
  nestedifCondition.py
```

Example:

```
import os

if os.path.exists("file.txt"):

os.remove("file.txt")


print("File deleted successfully")

else:

print("The file does not exist")
```

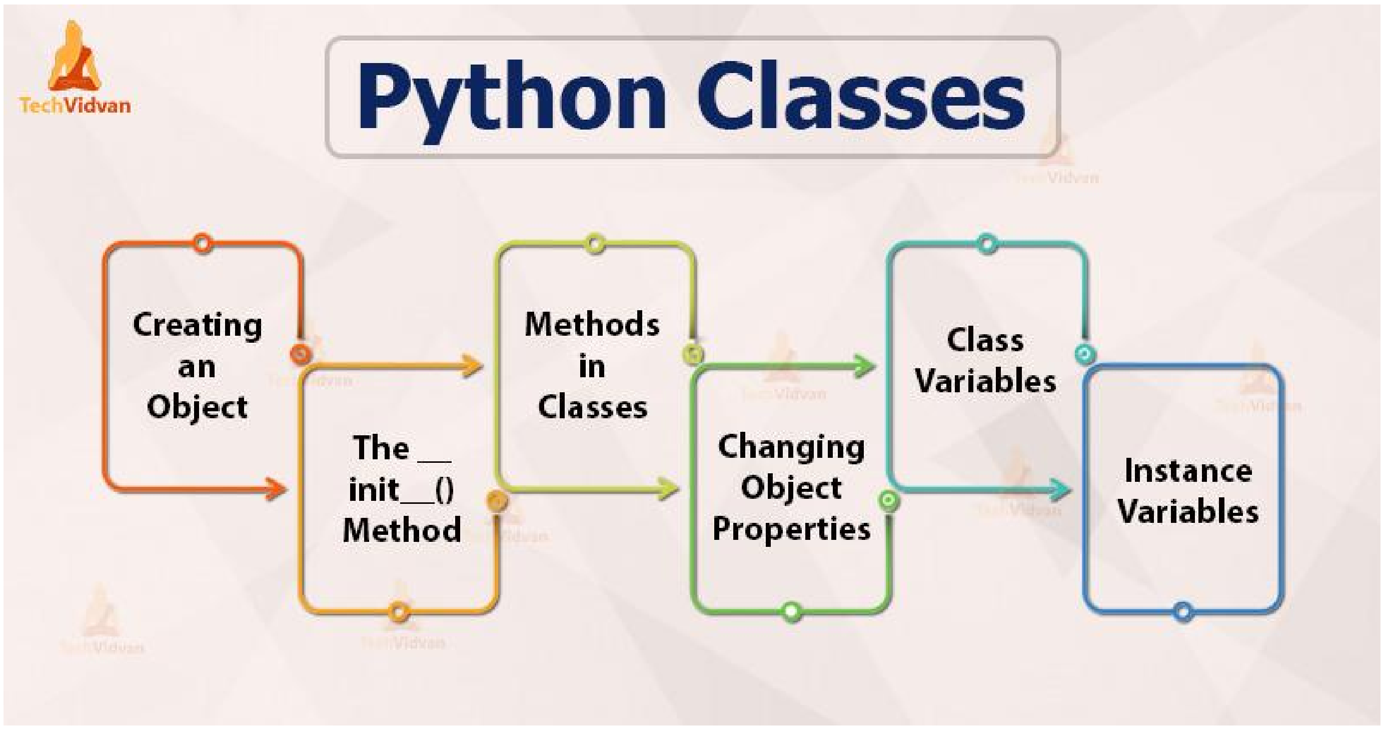
- Python Classes

Object-oriented programming (OOP) may be a programming model that organises software design around data, or objects, instead of functions and logic. An object is often defined as a knowledge field that has unique attributes and behaviour.

Python Classes

Python is an object-oriented language and everything in it is an object. By using Python classes and objects, we can model the real world.

A class is like a blueprint for objects – it has no values itself. It is an abstract data type.



We can have multiple objects of one class.

Defining a Class in Python

To define a class, we use the class keyword, not the def keyword we used to define functions.

Let’s take an example.

Code:

```
>>> class Fruit:

 pass
```

This was an empty class.

Now let’s declare a variable in it.

Code:

```
>>> class Fruit:

 name=’fruit’
```

This is a class Fruit.

We use PascalCase to name classes here. And because Python does not use curly braces, you need to indent the block of code in the class. Without that, it will raise an exception.

● Creating an Object in Python

There is no need to use the new keyword for creating objects in Python. Just use this class constructor:

Code:

```
>>> orange=Fruit()

>>> orange.name
```

Output:

```
'Fruit'
```

This calls the __init__() method in the Fruit class.

We did not declare it, but it uses the default one.

Python classes can also have docstrings to explain what they do. But if you add a docstring to a class, it should be the first thing in the class.

Code:

```
>>> class MyClass:

 '''This class does nothing'''

 pass
```

Code:

```
>>> MyClass

<class '__main__.MyClass'>
```

● The __init__() Method

So what is __init__()?

In most cases, we will need the __init__() method for classes.

This is a magic method (dunder method) which we can use to initialize values for classes (objects). So __init__() has initialization code.

Every class has __init__ and this is executed when we instantiate the class. You can also use this method to do anything you want to do when the object is being created.

Let's learn with an example.

Code:

```
>>> class Fruit:

 def __init__(self, color, size):

   self.name='fruit'

   self.color=color

   self.size=size
```

1. Accessing Values

We have created the class, now let's learn to create an object for it and access its values.

Code:

```
>>> orange=Fruit('orange',8)

>>> orange.name
```

Output:

```
'fruit'
```

Code:

```
>>> orange.color
```

Output:

```
'orange'
```

Code:

```
>>> orange.size
```

Output:

```
8
```

Here, class Fruit has an object orange. This has a name, color, and size.

The name for all fruits is name, and color and size are passed as arguments. So orange is the object with color 'orange' and size 8. We accessed the values with the dot operator.

• Python Classes Methods

Python classes can have methods and functions. Functions are simply functions. But methods act on an object and can modify it. Each method has to take the self parameter as the first parameter. It tells it to work on this object. However, you can call it anything you want.

Let's define a method in class Fruit.

Code:

```
>>> class Fruit:

 def __init__(self, color, size):

   self.name='fruit'

   self.color=color

   self.size=size

 def show(self):

   print(f'I am a {self.name}, I am {self.color} and of size {self.size}')
```

Code:

```
>>> orange=Fruit('orange',8)
```

Now, we can call the show() method on the orange object. For this, we use the dot operator.

Code:

```
>>> orange.show()
```

Output:

```
I am a fruit, I am orange and of size 8
```

You can call the self parameter anything.

Code:

```
>>> class Fruit:

 def __init__(obj, color, size):

   obj.name='fruit'

   obj.color=color
   obj.size=size
 def show(obj):
   print(f'I am a {obj.name}, I am {obj.color} and of size {obj.size}')
```

Code:

```
>>> orange=Fruit('orange',8)
>>> orange.show()
```

Output:

```
I am a fruit, I am orange and of size 8
```

You also have to use the self parameter even when a method does not take any parameters.

● Changing Object Properties

In Python classes, we use the dot operator to access object properties and methods, but we can also use it to modify them.

Code:

```
>>> orange.size=9
>>> orange.size
```

Output:

```
9
```

And we can also delete them with the del keyword.

Code:

```
>>> del orange.size

>>> orange.size
```

Output:

```
Traceback (most recent call last):

  File “<pyshell#33>”, line 1, in <module>

    orange.size

AttributeError: ‘Fruit’ object has no attribute ‘size’
```

We can also delete the complete object with the del keyword.

Code:

```
>>> del orange

>>> orange.size
```

Output:

```
Traceback (most recent call last):

  File “<pyshell#35>”, line 1, in <module>

    orange.size

NameError: name ‘orange’ is not defined
```

Python Class Variables

The Python Classes create new local namespaces with their attributes (data or functions). You can make a variable belong to a class. This is not like instance variables which belong to objects.

Code:

```
>>> class Fruit:

 name='fruit'

 def __init__(obj, color, size):

   obj.color=color

   obj.size=size

 def show(obj):

   print(f'I am a {obj.name}, I am {obj.color} and of size {obj.size}')
```

Code:

```
>>> orange=Fruit('orange',8)

>>> orange.name
```

Output:

```
‘fruit’
```

Here, the class Fruit has the class variable ‘orange’, but the orange object also has it.

1. Built-in Class Attributes

Python has some built-in class attributes that we can use to get more information about the class.

a. __dict__ – It is a dictionary consisting of the class’ namespace.

Code:

```
>>> Fruit.__dict__
```

Output:

```
mappingproxy({'__module__': '__main__', '__init__': <function Fruit.__init__ at
0x0000023450FB0048>, 'show': <function Fruit.show at 0x0000023450FB00D0>, '__dict__':
<attribute '__dict__' of 'Fruit' objects>, '__weakref__': <attribute '__weakref__' of
'Fruit' objects>, '__doc__': None})
```

b. __doc__ – It is the class docstring, but is None if there is no docstring in the class.

Code:

```
>>> Fruit.__doc__
```

Output:

```
>>>
```

c. __name__ – This is the class name.

Code:

```
>>> Fruit.__name__
```

Output:

```
'Fruit'
```

d. __module__ – This is the name of the module in which the class is defined.

In the interactive mode, this is __main__.

Code:

```
>>> Fruit.__module__
```

Output:

```
'__main__'
```

e. __bases__ – This is a tuple consisting of the base classes of this class.

Code:

```
>>> Fruit.__bases__
```

Output:

```
(<class 'object'>,)
```

2.Instance Variables in Python

In python classes, Instance variables belong to objects. We saw an example, but now we will see those other methods that __init__ can define them too.

Code:

```
>>> class Fruit:

 def __init__(obj, color):

   obj.name='fruit'

   obj.color=color

 def show(obj, size):

   obj.size=size

   print(f'I am a {obj.name}, I am {obj.color} and of size {obj.size}')
```

Code:

```
>>> orange=Fruit('orange')

>>> orange.show(8)
```

Output:

```
I am a fruit, I am orange and of size 8
```

Here, we define the size in the show() method using obj.size=size and it works like it should.

• What is Polymorphism in Python?

Polymorphism gives you the ability to represent objects of different types using a single interface.

A real-life example is You.

You act as a student when you are at college, you act like a son/daughter when you're at home, you act like a friend when you're surrounded by your friends.

Now the analogy here is, different personalities of that of a student, a son/daughter, a friend are all analogous to objects of different types.

And a single interface (i.e., you) represents all these different types(i.e. your different personalities).

● Understanding Polymorphism in Python

Python can implement polymorphism in many different ways. Python, like many other languages, also provides built-in implementations of Polymorphism.

Let’s look at the examples that illustrate built-in implementations of polymorphism in Python.

Later in this article, we’ll see the various ways in which we can implement Polymorphism in our own programs.

● Built-in implementation of Polymorphism

a. Polymorphism in ‘+’ operator

- You might have used the ‘+’ arithmetic python operator multiple times in your programs.
- And chances are, you might have used it with multiple different types.
- This right here is an implementation of polymorphism in Python.
- You use the same + symbol whether you want to add two integers, or concatenate two strings, or extend two lists.
- The + operator acts differently depending on the type of objects it is operating upon.

For integers, it performs arithmetic addition and returns an integer:

```
>>> x = 1 + 2

>>> print(x)
```

Output:

```
3


>>>
```

Whereas for strings, it concatenates them and returns a new string:

```
>>> x = "TechVidvan says " + "Hello"

>>> print(x)
```

Output:

```
'TechVidvan says Hello'

>>>
```

And for two lists, it returns a new list which contains elements of both the original lists:

```
>>> x = [1, 2, 3] + [10, 'TechVidvan']

>>> print(x)
```

Output:

```
[1, 2, 3, 10, 'TechVidvan']

>>>
```

b. Polymorphism in built-in method

- Python also implements polymorphism using methods.
- For instance, take the len() method that returns the length of an object.
- The len() method is capable of processing objects of different data types.
- Let's look at the code example below:

For a list, it returns the number of elements in the list.

```
>>> l = [1, 2, 3, 4, 5]

>>> length = len(l)
>>> print(length)
```

Output:

```
5

>>>
```

For a string, it returns the number of characters within it.

```
>>> s = "TechVidvan"

>>> length = len(s)

>>> print(length)
```

Output:

```
10

>>>
```

- **For python dictionary, it returns the number of keys.**

```
>>> d = {1: "Archie", 2: "Brady", 3: "Charlie"}

>>> length = len(d)

>>> print(length)
```

Output:

```
3

>>>
```

We have got a high-level view of what polymorphism is and how Python implements it.

● Polymorphism in user-defined methods

In the below example, we have two classes ‘Rectangle’ and ‘Square’. Both these classes have a method definition ‘area’, that calculates the area of the corresponding shapes.

```
class Rectangle:

   def __init__(self, length, breadth):

       self.l = length

       self.b = breadth

   def area(self):

       return self.l * self.b

class Square:

   def __init__(self, side):

       self.s = side

   def area(self):

       return self.s ** 2

rec = Rectangle(10, 20)

squ = Square(10)

print("Area of rectangle is: ", rec.area())

print("Area of square is: ", squ.area())
```

Output:

```
Area of rectangle is: 200

Area of square is: 100

>>>
```

Here, we implemented polymorphism using the method area(). This method works on objects of the types– Rectangle and Square. And it operates differently on objects of different classes.

We can also achieve polymorphism with inheritance.

Let’s see how.

- ## Polymorphism with Inheritance in python

A child class inherits all the attributes and methods of its parent class. But we can provide one or more methods with a different method definition within the child class.

We call this process “method overriding” and such methods “overridden” methods.

So overridden methods have the exact same external interface (method name, number and type of method parameters) as the parent class’s method but have a different internal implementation.

Let’s look at a very simple example:

```
#parent class

class Human:

   def who_am_i(self):

       print("I am a Human")

#child class

class Teacher(Human):

   def who_am_i(self):

       print("I am a Teacher")

t = Teacher()

t.who_am_i()
```

Output:

```
I am a Teacher
```

Here, we implement polymorphism by overriding the method who_am_i() and providing it with a different implementation in the child class Teacher.

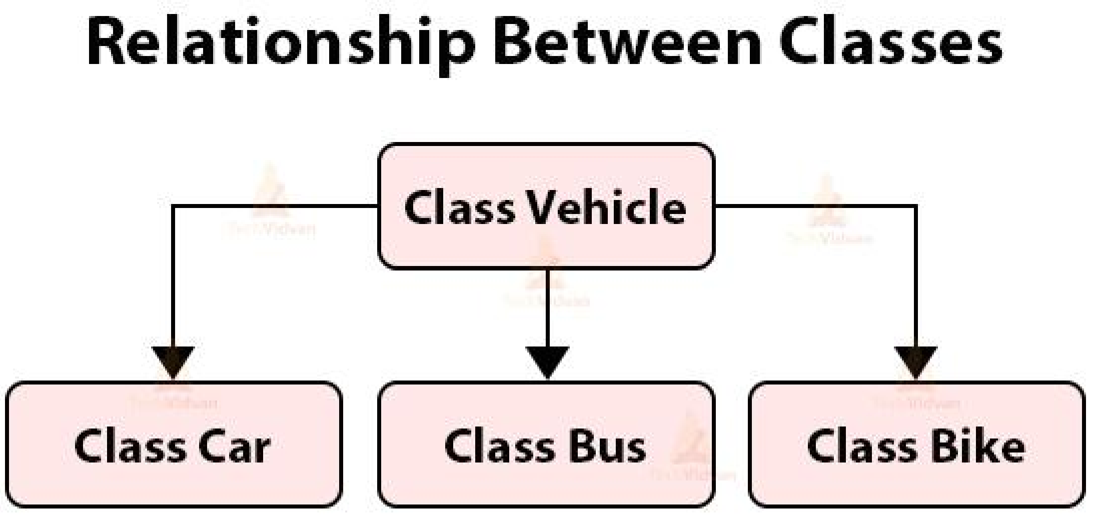
● What is Inheritance in Python?

Inheritance in object-oriented programming is inspired by the real-world inheritance in human beings. We acquire some of the traits of our parents during birth.

In Python, inheritance is the capability of a class to pass some of its properties or methods to it’s derived class(child class). With inheritance, we build a relationship between classes based on how they are derived.

For example, every car, bus, bikes are vehicles.

So we can build relationships between them and a car can inherit things from the vehicle. This can be represented as the given image.



Here, the Vehicle will be called parent or base class while the car, bus, and bike are its child or derived class.

- ## Python Inheritance Example

To derive a class from another class we can simply use the name of the parent class in parentheses after the class name.

Code:

```
class Fruit:

 pass

class Apple(Fruit):

 pass
```

Here, we have defined two classes Fruit and Apple. The Apple class is derived from the Fruit class.

The pass statement is used to create an empty class. Python has an in-built function issubclass() to check if a class is a subclass of another or not.

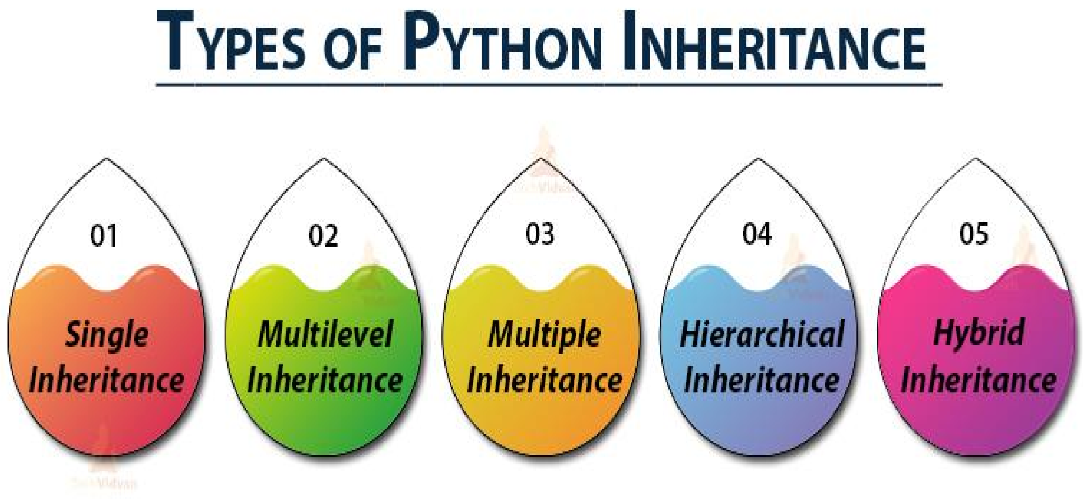
Code:

```
issubclass(Apple, Fruit)
```

Output:

```
True
```

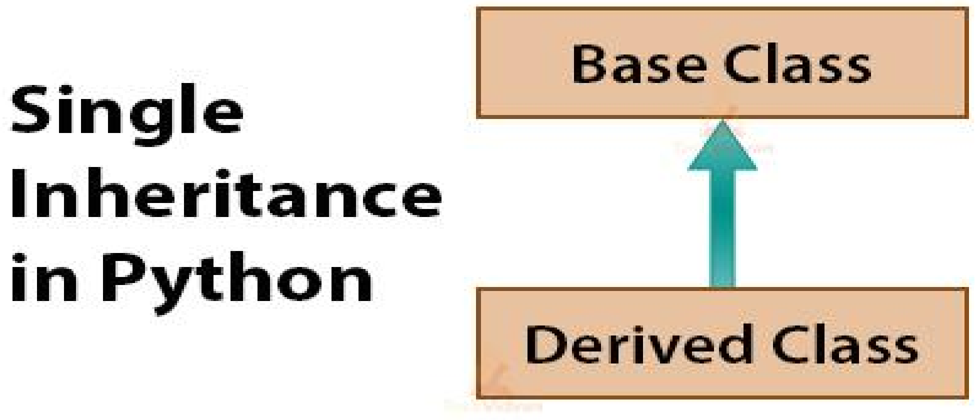
● Types of Inheritance in Python



We can build different types of relationships between classes by the way they are inherited. Python has 5 types of inheritance.

1. Single Inheritance in Python

In single inheritance, a single class inherits from a class. This is the simplest form of inheritance.



Code:

```
class Parent:

   def show(self):
        print("Parent method")

class Child(Parent):

   def display(self):

        print("Child method")

c = Child()

c.display()

c.show()
```

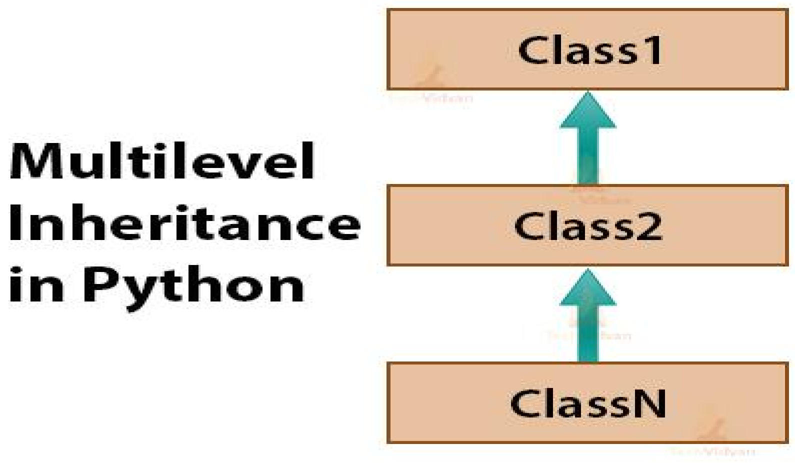
Output:

```
Child method

Parent method
```

Here, we created an object of the Child class and we saw that from the Child object we can even call the method of Parent class. This is the advantage of inheritance, we can reuse the code we have written.

2. Multilevel Inheritance in Python

Python supports multilevel inheritance, which means that there is no limit on the number of levels that you can inherit. We can achieve multilevel inheritance by



Inheriting one class from another which then is inherited from another class.

Code:

```
class A:

   def methodA(self):

       print("A class")

class B(A):

   def methodB(self):

       print("B class")

class C(B):

   def methodC(self):

       print("C class")

c = C()

c.methodA()

c.methodB()

