# PYTHON

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991. It is mostly used for web development (server-side), software development, mathematics, system scripting.

The most popular use cases for python are as follows:

* Python can be used on a server to create web applications.
* Python can be used alongside software to create workflows.
* Python can connect to database systems. It can also read and modify files.
* Python can be used to handle big data and perform complex mathematics.
* Python can be used for rapid prototyping, or for production-ready software development.

The advantages that python offers over other programming languages are:

* Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
* Python has a simple syntax similar to the English language.
* Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
* Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
* Python can be treated in a procedural way, an object-oriented way or a functional way.

**Install Python on Windows system:**

* Download the latest version of python from the website from https://www.python.org/downloads/
* Choose the suitable OS version and CPU architecture.
* Run the .exe file downloaded and follow the instructions to install Python on your system.

**Install Python on Linux based OS systems:**

* Use the appropriate package manager and run the command:
  + [yum/apt-get/dnf] install python3

## Variables: Variables are containers for storing data values. Python has no command for declaring a variable. A variable is created the moment you first assign a value to it. You can get the data type of a variable with the type() function.

String variables can be declared either by using single or double quotes. Variable names are case-sensitive.

**Casting**

If you want to specify the data type of variable.

x= float(3)

print(x)

output: 3.0

**Type**

To get data type of variable with type() function.

x=5

print(type(x))

output: <class ‘init’>

**Variable Name**

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)
* A variable name cannot be any of the Python keywords..

Mutil word variable names:

Camel case: myVariableName= “Sample”

Pascal case: MyVariableName=”Sample”

Snakecase: my\_variable\_name=”Sample”

**Many values to multiple Variables**

Example:

x, y, z = “Orange” , “banana”, Cherry”

Print(x)

Print(y)

Print(z)

Output:

Orange

Banana

Cherry

One value to Multiple variables: same value to multiple variables

Example:

x = y = z = “orange”

Print(x)

Print(y)

Output:

Orange

Orange

**Unpack a collection:**

If you have a collection of values in a list, tuples etc. Python allows you to extract the values into variables.

Vegetables = [“Cabbage”, “capsicums”, “carrot”]

x, y, z = vegetables

Print(x)

Print(y)

Output:

Cabbage

capsicums

**Output variables**:

In print() function,you output multiple variables, separated by a comma.

x = “Python”

y = “is awesome”

print(x, y)

output: Pythonisawesome

we can also use + in place of comma(,)

print (x + y)

When we try to combine string and a number with + operator , then it through an error.

x = 5

y = “John”

print(x + y)

**Global variables**

Variables that are created outside the functions are known as global variables

Example:

x = "awesome"

def myfunc():

print("Python is " + x)

myfunc()

If we create variable inside function, then it is called as local Variable.

x = "awesome"

def myfunc():

x = "fantastic"

print("Python is " + x)

myfunc()

print("Python is " + x)

# 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, memoryview |
| None Type: | NoneType |

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 |
| x = memoryview(bytes(5)) | memoryview |
| x = None | NoneType |

## Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

|  |  |
| --- | --- |
| **Example** | **Data Type** |
| x = str("Hello World") | str |
| x = int(20) | int |
| x = float(20.5) | float |
| x = complex(1j) | complex |
| x = list(("apple", "banana", "cherry")) | list |
| x = tuple(("apple", "banana", "cherry")) | tuple |
| x = range(6) | range |
| x = dict(name="John", age=36) | dict |
| x = set(("apple", "banana", "cherry")) | set |
| x = frozenset(("apple", "banana", "cherry")) | frozenset |
| x = bool(5) | bool |
| x = bytes(5) | bytes |
| x = bytearray(5) | bytearray |
| x = memoryview(bytes(5)) | memoryview |

We can convert the datatype of a variable by using typecasting.

* int() - constructs an integer number from an integer literal, a float literal (by removing all decimals), 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

**Strings:** Strings in python are surrounded by either single quotation marks, or double quotation marks. 'hello' is the same as "hello".

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

a = "Hello"

print(a)

You can assign a multiline string to a variable by using three quotes:

a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""  
print(a)

Strings in Python are arrays of bytes representing unicode characters. Python does not have a character datatype; a single character is simply a string with a length of 1.

