Class: 5th semester Computer Engg.
(Subject: Python Programming)

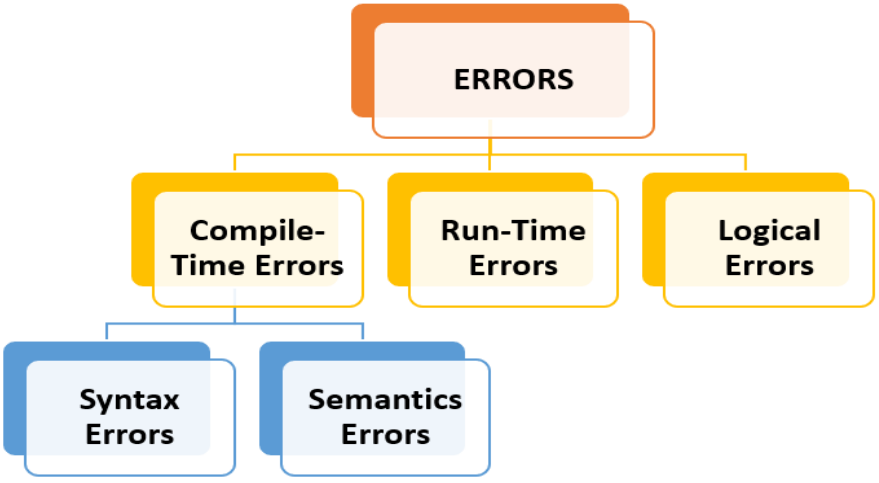
UNIT-1

Debugging

Debugging means the complete control over the program execution. Developers use debugging to overcome program from any bad issues. So debugging is a healthier process for the program and keeps the bugs far away. Python also allows developers to debug the programs using pdb module that comes with standard Python by default. We just need to import pdb module in the Python script. Using pdb module, we can set breakpoints in the program to check the current status. We can Change the flow of execution by using jump, continue statements

Errors

Errors are problems in a program that causes the program to stop its execution. On the other hand, exceptions are raised when some internal events change the program's normal flow. There are 3 types of errors.



1) Compile-Time Errors

As the name suggests, the errors that occurs while compilation or at the compile-time are called compile-time errors.

Now, you must be wondering, how errors occur at compile time?

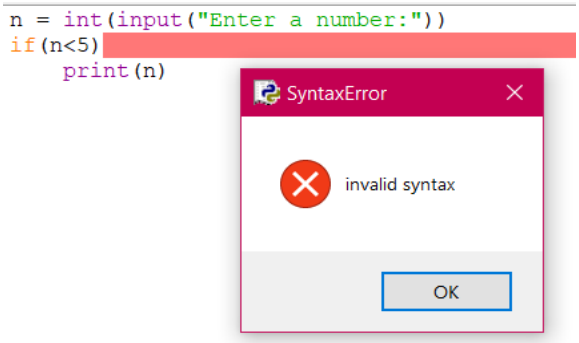
Answer to the question is; At the time of compilation, the system checks the source code. Now if there's any fault or violation of programming convention or you can say violation of language's rules then compilation stops and system gives us compilation error.

There are 2 types of compile-time errors.

1.1) Syntax Errors

Syntax refers to the formal rules those are used for writing valid statements in a programming language. You can understand it as a structure of the code and conventions that you have to follow to construct that structure.

For example,

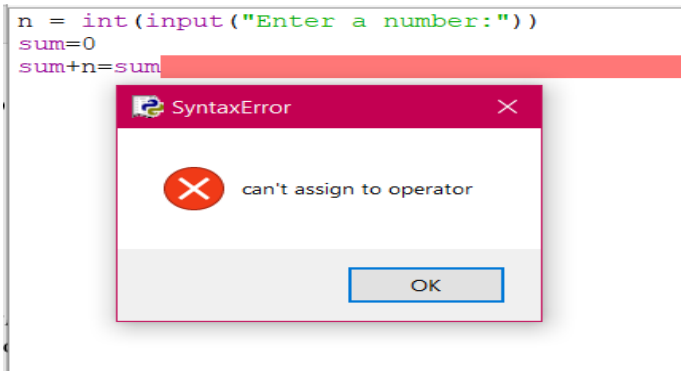


In the above example, we can see a syntax error. We know that in Python; the syntax for **if** statement is:
if(condition):
In the example ":" was missing, hence the compilation stopped and there was an error.

The correct code would be:

```
n = int(input("Enter a number:"))
if(n<5):
  print(n)
```

1.2) Semantics Errors
Semantics refers to the meaning of a statement. So, when a statement doesn't make any sense and is not meaningful then we can say it is a Semantic error.
For example**,**



In the above example, we can see a syntax error but actually it is a semantic error, because the statement:
sum+n=sum
doesn't make any sense. This is because an expression cannot come on the left side of an assignment according to Python language's rule.

The correct statement would be: -
sum= sum+n

```
n = int(input("Enter a number:"))
sum=0
sum= sum+n
```

2) Run-Time Errors
This type of errors occur after the compilation of the code; in-between when execution is going on. Due to when program is "crashed" or "abnormally abrupted".
These are also known as "Bugs" and are found during the process of debugging.
For example:

- Infinite loop

- Wrong value as Input
- Invalid function call
- Divide by Zero

3. Logical Errors

In many cases, programmers get demented and are not able to find the exact error. This is because errors are seen while compiling or at run-time. Sometimes programmer's analysis of the problem is wrong, in those cases the errors are logical errors.

For example,

- Incorrect implementation of an algorithm
- Unmarked end of loop
- Wrong parameters passed

Sometimes logical errors are treated as a subcategory of run-time errors. These types of errors are hardest to prevent and locate.

For Instance

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))
z = x+y/2
print ('The result is:',z)
```

Output

```
Enter a number: 10
Enter a number: 5
The result is: 12.5
```

This output seems fine, but if you notice closely, the actual output should be 7.5 (calculating average) and here it is 12.5. This is because it divides y/2 first and then moves to add with x. So it is computing 5/2= 2.5 first and then moving on to x+2.5=10+2.5=12.5. Hence, output 12.5

To get the desired output, we just have to add parenthesis.

```
z =( x+y)/2
```

Now the output would be,

```
Enter a number: 10
Enter a number: 5
The result is: 7.5
```

So by inserting parenthesis, it computes x+y first i.e 10+5=15, and then computes 15/2= 7.5. Hence, output 7.5

EXAMPLES OF ERRORS

a) Index Error

```
list_example= [10, 20, 30]
print(list_example[3])
```

Output

```
Traceback (most recent call last):
  File "C:\Users\Dell\Desktop\test.py", line 2, in <module>
    print(list_example[3])
IndexError: list index out of range
```

b) Module Error

```
import myModule
```

```
list_example= [10,20,30]
print(list_example[3])
```

Output

```
Traceback (most recent call last):
  File "C:\Users\Dell\Desktop\test.py", line 1, in <module>
    import myMdule
ModuleNotFoundError: No module named 'myMdule'
```

c) Value Error

```
my_value= int('abc')
print(my_value)
```

Output

```
Traceback (most recent call last):
  File "C:\Users\Dell\Desktop\test.py", line 1, in <module>
    my_value= int('abc')
ValueError: invalid literal for int() with base 10: 'abc'
```

d) DivideByZero Error

```
my_value= 10/0
print(my_value)
```

Output

```
Traceback (most recent call last):
  File "C:\Users\Dell\Desktop\test.py", line 1, in <module>
    my_value= 10/0
ZeroDivisionError: division by zero
```

e) Name Error

```
my_value= 10/5
print(age)
```

Output

```
Traceback (most recent call last):
  File "C:\Users\Dell\Desktop\test.py", line 2, in <module>
    print(age)
NameError: name 'age' is not defined
```

INTRODUCTION:

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. 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 C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Advantages of Python:

Some of the key advantages of learning Python are:

- **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.

Characteristics of Python:

- 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 supports automatic garbage collection.
- It can be easily integrated with C, C++, ActiveX, CORBA, and Java.

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 eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **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, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Python Identifiers:

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.

Reserved Words:

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	Exec	Not
assert	Finally	Or
break	For	Pass
class	From	Print
continue	Global	Raise
def	If	Return
del	Import	Try
elif	In	While
else	Is	With

Lines and Indentation:

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code. For example:

```
if 5 > 2:
   print("Five is greater than two!")
```

Python will give you an error if you skip the indentation. For example:

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you, but it has to be at least one. For example:

```
if 5 > 2:
   print("Five is greater than two!")
if 5 > 2:
      print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error. For example:

```
if 5 > 2:
  print("Five is greater than two!")
    print("Five is greater than two!")
```

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

```
if True:
  print "True"
else:
  print "False"
```

However, the following block generates an error −

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block.

Multi-Line Statements:

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
      item_two + \
      item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday',
      'Thursday', 'Friday']
```

IDLE:

It is Python's **Integrated Development and Learning Environment**. It has following features:

- coded in 100% pure Python, using the tkinter GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- Provides text editor options with multiple features like colorizing etc.

Quotation in Python:

Python accepts single ('), double (") and triple ('" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = ‘’’
This is a paragraph. It is
made up of multiple lines and sentences.’"
```

Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
# First comment
print "Hello, Python!"   # second comment
```

This produces the following result −

Hello, Python!

You can type a comment on the same line after a statement or expression −

```
name = "Madisetti"      # This is again comment
```

Comments does not have to be text to explain the code, it can also be used to prevent Python from executing code.

You can comment multiple lines as follows −

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as multiline comments:

```
'''
This is a multiline
comment.
'''
```

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon −

```
import sys;   x = 'foo';    sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example −

```
if expression :
   suite
elif expression :
   suite
```

```
else :
   suite
```

Variable Names

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 −

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
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
```

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are ***case-sensitive (age, Age and AGE are three different variables)***

```
#Legal variable names:
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"

#Illegal variable names:
2myvar = "John"
```

```
my-var = "John"
my var = "John"
```

Assign Value to Multiple Variables:

Python allows you to assign values to multiple variables in one line. e.g.

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

And you can assign the *same* value to multiple variables in one line.e.g.

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Output Variables:

The Python print statement is often used to output variables.

To combine both text and a variable, Python uses the + character:

```
x = "awesome"
print("Python is " + x)
```

You can also use the + character to add a variable to another variable:

```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```

For numbers, the + character works as a mathematical operator:

```
x = 5
y = 10
print(x + y)
```

If you try to combine a string and a number, **Python will give you an error:**

```
x = 5
y = "John"
print(x + y)
```

Global Variables:

Variables that are created outside of a function (as in all of the examples above) are known as global variables. Global variables can be used by everyone, both inside of functions and outside.

Create a variable outside of a function, and use it inside the function

```
x = "awesome"
def myfunc():
  print("Python is " + x)
myfunc()
```

(Result – Python is awesome)

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value. e.g.

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"   # Global variable
def myfunc():
  x = "fantastic"     # Local variable
  print("Python is " + x)
myfunc()       # Function calling
print("Python is " + x)
```

(Result – Python is fantastic
Python is awesome)

The global Keyword:

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function. To create a global variable inside a function, you can use the global keyword.

```
def myfunc():
  global x
  x = "fantastic"
myfunc()
print("Python is " + x)
```

(Result – Python is fantastic)

Also, use the global keyword if you want to change a global variable inside a function.

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = "awesome"   # Global variable
def myfunc():
  global x
  x = "fantastic"

myfunc()
print("Python is " + x)
```

(Result – Python is fantastic)

Built-in Data Types:

Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Text Type: Str

Numeric Types: int, float, complex

Sequence Types: list, tuple, range

Mapping Type: Dict

Set Types: set, frozenset

Boolean Type: Bool

Binary Types: bytes, bytearray

Getting the Data Type:

You can get the data type of any object by using the type() function:

Print the data type of the variable x:

```
x = 5
print(type(x))
```

(Result – class 'int')

Setting the Data Type:

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray

Python Casting:

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Integers:

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Floats:

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

Built-in Data Types:

Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Text Type:	Str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray

Getting the Data Type:

You can get the data type of any object by using the type() function:

Print the data type of the variable x:

```
x = 5
print(type(x))
```

(Result – class 'int')

Setting the Data Type:

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray

Python Casting:

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Integers:

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Floats:

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

UNIT – 2

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 -

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

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

```
del var
del var_a, var_b
```

Python supports four different numeric 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 −

Int	Long	float	complex
10	51924361L	0.0	3.14j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

- A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

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 −

```
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
```

This will produce the following result −

```
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 −

```
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 −

```
['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']

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 −

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

print tuple              # Prints the complete tuple
print tuple[0]           # Prints first element of the tuple
print tuple[1:3]         # Prints elements of the tuple starting from 2nd till 3rd
print tuple[2:]          # Prints elements of the tuple starting from 3rd element
print tinytuple * 2      # Prints the contents of the tuple twice
print tuple + tinytuple  # Prints concatenated tuples
```

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 −

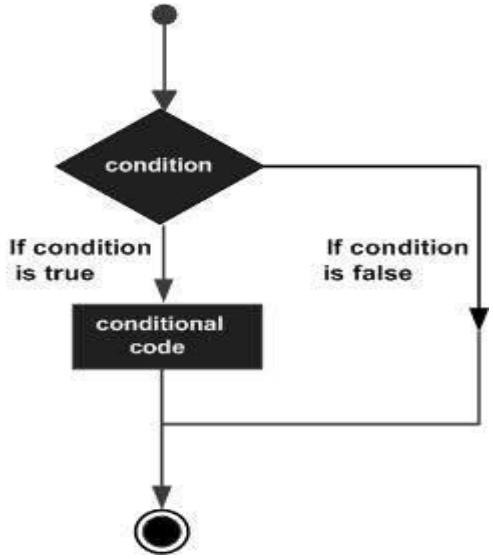
```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Decision Making Statements

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.

Following is the general form of a typical decision making structure found in most of the programming languages −



Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

An "if statement" is written by using the if keyword.

Example- If statement:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example - If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

The elif keyword is python's way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 23
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

Else

The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line. This technique is known as **Ternary**

Operators or Conditional Expression.

Example - One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example - Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or

The or keyword is a logical operator, and is used to combine conditional statements:

Example - Test if a is greater than b, OR if a is greater than c:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Nested If

You can have if statements inside if statements, this is called *nested* if statements.