c.methodC()
```

Output:

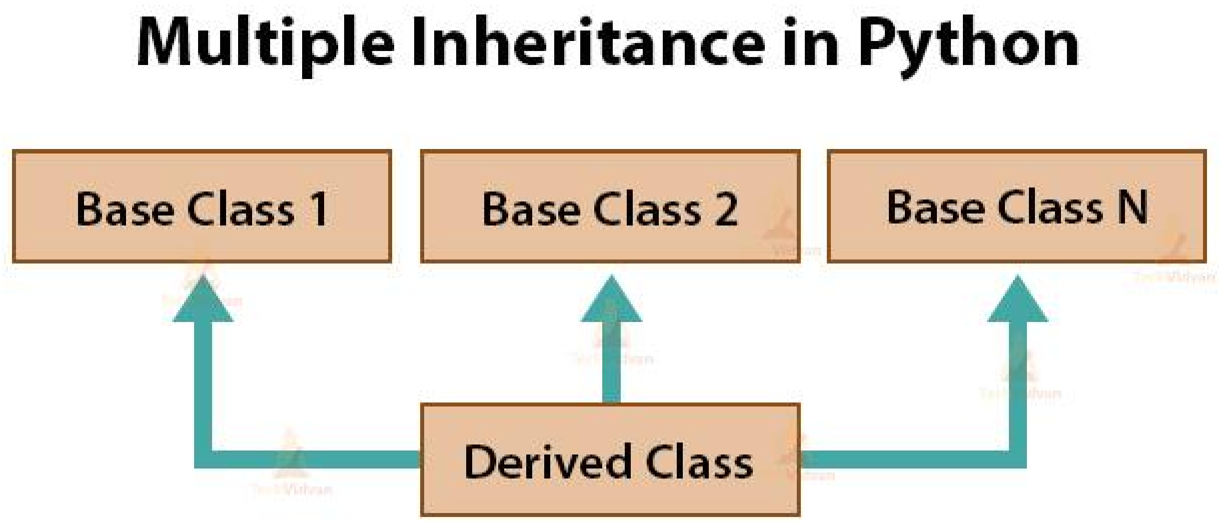
```
A class

B class

C class
```

Here, the object of C class can access the methods and properties of both A and B class because they were inherited from top to bottom. But take note that the object of a B class cannot access methods of the C class.

3. Multiple Inheritance in Python

Till now, we were inheriting from only one class at a time.

In multiple inheritance, we will see that Python also allows us to inherit from more than one class. To achieve this we can provide multiple classes separated by commas.



Code:

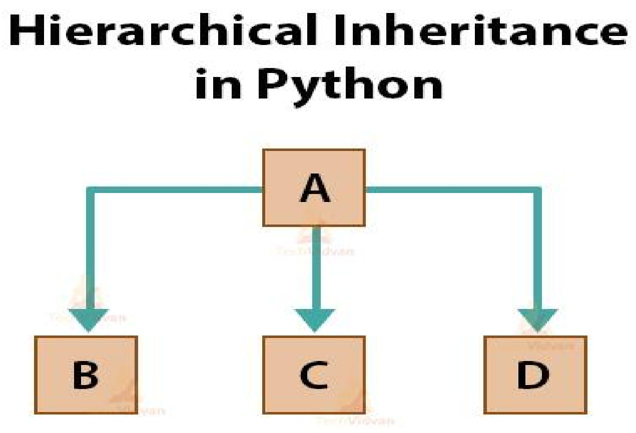
```
class A:

   def methodA(self):

       print("A class")

class B:

   def methodB(self):

       print("B class")

class C:

   def methodC(self):

       print("C class")

class D(A, B, C):
    def methodD(self):

        print("D class")

d = D()

d.methodA()

d.methodB()

d.methodC()

d.methodD()
```

Output:

```
A class

B class

C class

D class
```

The object of a D class has directly inherited the properties and methods of A, B, and C classes.

4. Hierarchical Inheritance in Python



In a hierarchical inheritance, a class is inherited by more than one class. It is simple to understand with a diagram.

Code:

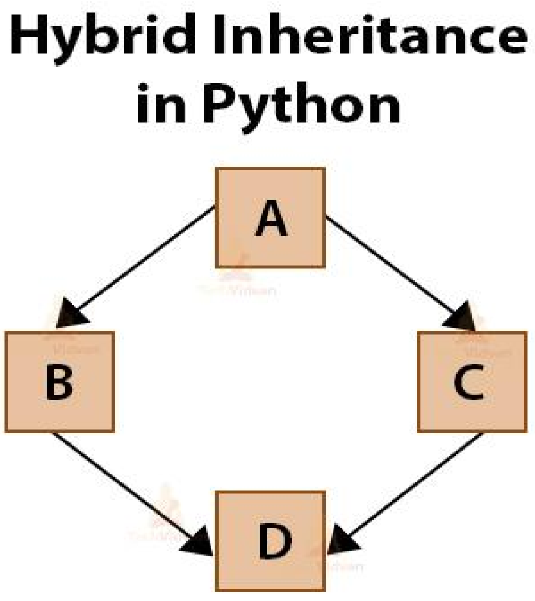
```
class A:

   def methodA(self):

       print("A class")

class B(A):

   def methodB(self):

       print("B class")

class C(A):

   def methodC(self):

       print("C class")

b = B()

c = C()

b.methodA()

c.methodA()
```

Output:

```
A class

A class
```

5. Hybrid Inheritance in Python

The term Hybrid describes that it is a mixture of more than one type. Hybrid inheritance is a combination of different types of inheritance.



$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & D & \end{tikzcd}$$

Code:

```
class A:

   def methodA(self):

       print("A class")

class B(A):

   def methodB(self):
        print("B class")

class C(A):

    def methodC(self):

        print("C class")

class D(B,C):

    def methodD(self):

        print("D class")

d = D()

d.methodA()
```

Output:

```
A class
```

- ## Python Inheritance – super() function

When dealing with inheritance, super() is a very handy function. super() is a proxy object which is used to refer to the parent object. We can call super() method to access the properties or methods of the parent class.

Code:

```
class A:

    x=100

    def methodA(self):

        print("A class")

class B(A):

    def methodB(self):
        super().methodA()

        print("B class")

        print(super().x)

b = B()

b.methodB()
```

Output:

```
A class

B class

100
```

● Python Overriding Methods

Method overriding is an important concept in object-oriented programming. Method overriding allows us to redefine a method by overriding it.

For method overriding, we must satisfy two conditions:

- There should be a parent-child relationship between the classes.
- Inheritance is a must.
- The name of the method and the parameters should be the same in the base and derived class in order to override it.

What will happen when the method in the base and derived class are the same. Let's see an example of this.

Code:

```
class A:
   def method(self):
       print("A class")
class B(A):
   def method(self):
       print("B class")
b = B()
b.method()
```

Output:

```
B class
```

Here, the B class has inherited A class and we have the same function in both classes method().

Since the name and parameters are the same, the derived class overrides the method of the base class and when we call the method() the B class method is called. This is known as method overriding.

● Overloading Methods in Python

If you have some experience with object-oriented programming or other languages like C/C++ or Java then you might have used method overloading. That is why it is important to understand that Python doesn't support method overloading.

Let's see the example –

Code:

```
def getDetails():
   print("Name: Default")
```

```
def getDetails(name):

   print("Name:", name)

getDetails("Siri")

getDetails()
```

Output:

```
Name: Siri

Traceback (most recent call last):

  File
"C:\Users\Techvidvan\AppData\Local\Programs\Python\Python38-32\test.py", line 8, in
<module>

    getDetails()

TypeError: getDetails() missing 1 required positional argument: 'name'
```

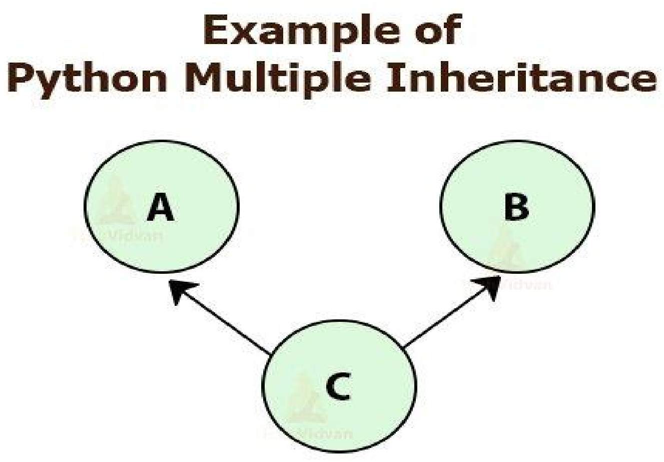
As you can see, the getDetails("Siri") function got executed but the getDetails() was not executed.

This is because Python overwrites the function with the same name and the latter function is used.

What is Multiple Inheritance in Python?

Multiple Inheritance is a type of inheritance in which one class can inherit properties(attributes and methods) of more than one parent classes.

In Multiple inheritance, there is 1 child class inheriting from more than 1 parent classes.



Here, the child class C inherits from 2 parent classes, A and B.

Let’s code it!

```
class hulk:

   def smash(self):

       return "I smash"

class banner:

   def speak(self):

       return "I've got the brains!"

class smarthulk(hulk, banner):

   pass

s1 = smarthulk()

print(s1.smash(), "and", s1.speak())
```

Output:

```
I smash and I’ve got the brain!
```

Regular Expressions in Python:

The term Regular Expression is popularly shortened as regex. A regex is a sequence of characters that defines a search pattern, used mainly for performing find and replace operations in search engines and text processors.

Python offers regex capabilities through the re module bundled as a part of the standard library.

- ## Raw strings

Different functions in Python's re module use raw string as an argument. A normal string, when prefixed with 'r' or 'R' becomes a raw string.

Example: Raw String Copy

```
>>> rawstr = r'Hello! How are you?'
>>> print(rawstr)
Hello! How are you?
```

The difference between a normal string and a raw string is that the normal string in print() function translates escape characters (such as \n, \t etc.) if any, while those in a raw string are not.

Example: String vs Raw String Copy

```
str1 = "Hello!\nHow are you?"
print("normal string:", str1)
str2 = r"Hello!\nHow are you?"
print("raw string:",str2)
```

Output

```
normal string: Hello! How are you? raw string: Hello!\nHow are you?
```

In the above example, \n inside str1 (normal string) has translated as a newline being printed in the next line. But, it is printed as \n in str2 - a raw string.

● Meta characters

Some characters carry a special meaning when they appear as a part pattern matching string. In Windows or Linux DOS commands, we use * and ? - they are similar to meta characters. Python's re module uses the following characters as meta characters:

. ^ $ * + ? [ ] \ | ( )

When a set of alpha-numeric characters are placed inside square brackets [], the target string is matched with these characters. A range of characters or individual characters can be listed in the square bracket. For example:

Pattern	Description
[abc]	match any of the characters a, b, or c
[a-c]	which uses a range to express the same set of characters.
[a-z]	match only lowercase letters.
[0-9]	match only digits.

The following specific characters carry certain specific meaning.

Pattern	Description
\d	Matches any decimal digit; this is equivalent to the class [0-9].
\D	Matches any non-digit character
\s	Matches any whitespace character
\S	Matches any non-whitespace character

\w	Matches any alphanumeric character
\W	Matches any non-alphanumeric character.
.	Matches with any single character except newline '\n'.
?	match 0 or 1 occurrence of the pattern to its left
+	1 or more occurrences of the pattern to its left
*	0 or more occurrences of the pattern to its left
\b	boundary between word and non-word. /B is opposite of /b
[..]	Matches any single character in a square bracket
\	It is used for special meaning characters like . to match a period or + for plus sign.
{n,m}	Matches at least n and at most m occurrences of preceding
a| b	Matches either a or b

● re.match() function

This function in re module tries to find if the specified pattern is present at the beginning of the given string.

re.match(pattern, string)

The function returns None, if the given pattern is not in the beginning, and a match objects if found.

Example: re.match() Copy

