Python

* r’…’ 🡪 raw string / raw file location
* array 🡪
  + -1; -2 --- last
  + [x:y] 🡪 x(included) y(excluded)
  + [:x] 🡪 start to x-1
  + [y:] 🡪 y(included) to end
  + [:-x] 🡪 start to count ‘x’ from last ‘x’ excluded
  + [:-x:] 🡪 count last ‘x’ included to end
* s[:i] + s[i:] 🡪 ‘s’
* s[:] 🡪 ‘s’
* s[:] = [] 🡪 empty list
* input(“…”) 🡪 int(), float(), str()
* comments 🡪 #
* len(array)
* array.append(value)
* type(variable)
* del x 🡪 delete variable
* swap 🡪 a,b=b,a
* keywords
  + import keyword
  + keyword.kwlist
* **Python Keywords by Category**
  + **Value Keywords**
    - True
    - False
    - None
  + **Operator Keywords**
    - and
    - or
    - not
    - in
    - is
  + **Control Flow Keywords**
    - if
    - else
    - elif
    - for
    - while
    - break
    - continue
    - pass
    - try
    - except
    - finally
    - raise
    - assert
  + **Function and Class Keywords**
    - def
    - return
    - lambda
    - yield
    - class
  + **Context Management Keywords**
    - with
    - as
  + **Import and Module Keywords**
    - import
    - from
    - as
  + **Scope and Namespace Keywords**
    - global
    - nonlocal
  + **Async Programming Keywords**
    - async
    - await
* False == 0
* True == 1
* None != 0
* None != []
* ‘in’ keyword (membership operator) – Check if a value exists in a sequence (like a list, tuple, or string). It returns True if value is found.
* ‘is’ keyword – Check if two variables point to the same object in memory. It returns True if the objects are identical.
* **if | else | elif**
* **for | while**
* break: “break” is used to control the flow of the loop. The statement is used to break out of the loop and passes the control to the statement following immediately after loop.
* continue: “continue” is also used to control the flow of code. The keyword skips the current iteration of the loop but does not end the loop.
* pass keyword: “pass” is the null statement in python. Nothing happens when this is encountered. This is used to prevent indentation errors and used as a placeholder. The code contains a for loop that iterates 10 times with a placeholder statement ‘pass', indicating no specific action is taken within the loop. 🡪 used in empty loop
* .py 🡪 source code (python interpreter language)
* .pyc 🡪 byte code (complied file)
* Errors

**ValueError** - Correct type, inappropriate value. | x = int("hello")  
**TypeError** - Operation/function applied to the wrong type. | x = 5 + "hello"  
**SyntaxError** - Code structure invalid, grammar rules broken.  
**NameError** - Variable or function not defined. | print(y) # y is not defined  
**IndexError** - Accessing a list/sequence index out of range. | no specified index value in the array  
**KeyError** - Accessing a non-existent key in a dictionary. No specified key in the dic  
**AttributeError** - Object has no attribute or method being accessed. | x = 10 🡪 x.append(5)  
**ZeroDivisionError** - Division or modulo by zero. | x = 5/0  
**ImportError** - Module or function not found during import.  
**IndentationError** - Improper indentation in code.

* [try :](https://www.geeksforgeeks.org/python-try-except) This keyword is used for exception handling, used to catch the errors in the code using the keyword except. Code in “try” block is checked, if there is any type of error, except block is executed.
* [except :](https://www.geeksforgeeks.org/python-try-except) As explained above, this works with “try” to catch exceptions.
* [finally :](https://www.geeksforgeeks.org/finally-keyword-in-python) No matter what is result of the “try” block, “finally” is always executed.
* **raise:** We can raise an exception explicitly with the raise keyword
* [assert:](https://www.geeksforgeeks.org/python-assert-keyword) This function is used for **debugging purposes**. Usually used to check the correctness of code. If a statement is evaluated to be true, nothing happens but when it is false, ” **AssertionError**” is raised. One can also **print a message with the error, separated by a comma**.

🡪 [**del**](https://www.geeksforgeeks.org/python-del-to-delete-objects) is used to delete a reference to an object. Any variable or list value can be deleted using del.

* [**def keyword**](https://www.geeksforgeeks.org/python-def-keyword/)**–**Defines a function named fun using the def keyword. When the function is called using fun().
* [**class**](https://www.geeksforgeeks.org/python-classes-and-objects) keyword is used to declare user defined classes.  
  This code defines a Python class named **Dog**with two class attributes, **attr1** and **attr2**.

**Return Keywords – Return, Yield**

* [return :](https://www.geeksforgeeks.org/python-return-statement) This keyword is used to return from the function.
* [yield :](https://www.geeksforgeeks.org/python-yield-keyword) This keyword is used like return statement but is used to return a generator.

**Return and Yield Keyword use in Python**

The **‘return'** keyword is used to return a final result from a function, and it exits the function immediately. In contrast, the ‘**yield'** keyword is used to create a generator, and it allows the function to yield multiple values without exiting. When ‘**return'** is used, it returns a single value and ends the function, while ‘**yield'** returns multiple values one at a time and keeps the function’s state.

* + [**Lambda**](https://www.geeksforgeeks.org/python-lambda) keyword is used to make inline returning functions with no statements allowed internally.
* Walrus Operator 🡪 The **walrus operator (:=)** is a syntax for assigning variables within expressions in Python.
* newList.append(num) if num != 0 else (input\_ := False)
* input\_ = True  
  newList=[]  
  while(input\_):  
   num = int(input("Give me a num: "))  
   newList.append(num) if num != 0 else (input\_ := False)  
  print(newList)

lambda 🡪 lambda arguments: expression ------------------- anonymous function

* result = (lambda x: (lambda y: x + y)(x\*\*2))(3)
* def complex\_calculation(x):  
   *# Perform some complex operations*  
   return x\*\*2 + 2\*x + 1  
  result = (lambda x: complex\_calculation(x))(5)
* numbers = [1, 2, 3, 4]

squared\_numbers = list(map(lambda x: x \*\* 2, numbers))

print(squared\_numbers) # Output: [1, 4, 9, 16]

* numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers) # Output: [2, 4, 6]

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

sorted\_words = sorted(words, key=lambda word: len(word))

print(sorted\_words) # Output: ['apple', 'cherry', 'banana']

* def combinator(y):

return (lambda x: lambda y: x(y))(lambda x:y)

combinator(lambda x:x\*10)(11)(12)

* alphabet\_dict = lambda: {chr(97 + i): i + 1 for i in range(26)}

print(alphabet\_dict())

* alphabet\_dict = lambda: {chr(i): i - 96 for i in range(97, 123)}

print(alphabet\_dict())

Keywords

* with 🡪 file\_path
* as 🡪 import
* import & from
* global variable 🡪 use global keyword
* nonlocal variable (enclosing nearest outer function) 🡪 use global keyword
* async & await 🡪 import asyncio
* pass 🡪 empty function or empty class
* format 🡪

print("Amount: ${:.2f}".format(amount:=6.54321))

print("Hello, {}! Your score is {:.1f}.".format(name, score))

* end parameter 🡪 in print statement any symbol
* sep parameter 🡪 in print statement any symbol or space or no space
* f”…” string 🡪
* %

%d –integer

%f – float

%s – string

%x –hexadecimal

%o – octal

* .split() 🡪 with space or any specified symbol or character
* type(variable) 🡪 gives type int, float, str, dict, list, tuple, complex, bool, range, set, nonetype, bytes, bytearray,
* typecast 🡪 change values int to float to str to …
* input() function
* output 🡪 print()
* f*ormat output in several ways:*

***Using the format() method****:*

***Using f-strings (Python 3.6+)****:*

***Using the % operator****:*

* alignment 🡪 center, ljust, rjust
* quit() & exit() 🡪 terminate python (system exit)
* inputimeout 🡪 take input within given time limit
* ways of taking input

sys.stdin

input()

fileinput.input()

* input() vs raw\_input()
* a = list(map(int, input().split())) 🡪 multiple inputs and return ‘a’ as list
* input()[0] 🡪 takes only 1st character
* input()[i] 🡪 takes only ‘i’ character
* input()[-1] 🡪 last character

**Operators 🡪**

* Arithmetic 🡪 + - \* / % // \*\*
* Relational 🡪 = > < >= <= == != (here, != and <> are same)
* Logical 🡪 and not or
* Bitwise 🡪 & | >> << ~ ^ (only on int)
* Assignment 🡪 = += -+ \*= /= %=
* **is** True if the operands are identical   
  **is not** True if the operands are not identical
* **in** True if value is found in the sequence  
  **not in** True if value is not found in the sequence
* in 🡪 operator.contains() method alternative
* is 🡪 access memory location (== says only values are equal)
* ternary operator 🡪 ***Syntax:*** *value\_if\_true if condition else value\_if\_false*

tuple 🡪 ***Syntax:*** *(condition\_is\_false, condition\_is\_true)[condition]*

Dict 🡪 ***Syntax:*** *condition\_dict = {True: value\_if\_true, False: value\_if\_false}[condition]*

Lambda 🡪 ***Syntax:*** *lambda x: value\_if\_true if condition else value\_if\_false*

* walrus operator 🡪 **Syntax:** a := expression

assign value to a variable within expression

* Dict 🡪

dict1.update(dict2) 🡪 merge 2nd dict into 1st dict

**\*\*** in dict 🡪 merge 2 dict and put it in 3rd dict --- eg: dict3 **=** {**\*\***dict1, **\*\***dict2}

(|) and (|=) operators 🡪 merge dict

* ***Syntax :*** *f'{expr=}'****Return :*** *Return the formatted string.* 🡪 gives value or length
* Membership operator 🡪 in, not in
* Identity operator 🡪 is, not is

**Data Types**

* **Numeric – int, float, complex(a+bj)**
* **Sequence Type – string, list, tuple**
* **Mapping Type – dict**
* **Boolean – bool**
* **Set Type – set, frozenset**
* **Binary Types –**[**bytes**](https://www.geeksforgeeks.org/python-bytes-method/)**,**[**bytearray**](https://www.geeksforgeeks.org/python-bytearray-function/)**,**[**memoryview**](https://www.geeksforgeeks.org/memoryview-in-python/)
* List – mutable 🡪 []
* Tuple – ordered & immutable 🡪 () --- access tuple convert to list and use it
* Set – unordered & mutable & no duplicate item 🡪 {}
* Dict 🡪 key & value pair --- keys are immutable & can’t be repeated
* complex 🡪 .conjugate(), .real, .imag
* random 🡪

import random

random.randint(a,b) 🡪 range

random.uniform(a,b) 🡪 float range

* nan 🡪 import math | math.nan
* float(+inf), float(-inf) 🡪 both are + and - infinity

**Python String**

* default values are str (eg: numbers) and strings are immutable
* left to right 🡪 index from 0
* right to left 🡪 negative indexing start from -1
* slicing 🡪 s[:] --- complete string

s[start:end] --- end🡪excluded, start🡪included

s[:]==s[::]==s==s[::1] = "1234567890"

s[::-1] 🡪 reverse string 0987654321

s[::-2] 🡪 08642

s[::3] 🡪 1470

* substring 🡪 s[start:end:step]

default

start 0

end --- last

step 1

* deleting string 🡪 del s
* string methods

s.replace(“old”,”new”)

len(s)

s.upper()

s.lower()

s.capitalize()

s.title()

s.swapcase()

s.strip() 🡪 remove whitespace start and end --- also called trim

s.strip(“some char”) 🡪 remove char start and end

concatenating 🡪 + --- merge strings

repeating 🡪 \* --- repeat ‘n’ no. of times

s.lstrip() 🡪 only start or left

s.rstrip() 🡪 only end or right

format()

f”….”