```
x = 11
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

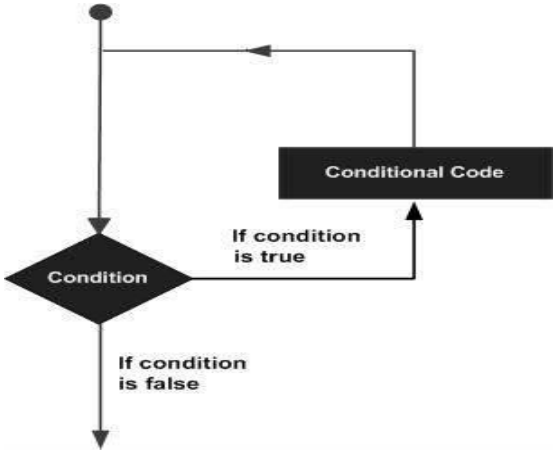
The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 33
b = 200
if b > a:
  pass
```

WHILE LOOP

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 number of times. A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −

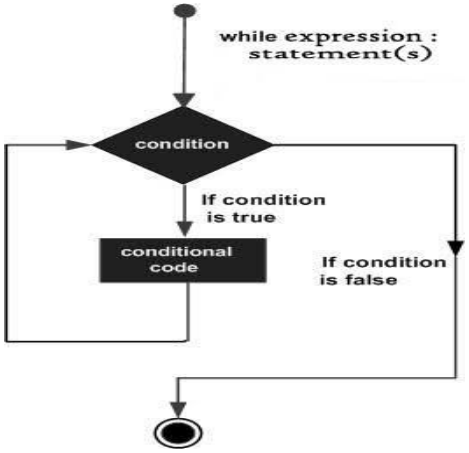


Python programming language provides various loops to handle looping requirements.

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true. The syntax of a **while** loop in Python programming language is −

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

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.



In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. **Python uses indentation as its method of grouping statements**.

```
count = 0
while (count < 4):
   print 'The count is:', count
   count = count + 1

print "Good bye!"
```

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

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 4. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
var = 1
while var == 1 :  # This constructs an infinite loop
   num = input("Enter a number  :")
   print "You entered: ", num

print "Good bye!"
```

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

```
Enter a number  :20
You entered:  20
Enter a number  :3
You entered:  3
Enter a number between :Traceback (most recent call last):
   File "test.py", line 5, in <module>
      num = raw_input("Enter a number :")
KeyboardInterrupt
```

Above example goes in an infinite loop and you need to use CTRL+C to exit the program.

Using else Statement with While Loop

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

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

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 3, otherwise else statement gets executed.

```
count = 0
while count < 3:
   print count, " is  less than 3"
   count = count + 1
else:
   print count, " is not less than 3"
```

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

```
0 is less than 3
```

1 is less than 3
2 is less than 3
3 is not less than 3

Single Statement Suites

Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Here is the syntax and example of a **one-line while** clause −

```
flag = 1
while (flag): print 'Given flag is really true!'
print "Good bye!"
```

It is better not try above example because it goes into infinite loop and you need to press CTRL+C keys to exit.

Nested while loop

The syntax for a **nested while loop** statement in Python programming language is as follows −

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

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

Example: Following program uses a nested for loop to find the prime numbers

```
a=61
while(a < 81):
   k=0
   i = 2
   while(i <= (a/i)):
      if (a%i==0):
         k=k+1
         break
      i = i + 1
   if (k==0):
       print(a, 'is prime')
a =a + 1

print "Good bye!"
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
Good bye!
```

FOR LOOP

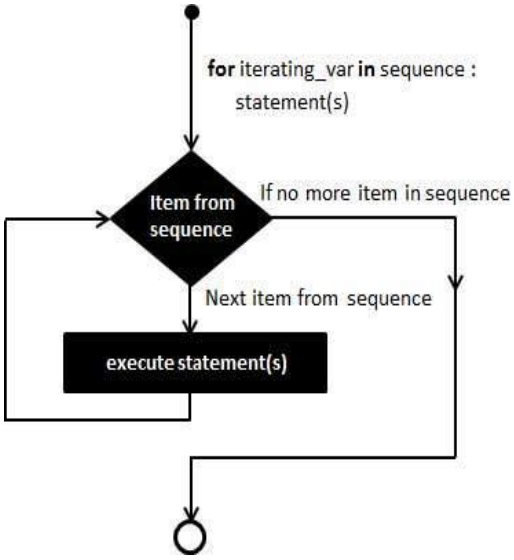
It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

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

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



Example:

```
for letter in 'Python':     # First Example
  print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for i in fruits:        # Second Example
  print 'Current fruit :', i

print "Good bye!"
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index:

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example −

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print 'Current fruit :', fruits[index]

print "Good bye!"
```

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

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Using else Statement with For Loop:

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

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 17.

```
for num in range(10,18):     #to iterate between 10 to 17
   for i in range(2,num/2+1):    #to iterate on the factors of the number
      if num%i == 0:         #to determine the first factor
         j=num/i            #to calculate the second factor
         print '%d equals %d * %d' % (num,i,j)
         break #to move to the next number, the #first FOR
   else:                # else part of the loop
      print num, 'is a prime number'
                        break
```

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

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example: Using the range() function

```
for x in range(6):
  print(x)
```

Output is: 0 1 2 3 4 5

Note that range(6) is not the values of 0 to 6, but the values 0 to 5

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

Example: Using the start parameter:

```
for x in range(2, 6):
  print(x)
```

Output is: 2 3 4 5

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example: Increment the sequence with 3 (default is 1):

```
for x in range(2, 20, 3):
  print(x)
```

Output is: 2 5 8 11 14 17

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 3, and print a message when the loop has ended

```
for x in range(4):
  print(x)
else:
  print("Finally finished!")
```

Output is: 0 1 2 3 Finally finished!

Nested Loops

A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop":

Example: Print each adjective for every fruit

```
adj = ["red", "big"]
fruits = ["apple", "banana", ”Mango”]

for x in adj:
```

```
  for y in fruits:
    print(x, y)
```

Output is:
red apple
red banana
red Mango
big apple
big banana
big Mango

The Collatz 3n+1 sequence

The **Collatz conjecture** is one of the most famous unsolved problems in mathematics. It concerns sequences of integers in which each term is obtained from the previous term as follows:

a) if the previous term is even, the next term is one half of the previous term.
b) If the previous term is odd, the next term is 3 times the previous term plus 1.

The conjecture is that these sequences always reach 1, no matter which positive integer is chosen to start the sequence.

String Comparison in Python
String comparison is a fundamental operation in any programming language, including Python. It enables us to ascertain strings' relative positions, ordering, and equality. Python has a number of operators and techniques for comparing strings, each with a specific function.

String Comparison in Python u**sing is Operator**
The == operator compares the values of both operands and checks for value equality. Whereas **is** operator checks whether both the operands refer to the same object or not. The same is the case for **!=** and **is not**.

What is the String find() Method?

String find() is an in-built function in Python that is used to find the index of a substring within a given string.
It is a very easy and useful string function, that can help us find a given substring. It returns the index of the substring if it is found in the string, but if the substring is not present in the string it returns -1.

How to use the string find() method?

String.find() method returns the index of a substring in a string and using the find() function is very easy. You just need to call the find function with the object string and pass the substring as a parameter.

Python String split()
Python String split() method splits a string into a list of strings after breaking the given string by the specified separator.

Using the list split() method is very easy, just call the split() function with a string object and pass the separator as a parameter. We can use Python split() function to split different Strings into a list, separated by different characters in each case.

Break Statement

With the break statement we can stop the loop even if the while condition is true:

Example: Exit the loop when i is 3

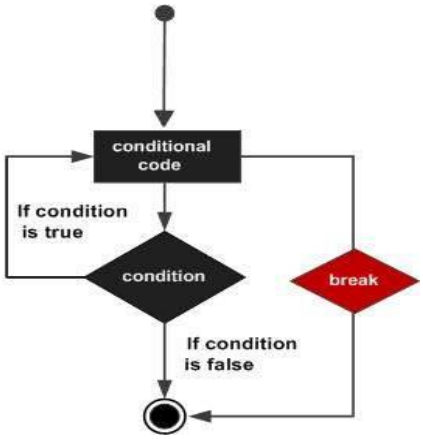
```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

Output is:
1 2 3

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

If you are using nested loops, the break statement **stops the execution of the innermost loop and start executing the next line of code after the block**.



Example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

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

```
Current Letter : P
Current Letter : y
```

```
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

Continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

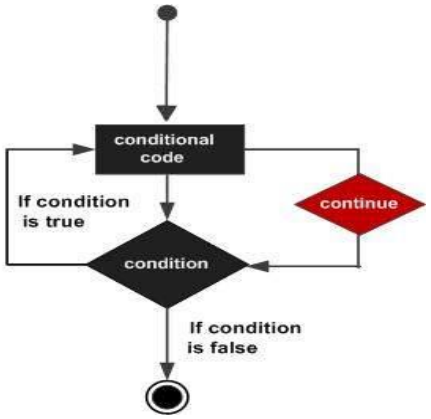
Example: Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Output is: # Note that number 3 is missing in the result

```
1       2       4       5       6
```

It returns the control to the beginning of the while loop. **The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop**. The **continue** statement can be used in both *while* and *for* loops.




```
for letter in 'Python':     # First Example
  if letter == 'h':
     continue
  print 'Current Letter :', letter

var = 10                    # Second Example
while var > 0:
  var = var -1
  if var == 5:
     continue
  print 'Current variable value :', var
print "Good bye!"
```

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

```
Current Letter : P
```

```
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute. **The pass statement is a *null* operation; nothing happens when it executes**.

```
for letter in 'Python':
  if letter == 'h':
     pass
     print 'This is pass block'
  print 'Current Letter :', letter

print "Good bye!"
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Python Collections (Arrays)

There are 4 collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered and changeable. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

LIST:

Lists are used to store multiple items in a single variable. Lists are created using square brackets:

Example: Create a List

```
x = ["apple", "banana", "cherry"]
print(x)
```

Result: [‘apple’, ‘banana’, ‘cherry’]

a) List Items: List items are ordered, changeable, and allow duplicate values. List items are indexed, the first item has index [0], the second item has index [1] etc.

b) Ordered: When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

c) Changeable: The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

d) Allow Duplicates: Since lists are indexed, lists can have items with the same value.

Example: Lists allow duplicate values

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

Result: [‘apple’, ‘banana’, ‘cherry’, ‘apple’, ‘cherry’]

1. List Length: To determine how many items a list has, use the len() function.

Example: Print the number of items in the list

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

Result: 3

2. List Items - Data Types: List items can be of any data type.

Example: String, int and boolean data types

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types.

Example: A list with strings, integers and boolean values.

```
list1 = ["abc", 34, True, 40, "male"]
```

3. type(): From Python's perspective, lists are defined as objects with the data type 'list'.

Example: What is the data type of a list?

```
x = ["apple", "banana", "cherry"]
print(type(x))
```

Result: class 'list'

4. The list() Constructor: It is also possible to use the list() constructor when creating a new list.

Example: Using the list() constructor to make a list

```
x = list(("apple", "banana", "cherry")) # note the double round-brackets
print(x)
```

Result: ['apple', 'banana', 'cherry']

ACCESSING THE LIST ITEMS

List items are indexed and you can access them by referring to the index number.

Example: Print the second item of the list.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Result: banana

Note: The first item has index 0.

1. Negative Indexing: Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

Example: Print the last item of the list.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Result: cherry

2. Range of Indexes: You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.

Example: Return the third, fourth and fifth item.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

Result: ['cherry', 'orange', 'kiwi']

Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0. By leaving out the start value, the range will start at the first item:

Example: This returns the items from the beginning to, but NOT included, "kiwi".

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

Result: [‘apple’, ‘banana’, ‘cherry’, ‘orange’]

This will return the items from index 0 to index 4. Remember that index 0 is the first item, and index 4 is the fifth item. Remember that the item in index 4 is NOT included

By leaving out the end value, the range will go on to the end of the list.

Example: This example returns the items from “cherry” and to the end.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

Result: [‘cherry’, ‘orange’, ‘kiwi’, ‘melon’,’mango’]

This will return the items from index 2 to the end. Remember that index 0 is the first item, and index 2 is the third.

3. Range of Negative Indexes: Specify negative indexes if you want to start the search from the end of the list.

Example: This example returns the items from “orange”(-4) to, but NOT included “mango”(-1).

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

Result: [‘orange’, ‘kiwi’, ‘melon’]

Negative indexing means starting from the end of the list. This example returns the items from index -4 (included) to index -1 (excluded). Remember that the last item has the index -1,

4. Check if Item Exists: To determine if a specified item is present in a list use the in keyword.

Example: Check if “apple” is present in the list.

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Result: Yes, ‘apple’ is in the fruits list

CHANGE/ADD/REMOVE LIST ITEMS

1. Change Item Value: To change the value of a specific item, refer to the index number.

Example 1: Change the second item.