```
from re import match

mystr = "Welcome to TutorialsTeacher"
obj1 = match("We", mystr)
print(obj1)
obj2 = match("teacher", mystr)
print(obj2)
```

Output

```
<re.Match object; span=(0, 2), match='We'> None
```

The match object has start and end properties.

Example: Copy

```
>>> print("start:", obj.start(), "end:", obj.end())
```

Output

```
start: 0 end: 2
```

The following example demonstrates the use of the range of characters to find out if a string starts with 'W' and is followed by an alphabet.

Example: match() Copy

```
from re import match

strings=["Welcome to TutorialsTeacher", "weather
forecast","Winston Churchill", "W.G.Grace","Wonders of
India", "Water park"]

for string in strings:
    obj = match("W[a-z]",string)
    print(obj)
```

Output
```
<re.Match object; span=(0, 2), match='We'> None <re.Match object; span=(0, 2), match='Wi'> None <re.Match object; span=(0, 2), match='Wo'> <re.Match object; span=(0, 2), match='Wa'>
```

● re.search() function

The `re.search()` function searches for a specified pattern anywhere in the given string and stops the search on the first occurrence.

Example: re.search() Copy
```
from re import search

string = "Try to earn while you learn"

obj = search("earn", string)
print(obj)
print(obj.start(), obj.end(), obj.group())
7 11 earn
```

Output
```
<re.Match object; span=(7, 11), match='earn'>
```

This function also returns the Match object with start and end attributes. It also gives a group of characters of which the pattern is a part of.

● re.findall() Function

As against the search() function, the findall() continues to search for the pattern till the target string is exhausted. The object returns a list of all occurrences.

Example: re.findall() Copy
```
from re import findall
```

```
string = "Try to earn while you learn"

obj = findall("earn", string)
print(obj)
```

Output

```
['earn', 'earn']
```

This function can be used to get the list of words in a sentence. We shall use \W* pattern for the purpose. We also check which of the words do not have any vowels in them.

Example: re.findall() Copy

```
obj = findall(r"\w*", "Fly in the sky.")
print(obj)

for word in obj:
    obj= search(r"[aeiou]",word)
    if word!='' and obj==None:
        print(word)
```

Output

```
['Fly', '', 'in', '', 'the', '', 'sky', '', ''] Fly sky
```

• re.finditer() function

The re.finditer() function returns an iterator object of all matches in the target string. For each matched group, start and end positions can be obtained by span() attribute.

Example: re.finditer() Copy

```
from re import finditer

string = "Try to earn while you learn"
it = finditer("earn", string)
for match in it:
    print(match.span())
```

Output
```
(7, 11) (23, 27)
```

● re.split() function

The re.split() function works similar to the split() method of str object in Python. It splits the given string every time a white space is found. In the above example of the findall() to get all words, the list also contains each occurrence of white space as a word. That is eliminated by the split() function in re module.

Example: re.split() Copy
```
from re import split

string = "Flat is better than nested. Sparse is better
than dense."
words = split(r' ', string)
print(words)
```

Output
```
['Flat', 'is', 'better', 'than', 'nested.', 'Sparse', 'is', 'better',
'than', 'dense.']
```

● re.compile() Function

The re.compile() function returns a pattern object which can be repeatedly used in different regex functions. In the following example, a string 'is' is compiled to get a pattern object and is subjected to the search() method.

Example:
re.compile() Copy
```
from re import *

pattern = compile(r'[aeiou]')
```

```
string = "Flat is better than nested. Sparse is better than dense."
words = split(r' ', string)
for word in words:
    print(word, pattern.match(word))
```

Output

```
Flat None is <re.Match object; span=(0, 1), match='i'> better None than None nested. None Sparse None is <re.Match object; span=(0, 1), match='i'> better None than None dense. None
```

The same pattern object can be reused in searching for words having vowels, as shown below.

Example:

search() Copy

```
for word in words:
    print(word, pattern.search(word))
```

Output

```
Flat <re.Match object; span=(2, 3), match='a'> is <re.Match object; span=(0, 1), match='i'> better <re.Match object; span=(1, 2), match='e'> than <re.Match object; span=(2, 3), match='a'> nested. <re.Match object; span=(1, 2), match='e'> Sparse <re.Match object; span=(2, 3), match='a'> is <re.Match object; span=(0, 1), match='i'> better <re.Match object; span=(1, 2), match='e'> than <re.Match object; span=(2, 3), match='a'> dense. <re.Match object; span=(1, 2), match='e'>
```

• Quantifiers in Python:

A quantifier has the form {m,n} where m and n are the minimum and maximum times the expression to which the quantifier applies must match. We can use quantifiers to specify the number of occurrences to match.