Get the character at position 1:

a = "Hello, World!"

print(a[1])

output: e

To concatenate, or combine, two strings you can use the + operator. Python has a set of built-in methods that you can use on strings.

|  |  |
| --- | --- |
| **Method** | **Description** |
| capitalize() | Converts the first character to upper case |
| casefold() | Converts string into lower case |
| center() | Returns a centered string |
| count() | Returns the number of times a specified value occurs in a string |
| encode() | Returns an encoded version of the string |
| endswith() | Returns true if the string ends with the specified value |
| expandtabs() | Sets the tab size of the string |
| find() | Searches the string for a specified value and returns the position of where it was found |
| format() | Formats specified values in a string |
| format\_map() | Formats specified values in a string |
| index() | Searches the string for a specified value and returns the position of where it was found |
| isalnum() | Returns True if all characters in the string are alphanumeric |
| isalpha() | Returns True if all characters in the string are in the alphabet |
| isdecimal() | Returns True if all characters in the string are decimals |
| isdigit() | Returns True if all characters in the string are digits |
| isidentifier() | Returns True if the string is an identifier |
| islower() | Returns True if all characters in the string are lower case |
| isnumeric() | Returns True if all characters in the string are numeric |
| isprintable() | Returns True if all characters in the string are printable |
| isspace() | Returns True if all characters in the string are whitespaces |
| istitle() | Returns True if the string follows the rules of a title |
| isupper() | Returns True if all characters in the string are upper case |
| join() | Joins the elements of an iterable to the end of the string |
| ljust() | Returns a left justified version of the string |
| lower() | Converts a string into lower case |
| lstrip() | Returns a left trim version of the string |
| maketrans() | Returns a translation table to be used in translations |
| partition() | Returns a tuple where the string is parted into three parts |
| replace() | Returns a string where a specified value is replaced with a specified value |
| rfind() | Searches the string for a specified value and returns the last position of where it was found |
| rindex() | Searches the string for a specified value and returns the last position of where it was found |
| rjust() | Returns a right justified version of the string |
| rpartition() | Returns a tuple where the string is parted into three parts |
| rsplit() | Splits the string at the specified separator, and returns a list |
| rstrip() | Returns a right trim version of the string |
| split() | Splits the string at the specified separator, and returns a list |
| splitlines() | Splits the string at line breaks and returns a list |
| startswith() | Returns true if the string starts with the specified value |
| strip() | Returns a trimmed version of the string |
| swapcase() | Swaps cases, lower case becomes upper case and vice versa |
| title() | Converts the first character of each word to upper case |
| translate() | Returns a translated string |
| upper() | Converts a string into upper case |
| zfill() | Fills the string with a specified number of 0 values at the beginning |

**Booleans**: Booleans represent one of two values: True or False. You can evaluate any expression in Python, and get one of two answers, True or False. Almost any value is evaluated to True if it has some sort of content. Any string is True, except empty strings. Any number is True, except 0. Any list, tuple, set, and dictionary are True, except empty ones.

**Arithmetic Operators:**

Arithmetic operators are used with numeric values to perform common mathematical operations:

|  |  |  |
| --- | --- | --- |
| Operator | Name | Example |
| + | Addition | x + y |
| - | Subtraction | x - y |
| \* | Multiplication | x \* y |
| / | Division | x / y |
| % | Modulus | x % y |
| \*\* | Exponentiation | x \*\* y |
| // | Floor division | x // y |

## Assignment Operators:

Assignment operators are used to assign values to variables:

|  |  |  |
| --- | --- | --- |
| Operator | Example | Same As |
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| \*= | x \*= 3 | x = x \* 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| //= | x //= 3 | x = x // 3 |
| \*\*= | x \*\*= 3 | x = x \*\* 3 |
| &= | x &= 3 | x = x & 3 |
| |= | x |= 3 | x = x | 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

**Comparison Operators:**

Comparison operators are used to compare two values:

|  |  |  |
| --- | --- | --- |
| Operator | Name | Example |
| == | Equal | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

**Logical Operators:**

Logical operators are used to combine conditional statements:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | Returns True if both statements are true | x < 5 and  x < 10 |
| or | Returns True if one of the statements is true | x < 5 or x < 4 |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x < 10) |

**Identity Operators:**

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | Returns True if both variables are the same object | x is y |
| is not | Returns True if both variables are not the same object | x is not y |

**Membership Operators:**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Returns True if a sequence with the specified value is present in the object | x in y |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y |

**Bitwise Operators:**

Bitwise operators are used to compare (binary) numbers:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |
| & | AND | Sets each bit to 1 if both bits are 1 | x & y |
| | | OR | Sets each bit to 1 if one of two bits is 1 | x | y |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 | x ^ y |
| ~ | NOT | Inverts all the bits | ~x |
| << | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off | x << 2 |
| >> | Signed right shift | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off | x >> 2 |