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Result: [‘apple’, ‘blackcurrant’, ‘cherry’]

To insert more than one item, create a list with the new values, and specify the index number where you want the new values to be inserted.

Example 2: Change the second value by replacing it with *two* new values.

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = ["blackcurrant", "watermelon"]
print(thislist)
```

Result: ['apple', ['blackcurrant', 'watermelon'], 'cherry']

Note: The length of the list will change when the number of items inserted does not match the number of items replaced.

2. Change a Range of Item Values: To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example 3: Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon".

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

Result: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

3. Insert Items: To insert a new list item, without replacing any of the existing values, we can use the insert() method. The insert() method inserts an item at the specified index.

Example 4: Insert "watermelon" as the third item.

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

Result: ['apple', 'banana', 'watermelon', 'cherry']

Note: As a result of the example above, the list will now contain 4 items.

4. Append Items: To add an item to the end of the list, use the append() method.

Example 5: Using the append() method to append an item.

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Result: ['apple', 'banana', 'cherry', 'orange']

5. Remove Specified Item: The remove() method removes the specified item.

Example 6: Remove "banana".

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Result: ['apple', 'cherry']

6. Remove Specified Index: The pop() method removes the specified index.

Example 7: Remove the second item.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

Result: [‘apple’, ‘cherry’].
If you do not specify the index, the pop() method removes the last item.

Example 8: Remove the last item.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Result: [‘apple’, ‘banana’]

7. The del keyword also removes the specified index.

Example 9: Remove the first item.

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

Result: [‘banana’, ‘cherry’]

8. The del keyword can also delete the list completely.

Example 10: Delete the entire list.

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Result: Successfully deleted “thislist”

9. Clear the List: The clear() method empties the list. The list still remains, but it has no content.

Example 11: Clear the list content.

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Result: [ ]

LOOP THROUGH A LIST

1. You can loop through the list items by using for loop.

Example: Print all items in the list, one by one.

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Result: apple, banana, cherry

2. **Using a While Loop:** You can loop through the list items by using a while loop. Use the len() function to determine the length of the list, then start at 0 and loop your way through the list items by refering to their indexes. Remember to increase the index by 1 after each iteration.

Example: Print all items, using a while loop to go through all the index numbers.

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

Result: apple, banana, cherry

COPY A LIST

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a *reference* to list1, and changes made in list1 will automatically also be made in list2.

1. There are ways to make a copy, one way is to use the built-in List method copy().

Example: Make a copy of a list with the copy() method.

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Result: ['apple', 'banana', 'cherry']

2. Another way to make a copy is to use the built-in method list().

Example: Make a copy of a list with the list() constructor.

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Result: ['apple', 'banana', 'cherry']

JOIN TWO LISTS

There are several ways to join, or concatenate, two or more lists in Python.

1. One of the easiest ways are by using the + operator.

Example: Join two lists.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
```

```
list3 = list1 + list2
print(list3)
```

Result: [‘a’, ’b’, ‘c’, 1, 2, 3]

2. **Another way to join two lists is by appending all the items from list2 into list1, one by one**.

Example: Append list2 into list1.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

Result: [‘a’, ’b’, ‘c’, 1, 2, 3]

SORTING THE LISTS

1. **Sort List Alphanumerically:** List objects have a sort() method that will sort the list alphanumerically, ascending, by default.

Example: Sort the list alphabetically.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Result: [‘banana’, ‘kiwi’, ‘mango’, ‘orange’, ‘pineapple’]

Example: Sort the list numerically.

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

Result: [23, 50, 65, 82,100]

2. **Sort Descending:** To sort descending, use the keyword argument reverse = True.

Example: Sort the list descending.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Result: [‘pineapple’,’orange’, ‘mango’, ‘kiwi’, ‘banana’]

Example: Sort the list descending.

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

Result: [100, 82, 65, 50, 23]

3. **Case Insensitive Sort:** By default the sort() method is case sensitive, resulting in all ***capital letters being sorted before lower case letters.***

Example: Case sensitive sorting can give an unexpected result.

```
thislist = ["Banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

Result: [‘Banana’, ‘Kiwi’, ‘Orange’, ‘cherry’]

4. **Luckily we can use built-in functions as key functions when sorting a list**: So if you want a case-insensitive sort function, use str.lower as a key function.

Example: Perform a case-insensitive sort of the list.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

Result: [‘banana’, ‘cherry’, ‘Kiwi’, ‘Orange’]

5. **Reverse Order:** What if you want to reverse the order of a list, regardless of the alphabet? The reverse() method reverses the current sorting order of the elements.

Example: Reverse the order of the list items.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

Result: [‘cherry’, ‘Kiwi’, ‘Orange’, ‘banana’]

TUPLES

Tuples are used to store multiple items in a single variable. A tuple is a collection which is ordered and **unchangeable**. Tuples are written with round brackets & allow duplicate members.

Example: Create a Tuple

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Result: (‘apple’, ‘banana’, ‘cherry’)

1. **Tuple Items:** Tuple items are ordered, unchangeable, and allow duplicate values. Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered: When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable: Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates: Since tuple are indexed, tuples can have items with the same value:

Example: Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```
Result: ('apple', 'banana', 'cherry', 'apple', 'cherry')

2. **Tuple Length:** To determine how many items a tuple has, use the len() function:

Example: Print the number of items in the tuple

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```
Result: 3

3. **Tuple Items - Data Types:** Tuple items can be of any data type

Example: String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

4. **A tuple can contain different data types**

Example: A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

5. **type( ):** From Python's perspective, tuples are defined as objects with the data type 'tuple':

Example: What is the data type of a tuple?

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```
Result: class 'tuple'

6. **The tuple() Constructor:** It is also possible to use the tuple() constructor to make a tuple.

Example: Using the tuple() method to make a tuple

```
thistuple = tuple(("apple", "banana", "cherry"))          # note the double round-brackets
print(thistuple)
```

Result: ('apple', 'banana', 'cherry')

ACCESS TUPLE ITEMS

7. **Access Tuple Items:** You can access tuple items by referring to the index number, inside square brackets:

Example: Print the second item in the tuple

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Result: banana

Note: The first item has index 0.

8. **Negative Indexing:** Negative indexing means start from the end. -1 refers to the last item, -2 refers to the second last item etc.

Example: Print the last item of the tuple

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Result: cherry

9. **Range of Indexes:** You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items.

Example: Return the third, fourth, and fifth item

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Result: ('cherry', 'orange', 'kiwi')

Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0.

10. By leaving out the start value, the range will start at the first item

Example: This example returns the items from the beginning to, but NOT included, "kiwi":

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
```

Result: ('apple', 'banana', 'cherry', 'orange')

11. By leaving out the end value, the range will go on to the end of the list

Example: This example returns the items from "cherry" and to the end:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])
```

Result: ('cherry', 'orange', 'kiwi', 'melon', 'mango')

12. **Range of Negative Indexes:** Specify negative indexes if you want to start the search from the end of the tuple:

Example: This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

Result: ('orange', 'kiwi', 'melon')

13. **Check if Item Exists:** To determine if a specified item is present in a tuple use the in keyword:

Example: Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Result: Yes, ‘apple’ is in the fruits tuple

Python - Update Tuples

1. **Change Tuple Values:** Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example: Convert the tuple into a list to be able to change it

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Result: (“apple”, “kiwi”, “cherry”)

2. **Add Items:** Once a tuple is created, you cannot add items to it.

Example: You cannot add items to a tuple

```
thistuple = ("apple", "banana", "cherry")
thistuple.append("orange") # This will raise an error
print(thistuple)
```

Result: ‘tuple’ object has no attribute ‘append’

Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example: Convert the tuple into a list, add "orange", and convert it back into a tuple

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Result: (‘apple’, ‘banana’, ‘cherry’, ‘orange’)

3. **Remove Items:** Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example: Convert the tuple into a list, remove "apple", and convert it back into a tuple

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
```

```
y.remove("apple")
thistuple = tuple(y)
```
Result: ('banana', 'cherry')

4. **Or you can delete the tuple completely**

Example: The del keyword can delete the tuple completely

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Loop through a Tuple

5. **Loop Through a Tuple:** You can loop through the tuple items by using a for loop.

Example: Iterate through the items and print the values

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```
Result: apple, banana, cherry

6. **Loop Through the Index Numbers:** You can also loop through the tuple items by referring to their index number.

Use the range() and len() functions to create a suitable iterable.

Example: Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")

for i in range(len(thistuple)):
  print(thistuple[i])
```

Result: apple, banana, cherry

7. **Using a While Loop:** You can loop through the list items by using a while loop.

Use the len() function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Example: Print all items, using a while loop to go through all the index numbers:

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

8. **Join Two Tuples:** To join two or more tuples you can use the + operator:

Example: Join two tuples

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

Result: ('a', 'b', 'c', 1, 2, 3)

9. **Multiply Tuples:** If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Example: Multiply the fruits tuple by 2

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

Result: ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')

Python Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Advantages of functions in Python:

a) By using functions, we can avoid rewriting same logic/code again and again.

b) We can call python functions any number of times in a program & from any place in a program.

c) We can track a large python program easily when it is divided into multiple functions.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called ***user-defined functions***.

1. **Defining a Function:** Here are simple rules to define a function in Python.

   - Function blocks begin with the keyword **def** followed by the function name and parentheses ( ).
   - Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
   - The code block within every function starts with a colon (:) and is indented.
   - The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
def my_function():
        print("Hello from a function")
```

2. **Calling a Function:** To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")
my_function()
```

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
# Function definition is here
def pr(str):
   "This prints a passed string into this function"
   Print(str)
   return

# Now you can call printme function
pr("I'm first call to user defined function!")
pr("Again second call to the same function")
```

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

I'm first call to user defined function!
Again second call to the same function

3. **Arguments:** Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Haryana")

my_function("Hisar")
my_function("Sirsa")
my_function("Jind")
```

4. **Parameters or Arguments?:** The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective: ***A parameter is the variable listed inside the parentheses in the function definition***.

An argument is the value that is sent to the function when it is called.

Ex: Function to calculate the sum of 2 variables

```
def sum(c,d):
  return c*d;
a=int(input("Enter a: "))
b=int(input("Enter b: "))
print("Sum= ",sum(a,b))
```

5. **Number of Arguments:** By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example: This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

Result: Emil Refsnes

If you try to call the function with 1 or 3 arguments, you will get an error:

Example: This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil")
```

Result: Error

Function Arguments

6. **Pass by reference vs value:** All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

```
# Function definition is here
def changeme(mylist):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme(mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. ***So, this would produce the following result*** −

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

7. **Arbitrary/variable length Arguments, *args:** If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example: If the number of arguments is unknown, add a * before the parameter name.

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Result: The youngest child is Linus

8. **Keyword Arguments:** You can also send arguments with the *key* = *value* syntax. This way the order of the arguments does not matter.

Example:

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)
```

```
my_function(child1= "Emil", child2 = "Tobias", child3 = "Linus")
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

9. **Default Parameter Value:** The following example shows how to use a default parameter value. If we call the function without argument, it uses the default value:

Example:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
```

Result: I am from Sweden
I am from India
I am from Norway

1. **Passing List as an Argument:** You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

Return Values: To let a function return a value, use the return statement:

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Result: 15, 25, 45

2. **The pass Statement:** function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def myfunction():
  pass
```

3. **Recursion:** Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

In this example, gph() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

```
def gph(k):
  if(k > 0):
    result = k + gph(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
gph(6)
```

Result: Recursion Example Results

1, 3, 6, 10, 15, 21

Unit - 3

Python Module is a file that contains built-in functions, classes, and **its** variables. There are many **Python modules**, each with its specific work.

What is Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Create a Python Module

To create a Python module, write the desired code and save that in a file with **.py** extension. Let's understand it better with an example:

Example:

Let's create a simple calc.py in which we define two functions, one **add** and another **subtract**.

Import module in Python

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

***Note**: A search path is a list of directories that the interpreter searches for importing a module.*

For example, to import the module calc.py, we need to put the following command at the top of the script.

Syntax to Import Module in Python

```
import module
```

Python Import From Module

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

Import Specific Attributes from a Python module

Here, we are importing specific sqrt and factorial attributes from the math module.

Import all Names

The * symbol used with the import statement is used to import all the names from a module to a current namespace.

Syntax:

```
from module_name import *
```

What does import * do in Python?

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

Locating Python Modules

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path. Python interpreter searches for the module in the following manner:

- First, it searches for the module in the current directory.

- If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.
- If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

Directories List for Modules

Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

Renaming the Python Module

We can rename the module while importing it using the keyword.

Syntax: *Import* ***Module_name*** *as* ***Alias_name***

Python Built-in modules

There are several built-in modules in Python, which you can import whenever you like.

Random Numbers in Python

Python defines a set of functions that are used to generate or manipulate random numbers through the random module.

Functions in the random module rely on a pseudo-random number generator function random(), which generates a random float number between 0.0 and 1.0. These particular type of functions are used in lot of games, lotteries, or any application requiring a random number generation.

Let us see an example of generating a random number in Python using the random() function.

Generating a Random number using choice()

Python random.choice() is an inbuilt function in the Python programming language that returns a random item from a list, tuple, or string.

Generating a Random Number using randrange()

The random module offers a function that can generate Python random numbers from a specified range and also allows room for steps to be included, called randrange().

Generating a Random number using shuffle()

The shuffle() function is used to shuffle a sequence (list). Shuffling means changing the position of the elements of the sequence. Here, the shuffling operation is in place.

Math Module in Python?

Math Module is an in-built Python library made to simplify mathematical tasks in Python. It consists of various mathematical constants and functions that can be used after importing the math module.

Constants in Math Module

The Python math module provides values of various constants like **pi.** We can easily write their values with these constants. The constants provided by the math module are :

- Pi
- Infinity
- Not a Number (NaN)

1. Pi

You all must be familiar with pi. The pi is depicted as either 22/7 or 3.14. **math.pi** provides a more precise value for the pi.

Example 1: This code imports the math module and then prints the value of the mathematical constant **pi**.

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

Output:

```
3.141592653589793
```

Example 2: Let's find the area of the circle

The code utilizes the math module in Python, defines a radius and the mathematical constant pi, and calculates the area of a circle using the formula**' A = pi * r * r'**. It demonstrates the application of mathematical concepts and the usage of the math module for numerical calculations

```
import math
r = 4
pie = math.pi
print(pie * r * r)
```

Output:

```
50.26548245743669
```

2. Infinity

Infinity basically means something which is never-ending or boundless from both directions i.e. negative and positive. It cannot be depicted by a number. The Python **math.inf** constant returns of positive infinity. For negative infinity, use **-math.inf**.

Example 1: This code imports the math module and then prints the values of positive and negative infinity.

