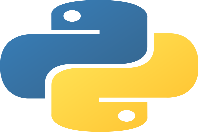
* **Python** 

**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)
* No whitespace is allowed within the variable name.
* A variable name must not be any reserved word or keyword, e.g. int, goto, etc.

**\_\_ is a special kind of variable which stores value of previous expression.**

20 + 10 – 8 \* 5 \

- 6 + 9 // 10 + \

7 \*\* 3 – 1

**Here \ can be used to continue previous formulae or expression in next line.**

**Dunder** or **magic methods** in [Python](https://www.geeksforgeeks.org/python-programming-language/) are the methods having two prefix and suffix underscores in the method name. Dunder here means “Double Under (Underscores)”. These are commonly used for operator overloading. Few examples for magic methods are: **\_\_init\_\_, \_\_add\_\_, \_\_len\_\_, \_\_repr\_\_** etc.

**LOCAL VARIABLE: -** **A variable with a local scope can be accessed only within the function/block that it is created in**. When a variable is created inside the function/block, the variable becomes local to it. A local variable only exists while the function is executing.

**GLOBAL VARIABLES are variables declared outside a function.** **Local variables are variables declared inside a function**. While global variables cannot be directly changed in a function, you can use the global keyword to create a function that will change the value of a global variable.

a **=** 10 *# global varible*

**def** random():

a **=** 50 *# local variable -> Only accessible within the block*

print("Inside Random1 -", end **=** " ")

print(a)

The **globals() and locals()** functions **can be used to return the names in the global and local namespaces depending on the location from where they are called**. If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

# program to illustrate access modifiers of a class

# super class

**class** Super:

**# public data member \*\*\* never initiate with any \_ undercsore**

     var1 **=** None

**# protected data member \*\*\* use single \_ underscore**

     \_var2 **=** None

**# private data member \*\*\* use double \_\_ underscore**

     \_\_var3 **=** None

**# constructor always gets called automatically when any object is created.**

**def** \_\_init\_\_(self, var1, var2, var3):

          self.var1 **=** var1

          self.\_var2 **=** var2

          self.\_\_var3 **=** var3

    # public member function

**def** displayPublicMembers(self):

          # accessing public data members

**print**("Public Data Member: ", self.var1)

     # protected member function

**def** \_displayProtectedMembers(self):

          # accessing protected data members

**print**("Protected Data Member: ", self.\_var2)

     # private member function

**def** \_\_displayPrivateMembers(self):

          # accessing private data members

**print**("Private Data Member: ", self.\_\_var3)

     # public member function

**def** accessPrivateMembers(self):

          # accessing private member function

          self.\_\_displayPrivateMembers()

# derived class

**class** Sub(Super):

      # constructor

**def** \_\_init\_\_(self, var1, var2, var3):

                Super.\_\_init\_\_(self, var1, var2, var3)

      # public member function

**def** accessProtectedMembers(self):

                # accessing protected member functions of super class

                self.\_displayProtectedMembers()

# creating objects of the derived class

obj **=** Sub("Geeks", 4, "Geeks !")

# calling public member functions of the class

obj.displayPublicMembers()

obj.accessProtectedMembers()

obj.accessPrivateMembers()

**# Object can access protected member**

print("Object is accessing protected member:", obj.\_var2)

**# object can not access private member, so it will generate Attribute error**

**#print(obj.\_\_var3)**

**Type Conversion**

**# 1. float()**

**# 2. bool()**

**# 3. str()**

**# 4. int()**

**Arithmetic Operators**

**print(5 + 2)**

**print(5 - 2)**

**print(5 \* 2)**

**print(5 / 2)**

**print( 5 // 2)**

**print(5 % 2)**

**print(5 \*\* 2)**

**If statements**

a = 330  
b = 330

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

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

**if condition1:**

**this**

**elif condition2:**

**this**

**elif condition3:**

**this**

**else:**

**this**

**Range in Python**

range() function returns a range object that is a sequence of numbers.

**range(1,10) *# -> [start, end) (start , end , jump)***

bin(56) # will return '0b111000'

bin() # It converts only integer => binary

int(0b111000) # will return 56 again

int() converts anything to integer if possible

**While Loop**

*# Initialization*

number **=** 1

*# Condition*

**while** number **<=** 5:

*# Task / Logic goes here*

print(number)

*# Updation*

number **+=** 1

\*\*\* With the**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")

**For Loop** (to iterate over a list)

for i in range(5):

   print(i)

Enumerate() in Python  
  
Enumerate() method **adds a counter to an iterable and returns it in a form of enumerating object**.