**Operator Precedence:**

Operator precedence describes the order in which operations are performed. The precedence order is described in the table below, starting with the highest precedence at the top:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| () | Parentheses |
| \*\* | Exponentiation |
| +x  -x  ~x | Unary plus, unary minus, and bitwise NOT |
| \*  /  //  % | Multiplication, division, floor division, and modulus |
| +  - | Addition and subtraction |
| <<  >> | Bitwise left and right shifts |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| | | Bitwise OR |
| ==  !=  >  >=  <  <=  is  is not  in  not in | Comparisons, identity, and membership operators |
| not | Logical NOT |
| and | AND |
| or | OR |

**Python Keywords:**

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers.

We can use the keyword module to get the list of python keywords:

import keyword

print(keyword.kwlist)

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| and | A logical operator |
| as | To create an alias |
| assert | For debugging |
| break | To break out of a loop |
| class | To define a class |
| continue | To continue to the next iteration of a loop |
| def | To define a function |
| del | To delete an object |
| elif | Used in conditional statements, same as else if |
| else | Used in conditional statements |
| except | Used with exceptions, what to do when an exception occurs |
| false | Boolean value, result of comparison operations |
| finally | Used with exceptions, a block of code that will be executed no matter if there is an exception or not |
| for | To create a for loop |
| from | To import specific parts of a module |
| global | To declare a global variable |
| if | To make a conditional statement |
| import | To import a module |
| in | To check if a value is present in a list, tuple, etc. |
| is | To test if two variables are equal |
| lambda | To create an anonymous function |
| none | Represents a null value |
| nonlocal | To declare a non-local variable |
| not | A logical operator |
| or | A logical operator |
| pass | A null statement, a statement that will do nothing |
| raise | To raise an exception |
| return | To exit a function and return a value |
| true | Boolean value, result of comparison operations |
| try | To make a try...except statement |
| while | To create a while loop |
| with | Used to simplify exception handling |
| yield | To end a function, returns a generator |

**List:**

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage. Lists are created using square brackets:

* thislist = ["apple", "banana", "cherry"]

Python has a set of built-in methods that you can use on lists.

|  |  |
| --- | --- |
| **Method** | **Description** |
| append() | Adds an element at the end of the list |
| clear() | Removes all the elements from the list |
| copy() | Returns a copy of the list |
| count() | Returns the number of elements with the specified value |
| extend() | Add the elements of a list (or any iterable), to the end of the current list |
| index() | Returns the index of the first element with the specified value |
| insert() | Adds an element at the specified position |
| pop() | Removes the element at the specified position |
| remove() | Removes the item with the specified value |
| reverse() | Reverses the order of the list |
| sort() | Sorts the list |

1. Sort list alphanumerically

List objects have a sort() method that will sort the list alphanumerically, ascending, by default:

thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]

thislist.sort()

print(thislist)

1. Sort descending

thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]

thislist.sort(reverse = True)

print(thislist)

1. Reverse order

The reverse() method reverses the current sorting order of the elements.

thislist = ["banana", "Orange", "Kiwi", "cherry"]

thislist.reverse()

print(thislist)

**Copy lists**:

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

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

mylist = thislist.copy()

print(mylist)

**Join lists**

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

list1 = ["a", "b", "c"]

list2 = [1, 2, 3]

list3 = list1 + list2

print(list3)

**Tuple:**

Tuples are used to store multiple items in a single variable. Tuple items are ordered, ***unchangeable***, and allow duplicate values. Tuples are written with round brackets.

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

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

* [**List**](https://www.w3schools.com/python/python_lists.asp) is a collection which is ordered and changeable. Allows duplicate members.
* **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
* **[Set](https://www.w3schools.com/python/python_sets.asp)** is a collection which is unordered, changeable, and unindexed. ***No duplicate members***.
* [**Dictionary**](https://www.w3schools.com/python/python_dictionaries.asp) is a collection which is ordered\*\* and changeable. No duplicate members.