```
import math
print (math.inf)
print (-math.inf)
```

Output:

```
inf
-inf
```

Example 2: Comparing the values of infinity with the maximum floating point value

This code imports the math module and then compares the values of positive and negative infinity to the values of 10e108 and -10e108, respectively.

```
import math
print (math.inf > 10e108)
print (-math.inf < -10e108)
```

Output:

```
True
True
```

3. NaN Values

The Python **math.nan** constant returns a floating-point nan (Not a Number) value. This value is not a legal number. The nan constant is equivalent to float("nan").

Example: This code imports the math module and then prints the value of math.nan. math.nan represents **Not a Number, which is a special value that is used to indicate that a mathematical operation is undefined or the result is not a number**.

```
import math
print (math.nan)
```

Output:

```
nan
```

Numeric Functions in Math Module

1. Finding the ceiling and the floor value

Ceil value means the smallest integral value greater than the number and the floor value means the greatest integral value smaller than the number. This can be easily calculated using the **ceil()** and **floor()** method respectively.

Example:

This code imports the math module, assigns the value 2.3 to the variable a, and then calculates and prints the ceiling and floor of a.

```
import math
a = 2.3
print ("The ceil of 2.3 is : ", end="")
print (math.ceil(a))
print ("The floor of 2.3 is : ", end="")
print (math.floor(a))
```

Output:

```
The ceil of 2.3 is : 3
The floor of 2.3 is : 2
```

2. Finding the factorial of the number

Using the **factorial()** function we can find the factorial of a number in a single line of the code. An error message is displayed if number is not integral.

Example: This code imports the math module, assigns the value 5 to the variable a, and then calculates and prints the factorial of a.

```
import math
a = 5
print("The factorial of 5 is : ", end="")
print(math.factorial(a))
```

Output:

```
The factorial of 5 is : 120
```

3. Finding the GCD

gcd() function is used to find the greatest common divisor of two numbers passed as the arguments.

Example: This code imports the math module, assigns the values 15 and 5 to the variables a and b, respectively, and then calculates and prints the greatest common divisor (GCD) of a and b.

```
import math
a = 15
b = 5
print ("The gcd of 5 and 15 is : ", end="")
print (math.gcd(b, a))
```

Output:

```
The gcd of 5 and 15 is : 5
```

4. Finding the absolute value

fabs() function returns the absolute value of the number.
Example: This code imports the math module, assigns the value -10 to the variable a, and then calculates and prints the absolute value of a.

```
import math
a = -10
print ("The absolute value of -10 is : ", end="")
print (math.fabs(a))
```

Output:

```
The absolute value of -10 is : 10.0
```

Logarithmic and Power Functions

1. Finding the power of exp

exp() method is used to calculate the power of e.
Example: This code imports the math module and then calculates and prints the exponential values of three different input values: an integer, a negative integer, and a float.

```
import math
test_int = 4
test_neg_int = -3
test_float = 0.00
print (math.exp(test_int))
print (math.exp(test_neg_int))
print (math.exp(test_float))
```

Output:

```
54.598150033144236
0.049787068367863944
1.0
```

2. Finding the power of a number

pow() function computes x**y. This function first converts its arguments into float and then computes the power.
Example: This code first prints the string **"The value of 3**4 is : "** to the console. Then, it calculates the value of 3 raised to the power of 4 using the **pow()** function and prints the result to the console.

```
print ("The value of 3**4 is : ",end="")
print (pow(3,4))
```

Output:

```
The value of 3**4 is : 81.0
```

3. Finding the Logarithm

- **log(a)** function returns the logarithmic value of a with base b. If the base is not mentioned, the computed value is of the natural log.
- **log2(a)** function computes value of log a with base 2. This value is more accurate than the value of the function discussed above.
- **log10(a)** function computes value of log a with base 10. This value is more accurate than the value of the function discussed above.

This code imports the math module and then calculates and prints the logarithms of three different numbers. The math module provides several functions for working with logarithms, including log(), log2(), and log10().

```
import math
print ("The value of log 2 with base 3 is : ", end="")
print (math.log(2,3))
print ("The value of log2 of 16 is : ", end="")
print (math.log2(16))
print ("The value of log10 of 10000 is : ", end="")
print (math.log10(10000))
```

Output:

```
The value of log 2 with base 3 is : 0.6309297535714574
The value of log2 of 16 is : 4.0
The value of log10 of 10000 is : 4.0
```

4. Finding the Square root

sqrt() function returns the square root of the number.

Example: This code imports the math module and then calculates and prints the square roots of three different numbers: 0, 4, and 3.5. The math module provides several functions for working with mathematical operations, including the square root function **sqrt()**.

```
import math
print(math.sqrt(0))
print(math.sqrt(4))
print(math.sqrt(3.5))
```

Output:

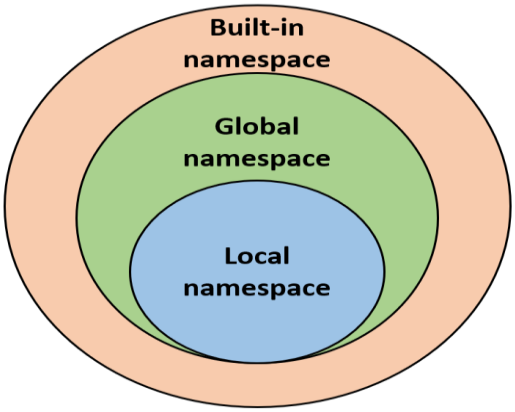
```
0.0
2.0
1.8708286933869707
```

What is namespace:

A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary. e.g. a directory-file system structure in computers. One can have multiple directories having a file with the same name inside every directory. But one can get directed to the file, just by specifying the absolute path to the file.

Types of namespaces :

When Python interpreter runs solely without any user-defined modules, methods, classes, etc. some functions like print(), id() are always present, **these are built-in namespaces**. When a user creates a module, a global namespace gets created, later the creation of local functions creates the local namespace. The **built-in namespace** encompasses the **global namespace** and the global namespace encompasses the **local namespace**.



Type of Namespaces

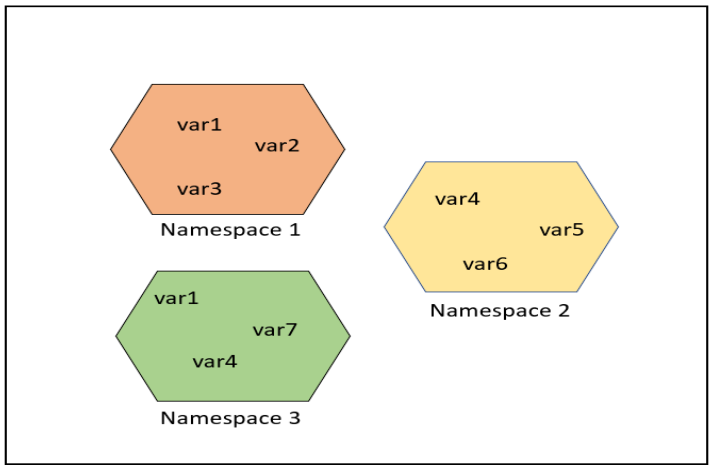
The lifetime of a namespace:
A lifetime of a namespace depends upon the scope of objects, if the scope of an object ends, the lifetime of that namespace comes to an end. Hence, it is not possible to access the inner namespace's objects from an outer namespace.

```
# var1 is in the global namespace
var1 = 5
def some_func():

    # var2 is in the local namespace
    var2 = 6
    def some_inner_func():

        # var3 is in the nested local namespace
        var3 = 7
```

As shown in the following figure, the same object name can be present in multiple namespaces as isolation between the same name is maintained by their namespace.



Python File Handling

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters and they form a text file. Each line of a file is terminated with a special character, called the **EOL or End of Line** characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

Advantages of File Handling

- **Versatility**: File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.
- **Flexibility**: File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files, etc.), and to perform different operations on files (e.g. read, write, append, etc.).
- **User–friendly**: Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.
- **Cross-platform**: Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling

- **Error-prone:** File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).
- **Security risks**: File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.
- **Complexity**: File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.
- **Performance**: File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.

Working of open() Function in Python

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python's inbuilt function open() but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.
f = open(filename, mode)

Where the following mode is supported:

1. **r:** open an existing file for a read operation.
2. **w:** open an existing file for a write operation. If the file already contains some data then it will be overridden but if the file is not present then it creates the file as well.
3. **a:** open an existing file for append operation. It won't override existing data.
4. **r+:** To read and write data into the file. The previous data in the file will be overridden.
5. **w+:** To write and read data. It will override existing data.
6. **a+:** To append and read data from the file. It won't override existing data.

Let us consider the following "**abc.txt**" file as an example.

Hello world
123 456

Working in Read mode

Example 1: The open command will open the file in the read mode and for loop will print each line present in the file.

```
# a file named "abc", will be opened with the reading mode.
file = open('abc.txt', 'r')

# This will print every line one by one in the file
for each in file:
```

```
    print (each)
```

Example 2: In this example, we will extract a string that contains all characters in the file then we can use **file.read()**.

```
# Python code to illustrate read() mode
file = open("abc.txt", "r")
print (file.read())
```

Example 3: In this example, we will see how we can read a file using the **with** statement.

```
# Python code to illustrate with()
with open("abc.txt") as file:
    data = file.read()
 print(data)
```

Example 4: Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("abc.txt", "r")
print (file.read(5))
```

Output:

```
Hello
```

Example 5:
We can also split lines while reading files in Python. The split() function splits the variable when space is encountered. You can also split using any characters as you wish.

```
# Python code to illustrate split() function
with open("abc.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

Output:

```
['Hello', 'world']
['123', '456']
```

Creating a File using the write() Function

Just like reading a file in Python, there are a number of ways to write in a file in Python. Let us see how we can write the content of a file using the write() function in Python.

Working in Write Mode

Example 1: In this example, we will see how the write mode and the write() function is used to write in a file. The close() command terminates all the resources in use and frees the system of this particular program.

```
# Python code to create a file
file = open('abc.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

Output:

This is the write command. It allows us to write in a particular file

Example 2: We can also use the written statement along with the with() function.

```
# Python code to illustrate with() alongwith write()
with open("file.txt", "w") as f:
    f.write("Hello World!!!")
```

Output:

Hello World!!!

Working of Append Mode

```
# Python code to illustrate append() mode
file = open('abc.txt', 'a')
file.write("This will add this line")
file.close()
```

Output:

This is the write command. It allows us to write in a particular file. This will add this line.

How Files are Loaded into Primary Memory

Python interacts with files loaded in primary memory or main memory through **"file handlers"**. **File handler is like a cursor, which defines from where the data has to be read or written in the file.**

Opening a File

It is done using the open() function. No module is required to be imported for this function.

File_object = open(r"File_Name","Access_Mode")

The file should exist in the same directory as the python program file else, the full address of the file should be written in place of the filename.

Note: The **r is placed before the filename to prevent the characters in the filename string to be treated as special characters.** For example, if there is \temp in the file address, then \t is treated as the tab character, and an error is raised of invalid address. The **r makes the string raw, that is, it tells that the string is without any special characters.** The r can be ignored if the file is in the same directory and the address is not being placed.