**for i,j in enumerate(list):**

**print(i,j)**

# This will give output as = (index , value) where i=index j=value of entire list

**Python zip() Function**The **zip() function** returns a zip object, which is an **iterator of tuples** where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc**.**

**\*It zips till the size of smallest list(any iterable) so different length don’t matter**

a = ("John", "Charles", "Mike")

b = ("Jenny", "Christy", "Monica", "Vicky")

print(set(zip(a, b))) # {('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica')}

x = zip(a, b)

print(dict(x)) # {'John': 'Jenny', 'Charles': 'Christy', 'Mike': 'Monica'}

c='yfghgjfukggjhg'

d='aaaaaaaaaa'

print(list(zip(d,c))) # [('a', 'y'), ('a', 'f'), ('a', 'g'), ('a', 'h'), ('a', 'g'), ('a', 'j'), ('a', 'f'), ('a', 'u'), ('a', 'k'), ('a', 'g')]

Else in For Loop

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

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

for x in range(6):  
  print(x)  
else:  
  print("Finally finished!")

[**List**](https://www.w3schools.com/python/python_lists.asp) is a collection which is ordered and changeable. Allows duplicate members.

[**Tuple**](https://www.w3schools.com/python/python_tuples.asp) is a collection which is ordered and unchangeable. Allows duplicate members.

**Set** is a collection which is unordered, unchangeable\*, 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.

\*Set items are unchangeable, but you can remove items and add new items.

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

**Lists**

List is a complex data type in Python.

Del list[0] will delete the index item in the list

Del list will delete whole list

List.Clear() will delete entire list

list(range(10)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**List slicing :-**

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

**Output** = ["apple", "blackcurrant", "watermelon", "orange", "kiwi", "mango"]

**List[ : : ]*=>[start=0: end=len(list): +1] (start , end , jump)***

*# Default end value = len(a) -> step size = +1*

*# Default end value = covering the first element -> step size = -1*

**List[ : :-1] *=> [start=0: end= : +1] (start , end , jump)***

## List Comprehension :-

fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
newlist = [x for x in fruits if "a" in x] # appending in a new list

**The Syntax :-**

newlist = [*expression* for *item* in *iterable* if *condition* == True]

newlist = [x if x != "banana" else "orange" for x in fruits]

New List = **[** **[**  **item to be appended** ( can be anything )  **]** **for loop** **if condition** **]**

A = **[ [** i\*j for j in range(11) if j!=0 **]** for i in range(10) if i!=0 **]**

A = **[ [** i\*j for j in range(1,11) **]** for i in range(1,int(input())+1) **]**

**List Methods :**

|  |  |
| --- | --- |
| Method | Description |
| [append()](https://www.w3schools.com/python/ref_list_append.asp) | Adds an element at the end of the list |
| [clear()](https://www.w3schools.com/python/ref_list_clear.asp) | Removes all the elements from the list |
| [copy()](https://www.w3schools.com/python/ref_list_copy.asp) | Returns a copy of the list |
| [count()](https://www.w3schools.com/python/ref_list_count.asp) | Returns the number of elements with the specified value |
| [extend()](https://www.w3schools.com/python/ref_list_extend.asp) | Add the elements of a list (or any iterable), to the end of the current list |
| [index()](https://www.w3schools.com/python/ref_list_index.asp) | Returns the index of the first element with the specified value |
| [insert()](https://www.w3schools.com/python/ref_list_insert.asp) | Adds an element at the specified position |
| [pop()](https://www.w3schools.com/python/ref_list_pop.asp) | Removes the element at the specified position |
| [remove()](https://www.w3schools.com/python/ref_list_remove.asp) | Removes the item with the specified value |
| [reverse()](https://www.w3schools.com/python/ref_list_reverse.asp) | Reverses the order of the list |
| [sort()](https://www.w3schools.com/python/ref_list_sort.asp) | Sorts the list |

**Split & join :-**

The **split()** method **splits a string into a list**. <string>.split(sep,maxsplit)

**Str.split()**  # default sep = ‘ ‘ || (space)

**my\_string.split(",")** || **a.split(sep=',')** # Using comma to seperate

my\_string.split("," , 2) # Using maxsplit

The **join()** method creates and returns a new string by concatenating all of the elements in an array (or an array-like object), separated by commas or a specified separator string.

**", ".join(my\_list)**

**<sep>.join(<iterable>)**

**Break & Continue & Pass**

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

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

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

for x in [0, 1, 2]:  
  pass

students = ["ram", "shyam", "kishan", "radha", "radhika"]

for student in students:

   if(student == "radha"):

       break

   print(student)

for student in students:

   if(student == "kishan"):

       continue

   print(student)

**Tuples**

They are like lists (sequence of objects) but they are **immutable** i.e. once they have been defined we cannot change them.

Parenthesis in tuples are optional.

Only 2 methods count() , index()

marks = (95, 98, 97, 97)

#marks[0] = 98

print(marks.count(97))

print(marks.index(97))

**Packing – Unpacking: -**

a,b,c = 1,2,3 || a,b,c=[1,3,5] || a,b,c = list || a,b,c = “def”

a,b,c = “ abcdef” where a=”a” , b=”b” , c=”c”

a,b,c,**\*d** = [1,2,3,4,5,6,7] where a=1 , b=2 , c=3 , **d=[4,5,6,7]**

**\*variable will store all remaining values.**

a,b,c,**\*d**,e,f = [1,2,3,4,5,6,7,8] where a=1, b=2, c=3, **d=[ 4,5,6 ]**, e=7, f=8

**Strings**

chr(38) *# chr converts ASCII value to a string*

ord("a") *# ord function gives the ASCII value of a character*

1. **split** -> "this is a string".split(" ") -> string to a list
2. **join** -> " ".join(["list", "of", "strings"]) -> list of strings to a string
3. **replace** -> "this is a string".replace(" ", "\_") -> replaces all occurences of the first character by the second character. **IMP** - CREATES A NEW STRING
4. **find** -> "this is a string".find(" a ") -> Finds the exact sequence or substring in the original string and returns the starting index of that substring.
5. **count** -> "this is a string".count("a") -> Counts the number of times a character or substring is present within a string
6. **isdigit** -> True if content inside the string is only digits.
7. **isalpha** -> True if content inside the string is only alphabets.
8. **islower** / **isupper** -> True if content inside the string is all lower case / all upper case
9. **isspace** -> True if the character is a space
10. **lower** / **upper** -> Converts a string to lowercase / uppercase

**F-String**

**print(f"Roll Number - {roll\_no} is {name}!")**

**print(f"string to print - {variables} string to print {values}!")**

**Sets**

Sets are a collection of all unique elements.

Indexing is not supported in sets.

marks = {98, 97, 95, 95}

print(marks)

for score in marks:

   print(score)

we cannot initialise empty set s = { } will be an empty dictionary

s=set() is only way to initialise empty set

**a = {{1,2,3,4,5}} *# sets can't have sets, lists and dictionaries within it***

**Set** is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.

\*Set items are unchangeable, but you can remove items and add new items.

**Method for sets:-**

**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

**Dictionary**

Dictionary is an ordered collection of Items. Dictionary stores a (key, value) pair.

marks = {"math" : 99, "chemistry" : 98, "physics" : 97}

print(marks)

print(marks["chemistry"])

marks["english"] = 95

print(marks)

marks["math"] = 96

print(marks)

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

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

The keys() method will return a **list of all the keys** in the dictionary.

The values() method will return a **list of all the values** in the dictionary.

The items() method will return each item in a dictionary, as tuples in a list.

**[ (key, value), (key, value), (key, value), (key, value), (key, value), (key, value) ] fromkeys() -  Returns a dictionary with the specified keys and value setdefault() - Returns the value of the specified key. If the key does not exist: insert the key, with the specified value**

**Dict1 | Dict2 = Union or add or update**

**Dict1 | Dict2 = Union or add or update**

**Dict1 | Dict2 = Union or add or update**

**Dict1 | Dict2 = Union or add or update**

**Functions in Python :-**

def my\_function():  
  print("Hello from a function")

*#* **Functions can accept arguments or parameters**

**def** sheldon\_knock(name):

print(name)

print("knock knock knock", name)

print("knock knock knock", name)

print("knock knock knock", name)

# Calling a function

sheldon\_knock("Penny")

**What are the 4 types of arguments in Python?**

**In Python, we have the following 4 types of function arguments.**

* Positional arguments # Normal arguments def my\_function(a,b,c):
  + An argument is a variable, value or object passed to a function or method as input. **Positional arguments are arguments that need to be included in the proper position or order.** The first positional argument always needs to be listed first when the function is called.
* Default argument # Gives Default Values def my\_function(a=1,b=2,c=0):
  + **Default values indicate that the function argument will take that value if no argument value is passed during the function call**.
* Arbitrary arguments (\*args) # Multiple arguments def my\_function(a,\*b):
  + Arbitrary arguments **allow us to pass a varying number of values during a function call**.
  + # Always use **\*args** as variable for passing Arbitrary argument def my\_function(a,**\*args**):
* Keyword arguments (\*\*kwargs) # Multiple Keyworded arguments

def my\_function(\*\*kwargs):

* + If you do not know how **many keyword arguments** that will be passed into your function, add two asterisk: \*\* before the parameter name in the function definition.
  + **This way the function will receive a dictionary of arguments**, and can access the items accordingly:
  + # Always use **\*kwargs** as variable for passing Arbitrary Keyword argument def my\_function(a,**\*kwargs**):

def my\_function(\*\*kwargs):

for i , j in kwargs.items():    **# unpacking a dictionary**

print(i , j)

for i in kwargs:              **# iterating over a dictionary**

print(i , kwargs[i])

# Calling a function

my\_function ( a = 'aaa' , b = 'bbb' , c ='ccc' , d = 'ddd' )

Output = a aaa b bbb c ccc d ddd a aaa b bbb c ccc d ddd

**lambda function :-**

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

**lambda arguments : expression**

x = lambda a : a + 10  
print(x(5))

-----------------------------------------------------------

x = lambda a, b, c : a + b + c  
print(x(5, 6, 2))

**Functional Programing in Python :-**

**Oops: -**

class Created\_a\_Class:

  variable1 = 'common variable => can be accessed by both class and objects and also inside functions of the class'

  def function1(self):

    print('Function 1 called')

    return self.variable1

  def class\_method():

    print("class\_method can be only accesible using class name")

    return "self is udsed to use variables inside functions once defined in class"

  def instance\_method(self):

    print("instance\_method can be accessible using object olny and not by class name")

    print(self.variable1)

object2=Created\_a\_Class()

object1=Created\_a\_Class()

object1.function1()

object1.instance\_method()

Created\_a\_Class.class\_method()

object1.variable1

object2.variable1='this will create a different instance variable with same name for this objest only'

object2.instance\_method()

**\_\_init\_\_ function:-**

**class** Student:

**def** \_\_init\_\_(self, custom\_name):

self**.**name **=** custom\_name

**class** Student:

**def** hello(): *# class method -> Can not be used by objects*

print("Hello!")

Student**.**hello() *# Can only be called using the class name itself*

a **=** Student() # here student is class & a is object

a**.**hello() *# Student.hello(a)*

'''

In case of a mutable Data structure ->

Updating it from any object will update the class variable directly!

'''

**class** BankAccount:

**def** \_\_init\_\_(self, opening\_balance):

self**.**\_\_balance **=** opening\_balance

*# using \_\_ creates a private property (variable)*

*# this variable is only accessible by class methods*

*# this variable now can't be accessed using class objects*

b2**.**\_BankAccount\_\_balance *# NAME MANGLING*

*# python adds prefix of \_ClassName to private variables*

**INHERITENCE**

*# super().\_\_init\_\_(a) # \_\_init\_\_ method of the first class gets called*

**class** SchoolMember:

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

self**.**name **=** name

**class** Student(SchoolMember):

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

self**.**grade **=** grade

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

**class** Staff(SchoolMember):

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

self**.**salary **=** salary

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

**class** Teacher(Staff):

**def** \_\_init\_\_(self, name, salary, subject):

self**.**subject **=** subject

*# call init of parent class*

*# Staff.\_\_init\_\_(self, name, salary)*

*# super() -> Reference to the immediate parent class*

super()**.**\_\_init\_\_(name, salary)

**File Handling: -**

# Working with Files

## File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

* **Read Only (‘r’)** : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
* **Read and Write (‘r+’)** : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
* **Write Only (‘w’)** : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
* **Write and Read (‘w+’)** : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
* **Append Only (‘a’)** : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
* **Append and Read (‘a+’)** : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

### Opening a file *# any\_variable\_name = open()*

**file = open("sample.txt", "r")**

here:

**file** = **variable** (eg:- a, b, c, d, i, j)

**open( )** = **function** to open ant text file

**“sample.txt”** = **name of text file**

**“r”** = is **mode** in which the text file is opened

(eg:- “r” “r+” “w” “w+” “a” “a+” )

**File.close()**  # verry important to close all the opened files

### Reading from a file

read

readline

readlines

file1**.**read()

file**.**readlines() ['Line 1\n', 'Line 2\n', 'Line 3\n']

file**.**readline() 'Line 1\n'

### **Smarter way of opening files...**

With the "with" statement, you get better syntax and exceptions handling.

"The with statement simplifies exception handling by encapsulating common preparation and cleanup tasks."

In addition, it will automatically close the file. The with statement provides a way for ensuring that a clean-up is always used.

**with** open("sample2.txt", "r+") **as** file1:

print(file1**.**read(5))

file1**.**seek(0)

print(file1**.**read())

file**.**read() *# this does not work*

**with** open("sample4.txt", "r+") **as** file:

**while** **True**:

chunk **=** file**.**read(20)

**if** **not** chunk:

**break**

print(chunk)

### Moving the cursor

seek(n) : takes the file read handle to the nth byte from the beginning

file**.**seek(5)

*# Cursor moved to right after character number 5*

### Writing to a file

* write
* writelines

file**.**write("Hello everyone. I am writing this text to the file!")

file**.**writelines(["Line 1\n", "Line 2\n", "Line 3\n"])