To determine if a specified item is present in a tuple use the in keyword:

* if "apple" in thistuple:

Python has two built-in methods that you can use on tuples:

|  |  |
| --- | --- |
| **Method** | **Description** |
| count() | Returns the number of times a specified value occurs in a tuple |
| index() | Searches the tuple for a specified value and returns the position of where it was found |

**tuple with data types**

tuple1 = ("apple", "banana", "cherry")

tuple2 = (1, 5, 7, 9, 3)

tuple3 = (True, False, False)

Change Tuple value:

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

x = ("apple", "banana", "cherry")

y = list(x)

y[1] = "kiwi"

x = tuple(y)

print(x)

**Loop tuples**

Loop through a Tuple

For loop

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

for x in thistuple:

print(x)

for i in range (len(thistuple)):

print(thistuple[i]) # Loop through index number

**While loop**

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

i = 0

while i < len(thistuple):

print(thistuple[i])

i = i + 1

Join Tuples

To join use “+” Operator

tuple1 = ("a", "b" , "c")

tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2

print(tuple3)

Multiply tuples

To multiply , use “\*” operator

fruits = ("apple", "banana", "cherry")

mytuple = fruits \* 2

print(mytuple)

**Set:**

Sets are used to store multiple items in a single variable. A set is a collection which is unordered, unchangeable\*, unindexed and do not allow duplicate values. Sets are written with curly brackets.

* thisset = {"apple", "banana", "cherry"}

Set items are unchangeable, but you can remove items and add new items. Python has a set of built-in methods that you can use on sets.

|  |  |
| --- | --- |
| **Method** | **Description** |
| add() | Adds an element to the set |
| clear() | Removes all the elements from the set |
| copy() | Returns a copy of the set |
| difference() | Returns a set containing the difference between two or more sets |
| difference\_update() | Removes the items in this set that are also included in another, specified set |
| discard() | Remove the specified item |
| intersection() | Returns a set, that is the intersection of two other sets |
| intersection\_update() | Removes the items in this set that are not present in other, specified set(s) |
| isdisjoint() | Returns whether two sets have a intersection or not |
| issubset() | Returns whether another set contains this set or not |
| issuperset() | Returns whether this set contains another set or not |
| pop() | Removes an element from the set |
| remove() | Removes the specified element |
| symmetric\_difference() | Returns a set with the symmetric differences of two sets |
| symmetric\_difference\_update() | inserts the symmetric differences from this set and another |
| union() | Return a set containing the union of sets |
| update() | Update the set with the union of this set and others |

**Access items:**

With for loop

thisset = {"apple", "banana", "cherry"}

for x in thisset:

print(x)

#or

print("banana" in thisset)

**Add set item**

Add items: Once a set is created, you cannot change its items, but you can add new items.

thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical) == > add sets

mylist = [ “Kiwi”, “Orange”]

thisset.update(mylist) #add any Iterable

**Join two sets**

There are several ways to join two or more sets in python.

1. union()

set1 = {"a", "b" , "c"}

set2 = {1, 2, 3}

set3 = set1.union(set2)

1. update()

set1.update(set2)

1. Keep ONLY Duplicates

It keeps only the items that are present in both sets.

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

x.intersection\_update(y)

print(x)

output: apple

**Keep ALL But NOT Duplicate**

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

x.symmetric\_difference\_update(y)

#or

z= x.symmetric\_difference(y)

## Dictionary:

Dictionaries are used to store data values in key:value pairs. A dictionary is a collection which is ordered\*, changeable and do not allow duplicates. Dictionaries are written with curly brackets, and have keys and values:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

You can access the items of a dictionary by referring to its key name, inside square brackets:

* x = thisdict["model"]

Python has a set of built-in methods that you can use on dictionaries.

|  |  |
| --- | --- |
| **Method** | **Description** |
| clear() | Removes all the elements from the dictionary |
| copy() | Returns a copy of the dictionary |
| fromkeys() | Returns a dictionary with the specified keys and value |
| get() | Returns the value of the specified key |
| items() | Returns a list containing a tuple for each key value pair |
| keys() | Returns a list containing the dictionary's keys |
| pop() | Removes the element with the specified key |
| popitem() | Removes the last inserted key-value pair |
| setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| update() | Updates the dictionary with the specified key-value pairs |
| values() | Returns a list of all the values in the dictionary |

**The dict() constructor:**It is possible by using dict() constructor

Thisdict = dict(name = “John”, age = 28, country = “India”)

**Access items**

x= thisdict.keys() == > list of all keys

Example:

car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change

x = thisdict.values() == > to get values

x =thisdict.items() == > Get items in a dictionary

for “model” in thisdict:

thisdict.update({“year” : 2020})

thisdict.pop(“model”) ==> remove item within specified key name

thisdict.popitem() == > remove the last inserted item

del thisdict[“model”]

thisdict.clear() == > empties directory.