* '\n' --> Leaves a line
* '\t' --> Leaves a space
* Mutable 🡪 List, Dict, Set
* Immutable 🡪 tuple, frozensets, strings, int, float, complex numbers & dict-keys
* **Ways to Deal with Immutability**

String Slicing and Reassembling

String Concatenation

Using the join() method

Using String Formatting

Converting to Mutable Data Structures (like string to list)

* join() 🡪 *string*.join(*iterable*)

“#”.join(some\_string\_array) --- each value in arrays are joined using ‘#’

* “in” keyword 🡪 check substring
* find() method also used to check substring
* reverse string

'#'.join(reversed(s)) 🡪 reverse and insert # between each char

* **Lexicographical Comparison**

kind of sorting with characters use >, <

* (s.startswith("hello"))
* (s.endswith("world"))
* ***find(substring)****: Returns the lowest index in the string where the substring is found.*
* ***strip()****: Removes any leading and trailing characters (space is the default).*
* ***replace(old, new)****: Replaces occurrences of a substring within the string.*
* ***split(delimiter)****: Splits the string at the specified delimiter and returns a list of substrings.*
* ***join(iterable)****: Concatenates elements of an iterable with a specified separator.*
* ***startswith(prefix)****: Checks if the string starts with the specified prefix.*
* ***endswith(suffix)****: Checks if the string ends with the specified suffix.*
* zip() 🡪 merge multiple strings can be displayed or accessed as list,set,tuple,dict
* map 🡪 map(function, \*iterables)

my\_strings = ['a', 'b', 'c', 'd', 'e']

my\_numbers = [1, 2, 3, 4, 5]

results = dict(map(lambda x, y: (x, y), my\_strings, my\_numbers))

* + above use dict or list or set or tuple
* filter 🡪 filter(function, iterable)

scores = [66, 90, 68, 59, 76, 60, 88, 74, 81, 65]

def is\_A\_student(score):

return score > 75

over\_75 = list(filter(is\_A\_student, scores))

--- only true values only printed or stored

* reduce 🡪 reduce(func, iterable[, initial])

from functools import reduce

numbers = [100,250,350]

def custom\_sum(first, second):

return first + second

result = reduce(custom\_sum, numbers, 10)

print(result) #710

from functools import reduce  
num = [1,2,3,10,100]  
x=num[0]  
val=lambda x: reduce(lambda a, b: a \* b, num)  
print(val(x)) #6000

* String Methods 🡪

Function Name Description

capitalize() Converts the first character of the string to a capital (uppercase) letter

casefold() Implements caseless string matching

center() Pad the string with the specified character.

count() Returns the number of occurrences of a substring in the string.

encode() Encodes strings with the specified encoded scheme

endswith() Returns “True” if a string ends with the given suffix

expandtabs() Specifies the amount of space to be substituted with the “\t” symbol in the string

find() Returns the lowest index of the substring if it is found

format() Formats the string for printing it to console

format\_map() Formats specified values in a string using a dictionary

index() Returns the position of the first occurrence of a substring in a string

isalnum() Checks whether all the characters in a given string is alphanumeric or not

isalpha() Returns “True” if all characters in the string are alphabets

isdecimal() Returns true if all characters in a string are decimal

isdigit() Returns “True” if all characters in the string are digits

isidentifier() Check whether a string is a valid identifier or not

islower() Checks if all characters in the string are lowercase

isnumeric() Returns “True” if all characters in the string are numeric characters

isprintable() Returns “True” if all characters in the string are printable or the string is empty

isspace() Returns “True” if all characters in the string are whitespace characters

istitle() Returns “True” if the string is a title cased string

isupper() Checks if all characters in the string are uppercase

join() Returns a concatenated String

ljust() Left aligns the string according to the width specified

lower() Converts all uppercase characters in a string into lowercase

lstrip() Returns the string with leading characters removed

maketrans() Returns a translation table

partition() Splits the string at the first occurrence of the separator

replace() Replaces all occurrences of a substring with another substring

rfind() Returns the highest index of the substring

rindex() Returns the highest index of the substring inside the string

rjust() Right aligns the string according to the width specified

rpartition() Split the given string into three parts

rsplit() Split the string from the right by the specified separator

rstrip() Removes trailing characters

splitlines() Split the lines at line boundaries

startswith() Returns “True” if a string starts with the given prefix

strip() Returns the string with both leading and trailing characters

swapcase() Converts all uppercase characters to lowercase and vice versa

title() Convert string to title case

translate() Modify string according to given translation mappings

upper() Converts all lowercase characters in a string into uppercase

zfill() Returns a copy of the string with ‘0’ characters padded to the left side of the string

**List**

* mutable, ordered (index), contains duplicate
* l.append(value) 🡪 adds at the end of the list
* l.extend(l) 🡪 adds multiple elements to the end of the list
* l.reverse() 🡪 reverse list
* l.insert(index,value) 🡪 adds value to the index
* create repeated elements 🡪 l = [7] \* 3 --- l = [7,7,7]
* l[index] = value 🡪 update list (index should be in range)
* l.remove(value) 🡪 remove that value (1st occurrence)
* l.pop() 🡪 [pop()](https://www.geeksforgeeks.org/python-list-pop-method/): Removes the element at a specific index or the last element if no index is specified.

l.pop() 🡪 removes last element & returns popped element

l.pop(index) 🡪 that index value is removed & returns popped element

* del l[index]
* nested list 🡪 matrix
* print([[0 for n in range(n)] for m in range(m)]) 🡪 m\*n matrix with all 0
* [val for val in a] 🡪

[val \*\* 2 for val in a] 🡪 square values in list

[val for val in a if val % 2 == 0] 🡪 even numbers

[val for val in a if val % 2 != 0] 🡪 odd numbers

[i for i in range(10)] 🡪 list of [0 to 9]

[i+1 for i in range(10)] 🡪 list of [1,10]

coordinates = [(x, y) for x in range(2) for y in range(3)]

print(coordinates := [(x, y) for x in range(2) for y in range(3)]) 🡪 [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]

* Flattening list 🡪

mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

res = [val for row in mat for val in row]

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

* ***[expression for item in list[start:stop:step] if condition]***

slicing with or without condition & can be used in nested list also

* *list\_name.reverse()*
* l.count(value) 🡪 returns no. of value (count) in the list
* l.copy() 🡪 for simple list no changes are reflected to either of the list

makes copy and any changes made to one of the list will reflect in other --- for nested list

* l.clear() 🡪 returns [] --- empty list

*the clear() method only removes the elements from the list.*

* ***insert()*** *method adds an element at a specified index.*
* ***append()*** *method adds an element to the end of the list,*
* *effectively equivalent to 🡪 list.insert(len(list), element).*
* dict.items() 🡪 access key and value in dict

for key, value in my\_dict.items():  
 print(key, ":", value)

* l.sort()

*list\_name.sort(key=None, reverse=False)*

*key 🡪*  len, lambda function, str.lower, str.upper, any function,

* iterate list 🡪 get both index and value

for i, val in enumerate(list):

print (i, val)

* [print(val) for val in list]
* Flatten List 🡪 used for nested list
  + 1. x = itertools.chain(\*list)

print([I for I in x])

* + 1. [val for row in mat for val in row]
    2. np.array(list).flatten()
    3. list(chain(\*nested\_list))
* itertools 🡪 zip(\*lists) & itertools.zip\_longest(\*lists)
* find elements in a list
  + 1. if value in list:
    2. any (x == value for x in list)
    3. list.count(value) > 0
    4. list(map(lst1.\_\_getitem\_\_, lst2)) 🡪 list1 is with values & list2 is with indexes
    5. list((itemgetter(**\***lst2)(lst1)))
    6. list(np.array(lst1)[lst2])
    7. index\_map = {index: element for index, element in enumerate(lst1)}

return [index\_map[index] for index in lst2]

* list.index(value) 🡪 returns index of the value

list.index(element, start, end) 🡪 syntax --- start & end are index values

* list(filter(lambda x: x == value, list))
* most frequent element in a list 🡪

max(set(List), key=List.count)

* remove element in a list

list(filter(lambda x: x != value, newList))

* remove multiple elements in a list

list(filter(lambda x: x not in remove\_set, newList))

* The [union()](https://www.geeksforgeeks.org/union-function-python/) method ensures that the resulting list contains unique elements. Here we convert the lists to sets perform a union and convert back to a list. This approach automatically removes duplicates.
  + union\_list = list(set(a).union(b))
  + otherways:
  + union\_list = list(set(a) | set(b))
  + union\_list = list(set([x for x in a + b]))
* # Elements in list1 but not in list2
  + c = list(set(a) - set(b))
  + c = [item for item in a if item not in b]
  + c = list(filter(lambda x: x not in b, a))
* # Check if all elements match
  + flag = all(x == y for x, y in zip(a, b))
* res **=** sorted(test\_list1 **+** test\_list2) 🡪 sort two sorted lists
* unpack list 🡪
  + a, b, c, \*rest = newlist
  + a, \*mid, c = newList
  + \*a, b, c = newList
* List Methods
  + [append()](https://www.geeksforgeeks.org/python-list-append-method/): Adds an element to the end of the list.
  + [copy()](https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/): Returns a shallow copy of the list.
  + [clear()](https://www.geeksforgeeks.org/python-list-clear-method/): Removes all elements from the list.
  + [count()](https://www.geeksforgeeks.org/python-list-count-method/): Returns the number of times a specified element appears in the list.
  + [extend()](https://www.geeksforgeeks.org/python-list-extend-method/): Adds elements from another list to the end of the current list.
  + [index()](https://www.geeksforgeeks.org/python-list-index/): Returns the index of the first occurrence of a specified element.
  + [insert()](https://www.geeksforgeeks.org/python-list-insert/): Inserts an element at a specified position.
  + [pop()](https://www.geeksforgeeks.org/python-list-pop/): Removes and returns the element at the specified position (or the last element if no index is specified).
  + [remove()](https://www.geeksforgeeks.org/python-list-remove/): Removes the first occurrence of a specified element.
  + [reverse()](https://www.geeksforgeeks.org/python-list-reverse/): Reverses the order of the elements in the list.
  + [sort()](https://www.geeksforgeeks.org/python-list-sort-method/): Sorts the list in ascending order (by default).
* array 🡪 same as list but it supports with only collection of same data type
  + import array as arr
  + arr.array('i', [1, 2, 3, 4, 5, 6]) --- ‘i’ int
  + arr.array('d', [2.5, 3.2, 3.3]) --- ‘d’ float
  + all list methods work for array