Open function to open the file "MyFile1.txt (same directory) in append mode and store its reference in the variable file1 and "MyFile2.txt" in D:\Text in file2

```
file1 = open("MyFile1.txt","a")
file2 = open(r"D:\Text\MyFile2.txt","w+")
```

Here, file1 is created as an object for MyFile1 and file2 as object for MyFile2

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

e.g. File_object.close()

```
# Opening and Closing a file "MyFile.txt" for object name file1.
file1 = open("MyFile.txt","a")
file1.close()
```

Writing to a file: There are two ways to write in a file:

write() : Inserts the string str1 in a single line in the text file.
File_object.write(str1)

writelines() : For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

File_object.writelines(L) for L = [str1, str2, str3]

Reading from a file

There are three ways to read data from a text file.

read() : Returns the read bytes in form of a string. Reads n bytes, if no n is specified, reads the entire file.
File_object.read([n])

readline() : Reads a line of the file and returns in form of a string. For specified n, reads at most n bytes. However, does not read more than one line, even if n exceeds the length of the line.
File_object.readline([n])

readlines() : Reads all the lines and return them as each line a string element in a list.
File_object.readlines()

```
# Program to show various ways to read and write data in a file.

file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]

# \n is placed to indicate EOL (End of Line)
file1.write("Hello \n")
file1.writelines(L)
file1.close() #to change file access modes

file1 = open("myfile.txt","r+")
```

```
print("Output of Read function is ")
print(file1.read())
print()

# seek(n) takes the file handle to the nth byte from the beginning.
file1.seek(0)

print( "Output of Readline function is ")
print(file1.readline())
print()

file1.seek(0)

# To show difference between read and readline
print("Output of Read(9) function is ")
print(file1.read(9))
print()

file1.seek(0)

print("Output of Readline(9) function is ")
print(file1.readline(9))

file1.seek(0)
# readlines function
print("Output of Readlines function is ")
print(file1.readlines())
print()
file1.close()
```

Output:

```
Output of Read function is
Hello
This is Delhi
This is Paris
This is London

Output of Readline function is
Hello

Output of Read(9) function is
Hello
Th

Output of Readline(9) function is
Hello

Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

Appending to a file

```
# Python program to illustrate Append vs write mode
```

```
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]
file1.writelines(L)
file1.close()

# Append-adds at last
file1 = open("myfile.txt","a")#append mode
file1.write("Today \n")
file1.close()

file1 = open("myfile.txt","r")
print("Output of Readlines after appending")
print(file1.readlines())
print()
file1.close()

# Write-Overwrites
file1 = open("myfile.txt","w")#write mode
file1.write("Tomorrow \n")
file1.close()

file1 = open("myfile.txt","r")
print("Output of Readlines after writing")
print(file1.readlines())
print()
file1.close()
```

Output:

```
Output of Readlines after appending
['This is Delhi \n', 'This is Paris \n', 'This is London \n', 'Today \n']

Output of Readlines after writing
['Tomorrow \n']
```

Binary files in Python:

Reading binary files is an important skill for working with data (non-textual) such as images, audio, and videos. Using file mode and the "**read"** method you can easily read binary files. Whether you are dealing with **multimedia files, compressed data, or custom binary format**s, Python's ability to handle binary data empowers you to create powerful and versatile applications for a wide range of use cases.

What are Binary files?
Generally, binary means two. In computer science, binary files are stored in a binary format having digits **0's** and **1's**. For example, **the number 9 in binary format is represented as '1001'**. In this way, our computer stores each and every file in a machine-readable format in a sequence of binary digits. The structure and format of binary files depend on the type of file. Image files have different structures when compared to audio files. However, decoding binary files depends on the complexity of the file format. In this article, let's understand the reading of binary files.

To read a binary file:
Step 1: Open the binary file in binary mode

To read a binary file in Python, first, we need to open it in binary mode (‘”rb”‘). We can use the ‘open()’ function to achieve this.

Step 2: Create a binary file
To create a binary file in Python, You need to open the file in binary write mode (wb).

Step 3: Read the binary data
After opening the binary file in binary mode, we can use the read() method to read its content into a variable. The” read()” method will return a sequence of bytes, which represents the binary data.

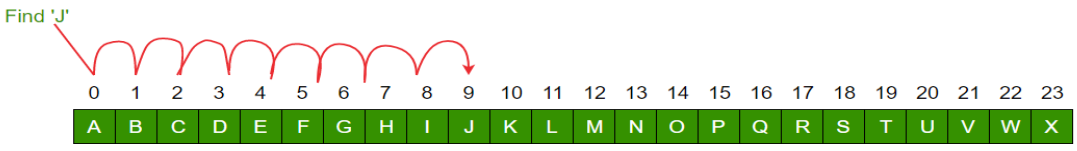
Step 4: Process the binary data
Once we have read the binary data into a variable, we can process it according to our specific requirements. Processing the binary data could involve various tasks such as decoding binary data, analyzing the content, or writing the data to another binary file.

Step 5: Close the file
After reading and processing the binary data, it is essential to close the file using the “close()” method to release system resources and avoid potential issues with file access.

Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[] using Python.

Example:
Input: arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
# Element to Find
x = 'J'
Output : 9



Explanation: A simple approach is to do a **linear search**, i.e
- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn’t match with any of the elements, return -1.

Binary Search

In a nutshell, this search algorithm takes advantage of a collection of elements that is already sorted by ignoring half of the elements after just one comparison.
1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else if x is greater than the mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.
4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

UNIT -4

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Let's consider an example, you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Syntax: Class Definition

```
class ClassName:
    # Statement
```

Syntax: Object Definition

```
obj = ClassName()
    print(obj.atrr)
```

The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Creating a Python Class

Here, the class keyword indicates that you are creating a class followed by the name of the class (Dog in this case).

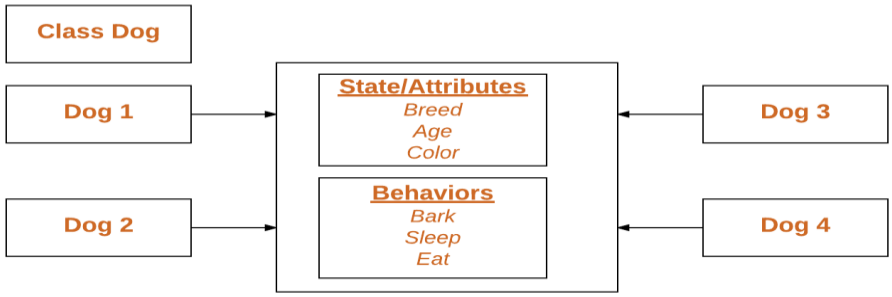
```
class Dog:
    sound = "bark"
```

Some points on Python class:

Classes are created by keyword class.
Attributes are the variables that belong to a class.
Attributes are always public and can be accessed using the dot (.) operator e.g.: Myclass.Myattribute

Declaring Class Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

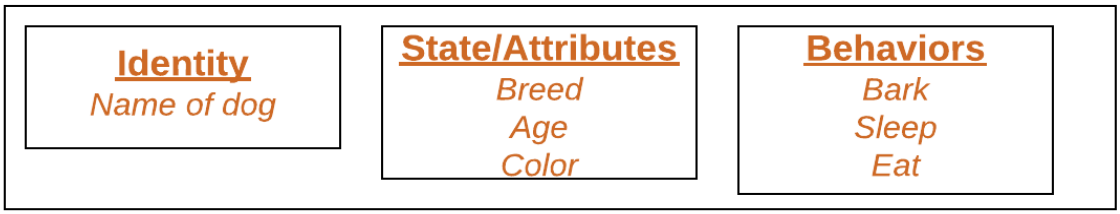


Python Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of:

- State: It is represented by the attributes of an object. It also reflects the properties of an object.
- Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- Identity: It gives a unique name to an object and enables one object to interact with other objects.



To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Example of Python Class and object

Creating an object in Python involves instantiating a class to create a new instance of that class. This process is also referred to as object instantiation.

```
# Program to demonstrate instantiating a class

class Dog:
    # A simple class attribute
    attr1 = "mammal"
    attr2 = "dog"

    # A sample method
    def fun(self):
```

```
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)

 # Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes and method through objects
print(Rodger.attr1)
Rodger.fun()
```

Output:
mammal
I'm a mammal
I'm a dog

In the above example, an object is created which is basically a dog named Rodger. This class only has two class attributes that tell us that Rodger is a dog and a mammal.

Explanation:
In this example, we are creating a Dog class and we have created two class attributes **attr1 and attr2**. We have created a function/method named **fun()** which returns the string "I'm a, {attr1}" and I'm a, {attr2}. We have created an object of the Dog class and we are printing the **attr1** of the object. Finally, we are calling the **fun()** function.

The Python __init__ Method

The __init__method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Self Parameter

When we call a method of this object as myobject.method(arg1,arg2), this is automatically converted by Python into MyClass.method(myobject,arg1,arg2).

Example:

```
class GFG:
    def __init__(self, name, company):
        self.name = name
        self.company = company

    def show(self):
        print("Hello my name is " + self.name+" and I work in "+self.company+".")

obj = GFG("John", "GP Hisar")
obj.show()
```

The Self Parameter does not call it to be Self, You can use any other name instead of it. Here we change the self to the word someone and the output will be the same.

```
class GFG:
    def __init__(somename, name, company):
        somename.name = name
        somename.company = company
```

```
    def show(somename):
        print("Hello my name is " + somename.name +
              " and I work in "+somename.company+".")

 obj = GFG("John", "GP Hisar")
obj.show()
```

Output: Output for both of the codes will be the same.
Hello my name is John and I work in GP Hisar.

Explanation:
In this example, we are creating a GFG class and we have created the **name, and company** instance variables in the constructor. We have created a method named show**()** which returns the string “Hello my name is ” + {name} +” and I work in “+{company}+”.”.We have created a class object and we are passing the name **John and Company** GP Hisar to the instance variable. Finally, we are calling the **show()** method of the class.

Pass Statement
The program’s execution is unaffected by the **pass** statement’s inaction. It simply permits the program to skip past that section of the code without doing anything.

```
class MyClass:
    pass

# Sample class with init method
class Person:
    # init method or constructor
    def __init__(self, name):
        self.name = name

   # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Nikhil')
p.say_hi()
```

Output:
Hello, my name is Nikhil

In this example, we are creating a Person class and we have created a **name** instance variable in the constructor. We have created a method named as say_hi() which returns the string “Hello, my name is {name}”.We have created a person class object and we pass the name Nikhil to the instance variable. Finally, we are calling the say_hi() method of the class.

Class and Instance Variables
Instance variables are for data, unique to each instance and **class variables** are for attributes and methods shared by all instances of the class. **Instance variables** are variables whose value is assigned inside a constructor or method with self whereas **class variables** are variables whose value is assigned in the class.

Defining instance variables using a constructor
Program to show that the **variables with a value assigned in the class declaration, are class variables** and **variables inside methods and constructors are instance variables**.

```
# Class for Dog
class Dog:
    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed, color):

        # Instance Variable
        self.breed = breed
        self.color = color

# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger details:')
print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('\nBuzo details:')
print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using class name also
print("\nAccessing class variable using class name")
print(Dog.animal)
```

Output:

```
Rodger details:
Rodger is a dog
Breed:  Pug
Color:  brown

Buzo details:
Buzo is a dog
Breed:  Bulldog
Color:  black

Accessing class variable using class name
dog
```

Explanation:

A class named Dog is defined with a class variable animal set to the string “dog”. Class variables are shared by all objects of a class and can be accessed using the class name. Dog class has two instance variables **breed and color.** Later we are creating two objects of the **Dog** class and we are printing the value of both objects with a class variable named animal.

Defining instance variables using the normal method:

```
# Program to show that how to create instance variables inside methods
 # Class for Dog
class Dog:
```

```
    # Class Variable
    animal = 'dog'

    # The init method or constructor
    def __init__(self, breed):
        # Instance Variable
        self.breed = breed

    # Adds an instance variable
    def setColor(self, color):
        self.color = color

    # Retrieves instance variable
    def getColor(self):
        return self.color

# Driver Code
Rodger = Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

Output:
brown

Explanation:
In this example, we have defined a class named **Dog** and we have created a class variable animal. We have created an instance variable breed in the **constructor**. The class Dog consists of two methods **setColor** and **getColo**r, they are used for creating and initializing an instance variable (i.e. color) and retrieving the value of that instance variable. We have made an object of the **Dog** class (i.e. Rodger) and we have set the instance variable value to brown and we are printing the value in the terminal.

Creating a class and object with class and instance attributes

```
class Dog:
    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy

Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

- attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.
- __init__ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.
- The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The __init__ method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

```
class Dog:
    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("My name is {}".format(self.name))

# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

Output
My name is Rodger
My name is Tommy

Exception Handling

What is Exception? An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

A python program terminates as soon as it encounters an unhandled error. These errors can be classified into two classes:

1. Syntax errors: Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.
   If a>3
   Here a colon (:) is missing in the if statement.

2. Logical errors (Exceptions): Errors that occur at runtime (after passing the syntax test) are called **Exceptions or Logical errors**

   Some of the Common exceptions are:

SN.	Exception	Cause of error
1.	AssertionError	Raised when an assert statement fails
2.	FileNotFoundError	To open a file (for reading) that does not exist
3.	IOError	Raised when Input Output operation fails
3.	EOFError	Raised when end of file is reached, and yet operations are being performed
4.	OverflowError	Raised when result of an arithmetic operation is too large
5.	SystemError	Raised when interpreter detects internal error
6.	ZeroDivisionError	To divide a number by zero

Example 1:

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

Result:

```
Enter a: 10
Enter b: 0
ZeroDivisionError: division by zero
```

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

The try block lets you test a block of code for errors.
The except block lets you handle the error.
The finally block lets you execute code, regardless of the result of the try- and except blocks.

Handling an exception: If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message. These exceptions can be handled using the try statement:

Example 2: The try block will generate an exception, because x is not defined

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed. Without the try block, the program will crash and raise an error:

This statement will raise an error, because x is not defined:

```
print(x)
```

Handling Exception using try....except...else block

```
try:
   You do your operations here;
   ......................
except Exception1:
   If there is ExceptionI, then execute this block.
except Exception2:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Example 3: This example opens a file, writes content in the file and comes out gracefully because there is no problem at all −

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

This produces the following result:

Written content in the file successfully

Example 4: This example tries to open a file which does not exist, so it raises an exception −

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

This produces the following result − Error: can't find file or read data

The *except* Clause with Multiple Exceptions: You can also use the same except statement to handle multiple exceptions as follows −

```
try:
   You do your operations here;
   ......................

except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list, then execute this block.
   ......................
else:
```

If there is no exception then execute this block.

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example 5: Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Result: Variable x is not defined

Else block: You can use the else keyword to define a block of code to be executed if no errors were raised:

Example 6: In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Result: Hello
Nothing went wrong

The try-finally clause: The finally block, if specified, will be executed regardless of the try block raises an error or not.

Example 7:

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Result: Something went wrong
The 'try except' is finished

This can be useful to close objects and clean up resources:

Example 8: Try to open and write to a file that is not writable (i.e. read only file)

```
try:
  f = open("demofile.txt")
  f.write("Test the program")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

Result: Something went wrong when writing to the file

You can use a **finally:** block along with a **try:** block. **The finally block is a place to put any code that must execute, whether the try-block raised an exception or not**. The syntax of the try-finally statement is this −

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use *else* clause as well along with a finally clause.

Example 9:

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the ***following result*** − Error: can't find file or read data

Example 10: Same example can be written more clearly as follows −

```
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print ("Going to close the file")
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

Result: Going to close the file

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs. To throw (or raise) an exception, use the raise keyword.

Example 11: Raise an error and stop the program if x is lower than 0:

```
x = -1
if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

Result: Sorry, no numbers below zero

The raise keyword is used to raise an exception. You can define what kind of error to raise, and the text to print to the user.

Example 12: Raise a TypeError if x is not an integer:

```
x = "hello"
if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

Result: TypeError: Only integers are allowed

Example 13:

```
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")
```

Result:

```
   Enter the age? 16
   The age is not valid
```

Exception Handling in Python:

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

```
try :
   #statements in try block
except :
   #executed when error in try block
```

The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped.

If the exception does occur, the program flow is transferred to the except: block. The statements in the except: block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message.

You can specify the type of exception after the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

The rest of the statements after the except block will continue to be executed, regardless if the exception is encountered or not.

The following example will throw an exception when we try to divide an integer by a string.

Example: try...except blocks

```
try:
```

```
    a=5
    b='0'
    print(a/b)
except:
    print('Some error occurred.')
print("Out of try except blocks.")
```

Output:
Some error occurred.
Out of try except blocks.

You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except blocks with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

Example: Catch Specific Error Type

```
try:
  a=5
  b='0'
  print(a+b)
except TypeError:
  print('TypeError Occurred')
except:
  print('Some error occurred.')
print ("Out of try except blocks")
```

Output:
TypeError Occurred
Out of try except blocks

The default except: block must come after all the except block that catch specific errors; otherwise Python will raise an error.

Example: Default except: Block

```
try:
  a=5
  b='0'
  print(a+b)
except:
  print('Some error occurred.')  #except: before other blocks
except TypeError:
  print('TypeError Occurred')
print ("Out of try except blocks")
```

Output:
File "main.py", line 4
    print(a+b)
    ^
SyntaxError: default 'except:' must be last

As mentioned above, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

Example: Multiple except Blocks

```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
except:
  print('Some error occurred.')
print ('Out of try except blocks')
```

Output:
Division by zero not allowed
Out of try except blocks

However, if variable b is set to '0', TypeError will be encountered and processed by corresponding except block.

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

Example: try, except, else, finally blocks

```
try:
  x,y = 10, 2
  z=x/y
except ZeroDivisionError:
  print("except ZeroDivisionError block")
  print("Division by 0 not accepted")
except:
  print('Some error occurred.')
else:
  print("Division = ", z)
finally:
  print("Executing finally block")
print ("Out of try, except, else and finally blocks." )
```

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

```
Division =  5.0
Executing finally block
Out of try, except, else and finally blocks.
```

Executing 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.

Example: try, except, else, finally blocks

```
try:
  x,y = 10, 0
  z=x/y
```

```
except ZeroDivisionError:
  print("Cannot divide by zero. Try again")
  print("Division by 0 not accepted")
except:
  print('Some error occurred.')
else:
  print("Division = ", z)
finally:
  print("Executing finally block")
print ("Out of try, except, else and finally blocks." )
```

Output:

```
Cannot divide by zero. Try again
Division by 0 not accepted
Executing 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.

Example: try, except, else, finally blocks

```
try:
  x,y = 10, "xyz"
  z=x/y
except ZeroDivisionError:
  print("except ZeroDivisionError block")
  print("Division by 0 not accepted")
except:
  print('Error occurred.')
else:
  print("Division = ", z)
finally:
  print("Executing finally block")
print ("Out of try, except, else and finally blocks." )
```

Output:

```
Error occurred.
Executing finally block
Out of try, except, else and finally blocks.
```

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.

Raise an Exception

Python also provides the raise keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

The following code accepts a number from the user. The try block raises a ValueError exception if the number is outside the allowed range.

Example: Raise an Exception

```
try:
```

```
  x,y=100,2
  z=x/y
  if z > 10:
    raise ValueError(z)
except ValueError:
  print(z, "is out of allowed range")
else:
  print(z, "is within the allowed range")
```

Output:
50.0 is out of allowed range

Here, the raised exception is a ValueError type. However, you can also raise a custom exception type.

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the **derived class or child class** and the class from which the properties are being derived is called the **base class or parent class**. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
- Inheritance offers a simple, understandable model structure.
- Less development and maintenance expenses result from an inheritance.

Python Inheritance Syntax
The syntax of simple inheritance in Python is as follows:

```
Class BaseClass:
    {Body}
Class DerivedClass(BaseClass):
    {Body}
```

Creating a Parent Class
A parent class is a class whose properties are inherited by the child class. Let's create a parent class called **Person** which has a **Display** method to display the person's information.

```
# A Python program to demonstrate inheritance
class Person():

  # Constructor
  def __init__(self, name, id):
    self.name = name
    self.id = id

  # To check if this person is an employee
  def Display(self):
    print(self.name, self.id)

# Driver code
emp = Person("Satyam", 102) # An Object of Person
```

```
emp.Display()
```

Output:
Satyam 102

Creating a Child Class
A child class is a class that derives the properties from its parent class. Here **Emp** is another (child) class that is going to inherit the properties of the **Person** class(base class).

```
class Emp(Person):

  def Print(self):
    print("Emp class called")

Emp_details = Emp("Mayank", 103)

# calling parent class function
Emp_details.Display()

# Calling child class function
Emp_details.Print()
```

Output:
Mayank 103
Emp class called

Example of Inheritance in Python

```
# A Python program to demonstrate inheritance
# Base or Super class.
class Person():

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

    # To check if this person is an employee
    def isEmployee(self):
        return False

# Inherited or Subclass (Note Person in bracket)
class Employee(Person):

    # Here we return true
    def isEmployee(self):
        return True

# Driver code
emp = Person("Geek1")  # An Object of Person
print(emp.getName(), emp.isEmployee())
```

```
emp = Employee("Geek2")  # An Object of Employee
print(emp.getName(), emp.isEmployee())
```

Output:
Geek1 False
Geek2 True

Inheritance in Python
In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through the employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

```
# Python code to demonstrate how parent constructors are called.
# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))

# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using its instance
a.display()
a.details()
```

Output
Rahul
886012

```
My name is Rahul
IdNumber: 886012
Post: Intern
```

Python program to demonstrate error if we forget to invoke __init__() of the parent

If you forget to invoke the __init__() of the parent class then its instance variables would not be available to the child class. The following code produces an error for the same reason.

```
class A:
    def __init__(self, n='Rahul'):
        self.name = n

class B(A):
    def __init__(self, roll):
        self.roll = roll

object = B(23)
print(object.name)
```

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

```
# Python example to show the working of multiple inheritance
class Base1(object):
    def __init__(self):
        self.str1 = "Geek1"
        print("Base1")

class Base2(object):
    def __init__(self):
        self.str2 = "Geek2"
        print("Base2")

 class Derived(Base1, Base2):
    def __init__(self):

        # Calling constructors of Base1 and Base2 classes
        Base1.__init__(self)
        Base2.__init__(self)
        print("Derived")

    def printStrs(self):
        print(self.str1, self.str2)

ob = Derived()
ob.printStrs()
```

Output:

Base1
Base2
Derived
Geek1 Geek2

```
# A Python program to demonstrate multilevel inheritance

class Base(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

 # Inherited or Sub class (Note Person in bracket)
class Child(Base):

    # Constructor
    def __init__(self, name, age):
        Base.__init__(self, name)
        self.age = age

    # To get age
    def getAge(self):
        return self.age

# Inherited or Sub class
class GrandChild(Child):

    # Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address

    # To get address
    def getAddress(self):
        return self.address

# Driver code
g = GrandChild("Geek1", 23, "Noida")
print(g.getName(), g.getAge(), g.getAddress())
```

Output:
Geek1 23 Noida

Private members of the parent class
We don’t always want the instance variables of the parent class to be inherited by the child class i.e. we can make some of the instance variables of the parent class private, which won’t be available to the child class.

In Python inheritance, we can make an instance variable private by adding double underscores before its name. For example:

 # Python program to demonstrate private members of the parent class

```
class C(object):
    def __init__(self):
        self.c = 21

        # d is private instance variable
        self.__d = 42

 class D(C):
    def __init__(self):
        self.e = 84
        C.__init__(self)

object1 = D()

# produces an error as d is private instance variable
print(object1.c)
print(object1.__d)
```

Output :
Here we can see that when we tried to print the variable ‘c’, its value 21 is printed on the console. Whereas when we tried to print ‘d’, it generated the error. This is because the variable ‘d’ is made private by using the underscores. It is not available to the child class ‘D’ and hence the error.

What is Polymorphism: The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Example of inbuilt polymorphic functions:

```
# Python program to demonstrate in-built polymorphic functions
 # len() being used for a string
print(len("Hisar"))

# len() being used for a list
print(len([10, 20, 30]))
```

Output
5
3

```
# A simple Python function to demonstrate Polymorphism
def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output
5

Polymorphism with class methods:
The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.

Polymorphism with Inheritance:
In Python, Polymorphism let us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

```
class Bird:
  def intro(self):
    print("There are many types of birds.")
```

```
  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Code: Implementing Polymorphism with a Function

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

def func(obj):
```

```
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Output

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Polymorphism in Python using inheritance and method overriding:

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Create a list of Animal objects
animals = [Dog(), Cat()]

# Call the speak method on each object
for x in animals:
    print(x.speak())
```

Output

```
Woof!
Meow!
```

Method Overloading:

Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

Like other languages, python does not support method overloading by default. But there are different ways to achieve method overloading in Python.

The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

```
# First product method takes 02 arguments & prints their product
```

```
def product(a, b):
    p = a * b
    print(p)

# Second product method takes 03 arguments & prints their product

def product(a, b, c):
    p = a * b*c
    print(p)

# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

Output

```
100
```

In the above code, we have defined two product methods we can only use the second product method, as python does not support method overloading. We may define many methods of the same name and different arguments, but we can only use the latest defined method. Calling the other method will produce an error. Like here calling **product(4,5)** will produce an error as the latest defined product method takes three arguments.

Pure Functions

A pure function is a function that will return the same values given the same arguments. A function is not pure if there is something outside the function that can change its return, given the same arguments. Below is an example of a pure function. Given the same arguments, this function will always return the same output. It has no side effects. There is nothing outside the function that can change its return. For example, the argument that we have passed (12 and 45) will always return 12.0.

```
def abc(n1,n2):
    x=(n1*n2)/n2
    return x
print(abc(12,45)
```

Impure Functions

An impure function is 'influenced' by forces outside the function. With an impure function, given the same arguments, there is no guarantee that it will return the same output. The function below is not pure. This is because it is using a variable outside the function (global variable) — *num3*. If that variable changes, then given the same arguments, the return will also change. So, the function is not pure because it has side effects.

```
n3=10
def abc(n1,n2):
    x=(n1*n2)/n2*n3
    return x
print(abc(12,45)
```

UNIT -5

Python Tkinter

What is Tkinter in Python used for?

Tkinter is a built-in library in Python for creating graphical user interfaces (GUIs). You can use it to design desktop applications with familiar elements like buttons, windows, menus, and more. It allows you to build interactive programs that users can navigate visually.

What does TK() mean in Python?

In Python, Tk() is a function call from the tkinter module, which is the standard interface to the Tk GUI toolkit. Calling Tk() initializes a new Tkinter application, creating the main window where all the GUI components (widgets) will be placed.

Is Tkinter the only GUI for Python?

No, Tkinter is not the only GUI library for Python. There are other options like PyQt, Kivy, wxPython, and GTK+.

What is a Tkinter window in Python?

A Tkinter window is the main graphical element, representing a single on-screen window that can contain other UI elements.

Out of all the GUI methods, tkinter is the most commonly used method. It is a standard Python interface to the Tk GUI toolkit shipped with Python. Python Tkinter is the fastest and easiest way to create GUI applications. Creating a GUI using Tkinter is an easy task.

To create a Tkinter Python app, you follow these basic steps:

1. **Import the tkinter module**: This is done just like importing any other module in Python. Note that in Python 2.x, the module is named 'Tkinter', while in Python 3.x, it is named 'tkinter'.
2. **Create the main window (container)**: The main window serves as the container for all the GUI elements you'll add later.
3. **Add widgets to the main window**: You can add any number of widgets like buttons, labels, entry fields, etc., to the main window to design the interface as desired.
4. **Apply event triggers to the widgets**: You can attach event triggers to the widgets to define how they respond to user interactions.

Create First Tkinter GUI Application

There are two main methods used which the user needs to remember while creating the Python application with GUI.

Tk()

To create a main window, tkinter offers a method 'Tk (screenName =None, baseName = None, className = 'Tk', useTk = 1)'. To change the name of the window, you can change the className to the desired one.

mainloop()

There is a method known by the name mainloop() is used when your application is ready to run. mainloop() is an infinite loop used to run the application, wait for an event to occur, and process the event as long as the window is not closed.

What are Widgets in Tkinter?

Tkinter is Python's standard GUI (Graphical User Interface) package. Using **tkinter** we can use to build out interfaces – such as buttons, menus, **interfaces,** and various kind of entry fields and display areas. We call these elements **Widgets**.

In Tkinter, a widget is essentially a graphical component that the user can interact with. They can range from simple elements like buttons and labels to more complex ones like text entry fields, listboxes, and canvases. Each widget serves a specific purpose and can be customized to fit the design and functionality requirements of your application.

How Do Tkinter Widgets Work?
Each widget in Tkinter is an instance of a specific class defined in the Tkinter Module. These classes provide methods and attributes that allow you to configure the widget's appearance, behavior, and functionality. **Widgets are typically added to the application's window or frames using layout managers like pack(), grid(), or place(), which determine their position and size within the interface**.

Layout Managers:

- *The place() method in Tkinter is a geometry manager that allows you to explicitly set the position and size of a widget within a window or a frame by specifying the exact coordinates. It can be useful for precise widget placement, especially in applications where layout needs to be tightly controlled.*

```
import tkinter as tk
root = tk.Tk()
label = tk.Label(root, text="Hello, World!")
label.place(x=50, y=50)  # Placing the label at coordinates x=50, y=50
root.mainloop()
```

- ***grid()*** *places widgets in a grid layout where each widget is positioned using row and column indices. This method is more flexible and suitable for creating complex layouts as it allows precise placement of widgets in relation to each other.*

The grid() function in Tkinter places widgets in a grid layout where you define rows and columns. Each widget can be assigned to a specific row and column, and you can also specify how many rows or columns a widget should span.

```
import tkinter as tk
root = tk.Tk()
label1 = tk.Label(root, text="Label 1")
label1.grid(row=0, column=0)  # Place Label 1 at row 0, column 0
label2 = tk.Label(root, text="Label 2")
label2.grid(row=1, column=1)  # Place Label 2 at row 1, column 1
root.mainloop()
```

```
import tkinter as tk
root = tk.Tk()
for i in range(5):
    for j in range(5):
        label = tk.Label(root, text=f 'Row {i} Col {j}')
        label.grid(row=i, column=j)  # Each label is placed in its corresponding grid position
root.mainloop()
```

- ***pack()*** *is a geometry manager that organizes widgets in a block, which means it places them one after the other in the specified order and direction (top to bottom or left to right). It is simpler to use but less versatile when complex layouts are required.*

Key Differences:

- *pack() is easier for simple layouts and stacking widgets.*

- *grid() is better for creating complex, table-like layouts where precise positioning is necessary.*
- *Widgets managed by pack() and grid() cannot be mixed in the same master window or frame.*

WIDGETS

There are a number of widgets which you can put in your tkinter application. Some of the major widgets are explained below:

1. Label

It refers to the display box where you can put any text or image which can be updated any time as per the code. The general syntax is:

```
w=Label(master, option=value)
```

master is the parameter used to represent the parent window.

2. Button

To add a button in your application, this widget is used. The general syntax is:

```
w=Button(master, option=value)
```

master is the parameter used to represent the parent window. There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas.

3. Entry

It is used to input the single line text entry from the user. For multi-line text input, Text widget is used. The general syntax is:

```
w=Entry(master, option=value)
```

master is the parameter used to represent the parent window. There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

4. CheckButton

To select any number of options by displaying a number of options to a user as toggle buttons. The general syntax is:

```
w = CheckButton(master, option=value)
```

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas.

5. RadioButton

It is used to offer multi-choice option to the user. It offers several options to the user and the user has to choose one option. The general syntax is:

```
w = RadioButton(master, option=value)
```

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas.

6. Listbox

It offers a list to the user from which the user can accept any number of options. The general syntax is:

```
w = Listbox(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

7. Scrollbar

It refers to the slide controller which will be used to implement listed widgets. The general syntax is:

w = Scrollbar(master, option=value)
master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

8. Menu

It is used to create all kinds of menus used by the application. The general syntax is:

w = Menu(master, option=value)
master is the parameter used to represent the parent window.

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas.

9. Combobox

Combobox widget is created using the ttk.Combobox class from the tkinter.ttk module. The values for the Combobox are specified using the values parameter. The default value is set using the set method. An event handler function on_select is bound to the Combobox using the bind method, which updates a label with the selected item whenever an item is selected.

10. Scale

It is used to provide a graphical slider that allows to select any value from that scale. The general syntax is:

w = Scale(master, option=value)master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

11. TopLevel

This widget is directly controlled by the window manager. It doesn't need any parent window to work on. The general syntax is:

w = TopLevel(master, option=value)

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

12. Message

It refers to the multi-line and non-editable text. It works same as that of Label. The general syntax is:

w = Message(master, option=value)
master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

13. MenuButton

It is a part of top-down menu which stays on the window all the time. Every menubutton has its own functionality. The general syntax is:

w = MenuButton(master, option=value)
master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

14. Progressbar

Tkinter application with a Progressbar widget and a button to start the progress. When the button is clicked, the progressbar fills up to 100% over a short period, simulating a task that takes time to complete.

15. SpinBox

It is an entry of 'Entry' widget. Here, value can be input by selecting a fixed value of numbers.
The general syntax is:

```
w = SpinBox(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

16. Text

To edit a multi-line text and format the way it has to be displayed. The general syntax is:

```
w =Text(master, option=value)
```

There are number of options which are used to change the format of the text. Number of options can be passed as parameters separated by commas.

17. Canvas

It is used to draw pictures and other complex layout like graphics, text and widgets. The general syntax is:

```
w = Canvas(master, option=value)
```

master is the parameter used to represent the parent window.
There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

18. PannedWindow

It is a container widget which is used to handle number of panes arranged in it. The general syntax is:

```
w = PannedWindow(master, option=value)
```

Master is the parameter used to represent the parent window. There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

Color Option in Tkinter

This example demonstrates the usage of various color options in Tkinter widgets, including active background and foreground colors, background and foreground colors, disabled state colors, and selection colors. Each widget in the example showcases a different color option, providing a visual representation of how these options affect the appearance of the widgets.

Geometry Management

Tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

pack() method
It organizes the widgets in blocks before placing in the parent widget.

grid() method
It organizes the widgets in grid (table-like structure) before placing in the parent widget.

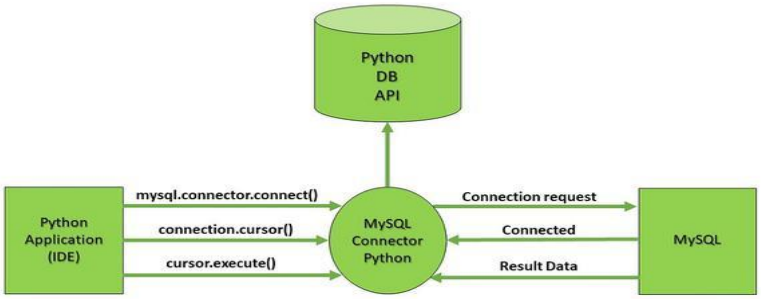
place() method
It organizes the widgets by placing them on specific positions directed by the programmer.

Python MySQL

Python MySQL Connector is a Python driver that helps to integrate Python and MySQL. This Python MySQL library allows the conversion between Python and MySQL data types. MySQL Connector API is implemented using pure Python and does not require any third-party library.

How to Connect Python with SQL Database?

Python is a high-level, general-purpose, and very popular programming language. The diagram given below illustrates how a connection request is sent to MySQL connector Python, how it gets accepted from the database and how the cursor is executed with result data.



SQL connection with Python

Connecting MySQL with Python

To create a connection between the MySQL database and Python, the **connect()** method of **mysql.connector** module is used. We pass the database details like HostName, username, and the password in the method call, and then the method returns the connection object.

The following steps are required to connect SQL with Python:
Step 1: Download and Install the free MySQL database.
Step 2: After installing the MySQL database, open your Command prompt.
Step 3: Navigate your Command prompt to the location of PIP.
Step 4: Now run the commands given below to download and install "MySQL Connector".

Here, mysql.connector statement will help you to communicate with the MySQL database.

Download and install "MySQL Connector"

```
pip install mysql-connector-python
```

Step 5: Test MySQL Connector

To check if the installation was successful, or if you already installed "MySQL Connector", go to your IDE and run the given below code :

```
import mysql.connector
```

If the above code gets executed with no errors, "MySQL Connector" is ready to be used.

Step 6: Create Connection

Now to connect SQL with Python, run the code given below in your IDE.

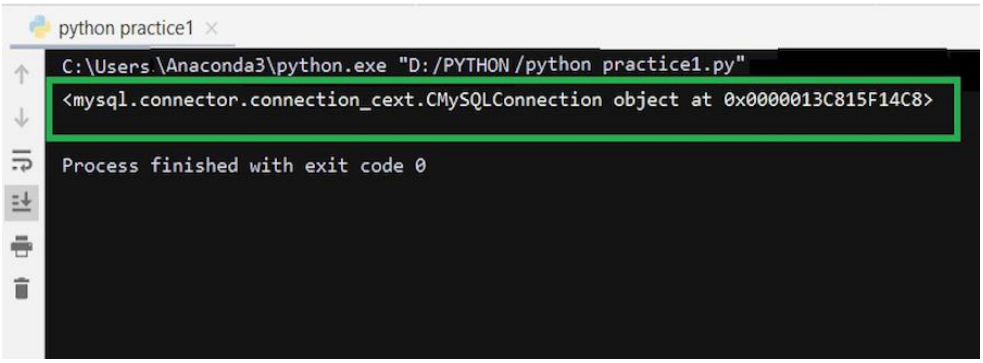
```
# Importing module
import mysql.connector

# Creating connection object
```

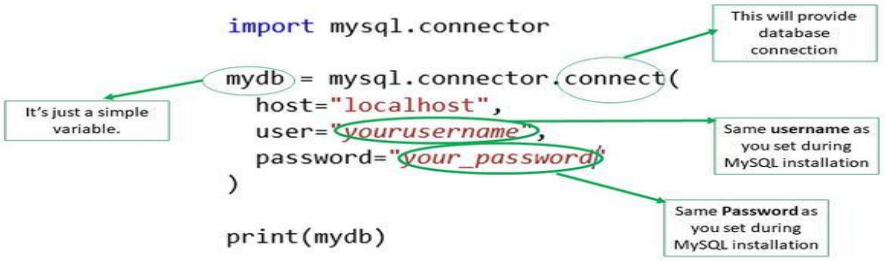
```
mydb = mysql.connector.connect(host = "localhost", user = "yourusername", password = "your_password")

# Printing the connection object
print(mydb)
```

Output:



Here, in the above code:



Creating MySQL Database

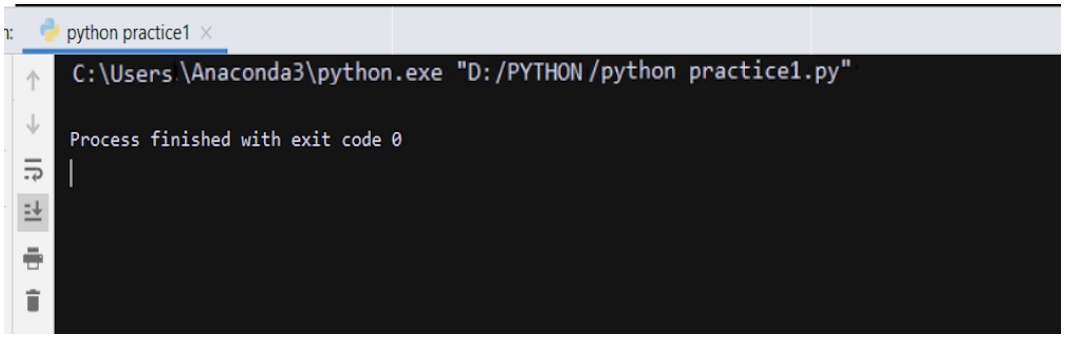
To create a database, we will use CREATE DATABASE database_name statement and we will execute this statement by creating an instance of the 'cursor' class.

```
import mysql.connector
mydb = mysql.connector.connect(host = "localhost", user = "yourusername", password = "your_password")

# Creating an instance of 'cursor' class which is used to execute the 'SQL' statements
cursor = mydb.cursor()

# Creating a database with name 'gphisar'
# execute() method is used to compile a SQL statement
cursor.execute("CREATE DATABASE gphisar")
```

Output:



If the database with the name 'gphisar' already exists then you will get an error, otherwise no error. So make sure that the new database that you are creating does not have the same name as the database already you created or exists previously.

Now to check the databases that you created, use *"SHOW DATABASES" – SQL statement* i.e. cursor.execute("SHOW DATABASES")

```
import mysql.connector
mydb = mysql.connector.connect(host = "localhost", user = "root", password = "1234")

# Creating an instance of 'cursor' class which is used to execute the 'SQL' statements
cursor = mydb.cursor()

# Show database
cursor.execute("SHOW DATABASE")
for x in cursor:
  print(x)
```

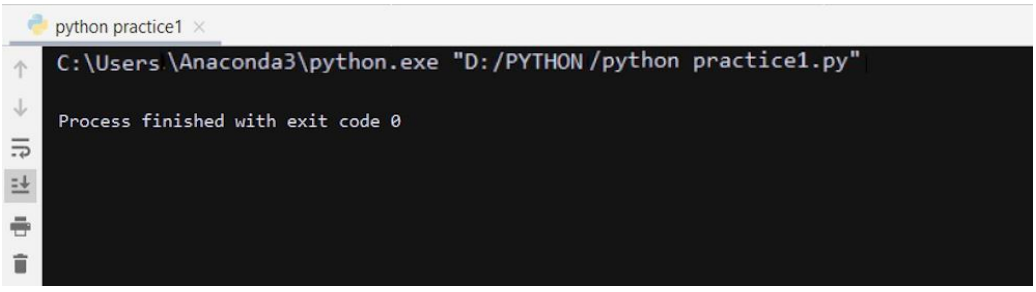
Creating Tables

Now to create tables in a database, first, we have to select a database and for that, we will pass *database = "NameofDatabase"* as your fourth parameter in connect() function. Since we have created a database with the name 'gphisar' above, so we will use that and create our tables. We will use *CREATE TABLE gph (variableName1 datatype, variableName2 datatype)* statement to create our table with the name 'gph'.

```
import mysql.connector
mydb = mysql.connector.connect(host = "localhost", user = "yourusername", password = "your_password", database = "gphisar")
cursor = mydb.cursor()

# Creating a table called 'gph' in the 'gphisar' database
cursor.execute("CREATE TABLE gph (name VARCHAR(255), user_name VARCHAR(255))")
```

Output:

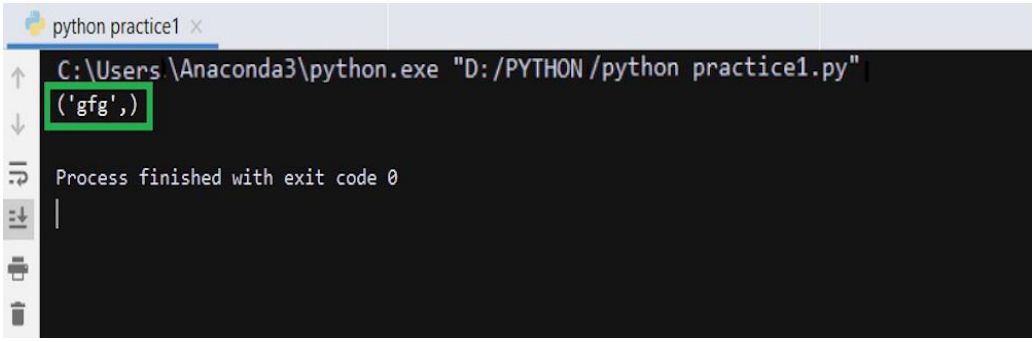


If the table with the name 'gph' already exists, you will get an error, otherwise no error. So make sure that the new table that you are creating does not have the same name as the table already you created or exists previously.
Now to check tables that you created, use *"SHOW TABLES" – SQL statement* i.e. cursor.execute("SHOW TABLES").

```
import mysql.connector
mydb = mysql.connector.connect(host = "localhost", user = "root, password = "1234",  database = "gphisar")
cursor = mydb.cursor()

# Show existing tables
cursor.execute("SHOW TABLES")
for x in cursor:
   print(x)
```

Output:



Notes:

- **mysql.connector** allows Python programs to access MySQL databases.
- **connect()** method of the MySQL Connector class with the arguments will connect to MySQL and would return a MySQLConnection object if the connection is established successfully.
- **user = "yourusername"**, here "yourusername" should be the same username as you set during MySQL installation.
- **password = "your_password"**, here "your_password" should be the same password as you set during MySQL installation.
- **cursor()** is used to execute the SQL statements in Python.
- **execute()** method is used to compile a SQL statement.

How to install MySQL connector package in Python?
MySQL is a Relational Database Management System (RDBMS) whereas the structured Query Language (SQL) is the language used for handling the RDBMS using commands i.e Creating, Inserting, Updating and Deleting the data from the databases.
A connector is employed when we have to use MySQL with other programming languages. The work of mysql-connector is to provide access to MySQL Driver to the required language. Thus, it generates a connection between the programming language and the MySQL Server.

Installation:
To install Python-mysql-connector module, one must have **Python** and **PIP**, preinstalled on their system. To check if your system already contains Python, go through the following instructions:
Open the Command line(search for cmd in the Run dialog(+ **R**).
Now run the following command:

```
python --version
```

If Python is already installed, it will generate a message with the Python version available.

PIP is a package management system used to install and manage software packages/libraries written in Python. These files are stored in a large "on-line repository" termed as Python Package Index (PyPI).

To check if PIP is already installed on your system, just go to the command line and execute the following command:

```
pip -V
```

mysql-connector method can be installed on Windows with the use of following command:

```
pip install mysql-connector-python
```

MySQL Connector module of Python is used to connect MySQL databases with the Python programs, it does that using the Python Database API Specification v2.0 (PEP 249). It uses the Python standard library and has no dependencies.

Connecting to the Database

The mysql.connector provides the connect() method used to create a connection between the MySQL database and the Python application. The syntax is given below.

```
Conn_obj= mysql.connector.connect(host = <hostname>, user = <username>, passwd = <password>)
```

The **connect()** function accepts the following arguments.
Hostname – It represents the server name or IP address on which MySQL is running.
Username – It represents the name of the user that we use to work with the MySQL server. By default, the username for the MySQL database is root.
Password – The password is provided at the time of installing the MySQL database. We don't need to pass a password if we are using the root.
Database – It specifies the database name which we want to connect. This argument is used when we have multiple databases.

Python MySQL – Select Query

Python Database API (Application Program Interface) is the Database interface for the standard Python. To connect with MySQL database server from Python, we need to import the mysql.connector module.

Syntax:

```
CREATE DATABASE DATABASE_NAME
```

Example:

```
# importing required libraries
import mysql.connector
dataBase = mysql.connector.connect(host ="localhost", user ="user", passwd ="gph")

# preparing a cursor object
cursorObject = dataBase.cursor()

# creating database
cursorObject.execute("CREATE DATABASE gphisar")
```

The above program illustrates the creation of MySQL database “gphisar” in which host-name is localhost, the username is user and password is gph.

Let’s suppose we want to create a table in the database, then we need to connect to a database. Below is a program to create a table in the geeks4geeks database which was created in the above program.

```
# importing required library
import mysql.connector

# connecting to the database
dataBase = mysql.connector.connect(host = "localhost", user = "user", passwd = "gph",
                database = "gphisar")

# preparing a cursor object
cursorObject = dataBase.cursor()

# creating table
studentRecord = """CREATE TABLE STUDENT (
                NAME  VARCHAR(20) NOT NULL,
                BRANCH VARCHAR(50),
                ROLL INT NOT NULL,
                SECTION VARCHAR(5),
                AGE INT
                )"""

# table created
cursorObject.execute(studentRecord)

# disconnecting from server
dataBase.close()
```

CRUD Operation in Python using MySQL

We will be seeing how to perform CRUD (CREATE, READ, UPDATE and DELETE) operations in Python using MySQL. For this, we will be using the Python MySQL connector.

In order to perform CRUD operations we need to have a database and a table. We shall first create the database.

We are going to create an employee database named **employee_db** and a table named **tblemployee** which consists of the following columns:

Column name	**Data type**	**Description**
Empid	INT	Stores the employee id and has auto-increment i.e. increments every time a record is added
Empname	VARCHAR(45)	Stores the employee’s name

Column name	Data type	Description
Department	VARCHAR(45)	Stores the department to which the employee belongs i.e. accounts, HR.
Salary	INT	Stores the salary of the employees.

Syntax for creating the Database:

```
CREATE DATABASE <DATABASE_NAME>
```

Creating Table

Now in order to create the table, we use the create table command. It is recommended to always keep a primary key which in this case is empid and helps to uniquely identify the employees.

The general syntax to create a table is:

```
CREATE TABLE
(
   column1 column1_data_type,
   column2 column2_data_type,
   column3 column3_data_type...
);
```

```
# Python implementation to create a table in MySQL
import mysql.connector

# connecting to the mysql server
db = mysql.connector.connect(host="localhost", user="root", passwd="password",
    database="employee_db")

# cursor object c
c = db.cursor()

# create statement for tblemployee
employeetbl_create = """CREATE TABLE `employee_db`.`tblemployee` (
  `empid` INT NOT NULL AUTO_INCREMENT,
  `empname` VARCHAR(45) NULL,
  `department` VARCHAR(45) NULL,
  `salary` INT NULL,
   PRIMARY KEY (`empid`))"""

c.execute(employeetbl_create)
c = db.cursor()

# fetch tblemployee details in the database
c.execute("desc tblemployee")

# print the table details
```

```
for i in c:
   print(i)

# finally closing the database connection
db.close()
```

Python MySQL – Update Query

A connector is employed when we have to use MySQL with other programming languages. The work of MySQL-connector is to provide access to MySQL Driver to the required language. Thus, it generates a connection between the programming language and the MySQL Server.

Update Clause

The update is used to change the existing values in a database. By using update a specific value can be corrected or updated. It only affects the data and not the structure of the table. The basic advantage provided by this command is that it keeps the table accurate.

Syntax:

```
UPDATE tablename
SET ="new value"
WHERE ="old value";
```

```
# Example 1: Python program to demonstrate update clause
import mysql.connector

# Connecting to the Database
mydb = mysql.connector.connect(host ='localhost', database ='College', user ='root')
cs = mydb.cursor()

# update clause
statement ="UPDATE STUDENT SET AGE = 23 WHERE Name ='Rishi Kumar'"
cs.execute(statement)
mydb.commit()

# Disconnecting from the database
mydb.close()

# Example 2: Python program to demonstrate update clause
 import mysql.connector

# Connecting to the Database
mydb = mysql.connector.connect(host ='localhost', database ='College', user ='root')
cs = mydb.cursor()

# update clause
statement ="UPDATE STUDENT SET Name = 'S.K. Anirban' WHERE Name ='SK Anirban'"
cs.execute(statement)
mydb.commit()

# Disconnecting from the database
mydb.close()
```

Select Query

After connecting with the database in MySQL we can select queries from the tables in it. **Syntax:**

- In order to select particular attribute columns from a table, we write the attribute names.

```
SELECT attr1, attr2 FROM table_name
```

- In order to select all the attribute columns from a table, we use the asterisk '*' symbol.

```
SELECT * FROM table_name
```

Inserting Data

Inserting the data into tables is a crucial part, it is required to make sure there is no mismatch of data i.e. the data type being sent should match the data type of the particular column.
The general syntax for insert statements:

```
INSERT INTO <TABLE_NAME> (column1, column2, column3...) VALUES (data1,data2,data3...);
```

We will be inserting multiple rows at one type, however, you can even insert one row at a time. After writing the insert statement, we will be creating a list or collections of row data to be passed. This is to be created right before the executed query.

Since multiple rows will be sent together, we need to use the executemany() function instead of execute().

```
# Python implementation to insert data into a table in MySQL
import mysql.connector
# connecting to the mysql server
db = mysql.connector.connect(
   host="localhost",
   user="root",
   passwd="password",
   database="employee_db"
)
 # cursor object c
c = db.cursor()
 # insert statement for tblemployee
# this statement will enable us to insert multiple rows at once.
employeetbl_insert = """INSERT INTO tblemployee (
  empname, department, salary)
  VALUES  (%s, %s, %s)"""

# we save all the row data to be inserted in a data variable
data = [("Vani", "HR", "100000"),
       ("Krish", "Accounts", "60000"),
       ("Aishwarya", "Sales", "25000"),
       ("Govind", "Marketing", "40000")]

# execute the insert commands for all rows and commit to the database
c.executemany(employeetbl_insert, data)
db.commit()

# finally closing the database connection
db.close()
```

```
# Python implementation to create a table in MySQL
```

```
import mysql.connector

# connecting to the mysql server

db = mysql.connector.connect(
   host="localhost",
   user="root",
   passwd="password",
   database="employee_db"
)

# cursor object c
c = db.cursor()

# create statement for tblemployee
employeetbl_create = """CREATE TABLE `employee_db`.`tblemployee` (
  `empid` INT NOT NULL AUTO_INCREMENT,
  `empname` VARCHAR(45) NULL,
  `department` VARCHAR(45) NULL,
  `salary` INT NULL,
   PRIMARY KEY (`empid`))"""

c.execute(employeetbl_create)

c = db.cursor()

# fetch tblemployee details in the database
c.execute("desc tblemployee")

# print the table details
for i in c:
   print(i)

 # finally closing the database connection
db.close()
```

Output:
tblemployee details are printed

Inserting Data

Inserting the data into tables is a crucial part, it is required to make sure there is no mismatch of data i.e. the data type being sent should match the data type of the particular column.
The general syntax for insert statements:

```
INSERT INTO <TABLE_NAME> (column1,column2,column3...) VALUES (data1,data2,data3...);
```

We will be inserting multiple rows at one type, however, you can even insert one row at a time. After writing the insert statement, we will be creating a list or collections of row data to be passed. This is to be created right before the executed query.

Since multiple rows will be sent together, we need to use the executemany() function instead of execute().

```
# Python implementation to insert data into a table in MySQL
import mysql.connector

# connecting to the mysql server

db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="password",
    database="employee_db"
)

# cursor object c
c = db.cursor()

# insert statement for tblemployee
# this statement will enable us to insert multiple rows at once.
employeetbl_insert = """INSERT INTO tblemployee (
   empname,
   department,
   salary)
   VALUES  (%s, %s, %s)"""

# we save all the row data to be inserted in a data variable
data = [("Vani", "HR", "100000"),
        ("Krish", "Accounts", "60000"),
        ("Aishwarya", "Sales", "25000"),
        ("Govind", "Marketing", "40000")]

# execute the insert commands for all rows and commit to the database
c.executemany(employeetbl_insert, data)
db.commit()

# finally closing the database connection
db.close()
```