**Loop Dictionaries**

For x in thisdict: or for x in thisdict.key():

Print(x) == > print all key

For x in thisdict.values(): == > print all values

Print(thisdict[x]) == > print all values

For x, y in thisdict.items():

Print(x, y) == > loop through both key and values.

**Copy dictionaries**

Makes a copy of a directory with the copy() method.

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

mydict = thisdict.copy()

print(mydict)

## Conditions and If statements:

Python supports the usual logical conditions from mathematics. Python relies on indentation (whitespace at the beginning of a line) to define scope in the code.

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")

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

* print("A") if a > b else print("B")

You can also have multiple else statements on the same line:

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

**Python Loops:**

Python has two primitive loop commands: **while** loops and **for** loops. With the **while** loop we can execute a set of statements as long as a condition is true.

i = 1

while i < 6:

print(i)

i += 1

**Break statement**: We can stop the loop even if the while condition is true.

i = 1

while i < 6:

print(i)

if i == 3:

break

i += 1

**Continous statement**: With continuous we can stop current iteration and continue with the next.

i = 0

while i < 6:

i += 1

if i == 3:

continue

print(i)

**Else** **statement**: we can run a block of code once when the condition no longer is true.

i = 1

while i < 6:

print(i)

i += 1

else:

print("i is no longer less than 6")

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

fruits = ["apple", "banana", "cherry"]

for x in fruits:

print(x)

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.

for x in range(6):

print(x)

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

for x in range(6):

print(x)

else:

print("Finally finished!")

The else block will NOT be executed if the loop is stopped by a break statement.

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

**Break statement**

With the break statement we can stop the loop before it has looped through all the items.

fruits = ["apple", "banana", "cherry"]

for x in fruits:

if x == "banana":

break

print(x)

**Continous statement**

With the continue statement we can stop the current iteration of the loop, and continue with the next.

fruits = ["apple", "banana", "cherry"]

for x in fruits:

if x == "banana":

continue

print(x)

**The Range function**

To loop through a set of code a specified number of times, we can use the range() function.

for x in range(6):

print(x)

for x in range(2, 6):

for x in range(2, 30, 3) == > Increment by sequence with 3

# 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. Python a function is defined using the def keyword:

def my\_function():

print("Hello from a function")

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

* **my\_function()**

**Arguments:** Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses.

* def my\_function(fname)

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.

* def my\_function(\*kids)

You can also send arguments with the key = value syntax. This way the order of the arguments does not matter.

* my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

If you do not know how many keyword arguments that will be passed into your function, add two asterisks: \*\* before the parameter name in the function definition.

* my\_function(fname = "Tobias", lname = "Refsnes")

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

def my\_function(country = "Norway")

my\_function()

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.

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

def my\_function(x):

return 5 \* x

The pass statement are functions definitions cannot be empty, but if you for some reasons have a function definition with no content, put in the pass statement to avoid getting an error.

def my\_function():

pass

**Recursion**

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. Python also accepts function recursion, which means a defined function can call itself.

def tri\_recursion(k):

if(k > 0):

result = k + tri\_recursion(k - 1)

print(result)

else:

result = 0

return result

print("\n\nRecursion Example Results")

tri\_recursion(6)

**Object Oriented Programming:**

In Python, Object-Oriented Programming (OOP) is a programming paradigm that allows you to define objects and classes that encapsulate data and behavior.

An object is an instance of a class, which is a blueprint for creating objects. The class defines the properties and methods that are common to all objects of that class.

The four main principles of OOP in Python are:

**Encapsulation**: This is the process of hiding implementation details and exposing only the necessary information to the user. In Python, you can use access modifiers such as public, private, and protected to encapsulate data.

**Inheritance**: This is the process of creating a new class from an existing class. The new class inherits the properties and methods of the existing class and can add new properties and methods.

**Polymorphism**: This is the ability of objects to take on multiple forms. In Python, you can use polymorphism to write functions that can take different types of objects as parameters.

**Abstraction**: This is the process of focusing on the essential features of an object and ignoring the non-essential ones. In Python, you can use abstract classes and interfaces to achieve abstraction

**FAQ’s**

**Q) Define class and object?**

A) **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.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, age.

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. Eg.: Myclass.Myattribute

**Syntax**:

class ClassName:

# Statement-1

.

.

# Statement-N

**Object:** 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.

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.

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**: Creating an object

obj = Dog()

This will create an object named obj of the class Dog defined above. Before diving deep into objects and class let us understand some basic keywords that will we used while working with objects and classes.