**Tuple**

* tup = ()
* concatenate, slicing, del 🡪 works
* index( ) Find in the tuple and returns the index of the given value where it’s available
* count( ) Returns the frequency of occurrence of a specified value
* all() Returns true if all element are true or if tuple is empty
* any() return true if any element of the tuple is true. if tuple is empty, return false
* len() Returns length of the tuple or size of the tuple
* enumerate() Returns enumerate object of tuple
* max() return maximum element of given tuple
* min() return minimum element of given tuple
* sum() Sums up the numbers in the tuple
* sorted() input elements in the tuple and return a new sorted list
* tuple() Convert an iterable to a tuple.
* Methods that **cannot** be used for tuples:
  + append(), insert(), remove(), pop(), clear(), sort(), reverse()
* unpacking operator 🡪 \*
  + (\*newList , ) --- output is tuple
  + Unpack it

x = [('a', 1), ('b', 4)]

a, b = zip(\*x)

a = list(a)

b = list(b)

a = ['a', 'b']

b = [1, 4]

|  |  |  |  |
| --- | --- | --- | --- |
| **List** | **Tuple** | **Set** | **Dictionary** |
| A list is a non-homogeneous data structure that stores the elements in columns of a single row or multiple rows. | A Tuple is a non-homogeneous data structure that stores elements in columns of a single row or multiple rows. | The set data structure is non-homogeneous but stores the elements in a single row. | A dictionary is also a non-homogeneous data structure that stores key-value pairs. |
| The list can be represented by [ ] | A tuple can be represented by  ( ) | The set can be represented by { } | The dictionary can be represented by { } |
| The list allows duplicate elements | Tuple allows duplicate elements | The Set will not allow duplicate elements | The dictionary doesn’t allow duplicate keys. |
| The list can be nested among all | A tuple can be nested among all | The set can be nested among all | The dictionary can be nested among all |
| Example: [1, 2, 3, 4, 5] | Example: (1, 2, 3, 4, 5) | Example: {1, 2, 3, 4, 5} | Example: {1: “a”, 2: “b”, 3: “c”, 4: “d”, 5: “e”} |
| A list can be created using the **list()**function | Tuple can be created using the **tuple()** function. | A set can be created using the **set()** function | A dictionary can be created using the **dict()**function. |
| A list is mutable i.e we can make any changes in the list. | A tuple is immutable i.e we can not make any changes in the tuple. | A set is mutable i.e we can make any changes in the set, its elements are not duplicated. | A dictionary is mutable, its Keys are not duplicated. |
| List is ordered | Tuple is ordered | Set is unordered | Dictionary is ordered (Python 3.7 and above) |
| Creating an empty list  l=[] | Creating an empty Tuple  t=() | Creating a set  a=set() b=set(a) | Creating an empty dictionary  d={} |

**Sets**

* {}
* s.add(value)
* s.update([value1, value2, …])
* remove() method removes a specified element from the set. If the element is not present in the set, it raises a KeyError.
* discard() method also removes a specified element from the set. Unlike remove(), if the element is not found, it does not raise an error.
* s.remove(value)
* s.discard(value)
* s.pop() 🡪 pop() method removes and returns an arbitrary element from the set. This means we don’t know which element will be removed. If the set is empty, it raises a KeyError.
* s.clear() 🡪 output set() --- empty set
* Set Methods
  + Function Description
  + add() Adds an element to a set
  + remove() Removes an element from a set. If the element is not present in the set, raise a KeyError
  + clear() Removes all elements form a set
  + copy() Returns a shallow copy of a set
  + pop() Removes and returns an arbitrary set element. Raise KeyError if the set is empty
  + update() Updates a set with the union of itself and others
  + union() Returns the union of sets in a new set
  + difference() Returns the difference of two or more sets as a new set
  + difference\_update() Removes all elements of another set from this set
  + discard() Removes an element from set if it is a member. (Do nothing if the element is not in set)
  + intersection() Returns the intersection of two sets as a new set
  + intersection\_update() Updates the set with the intersection of itself and another
  + isdisjoint() Returns True if two sets have a null intersection
  + issubset() Returns True if another set contains this set
  + issuperset() Returns True if this set contains another set
  + symmetric\_difference() Returns the symmetric difference of two sets as a new set
  + symmetric\_difference\_update() Updates a set with the symmetric difference of itself and another
* *| for union.  
  & for intersection.  
  – for difference  
  ^ for symmetric difference*
* test\_list = [{5, 3, 6, 7}, {1, 3, 5, 2}, {7, 3, 8, 5}, {8, 4, 5, 3}]

set.intersection(**\***test\_list) 🡪 {3,5}

* Set Methods
  + Functions Name Description
  + add() Adds a given element to a set
  + clear() Removes all elements from the set
  + copy() Returns a shallow copy of the set
  + - difference() Returns a set that is the difference between two sets
  + -= difference\_update() Updates the existing caller set with the difference between two sets
  + discard() Removes the element from the set
  + frozenset() Return an immutable frozenset object
  + & intersection() Returns a set that has the intersection of all sets
  + &= intersection\_update() Updates the existing caller set with the intersection of sets
  + isdisjoint() Checks whether the sets are disjoint or not
  + <= or < issubset() Returns True if all elements of a set A are present in another set B
  + >= or > issuperset() Returns True if all elements of a set A occupies set B
  + pop() Returns and removes a random element from the set
  + remove() Removes the element from the set
  + ^ symmetric\_difference() Returns a set which is the symmetric difference between the two sets
  + ^= symmetric\_difference\_update() Updates the existing caller set with the symmetric difference of sets
  + | union() Returns a set that has the union of all sets
  + |= update() Adds elements to the set

**Dict**

* access dict
  + d[key]
  + d.get(key)
* add or update
  + d[key] = value
  + if key exists it updates or else it appends
* del: Removes an item by key.
* [pop()](https://www.geeksforgeeks.org/python-dictionary-pop-method/): Removes an item by key and returns its value.
* [clear()](https://www.geeksforgeeks.org/python-dictionary-clear): Empties the dictionary.
* [popitem()](https://www.geeksforgeeks.org/python-dictionary-popitem-method/): Removes and returns the last key-value pair.
* Dict Methods
  + 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
* del d[key]
* val = d.pop(key)
* key, val = d.popitem()
* d.clear()
* # Iterate over keys
* for key in d:
* print(key)
* # Iterate over values
* for value in d.values():
* print(value)
* # Iterate over key-value pairs
* for key, value in d.items():
* print(f"{key}: {value}")
* d2 = {k: v \* 2 for k, v in d1.items() if k > 2}
* d2 = {k: v \* 2 if k > 1 else v for k, v in d1.items()}
* d.update({key: value, …})
* d.setdefault(key, value) 🡪 add an item to a dictionary. This method checks if a key already exists:
  + If the key exists, it returns its value.
  + If the key does not exist, it adds the key with a specified default value.

|  |  |
| --- | --- |
| **Using square brackets []** | Directly access the value using the key. |
| **Using**[**.get() method**](https://www.geeksforgeeks.org/python-dictionary-get-method/) | Access the value using the key with an optional default value. |
| **Using**[**.values() method**](https://www.geeksforgeeks.org/python-dictionary-values/) | Retrieve all values as a list-like object (not directly accessing by key). |
| **Using**[**.keys() method**](https://www.geeksforgeeks.org/python-dictionary-keys-method/) | Retrieve all the keys as a view object. |
| **Using**[**.items() method**](https://www.geeksforgeeks.org/python-dictionary-items-method/) | Returns key-value pairs as tuples. |

* dict()
* d.values()
* d.keys()
* d.items() 🡪 key & value as tuple
* del d[key]
* d.pop(key) 🡪 returns value
* d.popitem() 🡪 removes last item and returns item
* d.clear()
* len(d)
* for key in d:
* for key, value in d.items():
* replace key 🡪 d[key\_change] = d.pop(key\_old)
* merge & concatenate
  + d1.update(d2)
  + d3 = d1 | d2
  + d3 = {\*\*d1, \*\*d2}

**Ternary**

* ***Syntax:*** *value\_if\_true if condition else value\_if\_false*
* ***Syntax:*** *(condition\_is\_false, condition\_is\_true)[condition] 🡪 tuple*
* ***Syntax:*** *condition\_dict = {True: value\_if\_true, False: value\_if\_false} 🡪 dict*
* ***Syntax:*** *lambda x: value\_if\_true if condition else value\_if\_false 🡪 lambda*
* ***Syntax:*** *print(value\_if\_true if condition else value\_if\_false)*

Match Case --- (like switch case)

match subject:

case pattern1:

# Code block if pattern1 matches

case pattern2:

# Code block if pattern2 matches

case \_:

# Default case (wildcard) if no other pattern matches

* case pattern
  + constant
  + constant with if condition
  + sequences --- list, tuple
  + mapping dictionaries
  + python class
  + OR ‘|’ operator

**Jump Statements**

* pass 🡪 used in empty function or classes or conditional statements or loops
* break 🡪 Exits a loop prematurely, regardless of the loop condition.
* Continue 🡪 Skips the rest of the current iteration of a loop and moves on to the next iteration.
* return 🡪 Used in functions to return a value to the caller, Exits the function immediately

Other statements

* with 🡪  Statement to manage resources, ensuring they are properly cleaned up after use.
* try, except, finally

Loops

* for
  + for i in string:
  + for i in range():
  + control statements --- continue, break, pass, else (for…else)
  + enumerate --- value and index
* while
  + while expression:

statements

* + control statements --- continue, break, pass, else (while…else)

List Comprehension

* *[****expression*** *for* ***item*** *in* ***iterable*** *if* ***condition****(optional)]*

ASCII 🡪 **Python code using ord function :** [**ord()**](https://www.geeksforgeeks.org/ord-function-python/)**:**It converts the given string of length one, returns an integer representing the Unicode code point of the character. For example, ord(‘a’) returns the integer 97.