**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

# Driver code

# 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 Class and objects with methods:**

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))

# Driver code

# Object instantiation

Rodger = Dog("Rodger")

Tommy = Dog("Tommy")

# Accessing class methods

Rodger.speak()

Tommy.speak()

**Q) What are \_\_init\_\_ and \_\_str\_\_ in python?**

A)The \_\_init\_\_ Constructor are used to initializing the object’s state. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. 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.

# A 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 the above example, a person name Nikhil is created. While creating a person, “Nikhil” is passed as an argument, this argument will be passed to the \_\_init\_\_ method to initialize the object. The keyword self represents the instance of a class and binds the attributes with the given arguments. Similarly, many objects of the Person class can be created by passing different names as arguments.

\_\_str\_\_: \_\_str\_\_ is a method that returns a string representation of the object. It is called when you use the str() function on the object or when you use the object in a string context (e.g., printing the object).

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def \_\_str\_\_(self):

return f"{self.name} ({self.age})"

person = Person("John", 25)

print(person) # Output: John (25)

In this example, we defined a \_\_str\_\_ method that returns a string representation of the object person. When we print the object person, the \_\_str\_\_ method is called and returns the string "John (25)".

**Q) Explain the keyword del in Python OOPs context?**

A) In Python, del is a keyword used to delete an object or a reference to an object. When you delete an object, Python frees up the memory that was allocated to that object.

To delete an object, you can use the del keyword followed by the name of the object:

my\_list = [1, 2, 3, 4]

del my\_list

In this example, we created a list object my\_list with four elements. We then deleted the object using the del keyword.

You can also use del to delete a specific element from a list or a dictionary:

my\_list = [1, 2, 3, 4]

del my\_list[2]

print(my\_list) # Output: [1, 2, 4]

my\_dict = {"a": 1, "b": 2, "c": 3}

del my\_dict["b"]

print(my\_dict) # Output: {"a": 1, "c": 3}

In these examples, we used del to delete the third element of the list my\_list and the key-value pair with the key "b" from the dictionary my\_dict.

**Q) How can we achieve inheritance?**

A) In Python, you can achieve inheritance by creating a new class that inherits the properties and methods of an existing class. The new class is called the child class or the subclass, and the existing class is called the parent class or the superclass.

To create a child class, you define a new class and include the name of the parent class in parentheses after the name of the child class. You can then use the super() function to call the methods of the parent class.

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def make\_sound(self):

print("The animal makes a sound")

class Dog(Animal):

def \_\_init\_\_(self, name, breed):

super().\_\_init\_\_(name)

self.breed = breed

def make\_sound(self):

print("The dog barks")

dog = Dog("Fido", "Labrador")

print(dog.name) # Output: Fido

print(dog.breed) # Output: Labrador

dog.make\_sound() # Output: The dog barks

In this example, we defined an Animal class with an \_\_init\_\_ method that takes a name parameter and a make\_sound method. We then defined a Dog class that inherits from the Animal class and overrides the make\_sound method. The Dog class also has an additional breed attribute.

To call the \_\_init\_\_ method of the parent class in the Dog class, we use the super() function in the \_\_init\_\_ method. We pass the current class (Dog) and the instance (self) as arguments to super().

Q) Explain the different types of variables in Python in OOPs context?

A) In object-oriented programming (OOP) in Python, there are three types of variables:

**Instance variables:** These are variables that are defined inside a class and belong to a specific instance of that class. They can be accessed using the dot notation. Instance variables are unique to each instance of the class.

class Person:

def \_\_init\_\_(self, name, age):

self.name = name # instance variable

self.age = age # instance variable

person1 = Person("John", 25)

person2 = Person("Jane", 30)

print(person1.name) # Output: John

print(person2.name) # Output: Jane

In this example, name and age are instance variables defined inside the Person class. They belong to a specific instance of the Person class and can be accessed using the dot notation. person1 and person2 are two different instances of the Person class, so they have different values for their name instance variable.

**Class variables:** These are variables that are defined inside a class and belong to the class itself, rather than any specific instance of the class. They can be accessed using the class name. Class variables are shared by all variables by all instances of the class.

class Person:

population = 0 # class variable

def \_\_init\_\_(self, name, age):

self.name = name # instance variable

self.age = age # instance variable

Person.population += 1

person1 = Person("John", 25)

person2 = Person("Jane", 30)

print(Person.population) # Output: 2

In this example, population is a class variable defined inside the Person class. It belongs to the class itself, rather than any specific instance of the class. We can access it using the class name. population is incremented each time a new instance of the Person class is created.

**Local variables:** These are variables that are defined inside a method or function and can only be accessed within that method or function.

class Person:

def say\_hello(self):

message = "Hello" # local variable

print(message)

person = Person()

person.say\_hello() # Output: Hello

In this example, message is a local variable defined inside the say\_hello method. It can only be accessed within that method. When we call the say\_hello method on an instance of the Person class, it prints out the value of the message variable.

Q) What is private, protected and public in classes?

A) In object-oriented programming, Python has three levels of class attribute and method visibility:

Public: By default, all class attributes and methods are public, which means they can be accessed from outside the class.

class Person:

def \_\_init\_\_(self, name):

self.name = name # public attribute

person = Person("John")

print(person.name) # Output: John

In this example, name is a public attribute of the Person class. We can access it from outside the class using the dot notation.

Protected: In Python, a protected attribute or method is denoted by a single underscore prefix. This means that the attribute or method is intended for internal use only, but it can still be accessed from outside the class.

class Person:

def \_\_init\_\_(self, name):

self.\_name = name # protected attribute

person = Person("John")

print(person.\_name) # Output: John

In this example, \_name is a protected attribute of the Person class. It is intended for internal use only, but we can still access it from outside the class using the dot notation.

Private: In Python, a private attribute or method is denoted by a double underscore prefix. This means that the attribute or method is intended to be used only within the class itself, and cannot be accessed from outside the class.

class Person:

def \_\_init\_\_(self, name):

self.\_\_name = name # private attribute

person = Person("John")

print(person.\_\_name) # Output: AttributeError: 'Person' object has no attribute '\_\_name'

In this example, \_\_name is a private attribute of the Person class. It is intended to be used only within the class itself, and cannot be accessed from outside the class. When we try to access it from outside the class using the dot notation, we get an AttributeError.

Q) What is enum in python? Write one example.

A) An enum is a class that represents a group of named constants. Enums allow you to define a set of possible values that a variable can take, which can make your code more expressive and easier to read.

from enum import Enum

class Day(Enum):

MONDAY = 1

TUESDAY = 2

WEDNESDAY = 3

THURSDAY = 4

FRIDAY = 5

SATURDAY = 6

SUNDAY = 7

def is\_weekday(day):

if day in [Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY]:

return True

else:

return False

print(is\_weekday(Day.MONDAY)) # Output: True

print(is\_weekday(Day.SATURDAY)) # Output: False

**OS Module in Python:**

The OS module in Python provides functions for interacting with the operating system. OS comes under Python’s standard utility modules. This module provides a portable way of using operating system-dependent functionality.

**Current working directory(CWD):** Whenever the files are called only by their name, Python assumes that it starts in the CWD which means that name-only reference will be successful only if the file is in the Python’s CWD.

Note: The folder where the Python script is running is known as the Current Directory. This is not the path where the Python script is located.

Getting the Current working directory

To get the location of the current working directory os.getcwd() is used.

# Python program to explain os.getcwd() method

# importing os module

import os

# Get the current working

# directory (CWD)

cwd = os.getcwd()

# Print the current working

# directory (CWD)

print("Current working directory:", cwd)

**Changing current working directory:** To change the current working directory(CWD) os.chdir() method is used. This method changes the CWD to a specified path. It only takes a single argument as a new directory path.

# Python program to change the

# current working directory

import os

# Function to Get the current

# working directory

def current\_path():

print("Current working directory before")

print(os.getcwd())

print()

# Driver's code

# Printing CWD before

current\_path()

# Changing the CWD

os.chdir('../')

# Printing CWD after

current\_path()

**Creating a directory:** Using os.mkdir(): os.mkdir() method in Python is used to create a directory named path with the specified numeric mode. This method raises FileExistsError if the directory to be created already exists.

os.makedirs() method in Python is used to create a directory recursively. That means while making leaf directory if any intermediate-level directory is missing, os.makedirs() method will create them all.

**Listing out files and Directories:** os.listdir() method in Python is used to get the list of all files and directories in the specified directory. If we don’t specify any directory, then the list of files and directories in the current working directory will be returned.

# Python program to explain os.listdir() method

# importing os module

import os

# Get the list of all files and directories

# in the root directory

path = "/"

dir\_list = os.listdir(path)

print("Files and directories in '", path, "' :")

# print the list

print(dir\_list)

**Deleting Directory or files:** Using os.remove():os.remove() method in Python is used to remove or delete a file path. This method can not remove or delete a directory. If the specified path is a directory then OSError will be raised by the method.