**TODO**:

<https://www.geeksforgeeks.org/python-programming-examples/>

<https://www.geeksforgeeks.org/python-programming-language-tutorial/>

RegEx 🡪 \*

* Matrix Transpose 🡪 M = list(zip(\*M))
  + Other way --- T\_matrix **=** [[m[j][i] **for** j **in** range(len(m))] **for** i **in** range(len(m[0]))]
* Matrix length 🡪 length(M)
* Flatten Matrix 🡪 sum(M, [])
* Map 🡪 list(map(sum, B)) --- apply function
  + sum, min, max, len, sorted, all, any
* Add
  + A = [[1, 2], [3, 4]]
  + B = [[5, 6], [7, 8]]
  + C = [[A[i][j] + B[i][j] for j in range(len(A[0]))] for i in range(len(A))]
  + print(C) # [[6, 8], [10, 12]]
* Multiply
  + A = [[1, 2], [3, 4]]
  + B = [[5, 6], [7, 8]]
  + C = [[sum(A[i][k] \* B[k][j] for k in range(len(B))) for j in range(len(B[0]))] for i in range(len(A))]
  + result = [[sum(a \* b for a, b in zip(A\_row, B\_col))

for B\_col in zip(\*B)]

for A\_row in A]

* + print(C or result) # [[19, 22], [43, 50]]
* Use map(sum, B) for row sums and map(sum, zip(\*B)) for column sums.
* Use zip(\*B) for **column-wise** operations.
* print([i:=k\*i for k in [j for j in sum(test\_list, [])]][-1])
  + this works for nested list
  + flattens list and iterate as ‘j’
  + update ‘i’ product value with taking list values as ‘k’
  + it stores list in each iteration, so I am taking last value in list [-1]
  + using walrus operator for assigning
* create matrix
  + matrix = [[0 for \_ in range(n)] for \_ in range(n)]
* get K\_th column --- K is the column
  + list(zip(**\***test\_list)[K])
  + list(map(**lambda** x: x[K], test\_list))
  + [row[K] **for** row **in** test\_list]
* Vertical Concatenation in Matrix with String
  + test\_list **=** [["Gfg", "good"], ["is", "for"], ["Best"]]
  + str(["".join(ele) for ele in zip\_longest(\*test\_list, fillvalue ="")])

**String**

* Palindrome
  + s == s[::-1]
  + '(empty/no\_space)'.join(reversed(s)) == s
* Symmetrical (length 🡪 odd or even)
  + odd --- amaama
  + even --- khokho

half = len(s) // 2

s1 = s[:half]

s2 = s[half:] if len(s) % 2 == 0 else s[half + 1:]

s1 == s2

\*

def is\_symmetrical(s):

length = len(s)

mid = length // 2

if length % 2 == 0:

return s[:mid] == s[mid:]

else:

return s[:mid] == s[mid+1:]

* Reverse Words in a Given String in Python
  + ' (space) '.join(s.split()[::-1])
* .strip() 🡪 remove white space at the beginning and at the end
  + lstrip
  + rstrip
* reversed(any iterable object) 🡪 doesn't create a new reversed sequence, but rather an iterator that allows you to traverse the original sequence in reverse order.
  + list/tuple/set --- print(list(reversed(numbers)))
  + string --- print('(no space)'.join(reversed(text)))
* How to Remove Letters From a String in Python
  + s.replace("l", "") --- any letter
  + "(no spac/empty)".join([c for c in s if c != "o"])
  + ""(no spac/empty)".join(filter(lambda c: c != "o", s))
* Check if String Contains Substring in Python
  + if subString in givenString: \_\_\_\_\_\_or\_\_\_\_\_\_\_ if subString in givenString.split():
  + givenString.find(subString) != -1
    - find() 🡪 returns -1 if not found; or else returns 1st occurrence index value
  + givenString.count(subString) > 0
  + givenString.index(substring, startIndex, endIndex) != error --- start & end values optional
  + givenString.\_\_contains\_\_(“subString”) --- returns bool value
* Python – Words Frequency in String Shorthands --- each word count
  + from collections import Counter
  + Counter(map(str, s.split()))
  + Counter([word for word in s.split()])
  + Counter(s.split())
  + dict --- .get(key) 🡪 gets value from dict, default value is ‘0’
    - w\_freq = {}
    - for word in s.split():
    - w\_freq[word] = w\_freq.get(word, 0) + 1
    - Print(w\_freq)
  + empty dict
    - del dict[key]
    - dict.pop[key]
    - {key: test\_dict[key] for key in test\_dict if key != ‘deleting\_key’}
    - {key: val for key,

val in test\_dict.items() if key != ‘deleting\_key’}

* Convert Snake case to Pascal case
  + str(test\_str.replace("\_", " ").title().replace(" ", ""))
  + str(string.capwords(test\_str.replace("\_", " ")).replace(" ", ""))
  + "".join([word.capitalize() **for** word **in** snake\_str.split("\_")])
  + "".join([word.title() **for** word **in** snake\_str.split("\_")])
* Length of a string
  + print(len(test), test.count("")-1)
* dict 🡪 sort
  + Ascending Order
  + # Sort by key

sorted\_by\_key = dict(sorted(my\_dict.items()))

* + # Sort by value

sorted\_by\_value = dict(sorted(my\_dict.items(), key=lambda item: item[1]))

* + Descending Order

sorted\_by\_key\_desc = dict(sorted(my\_dict.items(), reverse=True))

sorted\_by\_value\_desc = dict(sorted(my\_dict.items(), key=lambda item: item[1], reverse=True))

* sorted(iterable, key=None, reverse=False)
  + **iterable** – The sequence (list, tuple, dictionary, etc.) to be sorted.
  + Sorting by Length of String (key parameter) --- sorted(words, key=len)
  + Sorting a List of Tuples by Second Element --- sorted(data, key=lambda x: x[1])
* to print even length words in a string
  + print(" ".join(i for i in a.split(" ") if len(i)%2==0))
  + print(" ".join(filter(lambda x: (len(x)%2==0), a.split(" "))))
* to Accept the Strings Which Contains all Vowels
  + all(i in s.lower() for i in v)
  + v.issubset(set(s.lower()))
  + re.search(r'(?=.\*a)(?=.\*e)(?=.\*i)(?=.\*o)(?=.\*u)', s.lower())
* min/max frequency characters in a string
  + s='Geeksforgeeks'
  + res=Counter(s.lower()) --- Counter({'e': 4, 'g': 2, 'k': 2, 's': 2, 'f': 1, 'o': 1, 'r': 1})
  + print(max(res, key = res.get)) --- ‘e’
  + print(min(res, key = res.get)) --- ‘f’
* String Methods
  + string.isalpha() --- upeer or lower not space
  + string.isdecimal() ---
  + string.isdigit() ---
  + string.isnumeric() ---
  + string.isalnum()
  + string.isspace() --- \t, \r, \n, \f includes
  + string.isprintable()
  + string.isidentifier()
  + str.lower()
  + str.upper()
  + str.capitalize() --- sentence 1st letter
  + str.title() --- each letter of the word
  + str.count(“value”)
  + str.find(“value”) --- false returns -1
  + str.index(“value”) --- false returns error
  + str.replace(“a”,”x”) --- replace all 🡪 a with x
  + str.strip() --- remove whitespace 🡪 str.rstrip() str.lstrip()
  + str.split() --- returns list
  + str.join()
  + str.startswith()
  + str.endswith()

**Python Functions**

* def function\_name (parameters):
  + function statements
  + return expression
* def function\_name(\*args):
  + \*[args](https://www.geeksforgeeks.org/args-kwargs-python/)is used to pass a variable number of arguments to a function.
  + The \* allows a function to accept any number of positional arguments.

def fun(\*args):

for arg in args:

print(arg)

# Calling the function with multiple arguments

fun(1, 2, 3, 4, 5)

* def function\_name(\*\*kwargs):
  + \*\*[kwargs](https://www.geeksforgeeks.org/args-kwargs-python/)is used to pass a variable number of keyword arguments to a function.
  + The \*\* syntax collects the keyword arguments into a [dictionary](https://www.geeksforgeeks.org/python-dictionary/), where the keys are the argument names and the values are the corresponding argument values.
  + This allows the function to accept any number of named (keyword) arguments.
* **Class & Object**
  + class Class\_name:

def \_\_init\_\_(self,any\_args):

self.arg=…

def func\_name(self,any\_args):

func statements

new\_object = Class\_name(if\_any\_args)

new\_object.func\_name(if\_any\_args)

* pass
* return --- one or more arguments
* global & local variable
* recursion --- need base case & recursive case
* \*args & \*\*kwargs
* *def example\_function(\*args, \*\*kwargs):  
   print(args) # tuple of positional arguments  
   print(kwargs) # dictionary of keyword arguments  
    
  example\_function(1, 2, 3, name='Alice', age=30)****Output:*** *(1, 2, 3)  
  {'name': 'Alice', 'age': 30}*
  + ***\*args*** *collects additional positional arguments as a tuple*
  + ***\*\*kwargs*** *collects additional keyword arguments as a dictionary.*

\*args

def fun(arg1, \*argv):

print("First argument :", arg1)

for arg in argv:

print("Argument \*argv :", arg)

fun('Hello', 'Welcome', 'to', 'GeeksforGeeks')

\*\*kwargs

def fun(arg1, \*\*kwargs):

print("First argument :", arg1)

for k, val in kwargs.items():

print("%s == %s" % (k, val))

fun("Hi", s1='Geeks', s2='for', s3='Geeks')

* “**self**” as default argument
* Functions
  + [**Assigned to Variables**](https://www.geeksforgeeks.org/assign-function-to-a-variable-in-python/)**:** We can assign functions to variables.
  + [**Passed as Arguments**](https://www.geeksforgeeks.org/passing-function-as-an-argument-in-python/)**:**We can pass functions as arguments to other functions.
  + [**Returned from Functions**](https://www.geeksforgeeks.org/returning-a-function-from-a-function-python/)**:** Functions can return other functions.
  + **Stored in Data Structures:**Functions can be stored in data structures such as lists, dictionaries, etc.
* *lambda arguments : expression*
* *lambda with if\_else*
* *lambda with filter, map, reduce*
* *map(function, iterable)*
* *filter(function, sequence)*
* *reduce(function, iterable[, initializer(optional)]) --- import functools*
* main
  + if \_\_name\_\_ == '\_\_main\_\_':
* decorator

def decorator\_name(func):

def wrapper(\*args, \*\*kwargs):

# Add functionality before the original function call

result = func(\*args, \*\*kwargs)

# Add functionality after the original function call

return result

return wrapper

@decorator\_name

def function\_to\_decorate():

# Original function code

pass

* multiple décor
  + @decor1
  + @decor2
  + func()
    - Implies 🡪 decor1(decor2(func())
* *A decorator is a higher-order function that takes another function as an argument, adds some functionality and returns a new function. It allows modifying or extending behavior of functions or methods.*
* ***\_\_init\_\_*** *is a special method (constructor) in Python classes. It’s automatically called when a new instance of a class is created. It initializes the object’s attributes or performs any setup needed for the object.*
* *class MyClass:  
   def \_\_init\_\_(self, arg1, arg2):  
   self.arg1 = arg1  
   self.arg2 = arg2*
* print(chr(ord('A')))
* print(id(object\_name)) --- returns unique id, each object has unique id

**Tuple**

* Size of Tuple
  + import sys

str(sys.getsizeof(newTuple))

* + str(newTuple.\_\_sizeof\_\_()) --- inbuilt
* sorted(newTuple)

**Dictionary**

* dict.values()
* dict.keys()
* sum(dict.values())
* sorted(iterable, key=None, reverse=False)
  + **iterable** – The sequence (list, tuple, dictionary, etc.) to be sorted.
  + Sorting by Length of String (key parameter) --- sorted(words, key=len)
  + Sorting a List of Tuples by Second Element --- sorted(data, key=lambda x: x[1])
* print(sorted(data\_list, key=itemgetter('age', 'name')))
* print(sorted(list, key**=lambda** i: (i['age'], i['name'])))
* merge
  + d1.update(d2) --- print(d1)
  + d3 = d1 | d2 --- print(d3)
  + d3 = {\*\*d1, \*\*d2}
* flatten dict
  + test\_dict = {'month': [1, 2, 3], 'name': ['Jan', 'Feb', 'March']}
  + res = {test\_dict['month'][i]: test\_dict['name'][i] for i in range(len(test\_dict['month']))}
  + print("Flattened dictionary:", res)
* move\_to\_end
  + from collections import OrderedDict
  + order=OrderedDict(newDict)
  + order.move\_to\_end(key,bool) --- default True
  + bool is True moves key to end
  + bool is False moves key to start
* order.popitem() --- removes last item
* order.pop(‘key’) --- removes specific item
* reversed(order) ---
  + for i in reversed(order):
  + print(i, order[i])
* Handling missing keys in Dict
  + print(dict.get(key, any\_error\_message) --- if key not found
  + dict.setdefault(key, value)
    - this appends item to the last of dict
  + defaultdict
    - import collections
    - defd **=** collections.defaultdict(**lambda** : 'Key Not found')
    - defd['a'] **=** 1
    - defd['b'] **=** 2
    - print (defd['c']) --- prints ‘Key Not Found’
  + try & except --- also used ‘KeyError’
* anagram --- sorted(n1)==sorted(n2)

**Linked List**

# Create a Node class to create a node

class Node:

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

self.data = data

self.next = None

# Create a LinkedList class

class LinkedList:

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

self.head = None

# Method to add a node at the beginning of the LL

def insertAtBegin(self, data):

new\_node = Node(data)

new\_node.next = self.head

self.head = new\_node

# Method to add a node at any index

# Indexing starts from 0.

def insertAtIndex(self, data, index):

if index == 0:

self.insertAtBegin(data)

return

position = 0

current\_node = self.head

while current\_node is not None and position + 1 != index:

position += 1

current\_node = current\_node.next

if current\_node is not None:

new\_node = Node(data)

new\_node.next = current\_node.next

current\_node.next = new\_node

else:

print("Index not present")

# Method to add a node at the end of LL

def insertAtEnd(self, data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

return

current\_node = self.head

while current\_node.next:

current\_node = current\_node.next

current\_node.next = new\_node

# Update node at a given position

def updateNode(self, val, index):

current\_node = self.head

position = 0

while current\_node is not None and position != index:

position += 1

current\_node = current\_node.next

if current\_node is not None:

current\_node.data = val

else:

print("Index not present")

# Method to remove first node of linked list

def remove\_first\_node(self):

if self.head is None:

return

self.head = self.head.next

# Method to remove last node of linked list

def remove\_last\_node(self):

if self.head is None:

return

# If there's only one node

if self.head.next is None:

self.head = None

return

# Traverse to the second last node

current\_node = self.head

while current\_node.next and current\_node.next.next:

current\_node = current\_node.next

current\_node.next = None

# Method to remove a node at a given index

def remove\_at\_index(self, index):

if self.head is None:

return

if index == 0:

self.remove\_first\_node()

return

current\_node = self.head

position = 0

while current\_node is not None and current\_node.next is not None and position + 1 != index:

position += 1

current\_node = current\_node.next

if current\_node is not None and current\_node.next is not None:

current\_node.next = current\_node.next.next

else:

print("Index not present")

# Method to remove a node from the linked list by its data

def remove\_node(self, data):

current\_node = self.head

# If the node to be removed is the head node

if current\_node is not None and current\_node.data == data:

self.remove\_first\_node()

return

# Traverse and find the node with the matching data

while current\_node is not None and current\_node.next is not None:

if current\_node.next.data == data:

current\_node.next = current\_node.next.next

return

current\_node = current\_node.next

# If the data was not found

print("Node with the given data not found")

# Print the size of the linked list

def sizeOfLL(self):

size = 0

current\_node = self.head

while current\_node:

size += 1

current\_node = current\_node.next

return size

# Print the linked list

def printLL(self):

current\_node = self.head

while current\_node:

print(current\_node.data)

current\_node = current\_node.next

# create a new linked list

llist = LinkedList()

# add nodes to the linked list

llist.insertAtEnd('a')

llist.insertAtEnd('b')

llist.insertAtBegin('c')

llist.insertAtEnd('d')

llist.insertAtIndex('g', 2)

# print the linked list

print("Node Data:")

llist.printLL()

# remove nodes from the linked list

print("\nRemove First Node:")

llist.remove\_first\_node()

llist.printLL()

print("\nRemove Last Node:")

llist.remove\_last\_node()

llist.printLL()

print("\nRemove Node at Index 1:")

llist.remove\_at\_index(1)

llist.printLL()

# print the linked list after all removals

print("\nLinked list after removing a node:")

llist.printLL()

print("\nUpdate node Value at Index 0:")

llist.updateNode('z', 0)

llist.printLL()

print("\nSize of linked list:", llist.sizeOfLL())

*The main types of linked lists include:*

* ***Singly Linked List:*** *Each node points to the next node in the sequence.*
* ***Doubly Linked List:*** *Each node points to both the next and previous nodes.*
* ***Circular Linked List:*** *Last node points back to the first node, forming a circle.*
* ***Sorted Linked List:*** *Elements are stored in a sorted order based on a specific criterion.*

Class 🡪

Attributes/Fields/Properties/Parameters/Arguments/values

Operations/Behaviours/Methods/(inside class functions)

**Classes & Objects**

* Class & Object

class ClassName:

class\_attribute=”value” #class attribute

def \_\_init\_\_ (self, para, num):

self.para=para #instance attribute

self.num=num #instance attribute

classObj = ClassName(“val”,3)

classObj.para 🡪 “val”

classObj.num 🡪 3

* \_\_str\_\_

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

return f”{self.para} has num {self.num}”

\_\_str\_\_ method in Python allows us to define a custom string representation of an object. By default, when we print an object or convert it to a string using str(), Python uses the default implementation, which returns a string like <\_\_main\_\_.ClassName object at 0x00000123>.

* Example

class Dog:

# Class variable

species = "Canine"

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

# Instance variables

self.name = name

self.age = age

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

return

f"{self.name} is {self.age} years old."

# Correct: Returning a string

print(dog1) # Buddy is 3 years old.

print(dog2) # Charlie is 5 years old.

# Create objects

dog1 = Dog("Buddy", 3)

dog2 = Dog("Charlie", 5)

# Access class and instance variables

print(dog1.species) # (Class variable)

print(dog1.name) # (Instance variable)

print(dog2.name) # (Instance variable)

# Modify instance variables

dog1.name = "Max"

print(dog1.name) # (Updated instance variable)

# Modify class variable

Dog.species = "Feline"

print(dog1.species) # (Updated class variable)

print(dog2.species)

* Output

Canine Buddy Charlie Max Feline Feline

* **Class Variable (species):**Shared by all instances of the class. Changing Dog.species affects all objects, as it’s a property of the class itself.
* **Instance Variables (name, age):** Defined in the \_\_init\_\_ method. Unique to each instance (e.g., dog1.name and dog2.name are different).
* **Accessing Variables:** Class variables can be accessed via the class name (Dog.species) or an object (dog1.species). Instance variables are accessed via the object (dog1.name).
* **Updating Variables:**Changing Dog.species affects all instances. Changing dog1.name only affects dog1 and does not impact dog2.
* **Self: Pointer to Current Object**
* The self is always pointing to the Current Object. When you create an instance of a class, you’re essentially creating an object with its own set of attributes and methods.
* id(self) --- inside the class while calling the obejct & id(object\_name) are same
* Self is the first argument to be passed in[Constructor](https://www.geeksforgeeks.org/constructors-in-python/)and Instance Method.Self must be provided as a First parameter to the Instance method and constructor. If you don’t provide it, it will cause an error.
* “self” is not a keyword --- instead we can use any words. Self is a convention and not a[Python keyword](https://www.geeksforgeeks.org/python-keywords/).
* **Class attributes:**[Class attributes](https://www.geeksforgeeks.org/g-fact-34-class-or-static-variables-in-python/) belong to the class itself they will be shared by all the instances. Such attributes are defined in the class body parts usually at the top, for legibility.
* **Instance Attributes**
* Unlike class attributes, instance attributes are not shared by objects. Every object has its own copy of the instance attribute (In case of class attributes all object refer to single copy). To list the attributes of an instance/object, we have two functions:- 1.
* vars() – This function displays the attribute of an instance in the form of an dictionary. 2.
* dir() – This function displays more attributes than vars function,as it is not limited to instance. It displays the class attributes as well. It also displays the attributes of its ancestor classes.
* subclass

class SubclassName(BaseClassName/ParentClassName/SuperClassName):

# Class attributes and methods for the subclass

# ...

* super().\_\_init\_\_(...) 🡪 arguments/parameters are given inside the parenthesis
  + super() returns a special object that allows you to call methods of the parent class.
  + super().\_\_init\_\_(...) calls the \_\_init\_\_ method of the parent class.
  + When we create a subclass, the parent class might have attributes or functionality that need to be initialized. By calling super().\_\_init\_\_(...), we make sure the parent class's initializer is executed, and all the necessary attributes from the parent class are set up.
  + If you don't explicitly call the parent class's \_\_init\_\_ method, the parent class's attributes won't be initialized, which can lead to errors if you try to access them.
  + It avoids duplicate code and ensures consistent initialization of attributes defined in the parent class.
  + super() is especially useful in multiple inheritance scenarios to manage the method resolution order (MRO).
* the [\_\_init\_\_()](https://www.geeksforgeeks.org/__init__-in-python/) built-in method for understanding the meaning of classes. Whenever the class is being initiated, a method namely \_\_init\_\_() is always executed. An \_\_init\_\_() method is used to assign the values to object properties or to perform the other method that is required to complete when the object is created.
* InnerClass / NestedClass
  + A class defined in another class is known as an inner class or nested class. If an object is created using child class means inner class then the object can also be used by parent class or root class. A parent class can have one or more inner classes but generally inner classes are avoided.
  + We can make our code even more object-oriented by using an inner class. A single object of the class can hold multiple sub-objects. We can use multiple sub-objects to give a good structure to our program.
  + Types of inner classes are as follows:
    - Multiple inner class
    - Multilevel inner class
* # create outer class
* class Doctors:
* def \_\_init\_\_(self):
* self.name = 'Doctor'
* self.den = self.Dentist()
* self.car = self.Cardiologist()
* def show(self):
* print('In outer class')
* print('Name:', self.name)
* # create a 1st Inner class
* class Dentist:
* def \_\_init\_\_(self):
* self.name = 'Dr. Savita'
* self.degree = 'BDS'
* def display(self):
* print("Name:", self.name)
* print("Degree:", self.degree)
* # create a 2nd Inner class
* class Cardiologist:
* def \_\_init\_\_(self):
* self.name = 'Dr. Amit'
* self.degree = 'DM'
* def display(self):
* print("Name:", self.name)
* print("Degree:", self.degree)
* # create a object
* # of outer class
* outer = Doctors()
* outer.show()
* # create a object
* # of 1st inner class
* d1 = outer.den
* # create a object
* # of 2nd inner class
* d2 = outer.car
* print()
* d1.display()
* print()
* d2.display()
* # create an outer class
* class Geeksforgeeks:
* def \_\_init\_\_(self):
* # create an inner class object
* self.inner = self.Inner()
* def show(self):
* print('This is an outer class')
* # create a 1st inner class
* class Inner:
* def \_\_init\_\_(self):
* # create an inner class of inner class object
* self.innerclassofinner = self.Innerclassofinner()
* def show(self):
* print('This is the inner class')
* # create an inner class of inner
* class Innerclassofinner:
* def show(self):
* print('This is an inner class of inner class')
* # create an outer class object
* # i.e.Geeksforgeeks class object
* outer = Geeksforgeeks()
* outer.show()
* print()
* # create an inner class object
* gfg1 = outer.inner
* gfg1.show()
* print()
* # create an inner class of inner class object
* gfg2 = outer.inner.innerclassofinner
* gfg2.show()

**MetaClass** 🡪 (later on…)

Syntax

*class ClassName:*

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

*# constructor code here*

*# Creating an instance object*

*instance\_object = ClassName(parameters)*

* **Dynamic attributes** in [Python](https://www.geeksforgeeks.org/python-programming-language/) are terminologies for attributes that are **defined at runtime**, after creating the objects or instances. In Python we call all functions, methods also as an object
  + class GFG:
  + employee = True
  + # Driver Code
  + e1 = GFG()
  + e2 = GFG()
  + e1.employee = False
  + e2.name = "Nikhil"
  + print(e1.employee)
  + print(e2.employee)
  + print(e2.name)
  + # this will raise an error # as name is a dynamic attribute # created only for the e2 object
  + print(e1.name)
* Output

False True Nikhil

Traceback (most recent call last):

File "/home/test.py", line 17, in

print(e1.name)

AttributeError: 'GFG' object has no attribute 'name'

* **Constructors**
* The method [**\_\_new\_\_**](https://www.geeksforgeeks.org/__new__-in-python/) is the **constructor**that creates a new instance of the class while [**\_\_init\_\_**](https://www.geeksforgeeks.org/__init__-in-python/) is the initializer that sets up the instance’s attributes after creation. These methods work together to manage object creation and initialization.
* \_\_new\_\_ Method 🡪 This method is responsible for creating a new instance of a class. It allocates memory and returns the new object. It is called before \_\_init\_\_.
  + class ClassName:
  + def \_\_new\_\_(cls, parameters):
  + instance = super(ClassName, cls).\_\_new\_\_(cls)
  + return instance
* \_\_init\_\_ Method 🡪 This method initializes the newly created instance and is commonly used as a constructor in Python. It is called immediately after the object is created by \_\_new\_\_ method and is responsible for initializing attributes of the instance. **I**t is called after \_\_new\_\_ and does not return anything (it returns **None**by default).
  + class ClassName:
  + def \_\_init\_\_(self, parameters):
  + self.attribute = value
* Types of Constructors
  + Default Constructor 🡪 A **default constructor** does not take any parameters other than **self**. It initializes the object with default attribute values.
  + Parameterized Constructor 🡪 A**parameterized constructor** accepts arguments to initialize the object’s attributes with specific values.

**Inheritance, Polymorphism, Encapsulation and Abstraction**

**Abstraction** (Hiding implementation details and exposing only necessary functionality)

**Encapsulation** (Restricting access to certain details of an object)

**Polymorphism** (Same interface, different implementations)

**Inheritance** (Allows a class to inherit properties and methods from another class)

A screenshot of a computer

Description automatically generated

Variables

* + local
  + global
  + instance
  + class (static)

A screenshot of a computer program

Description automatically generated

**Encapsulation**

* **Data Hiding**: The variables (attributes) are kept private or protected, meaning they are not accessible directly from outside the class. Instead, they can only be accessed or modified through the methods.
* **Access through Methods**: Methods act as the interface through which external code interacts with the data stored in the variables. For instance, getters and setters are common methods used to retrieve and update the value of a private variable.
* **Control and Security**: By encapsulating the variables and only allowing their manipulation via methods, the class can enforce rules on how the variables are accessed or modified, thus maintaining control and security over the data.
* **Public Access Modifier:**Theoretically, public methods and fields can be accessed directly by any class.
* **Protected Access Modifier:**Theoretically, protected methods and fields can be accessed within the same class it is declared and its subclass.
* **Private Access Modifier:**Theoretically, private methods and fields can be only accessed within the same class it is declared.
* Example

# program to illustrate access modifiers of a class

# super class

class Super:

# public data member

var1 = None

# protected data member

\_var2 = None

# private data member

\_\_var3 = None

# constructor

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()

print()

# Can also be accessed using

obj.\_displayProtectedMembers()

obj.\_Super\_\_displayPrivateMembers()

print()

# Object can access protected member

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

print("Object is accessing private member:", obj.\_Super\_\_var3)

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

# print(obj.\_\_var3)

* Output

Public Data Member: Geeks

Protected Data Member: 4

Private Data Member: Geeks!

Protected Data Member: 4

Private Data Member: Geeks!

Object is accessing protected member: 4

Object is accessing private member: Geeks!

* **Mangling** 🡪 Private

class MyClass:

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

self.\_\_private\_var = 10

def get\_private\_var(self):

return self.\_\_private\_var

obj = MyClass()

print(obj.get\_private\_var()) # Output: 10

print(obj.\_\_private\_var) # AttributeError: 'MyClass' object has no attribute '\_\_private\_var'

print(obj.\_MyClass\_\_private\_var) # Output: 10

* + In Python, name mangling is a technique used to protect class attributes from being accidentally overwritten by subclasses. It's achieved by adding a double underscore prefix (\_\_) to the attribute name. **Definition**: def \_\_private\_method(self), **Mangled Name**: \_MyClass\_\_private\_method
  + When you define an attribute with two leading underscores (e.g., \_\_my\_attr), Python automatically renames it to \_ClassName\_\_my\_attr, where ClassName is the name of the class.
  + This renaming makes it harder for subclasses to accidentally override the attribute, as they would need to explicitly use the mangled name.
* *class MyClass:  
   def \_\_private\_method(self):  
   return 'Private method'  
  obj = MyClass()  
  print(obj.\_MyClass\_\_private\_method()) # Output: Private method*

**Inheritance**

* Syntax

class ParentClass:

# Parent class code here

pass

class ChildClass(ParentClass):

# Child class code here

pass

* **Explanation of Python Inheritance Syntax**
  + **Parent Class**:
  + This is the base class from which other classes inherit.
  + It contains attributes and methods that the child class can reuse.
  + **Child Class:**
  + This is the derived class that inherits from the parent class.
  + The syntax for inheritance is class ChildClass(ParentClass).
  + The child class automatically gets all attributes and methods of the parent class unless overridden.
* We can use parent class inherit into child class in two ways (inside def \_\_init\_\_(self): )
  + ParentClassName.\_\_init\_\_(self)
  + super().\_\_init\_\_()
* The super() function is used inside the \_\_init\_\_() method of Child to call the constructor of Parent and initialize the inherited attributes
* **Types of Python Inheritance**
  + **Single Inheritance**: A child class inherits from one parent class.
  + **Multiple Inheritance**: A child class inherits from more than one parent class.
  + **Multilevel Inheritance**: A class is derived from a class which is also derived from another class.
  + **Hierarchical Inheritance**: Multiple classes inherit from a single parent class.
  + **Hybrid Inheritance**: A combination of more than one type of inheritance.
* Method Overriding
  + super().\_\_init\_\_() # Call parent constructor
  + # Calling the parent's class # method 🡪 ParentClassName.methodName(self)
  + # Calling the parent's class # method 🡪super().methodName()
* Operator Overloading

# Python Program illustrate how

# to overload an binary + operator

# And how it actually works

class A:

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

self.a = a

# adding two objects

def \_\_add\_\_(self, o):

return self.a + o.a

ob1 = A(1)

ob2 = A(2)

ob3 = A("Geeks")

ob4 = A("For")

print(ob1 + ob2)

print(ob3 + ob4)

# Actual working when Binary Operator is used.

print(A.\_\_add\_\_(ob1 , ob2))

print(A.\_\_add\_\_(ob3,ob4))

#And can also be Understand as :

print(ob1.\_\_add\_\_(ob2))

print(ob3.\_\_add\_\_(ob4))

* + Output 🡪 3; GeeksFor

Python magic methods or special functions for operator overloading

Binary Operators:

Operator Magic Method

+ \_\_add\_\_(self, other)

– \_\_sub\_\_(self, other)

\* \_\_mul\_\_(self, other)

/ \_\_truediv\_\_(self, other)

// \_\_floordiv\_\_(self, other)

% \_\_mod\_\_(self, other)

\*\* \_\_pow\_\_(self, other)

>> \_\_rshift\_\_(self, other)

<< \_\_lshift\_\_(self, other)

& \_\_and\_\_(self, other)

| \_\_or\_\_(self, other)

^ \_\_xor\_\_(self, other)

Comparison Operators:

Operator Magic Method

< \_\_lt\_\_(self, other)

> \_\_gt\_\_(self, other)

<= \_\_le\_\_(self, other)

>= \_\_ge\_\_(self, other)

== \_\_eq\_\_(self, other)

!= \_\_ne\_\_(self, other)

Assignment Operators:

Operator Magic Method

-= \_\_isub\_\_(self, other)

+= \_\_iadd\_\_(self, other)

\*= \_\_imul\_\_(self, other)

/= \_\_idiv\_\_(self, other)

//= \_\_ifloordiv\_\_(self, other)

%= \_\_imod\_\_(self, other)

\*\*= \_\_ipow\_\_(self, other)

>>= \_\_irshift\_\_(self, other)

<<= \_\_ilshift\_\_(self, other)

&= \_\_iand\_\_(self, other)

|= \_\_ior\_\_(self, other)

^= \_\_ixor\_\_(self, other)

Unary Operators:

Operator Magic Method

– \_\_neg\_\_(self)

+ \_\_pos\_\_(self)

~ \_\_invert\_\_(self)

* Example

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 2)

p2 = Point(2, 3)

result = p1 + p2

print(result.x, result.y) # Outputs: 3

* ***Arithmetic Operators****: +, -, \*, /, %, \*\* (exponentiation), // (floor division).*
* ***Comparison Operators****: ==, !=, <, >, <=, >=.*
* ***Logical Operators****: and, or, not.*
* ***Bitwise Operators****: &, |, ^, ~, <<, >>.*
* ***Assignment Operators****: =, +=, -=, etc.*
* ***Identity Operators****: is, is not.*
* ***Membership Operators****: in, not in.*

Super()

* ***Syntax:*** *super()*
* ***Return :*** *Return a proxy object which represents the parent’s class.*
* Need not remember or specify the parent class name to access its [methods](https://www.geeksforgeeks.org/list-methods-in-python/). This function can be used both in single and multiple inheritances.
* This implements modularity (isolating changes) and code reusability as there is no need to rewrite the entire function.
* The super function in Python is called dynamically because Python is a dynamic language, unlike other languages.
* Example --- Super with Multiple Inheritance

class Mammal():

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

print(name, "Is a mammal")

class canFly(Mammal):

def \_\_init\_\_(self, canFly\_name):

print(canFly\_name, "cannot fly")

# Calling Parent class

# Constructor

super().\_\_init\_\_(canFly\_name)

class canSwim(Mammal):

def \_\_init\_\_(self, canSwim\_name):

print(canSwim\_name, "cannot swim")

super().\_\_init\_\_(canSwim\_name)

class Animal(canFly, canSwim):

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

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

# Driver Code

Carol = Animal("Dog")

Output 🡪

Dog cannot fly  
Dog cannot swim  
Dog Is a mammal

***Is super() Automatically Called?***

*No, super() is not automatically called in Python. You need to explicitly call it when you want the superclass’s implementation of a method to be executed. This is particularly true for constructors or any overridden methods.*

**class** **A**: **def** age(self): print("Age is 21")

**class** **B**: **def** age(self): print("Age is 23")

**class** **C**(A, B): **def** age(self): super(C, self).age()

c = C()

print(C.\_\_mro\_\_)

print(C.mro())

Output 🡪

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>)  
[<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>]

* Example --- Super with Multi-Level Inheritance

class Mammal():

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

print(name, "Is a mammal")

class canFly(Mammal):

def \_\_init\_\_(self, canFly\_name):

print(canFly\_name, "cannot fly")

# Calling Parent class

# Constructor

super().\_\_init\_\_(canFly\_name)

class canSwim(canFly):

def \_\_init\_\_(self, canSwim\_name):

print(canSwim\_name, "cannot swim")

super().\_\_init\_\_(canSwim\_name)

class Animal(canSwim):

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

# Calling the constructor

# of both the parent

# class in the order of

# their inheritance

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

# Driver Code

Carol = Animal("Dog")

Output 🡪

Dog cannot swim  
Dog cannot fly  
Dog Is a mammal

* Multiple Inheritance
  + When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.

# Python program to demonstrate # super()

class Class1:

def m(self):

print("In Class1")

class Class2(Class1):

def m(self):

print("In Class2")

super().m() or Class1.m(self)

class Class3(Class1):

def m(self):

print("In Class3")

super().m() or Class1.m(self)

class Class4(Class2, Class3):

def m(self):

print("In Class4")

super().m()

or

Class2.m(self)

        Class3.m(self)

or (Class2.m(self)        Class3.m(self)        Class1.m(self))

print(Class4.mro()) #This will print list

print(Class4.\_\_mro\_\_) #This will print tuple

or

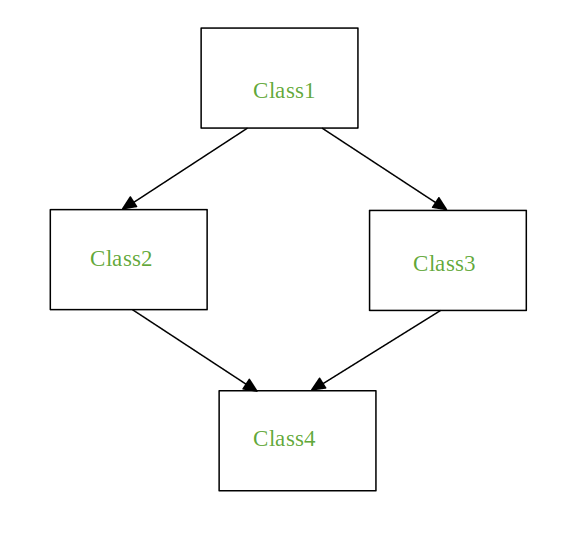
Class2.m(obj)

Class3.m(obj)

Class1.m(obj)

Output

*[<class ‘\_\_main\_\_.Class4’>, <class ‘\_\_main\_\_.Class2’>, <class ‘\_\_main\_\_.Class3’>, <class ‘\_\_main\_\_.Class1’>, <class ‘object’>]   
(<class ‘\_\_main\_\_.Class4’>, <class ‘\_\_main\_\_.Class2’>, <class ‘\_\_main\_\_.Class3’>, <class ‘\_\_main\_\_.Class1’>, <class ‘object’>)*

* Hybrid Inheritance 🡪 

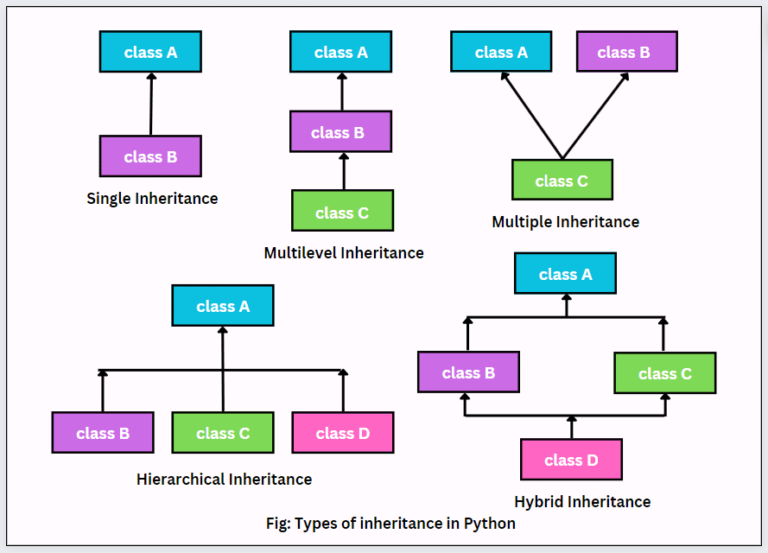
Class 4

Class 2

Class 3

Class 1

* MultiLevel Inheritance 🡪 is like parent, child, grandchild, and goes on…



**Polymorphism**

* Polymorphism is a foundational concept in programming that allows entities like functions, methods or operators to behave differently based on the type of data they are handling. Derived from Greek, the term literally means “many forms”.
* Python’s dynamic typing and duck typing make it inherently polymorphic. Functions, operators and even built-in objects like loops exhibit polymorphic behavior.

class Shape:

def area(self):

return "Undefined"

class Rectangle(Shape):

def \_\_init\_\_(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

class Circle(Shape):

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

self.radius = radius

def area(self):

return 3.14 \* self.radius \*\* 2

shapes = [Rectangle(2, 3), Circle(5)]

for shape in shapes:

print(f"Area: {shape.area()}")

* Types of Polymorphism
  + **Compile time** 🡪
    - Method overloading (not allowed in python) --- allowed in Java, C++
    - Operator overloading (allowed in python)
  + **Run time** 🡪 Method overriding
* Operator Overloading
  + print(5 + 10) # Integer addition
  + print("Hello " + "World!") # String concatenation
  + print([1, 2] + [3, 4]) # List concatenation
* Method Overloading is when multiple methods have the same name but different parameters within the same class (not typically supported directly in Python).
* Method Overriding happens when a subclass redefines a method from its parent class with the same signature to provide a specific implementation.

| **S.NO** | **Method Overloading** | **Method Overriding** |
| --- | --- | --- |
| 1. | In the method overloading, methods or functions must have the same name and different signatures. | Whereas in the method overriding, methods or functions must have the same name and same signatures. |
| 2. | Method overloading is a example of compile time polymorphism. | Whereas method overriding is a example of run time polymorphism. |
| 3. | In the method overloading, inheritance may or may not be required. | Whereas in method overriding, inheritance always required. |
| 4. | Method overloading is performed between methods within the class. | Whereas method overriding is done between parent class and child class methods. |
| 5. | It is used in order to add more to the behavior of methods. | Whereas it is used in order to change the behavior of exist methods. |
| 6. | In method overloading, there is no need of more than one class. | Whereas in method overriding, there is need of at least of two classes. |

**Abstraction**

* In Python, the @abstractmethod decorator is used to define abstract methods within an abstract base class (ABC).
* Key points:
  + **Abstract Base Classes (ABCs):**
  + These are classes that cannot be instantiated directly. They serve as blueprints for subclasses, defining a common interface that they must implement.
  + **Abstract Methods:**
  + Methods declared within an ABC that have no implementation. Subclasses are required to provide concrete implementations for these methods.

# Import required modules

from abc import ABC, abstractmethod

# Create Abstract base class

class Car(ABC):

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

# Create abstract method

@abstractmethod

def printDetails(self):

pass

# Create concrete method

def accelerate(self):

print("Speed up ...")

def break\_applied(self):

print("Car stopped")

# Create a child class

class Hatchback(Car):

def printDetails(self):

print("Brand:", self.brand)

print("Model:", self.model)

print("Year:", self.year)

def sunroof(self):

print("Not having this feature")

# Create a child class

class Suv(Car):

def printDetails(self):

print("Brand:", self.brand)

print("Model:", self.model)

print("Year:", self.year)

def sunroof(self):

print("Available")

# Create an instance of the Hatchback class

car1 = Hatchback("Maruti", "Alto", "2022")

# Call methods

car1.printDetails()

car1.accelerate()

car1.sunroof()

Output

Brand: Maruti

Model: Alto

Year: 2022

Speed up ...

Not having this feature

* Abstract Methods
* Concrete Methods

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def make\_sound(self):

pass # Abstract method, to be implemented by subclasses

def move(self):

return "Moving" # Concrete method with implementation

* Abstract Properties
  + **Abstract properties** work like abstract methods but are used for properties. These properties are declared with the @property decorator and marked as abstract using @abstractmethod. Subclasses must implement these properties.

from abc import ABC, abstractmethod

class Animal(ABC):

@property

@abstractmethod

def species(self):

pass # Abstract property, must be implemented by subclasses

class Dog(Animal):

@property

def species(self):

return "Canine"

# Instantiate the concrete subclass

dog = Dog()

print(dog.species)

* **Abstract methods** are methods that are defined in an abstract class but do not have an implementation. They serve as a blueprint for the subclasses, ensuring that they provide their own implementation.
* **Concrete methods** are methods that have full implementations in an abstract class. These methods can be inherited by subclasses and used directly without needing to be redefined.
* Abstract Class Instantiation
  + Abstract classes cannot be instantiated directly. This is because they contain one or more abstract methods or properties that lack implementations. Attempting to instantiate an abstract class results in a TypeError.
  + Only subclasses that implement all abstract methods can be instantiated.

***OOP’s***

***Abstraction 🡪 Abstract Base Class; @abstractmethod; @property***

***Encapsulation 🡪 private, public, protected --- methods & attributes***

***Inheritance 🡪 multiple, multilevel, hierarchical --- subclass & superclass***

***Polymorphism 🡪 method overriding; operator overloading***

***Class & Object***

**Dunder or magic methods in Python**

* Python Magic methods are the methods starting and ending with double underscores ‘\_\_’.
* They are defined by built-in classes in Python and commonly used for operator overloading.
* They are also called Dunder methods, Dunder here means “Double Under (Underscores)”.
* print(dir(int))
* ***Initialization and Construction***
  + [***\_\_new\_\_***](https://www.geeksforgeeks.org/__new__-in-python/)***:*** *To get called in an object’s instantiation.*
  + [***\_\_init\_\_***](https://www.geeksforgeeks.org/__init__-in-python/)***:*** *To get called by the \_\_new\_\_ method.*
  + [***\_\_del\_\_***](https://www.geeksforgeeks.org/python-__delete__-vs-__del__/)***:*** *It is the destructor.*
* ***Numeric magic methods***
  + [***\_\_trunc\_\_***](https://www.geeksforgeeks.org/g-fact-35-truncate-in-python/)***(self):*** *Implements behavior for math.trunc()*
  + [***\_\_ceil\_\_***](https://www.geeksforgeeks.org/__call__-in-python/)***(self):*** *Implements behavior for math.ceil()*
  + [***\_\_floor\_\_***](https://www.geeksforgeeks.org/python-math-floor-function/)***(self):*** *Implements behavior for math.floor()*
  + [***\_\_round\_\_***](https://www.geeksforgeeks.org/round-function-python/)***(self,n):*** *Implements behavior for the built-in round()*
  + ***\_\_invert\_\_(self):*** *Implements behavior for inversion using the ~ operator.*
  + ***\_\_abs\_\_(self):*** *Implements behavior for the built-in abs()*
  + ***\_\_neg\_\_(self):*** *Implements behavior for negation*
  + ***\_\_pos\_\_(self):*** *Implements behavior for unary positive*
* ***Arithmetic operators***
  + *\_\_add\_\_(self, other): Implements behavior for the + operator (addition).*
  + *\_\_sub\_\_(self, other): Implements behavior for the – operator (subtraction).*
  + *\_\_mul\_\_(self, other): Implements behavior for the \* operator (multiplication).*
  + *\_\_floordiv\_\_(self, other): Implements behavior for the // operator (floor division).*
  + *\_\_truediv\_\_(self, other): Implements behavior for the / operator (true division).*
  + *\_\_mod\_\_(self, other): Implements behavior for the % operator (modulus).*
  + *\_\_divmod\_\_(self, other): Implements behavior for the divmod() function.*
  + *\_\_pow\_\_(self, other): Implements behavior for the \*\* operator (exponentiation).*
  + *\_\_lshift\_\_(self, other): Implements behavior for the << operator (left bitwise shift).*
  + *\_\_rshift\_\_(self, other): Implements behavior for the >> operator (right bitwise shift).*
  + *\_\_and\_\_(self, other): Implements behavior for the & operator (bitwise and).*
  + *\_\_or\_\_(self, other): Implements behavior for the | operator (bitwise or).*
  + *\_\_xor\_\_(self, other): Implements behavior for the ^ operator (bitwise xor).*
  + *\_\_invert\_\_(self): Implements behavior for bitwise NOT using the ~ operator.*
  + *\_\_neg\_\_(self): Implements behavior for negation using the – operator.*
  + *\_\_pos\_\_(self): Implements behavior for unary positive using the + operator.*
* ***String Magic Methods***
  + [***\_\_str\_\_***](https://www.geeksforgeeks.org/str-vs-repr-in-python/)***(self):*** *Defines behavior for when str() is called on an instance of your class.*
  + [***\_\_repr\_***](https://www.geeksforgeeks.org/python-__repr__-magic-method/)***\_(self): T****o get called by built-int repr() method to return a machine readable representation of a type.*
  + ***\_\_unicode\_\_(self):*** *This method to return an unicode string of a type.*
  + ***\_\_format\_\_(self, formatstr):*** *return a new style of string.*
  + [***\_\_hash\_***](https://www.geeksforgeeks.org/python-hash-method/)***\_(self):*** *It has to return an integer, and its result is used for quick key comparison in dictionaries.*
  + ***\_\_nonzero\_\_(self):*** *Defines behavior for when bool() is called on an instance of your class.*
  + [***\_\_dir\_***](https://www.geeksforgeeks.org/python-dir-function/)***\_(self):*** *This method to return a list of attributes of a class.*
  + [***\_\_sizeof\_\_(***](https://www.geeksforgeeks.org/difference-between-__sizeof__-and-getsizeof-method-python/)***self):*** *It return the size of the object.*
* ***Comparison magic methods***
  + [***\_\_eq\_\_***](https://www.geeksforgeeks.org/difference-between-__eq__-vs-is-vs-in-python/)***(self, other):*** *Defines behavior for the equality operator, ==.*
  + ***\_\_ne\_\_(self, other):*** *Defines behavior for the inequality operator, !=.*
  + [***\_\_lt\_\_***](https://www.geeksforgeeks.org/python-__lt__-magic-method/)***(self, other):*** *Defines behavior for the less-than operator, <.*
  + [***\_\_gt\_\_***](https://www.geeksforgeeks.org/customize-your-python-class-with-magic-or-dunder-methods/)***(self, other):*** *Defines behavior for the greater-than operator, >.*
  + ***\_\_le\_\_(self, other):*** *Defines behavior for the less-than-or-equal-to operator, <=.*
  + [***\_\_ge\_\_***](https://www.geeksforgeeks.org/customize-your-python-class-with-magic-or-dunder-methods/)***(self, other):*** *Defines behavior for the greater-than-or-equal-to operator, >=.*
* main()
  + def main():
  + …statements
  + if \_\_name\_\_ == '\_\_main\_\_':
  + main()

**Class Method vs Static Method**

* A class method takes class as the first parameter while a static method needs no specific parameters.
* A class method can access or modify the class state while a static method can’t access or modify it.
* In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
* We use @classmethod decorator in Python to create a class method and we use @staticmethod decorator to create a static method in Python.

class Geeks:

course = 'DSA'

list\_of\_instances = []

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

self.name = name

Geeks.list\_of\_instances.append(self)

@classmethod

def get\_course(cls):

return f"Course: {cls.course}"

@classmethod

def get\_instance\_count(cls):

return f"Number of instances: {len(cls.list\_of\_instances)}"

@staticmethod

def welcome\_message():

return "Welcome to Geeks for Geeks!"

# Creating instances

g1 = Geeks('Alice')

g2 = Geeks('Bob')

# Calling class methods

print(Geeks.get\_course())

print(Geeks.get\_instance\_count())

# Calling static method

print(Geeks.welcome\_message())

Output

Course: DSA  
Number of instances: 2  
Welcome to Geeks for Geeks!

**DSA**

List; Tuple; Dict; Set; Frozen Set; String; Matrix(numpy); ByteArray; LinkedList; Stack; Queue; Priority Queue;

Heap; Binary Tree; Binary Search Tree; Graphs; Breath First Search; Depth First Search; Recursion;

Dynamic Programming; (**Tabulation:** Bottom Up; **Memoization:** Top Down); Searching; Sorting

RegEx\*

OpenCV, Scikit-learn, Pandas, NumPy, Matplotlib, SciPy, Seaborn, PyTorch, TensorFlow, Collections