# Python program to explain os.remove() method

# importing os module

import os

# File name

file = 'file1.txt'

# File location

location = "D:/Pycharm projects/GeeksforGeeks/Authors/Nikhil/"

# Path

path = os.path.join(location, file)

# Remove the file

# 'file.txt'

os.remove(path)

using os.rmdir():os.rmdir() method in Python is used to remove or delete an empty directory. OSError will be raised if the specified path is not an empty directory.

# Python program to explain os.rmdir() method

# importing os module

import os

# Directory name

directory = "Geeks"

# Parent Directory

parent = "D:/Pycharm projects/"

# Path

path = os.path.join(parent, directory)

# Remove the Directory

# "Geeks"

os.rmdir(path)

**File Operations in Python:**

To open a file in python we can use one of the following commands:

* f = open(“filename.txt”, “r”)
* with open(“filename.txt”. “r”) as f:

Using with open is recommended as it will automatically close the file after the operations.

When we use the first method, we need to close the file explicitly by using f.close() line.

* f.close()

The file can be opened in one of the following modes, which is specified by second argument in the open() method:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition, you can specify if the file should be handled as binary or text mode:

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Since “rt” is the default mode, open(“filename.txt”) will be same as open(“filename.txt”, “rt”)

**Example:**

#file1 = open("airtravel.csv","r")

filedata = []

with open("airtravel.csv", "r") as file1:

for line in file1:

data = line.split(',')

for i, x in enumerate(data):

y = ''

for letter in x:

if letter != '"':

y = y + letter

data[i] = y

filedata.append(data)

for lines in data:

print(lines, end='\t')

#file1.close()

month = input("Enter a month to query:")

for data in filedata:

if month in data:

print("Data for the month of :", month)

print("1958",":", data[1])

print("1959",":", data[2])

print("1960", ":", data[3])

break

else:

print("Invalid month")

**Output:**

****

**Error Handling in Python:**

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:

#The try block will generate an error, 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.

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:

#The try block will generate a NameError, because x is not defined:

try:

print(x)

except NameError:

print("Variable x is not defined")

except:

print("Something else went wrong")

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

#The try block does not raise any errors, so the else block is executed:

try:

print("Hello")

except:

print("Something went wrong")

else:

print("Nothing went wrong")

We can also define multiple except blocks for each type of error. Further an exception can be raised manually by ‘raise exception’ keyword

x = -1

if x < 0:

raise Exception("Sorry, no numbers below zero")

**Q) What are some popular Python frameworks?**

A. Python has several popular frameworks for web development, such as Flask and Django. Flask is a lightweight and flexible framework, while Django is a more comprehensive framework that follows the "batteries-included" philosophy.

**Q) Can I create a GUI (Graphical User Interface) application in Python?**

A. Yes, Python has several libraries for creating GUI applications, such as Tkinter, PyQt, and PySide. Tkinter comes bundled with Python, making it the most straightforward choice for simple GUIs.

**Q) What is the Python Standard Library?**

A. The Python Standard Library is a collection of modules and packages that come included with Python. It provides a wide range of functionality, including file I/O, networking, regular expressions, data types, math utilities, and more. It saves you from having to implement many common tasks from scratch.

**Q) What is the difference between Python 2 and Python 3?**

A. Python 2 and Python 3 are not fully compatible with each other, which led to the creation of Python 3. Some key differences include:

* Print statement: In Python 2, it is **print "Hello"**, while in Python 3, it's **print("Hello")**.
* Integer division: In Python 2, dividing two integers results in an integer, but in Python 3, it gives a float.
* Unicode: Python 3 handles text as Unicode by default, while Python 2 uses ASCII for strings.
* xrange: In Python 2, **xrange()** is used for range iteration, while Python 3's **range()** behaves like Python 2's **xrange()**.

**ASSIGMENTS:**

1. Write a Python program that converts temperatures between Celsius and Fahrenheit. The program should ask the user to enter a temperature along with the unit (Celsius or Fahrenheit) and then convert and display the temperature in the other unit.

2. Write a Python program that reads a text file and counts the frequency of each word. The program should then display the ten most frequent words along with their counts.

3. Write a Python function that takes a string as input and returns True if it is a palindrome (reads the same backward as forwards), and False otherwise.

4. Write a Python program to manage a contact book. The program should allow users to add contacts with names and phone numbers, search for a contact by name, and display the full contact list.

5. Write a Python function that takes two matrices as input and performs matrix multiplication. The function should return the resulting matrix.

6. Write a Python function that generates the first n numbers of the Fibonacci sequence. The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones.