# Requests Module

Intro

<https://www.linkedin.com/posts/gauravcode_api-reference-ugcPost-7193437847708708864-tBym?utm_source=share&utm_medium=member_desktop>

| **Method** | **Description** |
| --- | --- |
| [GET](https://www.geeksforgeeks.org/get-method-python-requests/) | GET method is used to r*etrieve information from* the given *server using a given UR*I. |
| [POST](https://www.geeksforgeeks.org/post-method-python-requests/) | *POST* request method *requests* that a *web server accepts the data enclosed in* the *body* of the request message, most likely for storing it |
| [PUT](https://www.geeksforgeeks.org/put-method-python-requests/) | The PUT method requests that the *enclosed entity* *be stored under the supplied URI*. *If* the *URI refers to* an a*lready existing resource*, *it is modified* and *if the URI does not point to an existing resource*, then the *server can create the resource* with that URI. |
| [DELETE](https://www.geeksforgeeks.org/delete-method-python-requests/) | The DELETE method *deletes the specified resource* |
| [HEAD](https://www.geeksforgeeks.org/head-method-python-requests/) | The HEAD method *asks for a response* *identical to that of a GET reque*st, *but without the response body*. |
| [PATCH](https://www.geeksforgeeks.org/patch-method-python-requests/) | It is *used* for *modifying capabilities*. The PATCH request *only needs to contain* the *changes to* the *resource*, not the complete resource |

| request(*method*, *url*, *args*) | *Sends* a *request of* the *specified method to* the *specified url* |
| --- | --- |

# Numpy vs Pandas

| **Pandas** | **NumPy** |
| --- | --- |
| When we *have to work on Tabular data*, we prefer the pandas module. | When we *have to work on Numerical data*, we prefer the NumPy module. |
| The *powerful tools* of pandas *are* *DataFrame and Series*. | Whereas the powerful *tool of NumPy is* *Arrays*. |
| Pandas *consume more memory*. | Numpy is *memory efficient*. |
| Pandas have a *better performance* *when* the number of *rows is 500K or more*. | Numpy has *better performance* *when* the number of r*ows is 50K or less*. |
| *Indexing of* the *Pandas* series *is very slow* as compared to Numpy arrays. | *Indexing* of Numpy arrays is *very fast*. |
| *Pandas have* a *2D table object* *called DataFrame*. | *Numpy* is *capable of providing* *multi-dimensional arrays.* |
| It was developed by Wes McKinney and was released in 2008. | It was developed by Travis Oliphant and was released in |
| It is used in a lot of organisations like Kaidee, Trivago, Abeja Inc., and a lot more. | It is being used in organisations like Walmart Tokopedia, Instacart, and many more. |
| It has a higher industry application. | It has a lower industry application. |

# Interview question

<https://www.interviewbit.com/python-interview-questions/>

##### Module vs Package vs Library

Python *packages* and Python *modules* are *two mechanisms* *that allow for modular programming* in Python. *Modularizing has several* *advantages* -

* *Simplicity*: Working on a single module helps you focus on a relatively small portion of the problem at hand. This makes development easier and less error-prone.
* *Maintainability*: Modules are designed to enforce logical boundaries between different problem domains. If they are written in a manner that reduces interdependency, it is less likely that modifications in a module might impact other parts of the program.
* *Reusability*: Functions defined in a module can be easily reused by other parts of the application.
* *Scoping*: Modules typically define a separate namespace, which helps avoid confusion between identifiers from other parts of the program.

*Modules*, in general, are *simply* Python *files with a .py extension* and can *have a set of functions, classes, or variables* defined and implemented. They can be *imported and initialised* once *using* the *import statement*. *If partial functionality* is *needed*, *import* the requisite classes or functions *using from foo import bar*.

*Packages* allow for *hierarchical structuring of* the *module* namespace *using dot notation*. As *modules* help *avoid clashes between* *global variable names*, in a similar manner, *packages* help *avoid clashes between* *module names*.  
*Creating a package* is *easy* *since it* makes *use* of the *system's inherent file structure*. *So* just *stuff* the *modules into a folder* *and* there you have it, the *folder name as* the *package name*. *Importing* a module or its *contents from this package* r*equires* the *package name* as prefix to the module name *joined by a dot*. *Python modules and also having \_\_init\_\_.py file by which the interpreter interprets it as a Package.*

*Note: You can technically import the package as well, but alas, it doesn't import the modules within the package to the local namespace, thus, it is practically useless.*

*library is a collection of packages.*

##### Role of \_ variable in python? Access modifiers

Underscore (*\_*) can be *used in many ways* in a Python program.

* *Use in Interpreter*:

Python immediately *save*s the *value of the last expression* *in* the *interpreter* *in this unique variable*. Underscore (\_) can also be used to value any other variable.



Underscore (*\_*) *can also be used as* a *normal variable*.

# Storing value in \_

\_ = 2 + 8

print(\_)

* *Use in Loops*:

In Python underscore (*\_*) can be *used as a variable in looping*. It will access each element of the data structure.

# Creating tuple

Tuple = (50, 40, 30)

# Using \_ to access index of each element

for \_ in range(3):

print(Tuple[\_])

* *Use in Ignoring Variables*:

In Python underscore (\_) is *often* *used to ignore a value*. If one doesn’t use some values when unpacking, just set the value to underscore (\_). Ignoring involves assigning values to a particular vector underscore (\_). We add values to underscore (\_) if this is not used in future code.

# Using \_ to ignore values

p, \_, r = 'Geeks', 4, 'Geeks!'

print(p, r)

* *Separating digit of Numbers*:

Underscores (\_) can also be *used to represent long digits*, it *separates* the *group of digits* *for better understanding*.

# Using \_ to separate digits

Crore = 10\_00\_000

print(Crore)

* *Use in Defining* *Access of Data members* *and Methods* in Class:

*Underscore* (\_) is u*sed as a prefix* *for a method or data member* in a class, *defines* its [Access Specifier](https://www.geeksforgeeks.org/access-modifiers-in-python-public-private-and-protected/), and u*ses double* *underscores* *(\_\_)* *as both suffix and prefix* *refer to a* [*Constructor*](https://www.geeksforgeeks.org/constructors-in-python/).

class Gfg:

a = None

\_b = None

\_\_c = None

# **Constructor**

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

# Data members

# **Public**

self.a = a

# **Protected**

self.\_b = b

# **Private**

self.\_\_c = c

# Methods

# **Private** method

def \_\_display(self):

print(self.a)

print(self.\_b)

print(self.\_\_c)

# **Public** method

def accessPrivateMethod(self):

self.\_\_display()

# Driver code

# Creating object

Obj = Gfg('Geeks', 4, "Geeks!")

# Calling method

Obj.accessPrivateMethod()

*Access Specifier* → Public, protect: \_, private: \_\_

# 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* *can be derived from child***

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 *for above private 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() **or** \_displayProtectedMembers()

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)

**Output:**

Public Data Member: Geeks

Protected Data Member: 4

Private Data Member: Geeks !

Object is accessing protected member: 4

##### Decorators in python

[***Decorators***](https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/) are a very powerful and useful tool in Python since it *allow*s programmers *to modify the behaviour of* *a function* *or class*. Decorators *allow us to wrap* *another function* *in order to extend* the *behaviour of* the *wrapped function*, *without permanently modifying* *it*. But *before diving deep into decorators* l*et us understand some concepts* that will come in handy in learning the decorators.

First Class Objects

In Python, functions are[*first class objects*](https://www.geeksforgeeks.org/first-class-functions-python/) which *means* that *functions* in Python *can be used* *or passed as* *arguments*.

Properties of first class functions:

* A *function* *is an instance of* the *Object type*.

# Python program to illustrate functions can be treated as objects

def shout(text):

return text.upper()

print(shout('Hello'))

yell = shout

print(yell('karan'))

HELLO

KARAN

In the above example, *we* have *assigned* the *function* shout *to a variable* yell. *This will not call the function* *instead it takes* the *function objec*t *referenced by* a *shout* *and creates* a *second name pointing to it*, yell.

* You *can store* the *function in a variable*.
* You *can pass* the *function as a parameter* to another function.

# Python program to illustrate functions can be passed as arguments to other functions

def shout(text):

return text.upper()

def whisper(text):

return text.lower()

def greet(func):

# storing the function in a variable

greeting = func("""Hi, I am created by a function passed as an argument.""")

print (greeting)

greet(shout)

greet(whisper)

HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.

hi, I am created by a function passed as an argument.

* You *can return* the *function from a function*.

# Python program to illustrate functions Functions can return another function

def create\_adder(x):

def adder(y):

return x+y

return adder

add\_15 = create\_adder(15) #add this call x is set to 15 and it returns adder func

print(add\_15(10)) #on this returned value we set y

25

In the above example, *we have created* a *function inside of another function* and then *have returned* the *function created inside.*

* You *can store them* *in* data structures such as hash tables, *lists*, …

Decorators

In Decorators, *functions* are *taken as* the *argument into another function* and then called inside the wrapper function.

Decorator can modify the behaviour:

# defining a decorator

def hello\_decorator(func):

# inner1 is a Wrapper function in which the argument is called

# inner function can access the outer local functions like in this case "func"

def inner1():

print("Hello, this is before function execution")

# calling the actual function now inside the wrapper function.

func()

print("This is after function execution")

return inner1

# defining a function, to be called inside wrapper

def function\_to\_be\_used():

print("This is inside the function !!")

# passing 'function\_to\_be\_used' inside the decorator to control its behaviour

function\_to\_be\_used = hello\_decorator(function\_to\_be\_used)

# calling the function

function\_to\_be\_used()

@hello\_decorator

def function\_to\_be\_used():

print("This is inside the function !!")

# calling the function

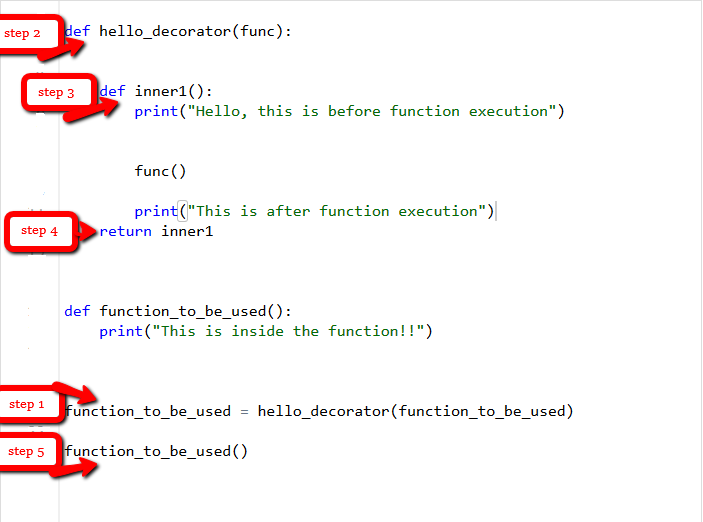
function\_to\_be\_used()

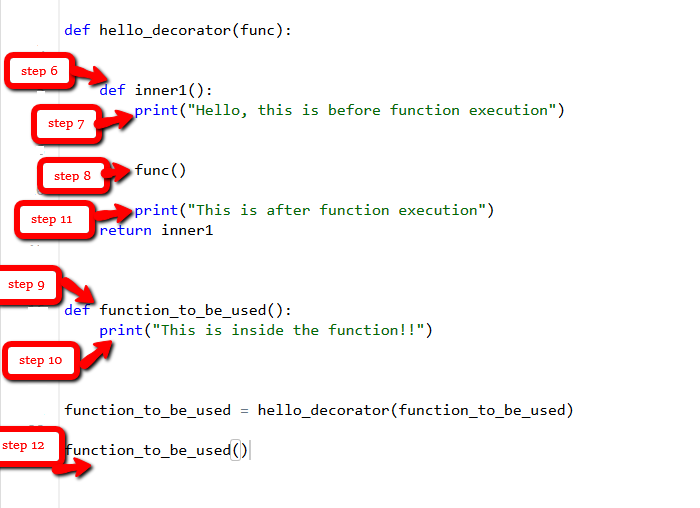
Both will have same result

Hello, this is before function execution

This is inside the function !!

This is after function execution





Chaining Decorators

In simpler terms *chaining decorators* *means decorating* *a function with multiple decorators*.

# code for testing decorator chaining

def decor1(func):

def inner():

x = func()

return x \* x

return inner

def decor(func):

def inner():

x = func()

return 2 \* x

return inner

@decor1

@decor

def num():

return 10

@decor

@decor1

def num2():

return 10

print(num())

print(num2())

Output:

400

200

The above example is similar to calling the function as –

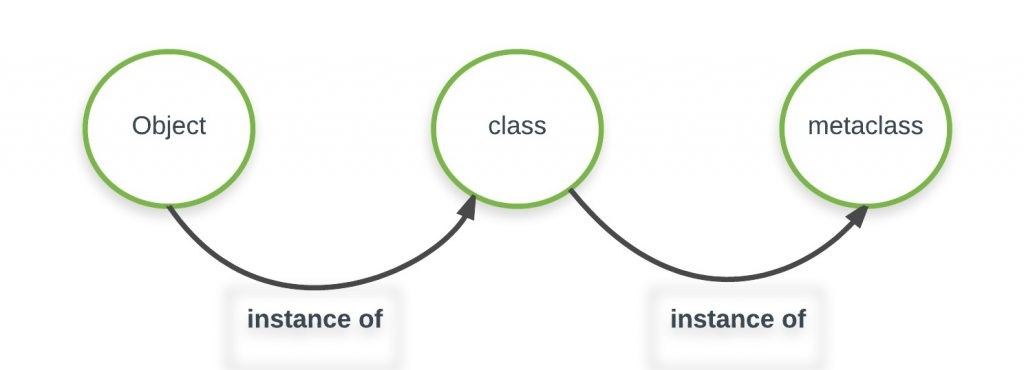
decor1(decor(num))

decor(decor1(num2))

##### Meta classes in python?

As quoted by Tim Peters

*“Metaclasses are deeper magic that 99% of users should never worry about. If you wonder whether you need them, you don’t (the people who actually need them know with certainty that they need them, and don’t need an explanation about why). “*



*Metaclass create Classes* and *Classes creates objects*

*Metaclass* is *responsible for* the *generation of classes*, so *we can write our custom metaclasses* *to modify the way classes are generated* by performing extra actions or injecting code. Usually, we do not need custom metaclasses but sometimes it’s necessary.

*In Python*, *everything has some type* associated with it. For example, *if* *we have* a variable having *an integer value* *then its type is int*. You can get the type of anything using the type() function.

num = 23

print("Type of num is:", type(num))

lst = [1, 2, 4]

print("Type of lst is:", type(lst))

name = "Atul"

print("Type of name is:", type(name))

Type of num is: <class 'int'>

Type of lst is: <class 'list'>

Type of name is: <class 'str'>

class Student:

pass

stu\_obj = Student()

# Print type of object of Student class

print("Type of stu\_obj is:", type(stu\_obj))

Type of stu\_obj is: <class '\_\_main\_\_.Student'>

*Every type in Python* *is* defined by *Class*. So in the above example, *unlike C++ or Java* *where int, char, float* *are primary data types*, *in* *Python they are* *objects of int class* or str class. So *we can make a new type* *by creating a class of that type*. For example, we can create a new type of Student by creating a Student class.

class Student:

pass

# Print type of Student class

print("Type of Student class is:", type(Student))

Type of Student class is: <class 'type'>

A *Class is also an object*, and just like any other object, i*t’s an instance of* something called *Metaclass*. A *special class* ***type*** *creates these* Class *objects*. The *type* class *is the default metaclass* which is responsible for making classes. *In* the *above example*, *if we try to* *find* out *the type of Student class*, *it comes out to be a type*.

*Because Classes are also an object*, *they can be modified in* the *same way*. ***We can add or subtract fields*** or methods in class in the same way we did with other objects. For example –

# Defined class without any class methods and variables

class test:pass

# Defining method variables

test.x = 45

# Defining class methods

test.foo = lambda self: print('Hello')

# creating object

myobj = test()

print(myobj.x)

myobj.foo()

45

Hello

***Just note that we don't send parameters to class. We send parameter to Class’s function***

*When we create* *an object of class* and pass *parameter* to it at that time it *goes to \_\_init\_\_() constructor*

When we create a class test(parameter) and pass some parameter to it at that time its performing inheritance.

Created class object and passing parameter at that time it's directly going to class’s function.

##### Abstract Class

An abstract class can be considered a *blueprint for other classes*. It *allows you to* *create a set of methods* that *must be created within any child classes* *built from* the *abstract class*. An *abstract method* is a *method that has a declaration* but *does not have an implementation*. This capability is especially *useful* in situations *where* a *third party is going to provide* *implementations*, such as with plugins, but can also help you when working in a large team or with a large code base where keeping all classes in your mind is difficult or not possible. By default, Python does not provide abstract classes. *Python* *comes with a module* *that provides* the *base for defining* Abstract Base classes(*ABC*) and that *module name is ABC*.

ABC w*orks by decorating methods* *of* the *base class as an abstract* and *then registering* *concrete classes as implementations* *of* the abstract *base*. A *method becomes abstract* *when decorated with* the keyword *@abstractmethod*.

# Python program showing abstract base class work

from abc import ABC, abstractmethod

class Polygon(ABC):

@abstractmethod

def noofsides(self):

pass

class Triangle(Polygon):

# overriding abstract method

def noofsides(self):

print("I have 3 sides")

class Pentagon(Polygon):

# overriding abstract method

def noofsides(self):

print("I have 5 sides")

# Driver code

R = Triangle()

R.noofsides()I have 3 sides

R = Pentagon()

R.noofsides()I have 5 sides

An *abstract class* *can have a non abstract method*, *when inheriting* this *sub class* we *don't need to overwrite* the *abstract class*’s method. *Can create* an *object of abstract class* but *not of abstract class* *with an abstractmethod.* Please check example below

from abc import ABC, abstractmethod

class Animal2(ABC):

@abstractmethod

def move(self):

pass

class Animal(ABC):

def move(self):

pass

class Human(Animal):

def movement(self):

print("I can walk and run")

R = Human()

R.movement() #I can walk and run

S = Animal() #<\_\_main\_\_.Animal object at 0x7af959c4b350>

S = Animal2() #Throws Error

##### Class method vs Static method in Python

The @*classmethod decorato*r is *a* built-in [*function decorator*](https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/) that is an expression that *gets evaluated after your function is defined*. The *result of that evaluation* *shadows* your *function definition*. A [***class method***](https://www.geeksforgeeks.org/classmethod-in-python/) ***receives*** the ***class as an implicit first argument cls***, *just like* an *instance method receives the instance*(self,.

class C(object):

@classmethod

def fun(**cls**, arg1, arg2, ...):

....

fun: function that needs to be converted into a class method

returns: a class method for function.

* A *class method* is a method that is *bound to the* [*class*](https://www.geeksforgeeks.org/python-classes-and-objects/) and *not the object* of the class.
* They have *access to the state* *of* the *class* *as it takes* a *class parameter* *that points to the clas*s and *not the object instance*.
* It *can modify a class state* that *would apply across all* the *instances* of the class. For example, it can modify a class variable that will be applicable to all the instances.

A [*static method*](https://www.geeksforgeeks.org/python-staticmethod/) ***does not receive*** an *implicit first argument*. A static method is *also* a method that is *bound to the class* *and not the object* of the class. This method *can’t access or modify the class state*. It is present in a class because it makes sense for the method to be present in class.

class C(object):

@staticmethod

def fun(arg1, arg2, ...):

...

returns: a static method for function fun.

Class method vs Static Method

The *difference between* the Class method and the static method is:

* A *class method takes cls* *as* the *first parameter* *while* a *static method needs no specific parameters*.
* A *class method* *can access or modify* the *class state* *while* a s*tatic method* *can’t* access or modify it.
* *In general*, s*tatic methods* *know nothing about* the *class state*. *They are* *utility-type method*s 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.

When to use the class or static method?

* We generally *use* the *class method to create factory methods*. *Factory methods return class objects* ( similar to a constructor ) for different use cases.
* We generally *use static methods* *to create utility functions*.

##### Method resolution order

Method Resolution Order(*MRO*) *denotes* the *way a programming language resolves* a *method* or attribute. In python, method resolution order *defines* the *order in which* the base *classes are searched* when executing a method. *First*, the *method or attribute is* *searched within a class* and *then it follows* the *order we specified* while *inheriting*. This order is also called Linearization of a class and the set of rules are called MRO(Method Resolution Order).

In multiple inheritance, the methods are executed based on the order specified while inheriting the classes. For the languages that support single inheritance, method resolution order is not interesting, but the languages that support multiple inheritance method resolution order plays a very crucial role. Let’s look over another example to deeply understand the method resolution order:

*To get* the *method resolution order of a class* we can *use either* *\_\_mro\_\_* attribute *or mro()* method. *By using these* methods we can *display the order in which* *methods are resolved*. For Example

# Python program to show the order

# in which methods are resolved

class A:

def rk(self):

print(" In class A")

class B:

def rk(self):

print(" In class B")

# classes ordering

class C(A, B):

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

print("Constructor C")

r = C()

# it prints the lookup order

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

print(C.mro())

Constructor C

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

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

##### What is monkey patching in python and its uses?

In Python, the term *monkey patch* *refers to* dynamic (or *run-time*) *modifications of a class* or module. In Python, *we can* actually *change* the *behaviour of code at run-time*.

# monk.py

class A:

def func(self):

print ("func() is being called")

We *use* the *above module* (monk) *in below code* *and change behaviour* *of func()* *at run-time* *by assigning different values*.

import monk

def monkey\_f(self):

print ("monkey\_f() is being called")

# replacing address of "func" with "monkey\_f"

monk.A.func = monkey\_f

obj = monk.A()

# calling function "func" whose address got replaced with function "monkey\_f()"

obj.func()

Output :monkey\_f() is being called

##### Polymorphism in python?

The word *polymorphism means* having *many forms*. In programming, polymorphism means the *same function name* (but different signatures) being *used for different types*. The key difference is the data types and number of arguments used in function.

Example of *inbuilt polymorphic functions*:

print(len("geeks"))

print(len([10, 20, 30]))

5

3

Examples of *user-defined polymorphic functions*:

# A simple Python function to demonstrate Polymorphism

def add(x, y, z = 0):

return x + y+z

# Driver code

print(add(2, 3))

print(add(2, 3, 4))

5

9

*Polymorphism with class methods:*

class India():

def capital(self):

print("New Delhi is the capital of India.")

class USA():

def capital(self):

print("Washington, D.C. is the capital of USA.")

obj\_ind = India()

obj\_usa = USA()

for country in (obj\_ind, obj\_usa):

country.capital()

New Delhi is the capital of India.

Washington, D.C. is the capital of USA.

*Polymorphism with a Function and objects:*

def func(obj):

obj.capital()

obj\_ind = India()

obj\_usa = USA()

func(obj\_ind)New Delhi is the capital of India.

func(obj\_usa)Washington, D.C. is the capital of USA.

*Polymorphism with Inheritance*:

In Python, Polymorphism lets us *define methods in* the *child* class *that have the same name* *as* the *methods in* the *parent* class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to *modify* a *method in* a *child class* *that it* *has inherited from* the *parent* class. This is particularly *useful* in cases *where* the *method* inherited *from* the *parent class* *doesn’t quite* *fit* the *child class*. In such cases, we re-implement the method in the child class. This *process of re-implementing* a *method* *in* the *child* class is *known as Method Overriding*.

class Bird:

def intro(self):

print("There are many types of birds.")

def flight(self):

print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

def flight(self):

print("Sparrows can fly.")

obj\_bird = Bird()

obj\_spr = sparrow()

obj\_bird.intro()There are many types of birds.

obj\_bird.flight()Most of the birds can fly but some cannot.

obj\_spr.intro()There are many types of birds.

obj\_spr.flight()Sparrows can fly.

##### How is memory management done in Python?

Python *uses* its *private heap space* *to manage* the *memory*. Basically, *all* the *objects and data structures* are *stored in* the *private heap* space. Even the *programmer* *can not access* this *private space* *as* the *interpreter takes care of this space*. Python also has an inbuilt garbage collector, which recycles all the unused memory and frees the memory and makes it available to the heap space.

##### Garbage collection in python

*Python*’s *memory allocation and deallocation* method *is automatic*. The *user does not* have to *preallocate or deallocate memory* similar to using dynamic memory allocation in languages such as C or C++.

Python uses two strategies for memory allocation:

1. Reference counting

*Python* and various other programming languages *employ reference counting*, a memory management approach, to automatically manage memory by *track*ing *how many times* *an object is referenced*. A reference count, or the number of references that point to an object, is a property of each object in the Python language. *When* an *object’s reference count* *reaches zero*, *it becomes* *un-referenceable* *and* its *memory* can be *freed up*.

import sys

# Create an object

**x** = [1, 2, 3]

# Get reference count

ref\_count = sys.getrefcount(**x**)

print("Reference count of x:", ref\_count)

Reference count of x: 2

1. Garbage collection

*Garbage collection* is *a memory management technique* used in programming languages *to automatically reclaim memory* *that is* *no longer accessible* or in use by the application. It helps prevent memory leaks, optimise memory usage, and ensure efficient memory allocation for the program.

*Generational Garbage Collection*

*When attempting to add* an *object to a reference counter*, *a cyclical reference* or reference cycle *is produced*. Because the *object’s* *reference counter* could *never reach 0 (due to cycle*), a *reference counter cannot destroy the object*. *Therefore, in situations like this*, *we employ* the *universal waste collector*. It operates and releases the memory used. A *Generational Garbage Collector* can be found in the standard library’s *gc module*.

*Automatic Garbage Collection of Cycles*

*Because reference cycles* *take computational work to discover*, *garbage collection* *must be a scheduled activity*. *Python schedules* *garbage collection* *based upon* a *threshold of object allocations* and object *deallocations*. When the *number of allocations* *minus* the number of *deallocations* is *greater than the threshold number,* the *garbage collector is run*. One can *inspect the threshold* for new objects (objects in Python known as generation 0 objects) *by importing the gc module* and asking for garbage collection thresholds:

# loading gc

import gc

# get the current collection

# thresholds as a tuple

print("Garbage collection thresholds:", *gc.get\_threshold()*)

Garbage collection thresholds: (700, 10, 10)

Here, the *default threshold* on the above system is *700*. *This means* *when* the number of *allocations* *vs.* the number of *deallocations* is *greater than 700* the automatic *garbage collector will run*. Thus any portion of your code which frees up large blocks of memory is a good candidate for running manual garbage collection.

Manual Garbage Collection

*Invoking* the *garbage collector manually* during the execution of a program can be a *good idea* *for how to handle* *memory being consumed by reference cycles*.

import gc

i = 0

# create a cycle and on each iteration x as a dictionary

# assigned to 1

def create\_cycle():

x = { }

x[i+1] = x

print(x)

# lists are cleared whenever a full collection or collection of the highest generation (2) is run

collected = gc.collect() # or gc.collect(2)

print("Garbage collector: collected %d objects." % (collected))

print("Creating cycles...")

for i in range(10):

create\_cycle()

collected = *gc.collect()*

print("Garbage collector: collected %d objects." % (collected))

Garbage collector: collected 0 objects.

Creating cycles...

{1: {...}}

{2: {...}}

{3: {...}}

{4: {...}}

{5: {...}}

{6: {...}}

{7: {...}}

{8: {...}}

{9: {...}}

{10: {...}}

Garbage collector: collected 10 objects.

# Delete references to objects

del obj1

del obj2

del obj3

# Force a garbage collection

gc.collect()

# Disable the garbage collector

gc.disable()

# Create some objects

obj1 = [1, 2, 3]

obj2 = {"a": 1, "b": 2}

obj3 = "Hello, world!"

# Delete references to objects

del obj1

del obj2

del obj3

# The garbage collector is disabled, so it will not run

There are *two ways for performing* manual *garbage collection*: time-based and event-based garbage collection.

* *Time-based* garbage collection is simple: the *garbage collector* is *called after a fixed time interval*.
* *Event-based* garbage collection *calls* the garbage *collector on event occurrence*. For example, *when* a *user exits* the application *or* when the *application enters* into an idle stat*e*.

##### Enable frequent garbage collection by design?

*To help garbage collection* *in Python*, you can *follow these* best practices:

* *Use* Context Managers and the *with Statement*: Utilise context managers and the with statement *when working with* *external resources* like files and database connections. This *ensures that resources are properly closed and released* after they're no longer needed.
* *Explicitly Close Resources*: For *resources that don't support* the *with statemen*t, e*xplicitly close them using the close()* method or appropriate cleanup functions to release resources promptly.
* *Use Generators*: *Generators* can be *memory efficient as they produce items on-the-fly* *instead of storing them* all *in memory* at once. *Consider using generators* *when* dealing with large datasets or *infinite sequences.*
* *Avoid Circular References*: Be mindful of creating circular references between objects, as this can prevent them from being garbage collected. *Break circular references* *when they're no longer needed* to allow the objects to be reclaimed by the garbage collector.
* *Avoid Unnecessary Object Creation*: Minimise unnecessary object creation, *especially within loops* or recursive functions. *Reuse* objects *when possible* to reduce memory usage and alleviate the burden on the garbage collector.
* *Profile and Optimise*: *Profile your code* t*o identify memory-intensive areas* and optimise them. Look for opportunities to reduce memory usage by optimising data structures, algorithms, and resource management.
* *Break unnecessary references*: When you're done with an object, *assign it to None* *to explicitly remove* the *reference*. This helps the garbage collector identify the object as unreachable.
* *Utilise temporary variables*: For calculations or short-lived data manipulation, consider using temporary variables \_ *instead of assigning* the result *to a named variable*. This avoids creating unnecessary objects in memory.

While the gc module provides functions to control garbage collection, it's generally recommended to rely on automatic collection and focus on writing clean, memory-conscious code.

For more advanced scenarios, you can explore the gc module for functions like gc.collect() to force collection or gc.get\_debug() to enable debugging flags. But remember, in most cases, Python's automatic garbage collection is sufficient.

##### Why is finalize used in python?

The [*weakref*](https://docs.python.org/3/library/weakref.html#module-weakref) module *allows* the *Python programmer* *to create weak references* to objects.

A *weak reference* to an object is *not enough to keep* the *object alive*: *when* the *only remaining references* to a referent *are* *weak references*, [*garbage collection*](https://docs.python.org/3/glossary.html#term-garbage-collection) is *free to destroy* the referent *and reuse* its *memory* for something else. However, *until* the *object* *is actually destroyed* the *weak reference may return the object* even if there are no strong references to it.

A *primary use* *for weak references* is *to implement caches* *or* *mappings holding large objects*, *where it’s desired that* a *large* *object not be kept alive* *solely because it appears in* a *cache* or mapping.

For example, *if you have* a number of large *binary image objects*, you may wish to associate a name with each. *If you use*d a *Python dict*ionary t*o map names to images*, or images to names, the image *objects would remain alive* *just because they appeared as* *values* or keys *in* the *dictionaries*. The [*WeakKeyDictionary*](https://docs.python.org/3/library/weakref.html#weakref.WeakKeyDictionary) and [*WeakValueDictionary*](https://docs.python.org/3/library/weakref.html#weakref.WeakValueDictionary) classes supplied by the [weakref](https://docs.python.org/3/library/weakref.html#module-weakref) module are *an alternative*, *using weak references* *to construct mappings* *that don’t keep objects alive* *solely because* *they appear in* the *mapping* objects

[*finalize*](https://docs.python.org/3/library/weakref.html#weakref.finalize) provides a *straightforward way* *to register* *a cleanup function* *to be called* *when an object is garbage collected*. This is *simpler to use* *than setting up a callback function* *on a* raw *weak reference*, *since* the *module* automatically *ensures* that the *finalizer remains alive* *until* the *object is collected*.

A *finalizer* is *considered alive* *until* *it is called* (either *explicitly or at garbage collection*), and *after that it is dead*. *Calling a live* *finalizer returns* the *result* *of evaluating func*(\*arg, \*\*kwargs), *whereas calling a dead finalize*r *returns* [*None*](https://docs.python.org/3/library/constants.html#None).

import weakref

class MyClass:

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

self.name = name

*# \_\_del\_\_() method is referred to as a destructor method. It is called after an object’s garbage*

*# collection occurs, which happens after all references to the item have been destroyed.*

def \_\_del\_\_(self):

print(f'{self.name} is being destroyed!')

def cleanup\_callback(obj\_ref):

obj = obj\_ref()

if obj is not None:

print(f'Cleaning up {obj.name}...')

obj = MyClass("example")

obj\_ref = weakref.finalize(obj, cleanup\_callback)

del obj *#Trigger garbage collection*

# Output:

# Cleaning up example...

# example is being destroyed!

##### Open() in python

The *open() function* *opens a file*, *and returns it as* a *file object*.

Parameter Values

| **Parameter** | **Description** |
| --- | --- |
| *file* | The *path and name of the file* |
| *Mode* | A string, *define which mode you want to open* the *file in*:  "r" - *Read* - Default value. *Opens* a file *for reading*, *error if* the *file does not exist*.  "a" - *Append* - *Opens* a file *for appending*, *creates* the *file if it does not exist*.  "w" - *Write* - *Opens* a file *for writing*, *creates* the *file if it does not exist*.  "x" - *Create* - Creates the specified file, *returns an error* *if the file exists*.  In addition you can specify if the file should be handled as binary or text mode  "t" - Text - Default value. *Text mode*  "b" - Binary - *Binary mode* (e.g. images) |

##### Use “with” in python?

*With statement* is *used in exception handling* *to make* the *code cleaner* and much more *readable*. It *simplifies* the *management of* common *resources* like file streams.

# file handling without using with statement

file = open('file\_path', 'w')

file.write('hello world !')

file.close()

file = open('file\_path', 'w')

try:

file.write('hello world')

finally:

file.close()

# using with statement

with open('file\_path', 'w') as file:

file.write('hello world !')

Notice that unlike the first two implementations, there is *no need to call file.close()* when using with statement. The *with statement* itself *ensures proper* *acquisition and release of resources*. An *exception during* the *file.write() call* *in* the ***first implementation*** *can prevent* the *file from closing properly* which may introduce several bugs in the code, i.e. *many changes* *in files* *do not go into effect until* the *file is* properly *closed*. The *second approach* in the above example *takes care of all* the *exceptions* but using the with statement *makes* the *code compact and much more readable*. Thus, “with” statement *helps avoid bugs and leaks* *by ensuring* that a *resource is properly released* when the code using the resource is completely executed. The with statement is *popularly used with file streams*, as shown above and with *Locks, sockets, subprocesses* and telnets etc.

##### Parse Json in Python

To *convert json string* *to python object* use json.load**s**()

# Python program to convert JSON to Python

import json

# JSON string

employee =**'**{"id":"09", "name": "Nitin", "department":"Finance"}**'**

# Convert string to Python dict

employee\_dict = json.load**s**(employee)

print(employee\_dict)

print(employee\_dict['name'])

*Use* open() function *to read JSON* files. Then, *parsed using* json.load() method *which gives us* a *dictionary named data*



import json

# Opening JSON file

f = open('data.json',)

# returns JSON object as a dictionary

data = json.load(f)

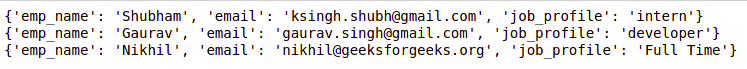
# Iterating through the json list

for i in data['emp\_details']:

print(i)

# Closing file

f.close()



*Convert*ing a [Python dictionary](https://www.geeksforgeeks.org/python-dictionary/) *to a JSON* object *using*[json.dump**s**()](https://www.geeksforgeeks.org/json-dumps-in-python/) method of JSON module in Python.

# Python program to convert Python dict to JSON

import json

# Data to be written

dictionary = {

"id": "04",

"name": "sunil",

"department": "HR"

}

# Serialising json

json\_object = [json.dump**s**](https://www.geeksforgeeks.org/json-dumps-in-python/)(dictionary, indent = 4)

print(json\_object)

*Write JSON to file* *using* json.dump(). In the below program, *we have open*ed a *file named sample.json in writing mode* using ‘w’. The *file will be created if it does not exist*. *Json.dump() will transform* the Python *dictionary to a JSON* string *and* it *will be* *saved in* the file *sample.json*.

# Python program to write JSON to a file

import json

# Data to be written

dictionary ={

"name" : "sathiyajith",

"rollno" : 56,

"cgpa" : 8.6,

"phonenumber" : "9976770500"

}

with open("sample.json", "w") as outfile:

json.dump(dictionary, outfile)

Comparison between load loads and dump dumps

Tip: S stands for string. Either convert **string** to python object(json.load**s**()) or vice-versa (json.dump**s**()). Other functions are to read **files** to a python object(json.load()) or vice-versa(json.dump()).

| **Category** | **json.loads()** | **json.load()** | **json.dumps()** | **json.dump()** |
| --- | --- | --- | --- | --- |
| Definition | To convert json string to python object use json.loads() | Use open() function **to** read JSON **files**. Then, parsed using json.load() method which gives us a dictionary named data | Converting a Python dictionary to a JSON object using json.dumps() | Write JSON **to file** using json.dump(). |
| Example | employee ='{"id":"09", "name": "Nitin", "department":"Finance"}'  employee\_dict = json.loads(employee) | f = **open**('data.json',)  data = json.load(f) | employee = {"id":"09", "name": "Nitin", "department":"Finance"}  json\_object = json.dumps(employee, indent = 4) | employee = {"id":"09", "name": "Nitin", "department":"Finance"}  with **open**("sample.json", "w") as outfile:  json.dump(employee, outfile) |

##### What are Pickling and Unpickling?

Pickling and unpickling are *processes* in Python used *for serialising and deserializing objects*. Here's a brief rundown:

Pickling: *Pickling* is the *process of converting* a *Python object* *into a byte stream*. This *byte stream* can *then* be *stored in a file*, *sent over* a *network*, or otherwise persisted. The *pickle module* in Python *facilitates this serialisation* process.

In this example, we are *creating* a *file named ‘person.pickle’* that *stores* the *serialised form of* a *Python object*. We will *create a dictionary* object ‘*person*’ *which will be serialised*. The file object represents the file that will be used for writing the pickled object. The *pickle.dump()* function is then *used to pickle* the *person object to* the *file*. It takes *two arguments* – the *object to be pickled* and the *file object to which* the *pickled object should be written*.

import pickle

# Define a Python object

person = {

"name": "Alice",

"age": 30,

"gender": "female"

}

# Pickle the object to a binary file

with open("person.pickle", "wb") as file:

pickle.dump(person, file)

print("Pickling completed")

Pickling completed

Unpickling: Unpickling is the inverse process of pickling. It involves *reconstructing a* *Python object* *from a byte stream*. This is typically *done by reading* the *byte stream from a file* or *receiving it* *over a network* and *using* the *pickle* module *to deserialize it back* into a Python object.

In this example, we will *load the pickle file* *in* our Python *code using* the *load() function* of the pickle module. The pickle.load*()* function is *used to deserialize* *and unpickle* the *object from the file*. It *takes one argument* – the *file object from which* the *object should be loaded*. The *unpickled object is stored in* the *variable data*.

import pickle

# load the data from a file

with open('data.pkl', 'rb') as f:

data = pickle.load(f)

# print the data

print(data)

{'name': 'Alice', 'age': 30, 'gender': 'female'}

The *purpose of* *pickling and unpickling* is *primarily to facilitate* the persistence and *transfer of Python objects between different sessions* or environments. This can be *useful for* tasks such as *caching*, *saving program state*, or *communicating between Python processes*.

##### How to delete a file using Python?

We can delete a file using Python by following approaches:

os.remove()

os.unlink()

##### Global interpreter lock in Python

Python *Global Interpreter Lock* (GIL) is a *type of process lock* which is *used by python* *whenever it deals with processes*. *Generally*, *Python* *only uses only one thread* to execute the set of written statements. *This means* that in python *only one thread* will be *executed at a time*. The *performance of* the *single-threaded process* *and* the *multi-threaded process* will be the *same in python* and this is *because of GIL* in python.

What problem did the GIL solve for Python :

*Python has* something that no other language has that is a *reference counter*. *With* the *help of* the *reference counter*, we can *count* the *total number of references* that are *made internally in python* to assign a value to a data object. Due to this counter, we can count the references and *when this count reaches zero* the *variable* or data object will be *released automatically*. For Example

# Python program showing

# use of reference counter

import sys

geek\_var = "Geek"

print(sys.getrefcount(geek\_var))

string\_gfg = geek\_var

print(sys.getrefcount(string\_gfg))

4

5

*This reference counter variable* *needed to be protected*, *because sometimes* two *threads increase or decrease its value* *simultaneously* by doing that it may lead to memory leaked so in order *to protect thread* *we add locks to all data structures* that are *shared across threads* *but sometimes by adding locks* there exists a multiple locks which *lead to* another problem that is *deadlock*. In order to avoid memory leak and deadlock problems, we used a single lock on the interpreter that is Global Interpreter Lock(GIL).

*Impact on multi-threaded/Multiprocessing Python programs* :

# Python program showing CPU bound program

import time

COUNT = 50000000

def countdown(n):

while n>0:

n -= 1

start = time.time()

countdown(COUNT)

end = time.time()

print('Time taken in seconds -', end - start)

Time taken in seconds - 2.5236213207244873

# Python program showing two threads running parallel

import time

from threading import Thread

COUNT = 50000000

def countdown(n):

while n>0:

n -= 1

t1 = Thread(target = countdown, args =(COUNT//2, ))

t2 = Thread(target = countdown, args =(COUNT//2, ))

start = time.time()

t1.start()

t2.start()

t1.join()

t2.join()

end = time.time()

print('Time taken in seconds -', end - start)

Time taken in seconds - 2.183610439300537

*As you can see*, In the above code two code where *CPU bound process* and *multithreaded process* *have* the *same performance* because in CPU bound program GIL restricts CPU to only work with a single thread. The *impact of CPU bound* thread and *multi-threading* *will be* the *same* in python.

*In this implementation*, *python provides* a *different interpreter to each process* *to run* so in this case the single thread is provided to each process in multiprocessing.

# Python program showing multiprocessing

import multiprocessing

import time

COUNT = 50000000

def countdown(n):

while n>0:

n -= 1

if \_\_name\_\_ == "\_\_main\_\_":

# creating processes

start = time.time()

p1 = multiprocessing.Process(target = countdown, args =(COUNT//2, ))

p2 = multiprocessing.Process(target = countdown, args =(COUNT//2, ))

# starting process 1

p1.start()

# starting process 2

p2.start()

# wait until process 1 is finished

p1.join()

# wait until process 2 is finished

p2.join()

end = time.time()

print('Time taken in seconds -', end - start)

Time taken in seconds - 2.5148496627807617

As you can see, there is *no difference between* the *time taken by* the *multi-threaded system* *and* the *multi-processing system*.

Hence,it’s not a good idea to use multi thread to speed up python code and its not possible **MULTIPROCESSING IS DEBATABLE IN INTERVIEW SAY ITS ALLOWED**

##### Try, Except, else and Finally in Python

An *Exception is an Unexpected Event*, which occurs during the execution of the program. It is also *known as* a *runtime error*. *When* that *error occurs*, [*Python*](https://www.geeksforgeeks.org/python-programming-language/) *generates an exception* during the execution and *that can be handled,* *which prevents your program from interrupting*.

Exception handling with try, except, else, and finally

* Try: This *block will test* the *expected error to occur*
* Except: Here you *can* *handle the error*. *Can have* *more than one except clause*.
* Else: *If* there is *no exception* then *this block will be executed*
* Finally: *Finally block always gets executed* either *exception is generated or not.* ***Even if you return* *in the except block still the finally block will execute***

try:

# Some Code....

except:

# optional block

# Handling of exception (if required)

else:

# execute if no exception

finally:

# Some code .....(always executed)

A *try statement* *can have* *more than one except clause*.

try:

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

result = 10 / x

except ZeroDivisionError:

print("You cannot divide by zero.")

except ValueError:

print("Invalid input. Please enter a valid number.")

except Exception as e:

print(f"An error occurred: {e}")

finally:

# this block is always executed regardless of exception generation.

print('This is always executed')

##### User Defined exception

[*Python*](https://www.geeksforgeeks.org/python-programming-language/) *throws* *errors and exceptions* *when* the *code goes wron*g, which may *cause the program to stop abruptly*. *Python* also *provides* an *exception handling method* *with* the help of [*try-except*](https://www.geeksforgeeks.org/python-try-except/). Some of the *standard exceptions* which *are* most frequent include *IndexError, ImportError, IOError, ZeroDivisionError, TypeError, and FileNotFoundError*.

*User-Defined Exception*s *in Python* need to be ***derived from the Exception class***, either *directly or indirectly*. Although not mandatory, *most of the exceptions* are *named as names* t*hat end in* ***“Error*”** similar to the naming of the standard exceptions in python.

class CustomError(Exception):

pass

raise CustomError("Example of Custom Exceptions in Python")

**Output:** CustomError: **Example of Custom Exceptions in Python**

*In* the *below* article, we have *created a class named “*Error” *derived from* the class *Exception*. This *base class is inherited by* *various user-defined classes* *to handle different types of* ***python raise an exception* with message**

# define Python user-defined exceptions

class Error(Exception):

"""Base class for other exceptions"""

pass

class zerodivision(Error):

"""Raised when the input value is zero"""

pass

try:

i\_num = int(input("Enter a number: "))

if i\_num == 0:

raise zerodivision

except zerodivision:

print("Input value is zero, try again!")

print()

Enter a number: 0

Input value is zero, try again!

##### Different types of exceptions in python:

In Python, there are *several built-in* Python *exceptions* that *can be raised when an error occurs* during the execution of a program. Here are some of the most common types of exceptions in Python:

* *SyntaxError*: This exception is *raised when* the *interpreter encounters a syntax error* in the code, such as a misspelt keyword, a missing colon, or an unbalanced parenthesis.
* *TypeError*: This exception is *raised when* an *operation or function* is *applied to an object* *of* the *wrong type*, such as *adding a string to an integer*.
* *NameError*: This exception is raised *when a variable* *or function* name is *not found in* the *current scope*.
* *IndexError*: This exception is raised *when an index is out of range for a list*, tuple, or other sequence types.
* *KeyError*: This exception is raised *when* a *key is not found in a dictionary*.
* *ValueError*: This exception is raised *when a function* or method is *called with an invalid argument* or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
* *AttributeError*: This exception is raised *when an attribute* *or method* is *not found on an object*, such as trying to access a non-existent attribute of a class instance.
* *IOError*: This exception is raised *when an I/O operation*, such as reading or writing a file, *fails due to an input/output error*.
* *ZeroDivisionError*: This exception is raised *when an attempt* is *made to divide a number by zero*.
* *ImportError*: This exception is *raised when an import statement* *fails to* find or *load a module*.

##### What are Exception Groups in Python?

It is a collection/*group of different kinds of Exception*. *Without creating Multiple Exceptions* *we can group* *together different Exceptions* *which we can later fetch* *one by one whenever necessary*, the *order in which* the *Exceptions are stored* in the Exception Group *doesn’t matter* while calling them.

***Syntax:******ExceptionGroup****(*

*“<User\_Message/Description>”,*

***[****<SubException1(“exception message”)>, <SubException2(“exception message”)>,…. <SubExceptionN(“exception message”)>****]***

*)*

**Parameter:** ExceptionGroup *takes two parameters*.

* **First\_Parameter** – A *message*/description *of what kind of Exceptions* *it will be storing* or anything the user wants to write there as a message of the ExceptionGroup.
* **Second\_Parameter** – *Any amount of sub-exceptions* *users want to store* with their respective messages.

As you can see, *even though we have created* a *function* ***exceptionGroup()* *we are not returning any value***, *rather we are* ***raising*** the *variable* in which we are storing the ExceptionGroup, *because* as we know in case of exception handling *we* ***raise*** *an Exception*.

def exceptionGroup():

exec\_gr = ExceptionGroup('ExceptionGroup Message!',

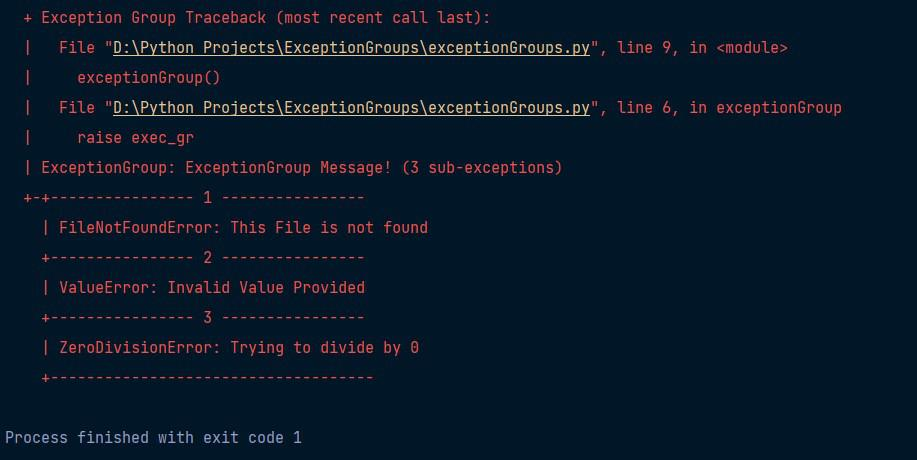
[FileNotFoundError("This File is not found"),

ValueError("Invalid Value Provided"),

ZeroDivisionError("Trying to divide by 0")])

raise exec\_gr

exceptionGroup()



There is *another* great *way to* ***handle*** ***Sub* *Exceptions one by one*** *without handling them* all together l*ike the above* traditional *method*. For that, there *is a* new syntax – *except\**, which is *used specifically with ExceptionGroup*.

**except\*** <ExceptionName> as <any\_alias>:

print(<any\_alias>.exception)

try:

exceptionGroup()

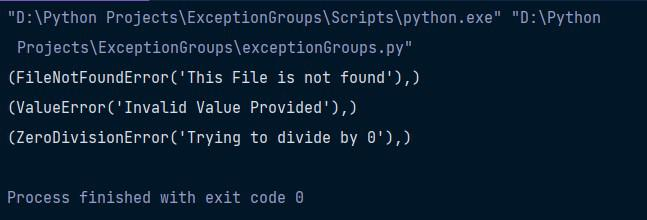
except\* FileNotFoundError as fnf:

print(fnf.exceptions)

except\* ValueError as ve:

print(ve.exceptions)

except\* ZeroDivisionError as zde:

print(zde.exceptions)

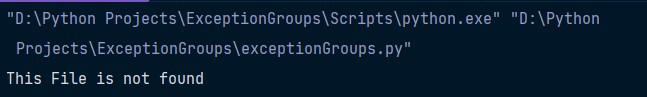
if we want to *only print* the *messages then* we have to *write code as below*:

# only one except part is given here, the try block would be the same. the way of printing the rest of the except blocks will be same also

except\* FileNotFoundError as fnf:

for err in fnf.exceptions:

print(err)



##### Why isn't all memory freed when python exits?

*CPython interacts with* the *C library for low-level tasks*. Sometimes, m*emory allocated through* the *C library* *might not be managed by Python's garbage collector*. The operating system will typically reclaim this memory when the entire process (not just the Python interpreter) exits.

##### Assert, pass, continue, is ==, isinstance, issubclass, super() in python

**Assert**

The *assert* keyword is *used when debugging code*. The assert keyword lets you *test if a condition in your code returns True*, *if not*, the program will *raise an AssertionError*. You can write a message to be written if the code returns False, check the example below.

x = "hello"

#if condition returns True, then nothing happens:

assert x == "hello"

#if condition returns False, AssertionError is raised:

assert x == "goodbye"

Traceback (most recent call last):

File "demo\_ref\_keyword\_assert.py", line 5, in <module>

assert x == "goodbye"

AssertionError

**issubclass**()

Syntax: issubclass( object, classinfo )

Parameters:

Object: class to be checked

classinfo: class, types or a tuple of classes and types

Return Type: *True* *if* the *object is a subclass of a class*, or any element of the tuple, otherwise False.

# Defining Parent class

class Vehicles:

# Constructor

def \_\_init\_\_(vehicleType):

print('Vehicles is a ', vehicleType)

# Defining Child class

class Car(Vehicles):

# Constructor

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

Vehicles.\_\_init\_\_('Car')

# Driver's code

print(issubclass(Car, Vehicles))True

print(issubclass(Car, list))False

print(issubclass(Car, Car))True

print(issubclass(Car, (list, Vehicles)))True

print( issubclass(type(1), int) )True

print(issubclass( 1, int ))TypeError: issubclass() arg 1 must be a class

**isinstance**()

Python isinstance() function *returns True* *if* the *object is of specified types*, and *if it does not* match *then* returns *False*. In this article we will see how isinstance() method works in Python.

**Input:** isinstance([1, 2, 3], list)

**Output:** True

**Explanation:** The first parameter passed is of list type.

**Input:** isinstance(10, str)

**Output:** False

**Explanation:** The first parameter, 10 is an integer and not a string.

*Syntax:*

isinstance(obj, class)

Parameters :

* obj : The *object that needs to be checked* *as a part of class or not*.
* class : *class/type/tuple* of class or type, *against which object is needed to be checked*.

Returns : True, if object belongs to the given class/type if single class is passed or any of the class/type if tuple of class/type is passed, else returns False.

TypeError: if anything other than mentioned valid class type.

test\_int = 5

# testing with tuple

print("Is test\_int integer or list or string? : " + str(isinstance(test\_int, (int, list, str))))

Is test\_int integer or list or string? : True

| isinstance() | type() |
| --- | --- |
| Syntax: isinstance(object, class) | Syntax: type(object) |
| It checks if an object is of a specific class type | It returns the class type of an object |
| It can check if the object belongs to a class and its subclasses | It cannot deal with inheritance |
| It is faster as compared to type() | It is slower than isinstance() |
| It returns either True or False | It returns the type of the object |
| It can check for multiple classes at a time | It cannot do this |
| Example: isinstance(10, (int, str)) | Example: type(10) |

**Pass**

The [Python](https://www.geeksforgeeks.org/python-programming-language/) *pass* statement *is a null statemen*t. But the *difference between* *pass and* [*comment*](https://www.geeksforgeeks.org/statement-indentation-and-comment-in-python/) is that *comment is ignored* by the interpreter whereas *pass is not ignored*.

When the user does not know what code to write, the user simply places a pass at that line. Sometimes, the *pass is used when* the *user doesn’t want any code* *to execute*. So users can *simply place a pass* *where empty code* is *not allowed*, like in loops, function definitions, class definitions, or in if statements. So using a pass statement user avoids this error.

Python Pass keyword can be used in empty functions/Class.

def function():

pass

class geekClass:

pass

Continue

Continue statement is a *loop control statement* that *forces to execute* the *next iteration* of the loop while *skipping the rest* *of* the *code* inside the loop *for the current iteration only*

Break

The *break* statement in [Python](https://www.geeksforgeeks.org/python-programming-language/) is used to *terminate* the *loop* or statement *in which it is present*. After that, the c*ontrol will pass to* the *statements that are present after* the *break statement* (*it wont exit two for loops*), if available. If the break statement is present in the nested loop, then it terminates only those loops which contain the break statement.

is,==

== To *compare objects* *based on* their *values*, Python’s equality operators (==) are employed. == returns *True if two objects*, *have* the *same value*, and returns *False if* both have *different* value

(is, is not) are *used to compare objects based on* their *identity*. *When* the *variables on either side* of an operator *point at* the *exact same object*

list1, list2 = [], []

list3 = list1

#True

print("True" if list1 == list2 else "False")

#False

print("True" if list1 is list2 else "False")

#True

print("True" if list1 is list3 else "False")

#False

list3 = list3 + list2

print("True" if list1 is list3 else "False")

super()

A *method from* a *parent class* *can be called* in Python *using* the *super()* function. It’s *typical practice in object-oriented programming* to call the methods of the superclass and enable method overriding and inheritance. *Even if* the *current class* has *replaced* those methods *with its own implementation*, calling *super() allows you to access* and use the *parent class’s methods*. By doing this, you may enhance and modify the parent class’s behaviour while still gaining from it.

class Emp():

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

self.id = id

self.name = name

self.Add = Add

# Class freelancer inherits EMP

class Freelance(Emp):

def \_\_init\_\_(self, id, name, Add, Emails):

super().\_\_init\_\_(id, name, Add)

self.Emails = Emails

Emp\_1 = Freelance(103, "Suraj kr gupta", "Noida" , "KKK@gmails")

print('The ID is:', Emp\_1.id)

print('The Name is:', Emp\_1.name)

print('The Address is:', Emp\_1.Add)

print('The Emails is:', Emp\_1.Emails)

The ID is: 103

The Name is: Suraj kr gupta

The Address is: Noida

The Emails is: KKK@gmails

##### How is python interpreted?

*In various books of python* programming, *it is mentioned* that *python* language *is interpreted*. But *that is half correct*; the python *program is first compiled* and *then interpreted*. The *compilation part is hidden from* the *programmer* thus, many programmers believe that it is an interpreted language. The *compilation part is done first* *when we execute* *our code* and *this will generate byte code* and internally this *byte code gets converted by* the python virtual machine(*p.v.m*) *according to the underlying platform*(machine+operating system).

Now the question is – if there is any proof that python first compiles the program internally and then run the code via interpreter?

The answer is yes! and note this *compiled part gets deleted by* the *python*(*as soon as you execute your code*) just it does not want programmers to get into complexity.

print("i am learning python")

print("i am enjoying it")

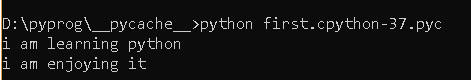
*run this code using command prompt*



*as you press enter* the *byte code will get generated*. A folder created and this will contain the byte code of your program. This folder is in the python\_prog folder where you will save your python codes.



now to *run the compiled byte code* just type the following command in the command prompt:-



the *extension .pyc is a python compiler*..

Thus, it is proven that python programs are both compiled as well as interpreted!! but the compilation part is hidden from the programmer.

*Python* is an interpreted language because it *executes line-by-line instructions*. There are actually two way to execute python code one is in Interactive mode and another thing is having Python prompts which is also called script mode. *Python does not convert* *high level code into low level* code as many other programming languages do; r*ather it will scan* the *entire code into something* *called* *bytecode*.

##### Iterators in python?

An *iterator* in Python is an *object* that is *used to iterate over iterable objects like* *lists, tuples, dicts, and sets*. The Python iterators object is *initialised using* the *iter() method*. It *uses* the *next() method for iteration*.

* \_\_iter\_\_(): The *iter()* method is *called for* the *initialization of an iterator*. This *returns an iterator object*
* \_\_next\_\_(): The *next* method r*eturns the next value* for the iterable. *When we use* a *for loop to traverse* *any iterable object*, *internally it uses* the *iter() method to get an iterator* object, *which further uses* the *next()* method to iterate over. This method *raises* a *StopIteration to signa*l the *end* of the iteration.

Iterable vs Iterator

Python iterable and Python iterator are different. The *main difference* between them is, ***iterable***(list,set) in Python ***cannot save*** the ***state* *of*** the ***iteration***, ***whereas* *in iterators*** the ***state*** of the current iteration ***gets saved***.

**Note:** *Every iterator* *is* also an *iterable*, but *not every iterable is an iterator* in [Python](https://www.geeksforgeeks.org/python-programming-language/).

# A simple Python program to demonstrate working of iterators using an example type that iterates from 10 to given value

# An iterable user defined type

class Test:

# Constructor

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

self.limit = limit

# Creates iterator object, Called when iteration is initialised

def \_\_iter\_\_(self):

self.x = 10

return self

# To move to the next element. In Python 3, we should replace next with \_\_next\_\_

def \_\_next\_\_(self):

# Store current value of x

x = self.x

# Stop iteration if limit is reached

if x > self.limit:

raise StopIteration

# Else increment and return old value

self.x = x + 1;

return x

# Prints numbers from 10 to 15

for i in Test **(** 15 **):**

print(i)

# Prints nothing

for i in Test(5):

print(i)

tup = ('a', 'b', 'c', 'd', 'e')

# creating an iterator from the tuple

tup\_iter = iter(tup)

print("Inside loop:")

# iterating on each item of the iterator object

for index, item in enumerate(tup\_iter):

print(item)

# break outside loop after iterating on 3 elements

if index == 2:

break

# we can print the remaining items to be iterated using next() thus, the **state was saved**

print("Outside loop:")

print(next(tup\_iter))

print(next(tup\_iter))

Inside loop:

a

b

c

Outside loop:

d

e

##### Generators in python

A *generator function* in Python is defined like a *normal function*, but *whenever it needs to* *generate a value*, *it does* so *with* the [*yield* keyword](https://www.geeksforgeeks.org/python-yield-keyword/) *rather than return*. *If* the *body* of a def *contains yield*, the *function automatically becomes* a Python *generator function*.

def function\_name():

yield statement

Generator Object

Python *Generator functions* r*eturn a generator* object *that is iterable*, i.e., *can be used as an* [*Iterator*](https://www.geeksforgeeks.org/iterators-in-python/). *Generator objects* are *used* either *by calling* the *next method* of the generator object or using the generator object in a “for in” loop.

Applications of Generators in Python

*Suppose* *we create a stream of* *Fibonacci numbers*, *adopting* the *generator approach* *makes it trivial*; we *just have to call next(x)* to get the next Fibonacci number without bothering about where or when the stream of numbers ends. A *more practical* type of stream processing i*s handling large data files such as log files*. *Generators provide* a *space-efficient method for* such *data processing* *as* *only parts of the file are handled* *at* one *given point in time*. We can also use Iterators for these purposes, but Generator provides a quick way (We don’t need to write \_\_next\_\_ and \_\_iter\_\_ methods here).

# A simple generator for Fibonacci Numbers

def fib(limit):

# Initialise first two Fibonacci Numbers

a, b = 0, 1

# One by one yield next Fibonacci Number

while a < limit:

yield a

a, b = b, a + b #executed in next function call

# *Create a generator object*

x = fib(5)

# Iterating over the generator object using next, In Python 3, \_\_next\_\_()

print(next(x))

print(next(x))

print(next(x))

print(next(x))

print(next(x))

# Iterating over the generator object using for in loop.

print("\nUsing for in loop")

for i in fib(5):

print(i)

0

1

1

2

3

Using for in loop

0

1

1

2

3

Python Generator Expression

In Python, *generator expression* is a*nother way of writing* the *generator function*. It *uses* the Python [*list comprehension*](https://www.geeksforgeeks.org/python-list-comprehension/) *technique* *but* *instead of storing* the *elements in* a *list* in memory, *it creates generator objects*.

Generator Expression Syntax

The *generator expression* in Python *has the following Syntax*: (expression for item in iterable)

# generator expression

generator\_exp = (i \* 5 for i in range(5) if i%2==0)

for i in generator\_exp:

print(i)

0

10

20

##### Difference between return and yield function in generators?

| S.NO. | YIELD | RETURN |
| --- | --- | --- |
| 1 | Yield is generally *used to convert* a *regular Python function* *into* a *generator*. | Return is generally *used for the end* *of* the *execution* and “*returns*” the *result to* the *caller statement*. |
| 2 | It *replaces* the *return of a function* *to* *suspend its execution* *without destroying local variables*. | It *exits from* a *function* and *hands back* a *value to its caller*. |
| 3 | It is *used when* the generator *return*s an *intermediate result to the caller*. | It is used *when* a *function is ready to send a value*. |
| 4 | ***Code*** written ***after*** the ***yield statement*** ***executes in the next* *function call***. | while, *code* written *after* the *return* *statement won't execute*. |
| 5 | It *can run multiple times*. | It *only runs once*. |
| 6 | Yield statement function is *executed* *from the last state* from *where* the *function gets paused*. | Every function call *runs* the function *from the start*. |

##### What is the difference between shallow and deep copy?

| **Shallow Copy** | **Deep Copy** |
| --- | --- |
| Shallow Copy *stores* the *reference*s of objects *to the original memory* address. | Deep copy *stores copies of* the *object’s value*. |
| Shallow Copy *reflects changes made to* the new/*copied object* *in* the *original object*. | Deep copy *doesn’t reflect changes* *made to* the new/*copied object* in the original object. |
| Shallow Copy *stores* the *copy of* the *original object* and *points* the r*eferences to the objects*. | Deep copy s*tores the copy of* the *original* object and r*ecursively copies* the *objects as well.* |
| A shallow copy is *faster*. | Deep copy is comparatively *slower*. |

# Python program to illustrate the difference between shallow and deep copy

import copy

class Car:

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

self.name = name

self.colors = colors

# Create a Honda car object

honda\_colors = ["Red", "Blue"]

honda = Car("Honda", honda\_colors)

# Basic reassign of Honda

h = honda

h.colors.append("Yellow")

print("Basic:", h.colors)Basic: ['Red', 'Blue', 'Yellow']

print("Original:", honda.colors)Original: ['Red', 'Blue', 'Yellow']

# Deepcopy of Honda

deepcopy\_honda = copy.deepcopy(honda)

deepcopy\_honda.colors.append("Green")

print("Deepcopy:", deepcopy\_honda.colors)Deepcopy: ['Red', 'Blue', 'Yellow', 'Green']

print("Original:", honda.colors)Original: ['Red', 'Blue', 'Yellow']

# Shallow Copy of Honda

copy\_honda = copy.copy(honda)

copy\_honda.colors.append("Black")

print("Shallow Copy:", copy\_honda.colors)Shallow Copy: ['Red', 'Blue', 'Yellow', 'Black']

print("Original:", honda.colors)Original: ['Red', 'Blue', 'Yellow', 'Black']

In list copy() vs deepCopy()

copy(): This is a shallow copy. It creates a new list with references to the same elements in the original list. So if the original list contains other lists (nested lists), copy() will only duplicate the outer list, not the inner ones. Changes to elements inside inner lists will affect both the original and the copied list. If your list has no inner lists (or nested mutable objects), then copy() works just fine. It creates a new list with copies of the original elements, so changes in the copied list won’t affect the original list, and vice versa.

import copy

list1 = [[1, 2], [3, 4]]

list2 = copy.copy(list1)

list2[0][0] = 9 # This change will reflect in both lists

print(list1) #[[9, 2], [3, 4]]

print(list2) #[[9, 2], [3, 4]]

list3 = [1, 2]

list4 = copy.copy(list3)

list4[0] = 9 # This change will not reflect in both lists

print(list3) #[1, 2]

print(list4) #[9, 2]

deepcopy(): This creates a deep copy. It duplicates everything, including any nested lists or objects. This means the new list and all its nested elements are fully independent of the original list. Changes in one won’t affect the other.

import copy

list1 = [[1, 2], [3, 4]]

list2 = copy.deepcopy(list1)

list2[0][0] = 9 # This change will NOT reflect in list1

print(list1) # [[1, 2], [3, 4]]

print(list2) # [[9, 2], [3, 4]]

##### xrange and range in python?

| **range()** | **xrange()** |
| --- | --- |
| *Returns* a *list of integers*. | *Returns* a *generator object*. |
| Execution *speed is slower* | Execution *speed is faster*. |
| *Takes more memory* *as it keep*s the *entire list* of elements *in memory*. | *Takes less memory* *as it keeps* *only one element* at a time in memory. |
| All arithmetic operations can be performed as it returns a list. | Such operations cannot be performed on xrange(). |
| In *python 3*, *xrange() is not supported*. | In *python 2*, x*range() is used to iterat*e in for loops. |
| Can perform list slicing  a[2:5] | Cannot perform list slicing  x[2:5] |
| a = range(1,10000) print (sys.getsizeof(a)) #80064 | x = xrange(1,10000)  print (sys.getsizeof(x)) #40 |

##### Scope of a variable and global

*Scope of a variable* entirely r*evolves around* a *function*. *Class has no relation* *to scope*. *Instances of class variables* *are* like *global variables* *to its functions*.

*Local variables* are those *that are initialised within a function* and are *unique to that function*. It *cannot be accessed outside* of the function.

def f():

# local variable

s = "I love Geeksforgeeks"

print("Inside Function:", s) #success

# Driver code

f()

print(s) #failure

*Global variables* are the ones that are *defined and declared outside* any function and are *not specified to any function*.

# This function uses global variable s

def f():

print(s)#I love Geeksforgeeks

# Global scope

s = "I love Geeksforgeeks"

f()

Now *suppose a variable* *with* the *same name* is *defined inside* the *scope of the function* as well *then it will print* the *value given inside* the function only and *not the global* value.

# This function has a variable with the same name as s.

def f():

s = "Me too."

print(s) #Me too.

# Global scope

s = "I love Geeksforgeeks"

f()

print(s) #I love Geeksforgeeks

*What will happen* *if we change* the value of s *inside of* the *function* f()? *Will it affect* the *global s as well*? We test it in the following piece of code:

def f():

print(s) #ERROR

# This program will NOT show error if we comment below line.

s = "Me too." #ERROR

print(s)

# Global scope

s = "I love Geeksforgeeks"

f()

print(s)

Why? *Python “assumes”* that *we want* a *local variabl*e *due to* the *assignment* *of s* *inside* of *f()*, *so* the *first print statement* *throws* this *error message*. *Any variable* which is *changed or created inside* of a *function* *is local*, *if it hasn’t been* *declared* as *a global variable*.

*If* we *want to use* the *global variable*, we have to *use the keyword global*,

def f():

global s

print(s)#I love Geeksforgeeks

s = "Me too."

print(s)#Me too.

# Global scope

s = "I love Geeksforgeeks"

f()

print(s)#Me too.

Unlike functions, *if statements* *do not create a new scope*. This means that *variables defined* *inside* an *if* statement *are* *accessible outside* of the if statement. Here's an example: (same is for while) but for creates a new

if True:

message = "Hello"

print(message) # This will print "Hello"

*Within functions*, *if statements* *have* the *same scope as* the *function*. This means that *variables defined inside* an *if* statement *are accessible* *within* that *function*, *but not outside* of it. Here's an example:

def is\_balance\_low(balance: int):

if balance <= 100:

message = "Warning: Low balance."

print(message)

is\_balance\_low(50) # This will print "Warning: Low balance."

print(message) # This will cause an error

##### What is the purpose of python non-local statements?

Python *nonlocal* keyword is *used to refer*ence a *variable in* the *nearest scope*. The *nonlocal keyword* *won’t work on* *local or global* *variables* and *therefore* *must be used to refer*ence *variables in another scope* *except* the *global and local one*. The *nonlocal* keyword is *used in nested functions* *to refer*ence a *variable in the parent function*.

global\_name = 'geeksforgeeks'

def foo():

def bar():

local\_name = 'geeksforgeeks'

nonlocal local\_name #LocalVar hence throws error

nonlocal global\_name #globalVar hence throws error

bar()

foo()

def foo():

# Local variable of foo()

name = "geek"

def bar():

name = "Geek"

def ack():

nonlocal name # Reference to the next upper variable with this name

print(name) # Print the value of the referenced variable

name = 'GEEK' # Overwrite the referenced variable

ack() # Calling second inner function

print(name)

bar() # Calling first inner function

print(name) # Printing local variable of bar()

foo()

Geek

GEEK

geek

##### Data Type in Python

Detailed For list, tuple, set, dictionary [Java/Python, Tips](https://docs.google.com/spreadsheets/d/1upeMZgIFjxoT3XalldV8y_DvifzhCIun9bOdILGqA54/edit?gid=532988201#gid=532988201)

Python *Data types* are the classification or categorization of data items. It *represent*s the *kind of value* that tells *what operations can be performed* on a particular data. *Since everything* *is an object in Python* programming, *Python data types* *are classes* and *variables are* instances (*objects) of these classes*. The following are the standard or built-in data types in Python:

*Numeric*

The numeric data type in Python *represents* the *data that has* a *numeric value*. A numeric value *can be* an *integer*, a *float*ing *number*, or even a *complex number*. These values are *defined as* [*Python int*](https://www.geeksforgeeks.org/python-int-function/)*,* [*Python float*](https://www.geeksforgeeks.org/float-in-python/), *and* [*Python complex*](https://www.geeksforgeeks.org/python-complex-function/) classes in [Python](https://www.geeksforgeeks.org/python-programming-language/).

* *Integers* – This value is represented by int class. It contains *positive or negative whole numbers* (without fractions or decimals). In Python, there is ***no limit to how long an integer value can be***.
* *Float* – This value is represented by the float class. It is a real *number with a floating-point representation*. It is *specified by a decimal* point. *Optionally,* the character *e or E* *followed by* a positive or negative *integer* may be *appended to specify* *scientific notation*.
* Complex Numbers – A complex number is represented by a complex class. It is *specified as (real part)* *+ (imaginary part)j*. For example – *2+3j*

*Sequence Type*

Sequences allow storing of multiple values in an organised and efficient fashion. There are several sequence data types of Python:

* [Python String](https://www.geeksforgeeks.org/python-data-types/#string)

A string is a *collection of one or more char*acters *put in a* *single* quote, *double*-quote, *or* triple-quote for multiline strings (*triple single quote*).

Can access individual char of strings using Indexing, negative indexing String1[0], String1[-1]

Can access range of string using string slicing String1[3:-2]

Can reverse a string using gfg[::-1] || gfg = "".join(reversed(gfg)) #reversed() returns an iterable in reverse order of input. Join takes any iterable and joins into one string.

Strings are *immutable* *Updation or deletion of characters is not allowed*. Alternative way to achieve this are:

Converting string to list: list1 = list(String1)**;** list1[2] = 'p'**;** String1 = ''.join(list1)

Using list slicing: String1 = String1[0:2] + 'p' + String1[3:]

Can assign entirely new value to it: String1 = "Hello, I'm a Geek"; String1 = "Welcome to the Geek World"

*Deleting a char*:

String1 = "He**l**lo, I'm a Geek"

del String1[2] #TypeError: 'str' object doesn't support item deletion

String1 = String1[0:2] + String1[3:]

*Can delete entire string*: del String1

Escape Sequencing in Python

*Escape sequences* *start with a backslash* **“\”** *and* can be *interpreted differently*. *If single quotes* are *used to represent a string*, *then* *all* the *single quotes* present *in* the *string* *must be escaped* and the same is done for Double Quotes.

String1 = 'I\'m a "Geek"' #Escaping Single Quote

String1 = "I'm a \"Geek\"" #Escaping Double Quotes

String1 = "C:\\Python\\Geeks\\" #Printing Paths with use of Escape Sequences

String1 = "Hi\tGeeks" #Printing with use of Tab

String1 = "Python\nGeeks" #Printing with the use of New Line

*To ignore* the *escape sequences* in a String, *r* or R *is used*, this implies that the string is a raw string and escape sequences inside it are to be ignored.

String1 = r"This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"

Formatting of Strings

*Strings in Python* or string data type in Python *can be formatted* *with* the *use of format() method* which is a very versatile and powerful tool for formatting Strings. *Format method in String contains* *curly braces {} as placeholders* *which can hold* *arguments according to* *position or keyword* *to specify* the *order*.

# Default order

String1 = "{} {} {}".format('Geeks', 'For', 'Life')

Geeks For Life

# Positional Formatting

String1 = "{1} {0} {2}".format('Geeks', 'For', 'Life')

For Geeks Life

# Keyword Formatting

String1 = "{l} {f} {g}".format(g='Geeks', f='For', l='Life')

Life For Geeks

# Formatting of Integers binary

String1 = "{0:b}".format(16)

10000

# Formatting of Floats exponential

String1 = "{0:e}".format(165.6458)

1.656458e+02

# Rounding off Integers

String1 = "{0:.2f}".format(1/6)

0.17

Alignment of strings

A *string can be* *left, right, or centre aligned* *with* the use of *format specifiers*, *separated by* a *colon(:)*. The (*<*) indicates that the *string should be aligned to* the *left*, (*>*) indicates that the *string should be aligned to* the *right* and (*^*) indicates that the *string should be aligned to* the *centre*. We can also specify the length in which it should be aligned. For example, (*<10*) means that the *string should be aligned to* the *left* *within a field of width of* *10 char*acters.

# String alignment

String1 = "|{:<10}|{:^10}|{:>10}|".format('Geeks','for','Geeks')

|Geeks | for | Geeks|

# To demonstrate aligning of spaces

String1 = "\n{0:^16} was founded in {1:<4}!".format("GeeksforGeeks",2009)

GeeksforGeeks was founded in 2009 !

* [Python List](https://www.geeksforgeeks.org/python-data-types/#list) [Java/Python, Tips](https://docs.google.com/spreadsheets/d/1upeMZgIFjxoT3XalldV8y_DvifzhCIun9bOdILGqA54/edit?gid=532988201#gid=532988201)
* [Python Tuple](https://www.geeksforgeeks.org/python-data-types/#tuple) [Java/Python, Tips](https://docs.google.com/spreadsheets/d/1upeMZgIFjxoT3XalldV8y_DvifzhCIun9bOdILGqA54/edit?gid=532988201#gid=532988201)

*Boolean*

Python Data type with one of the two built-in values, *True or False*.True and False *with capital ‘T’ and ‘F*’ *are valid booleans* otherwise python will throw an error. However *non-Boolean objects* *can be evaluated in a Boolean context* as well and determined to be true or false. It is denoted by the class bool.

*Set*

* [Java/Python, Tips](https://docs.google.com/spreadsheets/d/1upeMZgIFjxoT3XalldV8y_DvifzhCIun9bOdILGqA54/edit?gid=532988201#gid=532988201)

*Dictionary*

* [Java/Python, Tips](https://docs.google.com/spreadsheets/d/1upeMZgIFjxoT3XalldV8y_DvifzhCIun9bOdILGqA54/edit?gid=532988201#gid=532988201)

*Binary Types(* [*memoryview*](https://www.geeksforgeeks.org/memoryview-in-python/)*,* [*bytearray*](https://www.geeksforgeeks.org/python-bytearray-function/)*,* [*bytes*](https://www.geeksforgeeks.org/python-bytes-method/)*)*

This code assigns variable ‘x’ different values of various Python data types.

x = "Hello World" #string

x = 50 #integer

x = 60.5 #float

x = 3j #complex

x = ["geeks", "for", "geeks"] #list

x = ("geeks", "for", "geeks") #tuple

x = range(10) #range

x = {"name": "Suraj", "age": 24} #dictionary

x = {"geeks", "for", "geeks"} #

x = frozenset({"geeks", "for", "geeks"}) #frozenset

x = True #boolean

x = b"Geeks" #bytes

x = bytearray(4) #bytearray

x = memoryview(bytes(6)) #memoryview

x = None #special value ‘None’

##### Logical Operators in Python

In Python, *Logical operators are used on* *conditional statements* (either True or False). They perform *Logical AND, Logical OR, and Logical NOT* operations.

| **OPERATOR** | **DESCRIPTION** | **SYNTAX** | **Example** |
| --- | --- | --- | --- |
| and | Returns True if both the operands are true | *x and y* | x>7 and x>10 |
| or | Returns True if either of the operands is true | *x or y* | x<7 or x>15 |
| not | Returns True if the operand is false | *not x* | not(x>7 and x> 10) |

##### sort(), sorted(), join()

**sort()** :- sort() method sorts the list ascending by default.

Syntax

**list**.sort(reverse=True|False, key=myFunc)

Parameter Values

| Parameter | Description |
| --- | --- |
| reverse | Optional. reverse=True will *sort the list descending*. Default is reverse=False |
| key | Optional. A *function to specify* the *sorting criteria*(s) |

**sorted()** :- sorted() function *returns a sorted list* of the specified iterable object.

Syntax

sorted(iterable, key=key, reverse=reverse)

Parameter Values

| Parameter | Description |
| --- | --- |
| iterable | Required. The sequence to sort, list, dictionary, tuple etc. |
| key | Optional. A **Function to execute** to decide the order. Default is None |
| reverse | Optional. A Boolean. False will sort ascending, True will sort descending. Default is False |

*Python uses* the [***Tim Sort***](https://www.geeksforgeeks.org/timsort/) *algorithm* for sorting. It’s a stable sorting whose worst case is O(N log N). It’s a *hybrid* *sorting algo*rithm, *derived from merge sort and insertion sort*, designed to perform well on many kinds of real-world data.

**join()** :- string function in Python used to *join elements of* the *sequence separated by a string separator*.

Syntax: **string\_name**.join(**iterable**)

Parameters:

Iterable – *objects* *capable of returning* their *members one at a tim*e. Some examples are List, Tuple, String, Dictionary, and Set.

Return Value: The join() method returns a *string concatenated with* the *elements of iterable*.

Type Error: ***If*** the ***iterable contains*** any ***non-string values***, ***it raises a TypeError*** exception.

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

separator = "@ "

result = separator.join(word for word in words if word)

print(result) apple@ banana@ cherry

##### List Comprehension & Lambda function

List Comprehension : returns a new list

**Syntax:** newList **=** **[** expression(element) **for** element **in** iterable **if** condition **]**

**Parameter:**

* **expression**: Represents the *operation* you want *to execute* *on every item* *within* the *iterable*.
* **element**: The term “variable” refers to *each value* *taken from* the *iterable*.
* **iterable**: specify the *sequence of elements* you want to iterate through.(e.g., a list, tuple, or string).
* **condition**: (Optional) A *filter helps decide* *whether* or not an *element should be added* to the new list.

**Return:**The *return value* of a list comprehension *is a new list containing* the *modified elements* that satisfy the given criteria.

Python *List comprehension* provides a much more *short syntax for* *creating* a *new list* *based on* the *values of an existing list*

Both of below are same

matrix = []

for i in range(3):

# Append an empty sublist inside the list

matrix.append([])

for j in range(5):

matrix[i].append(j)

print(matrix)

# Nested list comprehension

matrix = [[j for j in range(5)] for i in range(3)]

print(matrix)

[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]

If condition in expression

lis = ["Even number" if i % 2 == 0 else "Odd number" for i in range(8)]

print(lis)

['Even number', 'Odd number', 'Even number', 'Odd number', 'Even number', 'Odd number', 'Even number', 'Odd number']

More details: <https://www.geeksforgeeks.org/python-list-comprehension/>

Lambda function

**Syntax:** *lambda* *arguments* : *expression*

* This function *can have any number of arguments* but *only one expression*, *which is evaluated* *and returned*.
* One is free to *use lambda functions* *wherever function* objects *are required*.
* You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
* It has various uses in particular fields of programming, besides other types of expressions in functions.

Max = lambda a, b : a if(a > b) else b

print(Max(1, 2))

2

Python Lambda Function with List Comprehension

On *each iteration inside* the *list* comprehension, we are *creat*ing *a new lambda function* *with* a *default argument* of *x* (where *x* *is* the *current item* *in* the *iteration*). *Later, inside* the for loop, we are *calling* the *same function object* *having* the *default argument* using item() and get the desired value. Thus, is\_even\_list stores the list of lambda function objects.

is\_even\_list = [lambda arg=x : arg \* 10 for x in range(1, 5)]

for item in is\_even\_list:

print(item())

10

20

30

40

Python Lambda Function within sorted function

print(count\_dict.items()) #dict\_items([(1, 2), (2, 2), (5, 1)])

sorted\_counts = sorted(count\_dict.items(), key=lambda x: x[1])

print(sorted\_counts) #[(5, 1), (1, 2), (2, 2)]

##### What is Dictionary Comprehension? Give an Example

Dictionary Comprehension is a syntax construction to *ease* the *creation of a dicti*onary based *on the existing iterable*.

For Example: my\_dict = **{**i:i+7 for i in range(1, 10)**}**

Set Comprehension

*Both* *dict*ionaries *and sets* *use curly braces*. The difference between them is that *dictionaries* *have* *key-value pairs* *separated* *by a colon*, while sets only have the elements themselves.

squared = **{**num \* num for num in nums**}**

##### What is a zip, Map function?

*zip()* method *takes iterable* containers *and returns* a *single iterator* *tuple* object, *having mapped values* from all the containers. It is *used to map* the *similar index* *of multiple container*s *so that they can be used* just *using a single entity*.

Syntax : zip(\*iterators)

Parameters : Python iterables or containers ( list, string etc )

Return Value : **Returns** a single iterator object. Which is a **tuple**

Python *zip() with enumerate* - *useful* *where you want to process* *multiple lists or tuples* *in parallel*, and *also need to access* *their indices* for any specific purpose.

names = ['Mukesh', 'Roni', 'Chari']

ages = [24, 50, 18]

for i, (name, age) in enumerate(zip(names, ages)):

print(i, name, age)

0 Mukesh 24

1 Roni 50

2 Chari 18

Python *zip() with Dictionary* - used to combine two or more iterable [dictionaries](https://www.geeksforgeeks.org/python-dictionary/) into a single iterable, where corresponding elements from the input iterable are paired together as tuples. When using zip() with dictionaries, *it pairs* the *keys and values* *of* the *dictionaries* *based on* *their position in* the *dictionary*.

stocks = ['GEEKS', 'For', 'geeks']

prices = [2175, 1127, 2750]

new\_dict = **{**stocks: prices for stocks,prices in zip(stocks, prices)**}**

print(new\_dict)

{'GEEKS': 2175, 'For': 1127, 'geeks': 2750}

***Unzipping*** Using zip() - Unzipping means *converting* the *zipped values back to* the *individual self* as they were. This is *done* *with* the *help of “****\*****” operator*.

Consider zipped value to be The zipped result is : [('Manjeet', 4, 40), ('Nikhil', 1, 50),

('Shambhavi', 3, 60), ('Astha', 2, 70)]

# unzipping values

namz, roll\_noz, marksz = zip(**\***mapped)

# printing initial lists

print("The name list is : ", end="")

print(namz)('Manjeet', 'Nikhil', 'Shambhavi', 'Astha')

print("The roll\_no list is : ", end="")

print(roll\_noz)(4, 1, 3, 2)

print("The marks list is : ", end="")

print(marksz)(40, 50, 60, 70)

***Map function***

The map() function *executes* a *specified function* *for each item* *in* an *iterable*. The *item* is *sent to* the *function as a parameter*.

Syntax

map(function, iterables)

| Parameter | Description |
| --- | --- |
| function | Required. The function to execute for each item |
| iterable | Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable. |

Code

def myfunc(a, b):

return a + b

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))

print(x) <map object at 0x034244F0>

#convert the map into a list, for readability:

print(list(x))['appleorange', 'bananalemon', 'cherrypineapple']

We *can also use* [*lambda expressions*](https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/) *with map()*.

numbers1 = [1, 2, 3]

numbers2 = [4, 5, 6]

result = map(lambda x, y: x + y, numbers1, numbers2)

print(list(result))

[5, 7, 9]

##### Trick Questions

1. What will be result of below code

i = 0

for i in range(10):

i = 100

print("karan")

The *loop runs* *10 times* because of range(10).

However, *inside the loop*, *setting i = 100* *doesn't affect* the *loop* *since* the *loop control variable (i*) is overwritten but then *gets reset by* the *for loop* mechanism *in each iteration*.

Therefore, "karan" will be printed 10 times. The value of i = 100 is irrelevant in this case for the loop's functionality.

1. Datatype post arithmetic operation in python?

The *result of division* is *always a float* in Python. *Floor Division* //: r*eturns integer,* *Divides* the *first* operand *by* the *second and rounds down* the result to an integer.

*For* the *other arithmetic operators*, the *result* will be an *integer if* *both operands are integer*s. *If one* of the operands *is a float*, the *result will be a float*.

1. Valid way to assign default arguments

# This is valid

def greet(**greeting, name="world"**):

print(greeting + ", " + name + "!")

greet(hello)

# This is NOT valid as, post 1st default it shouldn't be any not default arguments, issues comes in calling:

# As you cant call like greet(,”karan”)

def greet(**greeting="Hello", name**):

print(greeting + ", " + name + "!")

*Then how is* the r*ange() function implemented* *as* *its parameter as* range(start, **stop**, step) *with stop being mandatory* and the *other two have default values* of 0 and 1 respectively.

*It's actually defined in* a way that provides flexibility without violating the rules of default arguments. range() is a built-in function in *C* (implemented in C, not Python itself), *allowing* it t*o bypass* some of the *rules that* *Python* functions *must follow*.

*If you* *want*ed *to mimic range()’s flexibility* in a custom Python function,  *you’d need to handle* *arguments* conditionally, often *with \*args*, to accept a variable number of inputs:

def custom\_range(\*args):

if len(args) == 1:

start, stop, step = 0, args[0], 1

elif len(args) == 2:

start, stop, step = args[0], args[1], 1

elif len(args) == 3:

start, stop, step = args[0], args[1], args[2]

else:

raise TypeError("custom\_range expected 1 to 3 arguments, got {}".format(len(args)))

# Implement your range logic here, similar to Python's range()

1. 2D list declaration

grid = [[0] \* 3] \* 2

grid[0][0] = 1

print(grid) #[[1, 0, 0], [1, 0, 0]]

This code will not work as expected. The issue is that the *inner list* is a *reference to* the *same list object*. This means that *if we change* *one of the inner list*s, *all* the *inner lists will change*

This code works

grid = [[0 for i in range(3)] for j in range(2)]

grid[0][0] = 1

print(grid) # [[1, 0, 0], [0, 0, 0]]

But there's a more concise solution you may prefer:

grid = [[0] \* 3 for \_ in range(2)]

##### SOLID Principle

<https://arjancodes.com/blog/solid-principles-in-python-programming/>

###### Single Responsibility Principle (*SRP*)

* The Single Responsibility Principle *advocates for* a *class or module* *to have* *only one reason to change*. In simpler terms, *it should do one thing* and do it *well*. By adhering to SRP, your *code becomes* more *modular*, making it *easier to understand and maintain*.

class Order:

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

#initialise

def add\_item(self, name: str, quantity: int, price: float) -> None:

#Just appends

def total\_price(self):

#calculate price and return total

def pay(self, payment\_type: str, security\_code):

#check the payment type and change self.status if any issue raise Exception

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

print(order.total\_price())

order.pay("debit", "0372846")

This code violates the SRP *because* it is both *responsible for managing* the *order and* the *payment*. This results in our code being *highly coupled* and makes it *harder to* *understand*, *maintain*, and *test*.

class Order:

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

#initialise

def add\_item(self, name: str, quantity: int, price: float) -> None:

#Just appends

class PaymentProcessor:

def pay(self, order: Order, security\_code: str):

print("Processing payment")

print(f"Verifying security code: {security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = PaymentProcessor()

processor.pay(order, "0372846")

print(order.status)

This *code still violates the SRP* *because* the *order* is *responsible for both* the *prices and the quantities* and could be improved by separating these concerns.

***Over here* *we pass* *order object* *to*** processor.pay

###### Open-Closed Principle (OCP)

* The *Open-Closed Principle* states that software entities should be *open for extension* but *closed for modification*. This means that you should be able to *extend a class’s behaviour* *without modifying it*.

from abc import ABC, abstractmethod

class Order:

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

#initialise

def add\_item(self, name: str, quantity: int, price: float) -> None:

#add items

class PaymentProcessor(ABC):

@abstractmethod

def pay(self, order, security\_code: str):

...

class CreditCardPaymentProcessor(PaymentProcessor):

def pay(self, order, security\_code: str):

#custom credit card logic

order.status = "paid"

class DebitCardPaymentProcessor(PaymentProcessor):

def pay(self, order, security\_code: str):

#custom Debit card logic

order.status = "paid"

class PaypalPaymentProcessor(PaymentProcessor):

def pay(self, order, security\_code: str):

#custom Paypal logic

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = CreditCardPaymentProcessor()

processor.pay(order, "0372846") #again passing order as a parameter

print(order.status)

It’s *closed for modification* and *open for extension* *because* we *can add new payment methods* *without modifying* the *PaymentProcessor* class.

**Note**: Paypal *does not use* *security code* *still it has to use it*. *Not following Liskov* Substitution Principle. *We are using* a *subclass in a way* *that is not compatible with its parent* class. This is *because of* the concept of *Design by Contract*, which in this context dictates *classes should adhere to* the “*contract*” *set out by* their *interface* for consistency and integrity.

###### Liskov Substitution Principle (LSP)

* The *Liskov Substitution Principle* states that objects in a *program should be replaceable with instances of their subtypes* *without altering* the *correctness of the progra*m. In other words, a *subclass* *should be able to replace* its *parent class* *without breaking the code*.
* We are *using a subclass in a way* *that is not compatible with* *its parent class*. This is *because of* the *concept of Design by Contract*, which in this context *dictates classes should adhere to the “contract*” *set out by their interface* for consistency and integrity.

from abc import ABC, abstractmethod

class Order:

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

#initialise

def add\_item(self, name: str, quantity: int, price: float) -> None:

#add items

class PaymentProcessor(ABC):

@abstractmethod

def pay(self, order): **#removed security\_code from here**

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str): **#and moved to constructor**

self.security\_code = security\_code

def pay(self, order):

#Processed Payment

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str):

self.security\_code = security\_code

def pay(self, order):

#Processed Payment

class PaypalPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, email\_address: str):

self.email\_address = email\_address

def pay(self, order):

print("Processing paypal payment")

print(f"Verifying email: {self.email\_address}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = PaypalPaymentProcessor("hi@arjancodes.com") #passing email\_address as security\_codes

processor.pay(order)#again passing order as a parameter but with no security code

print(order.status)

We *moved* the *security code to* the *constructor class*.

###### Interface Segregation Principle (ISP)

* The *Interface Segregation Principle* states that *clients should not be forced to depend on methods* *they do not use*. This *means* that you should *not have to implement methods* *that you do not need*.
* It’s *better to have* *interfaces* *that are suited to specific tasks* *rather than one general-purpose interface*.

from abc import ABC, abstractmethod

class Order:

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

#initialise

def add\_item(self, name: str, quantity: int, price: float) -> None:

#add items

class PaymentProcessor(ABC):

@abstractmethod

def pay(self, order):

...

@abstractmethod

def auth\_sms(self, order, code: str):

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str):

self.security\_code = security\_code

self.authenticated = False

def auth\_sms(self, order, code: str):

print("Authenticating via SMS")

self.authenticated = True

def pay(self, order):

if not self.authenticated:

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str):

self.security\_code = security\_code

def pay(self, order):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

def auth\_sms(self, order, code: str):

raise Exception("Not implemented")

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = DebitPaymentProcessor("0372846")

processor.auth\_sms(order, "12345")

processor.pay(order)

print(order.status)

This *code violates* the *ISP because* the *CreditPaymentProcessor* class is *forced to implement* the *auth\_sms method*, *even though it does not use it*.

class PaymentProcessor(ABC):

@abstractmethod

def pay(self, order: Order):

...

class SmsPaymentProcessor(PaymentProcessor):

@abstractmethod

def auth\_sms(self, order: Order, code: str):

...

class DebitPaymentProcessor(SmsPaymentProcessor):

def \_\_init\_\_(self, security\_code: str):

self.security\_code = security\_code

self.authenticated = False

def pay(self, order):

if not self.authenticated:

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

def auth\_sms(self, order, code: str):

print("Authenticating via SMS")

self.authenticated = True

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str):

self.security\_code = security\_code

def pay(self, order):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = DebitPaymentProcessor("0372846")

processor.auth\_sms(order, "12345")

processor.pay(order)

print(order.status)

*Now* *abstract class inheritance* is *happening*, so CreditPaymentProcessor doesn’t need to implement auth\_sms of SmsPaymentProcessor but directly implement PaymentProcessor

*We could make* this *code even better by* *separating* the *authorization logic from* the *payment processor* *aka no inheritance like above*. This is *almost like* *dependency inversion principle* *but with thing missing* *was a abstract method* of SMSAuthorizer

class SMSAuthorizer:

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

self.authenticated = False

def verify\_code(self, code: str):

print("Verifying code")

self.authenticated = True

def is\_authenticated(self):

return self.authenticated

class PaymentProcessor(ABC):

@abstractmethod

def pay(self, order: Order):

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str, authorizer: SMSAuthorizer):

self.security\_code = security\_code

self.authorizer = authorizer

def pay(self, order):

if not self.authorizer.is\_authenticated():

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str):

self.security\_code = security\_code

def pay(self, order: Order):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

authorizer = SMSAuthorizer()

authorizer.verify\_code("12345")

processor = DebitPaymentProcessor("0372846", authorizer)

processor.pay(order)

print(order.status)

###### Dependency Inversion Principle (DIP)

* The *Dependency Inversion Principle* states that *high-level modules* should *not depend on low-level modules,* *but both* should *depend on abstractions*. This means that *you should not have to* *change* your *code* *when you change* the *implementation of a module*.
* In practice, this means that our *payment processor* *shouldn’t be concerned with* *how* its *payment is validated*, *whether* that be by *SMS, a robot check, or an email*.

from abc import ABC, abstractmethod

class Order:

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

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(self, name: str, quantity: int, price: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class Authorizer(ABC):

@abstractmethod

def is\_authenticated(self):

...

class SMSAuthorizer(Authorizer):

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

self.authenticated = False

def verify\_code(self, code: str):

print("Verifying code")

self.authenticated = True

def is\_authenticated(self):

return self.authenticated

class NotARobotAuthorizer(Authorizer):

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

self.authenticated = False

def ask(self):

print("Are you a robot?!!! [┐∵]┘")

self.authenticated = True

def is\_authenticated(self):

return self.authenticated

class PaymentProcessor(ABC):

@abstractmethod

def pay(self, order: Order):

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str, authorizer: Authorizer):

self.security\_code = security\_code

self.authorizer = authorizer

def pay(self, order: Order):

if not self.authorizer.is\_authenticated():

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(self, security\_code: str):

self.security\_code = security\_code

def pay(self, order: Order):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

authorizer = NotARobotAuthorizer()

authorizer.ask()

processor = DebitPaymentProcessor("0372846", authorizer)

processor.pay(order)

print(order.status)

##### What are \*args and \*\*kwargs?

The special syntax *\*args* *in function* definitions in Python are *used to pass* a *variable number of arguments to a function*. It is *used to pass* a *non-keyworded*, *variable-length argument list*.

* The syntax is to *use* the symbol *\** *to take in* a *variable number of arguments*; *by convention*, it is *often used with* the word *args*.
* *if* you're *not sure how many arguments will be passed* to your function, you can *use \*args to handle them flexibly*.
* For example, we want to make a multiply function that takes any number of arguments and is able to multiply them all together. It can be done using \*args.
* *Using* the *\**, the *variable that we associate with* the *\** *becomes iterable* meaning *you can do things like iterate over it*, run some higher-order functions such as map and filter, etc.

def myFun(arg1, \*argv):

print("First argument :", arg1)

for arg in argv:

print("Next argument through \*argv :", arg)

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

First argument : Hello

Next argument through \*argv : Welcome

Next argument through \*argv : to

Next argument through \*argv : GeeksforGeeks

The special syntax *\*\*kwargs* in function definitions in Python are *used to pass a keyworded*, *variable-length argument list*. *We use* the name *kwargs with the double star*. The reason is that the double star *allows us to pass through* *keyword arguments* (and any number of them).

* A *keyword argument* is where you *provide* a *name to the variable* *as you pass it into the function*.
* One can ***think of*** the ***kwargs as*** *being a* ***dictionary*** that *maps each keyword to the value* that we pass alongside it. That is why when we iterate over the kwargs there doesn’t seem to be any order in which they were printed out.

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

for key, value in kwargs.items():

print("%s == %s" % (key, value))

# Driver code

myFun("Hi", first='Geeks', mid='for', last='Geeks')

first == Geeks

mid == for

last == Geeks

def myFun(arg1, arg2, arg3):

print("arg1:", arg1)

print("arg2:", arg2)

print("arg3:", arg3)

# Now we can use \*args or \*\*kwargs to

# pass arguments to this function :

args = ("Geeks", "for", "Geeks")

myFun(\*args)

kwargs = {"arg1": "Geeks", "arg2": "for", "arg3": "Geeks"}

myFun(\*\*kwargs)

arg1: Geeks

arg2: for

arg3: Geeks

arg1: Geeks

arg2: for

arg3: Geeks

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

print("args: ", args)

print("kwargs: ", kwargs)

# Now we can use both \*args ,\*\*kwargs

# to pass arguments to this function :

myFun('geeks', 'for', 'geeks', first="Geeks", mid="for", last="Geeks")

args: ('geeks', 'for', 'geeks')

kwargs: {'first': 'Geeks', 'mid': 'for', 'last': 'Geeks'}

In Python, the *order matters* *when defining function parameters*. Typically, *you'd define* them in this order: *first \*args* to capture any number of positional arguments, *then \*\*kwargs* to capture any keyword arguments. This convention ensures that Python knows how to interpret the arguments correctly when you call the function.

Using \*args and \*\*kwargs in Python to set values of object

* *\*****args receives* *arguments as*** a [***tuple***](https://www.geeksforgeeks.org/python-tuples/).
* ***\*\*kwargs receives* *arguments as*** a [***dictionary***](https://www.geeksforgeeks.org/python-dictionary/)**.**

# defining car class

class car():

# args receives unlimited no. of arguments as an array

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

# access args index like array does

self.speed = args[0]

self.color = args[1]

# creating objects of car class

audi = car(200, 'red')

bmw = car(250, 'black')

# printing the colour and speed of the cars

print(audi.color)

print(bmw.speed)

red

250

# defining car class

class car():

# args receives unlimited no. of arguments as an array

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

# access args index like array does

self.speed = kwargs['s']

self.color = kwargs['c']

# creating objects of car class

audi = car(s=200, c='red')

bmw = car(s=250, c='black')

# printing the colour and speed of cars

print(audi.color)

print(bmw.speed)

red

250

##### How do you debug a Python program?

$ python -m pdb python-script.py

This *command runs* the Python debugger (*pdb*) *with* the *specified Python script* (python-script.py). It allows you to *interactively debug your Python code*, *stepping through it* *line by line*, inspecting variables, and diagnosing issues.

[Command-Line Python Debugging with pdb](https://youtu.be/a7qIcIaL4zs?si=sKaB0rk4W29uL4g4)

##### What is PIP?

*PIP* is an acronym for Python Installer Package which provides a *seamless interface to install* various *Python modules*. It is a *command-line tool* that can *search for packages over* the *internet* *and install them without any user interaction*.

##### What is a Python Switch/Match Statement?

From version 3.10 upward, Python has implemented a switch case feature called “structural pattern matching”. You can *implement* *this feature with* the *match and case keywords*. Note that the ***underscore symbol is what you use to define*** a ***default case*** for the switch statement in Python.

match term:

case pattern-1:

action-1

case pattern-2:

action-2

case pattern-3:

action-3

case \_:

action-default

##### What is a Walrus Operator?

[The Walrus Operator](https://docs.python.org/3/whatsnew/3.8.html) allows you to *assign values to variables* *as part of* a *larger expression*. This can be *useful when* you *need to use* a *value multiple times in a loop*, *but don’t want to repeat* the *calculation*.

The Walrus Operator is represented by the `:=` syntax and *can be used in a variety of contexts including while loops and if statements.*

names = ["Jacob", "Joe", "Jim"]

if (name := input("Enter a name: ")) in names:

print(f"Hello, {name}!")

else:

print("Name not found.")

##### What are Function Annotations in Python?

<https://www.geeksforgeeks.org/function-annotations-python/>

Function annotations are *arbitrary python expressions* that are associated with various part of functions. These expressions are *evaluated at compile time* and *have no life in python’s runtime env*ironment.

##### Various libraries used

###### Time

import time

start = time.time() #return the current time in seconds since epoch.

end = time.time()

print('Time taken in seconds -', end - start)

###### Logging

There are *five built-in levels of* the l*og message*.

Debug: These are used to *give Detailed info*rmation, typically of interest *only when diagnosing problems*.

Info: These are *used to confirm* that *things are working as expecte*d

Warning: These are *used as an indication* that *something unexpected happened*, or is indicative of some problem in the near future

Error: This tells that *due to* a *more serious problem*, the *software has not been able to perform* some *function.*

Critical: This *tells serious error*, i*ndicating* that the *program itself may be unable to continue* running

Some *developers use* the *concept of printing* the statements *to validate if* the *statements are executed correctly* or if some error has occurred. But printing is not a good idea. It may solve your issues for simple scripts but *for complex scripts*, the *printing approach will fail*.

[*Python*](https://www.geeksforgeeks.org/python-programming-language/) *has* a built-in module ***logging*** which *allows writing status messages to a file* or any other output streams. The file can contain *information on* *which part of* the *code is executed* and what problems have arisen.

*Handler* is an *object responsible* *for* *processing log records* *created by* the *application*. Handlers *determine what happens to* *each log record*, such as w*riting it to a file*, *sending it over* the *network*, or printing it to the console.

import logging

from google.cloud import logging as gcp\_logging

# Set up logging client

client = gcp\_logging.Client()

logger = client.logger('my\_cloud\_function\_name') # Replace 'my\_cloud\_function\_name' with your Cloud Function name

# Configure root logger to use Cloud Logging handler

logging.basicConfig(level=logging.INFO)

cloud\_handler = gcp\_logging.handlers.CloudLoggingHandler(client)

cloud\_handler.setLevel(logging.INFO)

logging.getLogger().addHandler(cloud\_handler)

# Now you can use logging as usual

logging.info("This log message will be sent to GCP Cloud Functions logs")

By default, any log whose severity level is at least INFO that is written by your application is sent to Cloud Logging.

We can just use this:

# Imports the Google Cloud client library

from google.cloud import logging

# Instantiates a client

logging\_client = logging.Client()

# The name of the log to write to

log\_name = "my-log"

# Selects the log to write to

logger = logging\_client.logger(log\_name)

# The data to log

text = "Hello, world!"

# Writes the log entry

log\_name.log\_text(text)

print("Logged: {}".format(text))

###### OS

#importing os module

import os

# Method 1

print("MY\_HOME:", os.environ.get('MY\_HOME', "Environment variable does not exist"))

# Method 2

try:

print("MY\_HOME:", os.environ['MY\_HOME'])

except KeyError:

print("Environment variable does not exist")

List all files

os.listdir() method in python is used to get the list of all files and directories in the specified directory.

###### UUID

import uuid

x = uuid.uuid1()

table\_postfix = str(x).replace('-', '\_')

ts = time.time()

timestamp\_postfix = datetime.datetime.fromtimestamp(ts).strftime('%Y%m%d%H%M%S')

*If* all y*ou want is a unique ID*, you should probably *call uuid1()* or uuid4(). Note that *uuid1()* *may compromise privacy* *since it creates a UUID containing the computer’s network address.*

###### DATETIME

create date objects from timestamps using the fromtimestamp() method.

from datetime import datetime

# Getting Datetime from timestamp

date\_time = datetime.fromtimestamp(1887639468)

print("Datetime from timestamp:", date\_time) #Datetime from timestamp: 2029-10-25 16:17:48

*print* the *current date and time using* the *Datetime.now()* function.

today = datetime.now()

print("Current date and time is", today)#Current date and time is 2019-10-25 11:12:11.289834

###### Requests and Retry

import requests

from retrying import retry

# Randomly wait 1 to 2 seconds between retries to handle connection reset issue

@retry(stop\_max\_attempt\_number=3, wait\_random\_min=1000, wait\_random\_max=2000)

def get\_google\_client(type, project\_id):

if type == 'storage':

return storage.Client(project\_id)

if type == 'bigquery':

return bigquery.Client(project\_id)

# Function to read the webhook URL from GCS bucket

def read\_webhook\_file(gcs\_client, webhook\_bucket\_name, webhook\_file):

gcs\_bucket = gcs\_client.get\_bucket(webhook\_bucket\_name)

blob = gcs\_bucket.get\_blob(webhook\_file)

contents = blob.download\_as\_string()

return contents.decode('utf-8')

def slack\_notification(project\_id, webhook\_bucket\_name, webhook\_file, log\_message):

print('slack channel')

# gcs\_client = storage.Client(project\_id)

gcs\_client = get\_google\_client('storage', project\_id)

webhook\_url = read\_webhook\_file(gcs\_client, webhook\_bucket\_name, webhook\_file)

response = requests.post(

webhook\_url, json={'text': str(log\_message)},

headers={'Content-Type': 'application/json'}

)

if response.status\_code != 200:

raise ValueError(

'Request to slack returned an error %s, the response is:\n%s'

% (response.status\_code, response.text)

)

Retry

@retry(ZeroDivisionError, tries=3, delay=2)

def make\_trouble():

'''Retry on ZeroDivisionError, raise error after 3 attempts, sleep 2 seconds between attempts.''

Status Codes

1xx: Information

2xx: Successful

3xx: Redirection

4xx: Client Error

5xx: Server Error

##### Numpy common questions

###### Intro

*NumPy* (aka Numerical Python) is a *third-party open-source Python library* *used for scientific computations*. It *offers powerful* *functions and tools for performing* a *variety of arithmetic and logical operations* on numerical data. An *N-dimensional array* type is *known as ndarray*. All *elements stored in* a *ndarray* *should be homogeneous* i.e., have the same type. *Ndarray* is the *same as an array*.

###### Add list to numpy array

**import numpy as np**

# Existing NumPy array

arr = **np**.**array**([1, 2, 3])

# List of elements to add

elements\_to\_add = [4, 5]

# Adding elements to the NumPy array

arr = **np.append(arr, elements\_to\_add)**

print(arr) #[1 2 3 4 5]

elements\_to\_be\_add = [True, False]

arr = np.append(arr, elements\_to\_be\_add)

print(arr) #[1 2 3 4 5 1 0]

###### Create numpy arrays

import numpy as np

#1D Array

arr1 = np.array([1,2,3])

print(arr1)



#2D Array

X = ([[1.8, 2.5, 3.14],

[4, 5, 6]])

arr2 = np.array(X)

print(arr2)



#3D Array

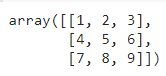
Y = ([[1,2,3],

[4,5,6],

[7,8,9]])

arr3 = np.array(Y)

print(arr3)

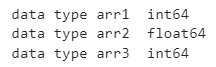


###### Identify type of an array

print('data type arr1 ', arr1.**dtype**)

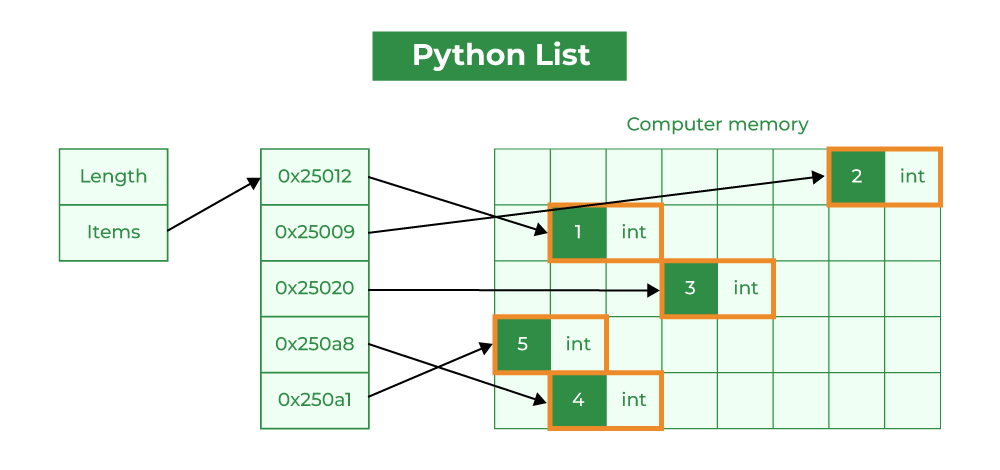
print('data type arr2 ', arr2.dtype)

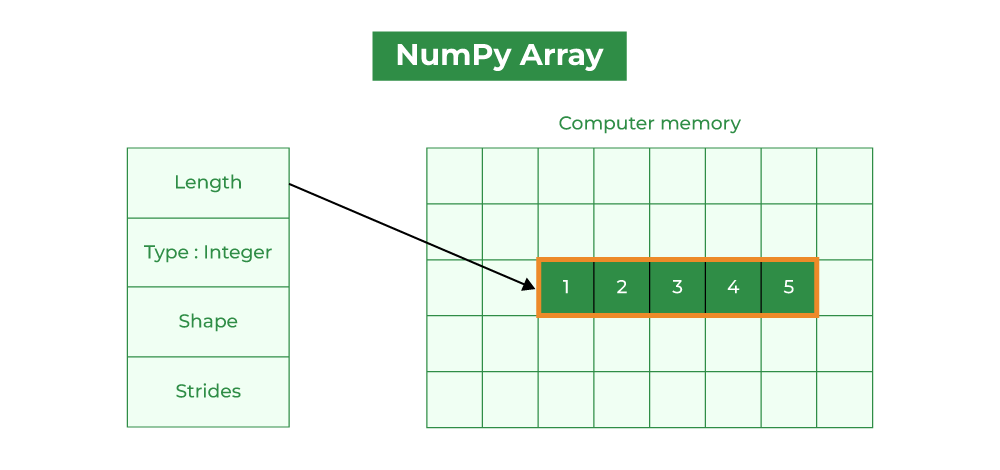
print('data type arr3 ', arr3.dtype)



###### Numpy array vs list

| **Feature** | **NumPy Arrays** | **Python Lists** |
| --- | --- | --- |
| Data type | Stores homogenous data i.e., data having the same types. *optimised for numerical computations*, s*tore elements of a single data type* (homogeneous) *for efficiency.* | *Stores heterogenous data* i.e., *elements can be of different data types.* |
| Flexibility | *Less flexible*, because *operations are performed element-wise* | Allows *more flexibility in adding and removing data* |
| Arithmetic and matrix operations | Many *vectors and matrix operations are in-built*. NumPy arrays e*xcel at numerical operations* and *vectorized calculations*, *making them significantly faster* for large datasets. | *Complex statistical and analytical libraries are not available*. Lists are *ideal for storing various data types* and offer flexibility in modifying elements. |
| Loops | *Explicit loop is required* to run through the elements | *Explicit loop is not required* to run through the elements |
| Declaration for usage | *NumPy module needs to be imported* for using NumPy arrays | *No import is required* to use Python lists |





Loops

import numpy as np

# Example with numpy array

numpy\_array = np.array([1, 2, 3, 4, 5])

print(numpy\_array \* 2) # No explicit loop required

#[ 2 4 6 8 10]

# Example with list

python\_list = [1, 2, 3, 4, 5]

result = []

for item in python\_list:

result.append(item \* 2) # Explicit loop required

print(result)

#[2, 4, 6, 8, 10]

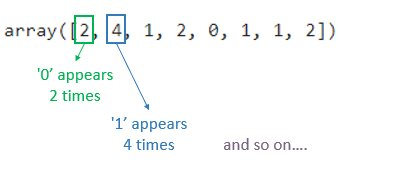
print(python\_list \* 2)

#[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

###### Count the number of times a value appears in a NumPy array of integers?

arr = np.array([**1**, 7, **1, 1**, 7, 6, 5, **1**, 3, 3, 2, **0, 0**])

np.bincount(arr)



import numpy as np

array = np.array([1, 2, 3, 1, 2, 3, 3, 4, 5, 6, 7, 5])

**unique**, **counts** = np.**unique**(array, **return\_counts**=True)

print(unique) #[1 2 3 4 5 6 7]

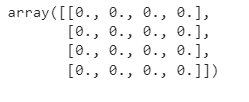
print(counts) #[2 2 3 1 2 1 1]

###### Create an array with all values as zeros or ones?

#Creating array of zeros

zeros\_arr = np.**zeros**((4,4)) #np.ones((4,4))

zeros\_arr



###### Print an array range between 1 to 35 and show 7 integer random numbers.

rand\_arr = np.**random.randint**(1,35,7)

rand\_arr



###### Change the data type of a NumPy array?

You can *use* the *astype() method* for this:

arr = np.array([2.7, 3.14, 5.89, 7])

new\_arr = arr.**astype**('i')

print(new\_arr)

print(new\_arr.dtype)



###### Reverse a 3×3 matrix?

import numpy as np

matrix = np.**arange**(30,39).**reshape**(3,3)

print(matrix)

[[30 31 32]

[33 34 35]

[36 37 38]]

flat = matrix.**flatten**()

print(flat)

[30 31 32 33 34 35 36 37 38]

reverse = flat[::-1]

print(reverse)

[38 37 36 35 34 33 32 31 30]

print(reverse.reshape(3,3))

[[38 37 36]

[35 34 33]

[32 31 30]]

###### Numpy array manipulation

numpy.**dot**(a, b)

import numpy as np

# Create two NumPy arrays

array1 = np.array([1, 2, 3, 4, 5])

array2 = np.array([6, 7, 8, 9, 10])

# Perform element-wise operations

result\_addition= array1 + array2 #[ 7 9 11 13 15]

result\_subtract = array1 - array2 #[-5 -5 -5 -5 -5]

result\_multiply = array1 \* array2 #[ 6 14 24 36 50]

result\_divide = array1 / array2 #[0.16666667 0.28571429 0.375 0.44444444 0.5 ]

result\_power = np.power(array1, 2) #[ 1 4 9 16 25]

###### Find the transpose of the matrix using NumPy?

#Construct a matrix

X = ([[38,13],

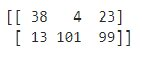
[4,101],

[23,99]])

#Find the transpose of the matrix

transpose = np.**transpose**(X)

print(transpose)



###### How do you stack matrices?

*NumPy arrays/matrices* *can be stacked* either *horizontally or vertically* through the following methods:

* np.*hstack*() method *adds* the *second* *ndarray* t*o the columns* *of* the *first ndarray*:

#Construct two matrices

A = ([[1,2,3],

[4,5,6,],

[7,8,9]])

B = ([[2,4,6],

[8,10,12],

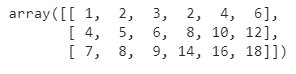
[14,16,18]])

arr\_A = np.array(A)

arr\_B = np.array(B)

#Stacking Horizontally

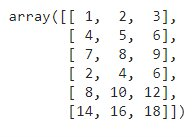
np.**hstack**((arr\_A, arr\_B))



* np.*vstack*() method *combines* the *second ndarray* *as new rows* *in* the *first ndarray*:

#Stacking Vertically

np.**vstack**((arr\_A, arr\_B))



###### Create a new array consisting of common values from two given arrays?

We need to *find* the *positions where* the *elements of the two* arrays *match*:

#Create two 1D arrays

a = np.array([1,2,3,2,3,4,3,4,5,6])

b = np.array([1,7,9,2,7,4,3,6,4,6])

np.where(a == b)

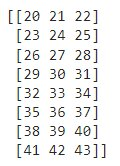


###### Split a given array into four equal-sized sub-arrays.

* For this, let’s create an 8×3 integer array:

arr = np.**arange**(20,44).**reshape**(8,3)

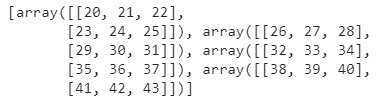
print(arr)



* Now, you can create 4 sub-arrays using the split() function:

sub\_arr = np.**split**(arr, 4)

print(sub\_arr)



###### Swap rows and columns

num\_arr[[0,3]] = num\_arr[[3,0]]

Original array:

[**[1 3 1]**

[3 1 3]

[2 9 2]

**[9 2 9]**]

Array after swapping the rows:

[**[9 2 9]**

[3 1 3]

[2 9 2]

**[1 3 1]**]

my\_array[:, [2, 0]] = my\_array[:, [0, 2]]

Original array:

[[ **0**  1 **2**]

[ **3**  4 **5**]

[ **6**  7 **8**]

[ **9** 10 **11**]]

After swapping arrays the last column and first column:

[[ **2**  1 **0**]

[ **5**  4 **3**]

[ **8**  7 **6**]

[**11** 10 **9**]]

##### Pandas Common questions

<https://www.geeksforgeeks.org/pandas-interview-questions/>

###### Intro

[Pandas](https://www.geeksforgeeks.org/pandas-tutorial/) is an open-source Python *library* that is *built on top of* the *NumPy* library. It is made for working with relational or labelled data.

###### What are the Different Types of Data Structures in Pandas?

The *two data structures* that are *supported* by Pandas *are* Series and DataFrames.

* Pandas [Series](https://www.geeksforgeeks.org/python-pandas-series/) is a *one-dimensional* labelled *array* *that can hold data of any type*. It is mostly used to *represent a single column* or row of data.
* Pandas [DataFrame](https://www.geeksforgeeks.org/python-pandas-dataframe/) is a *two-dimensional* heterogeneous *data structure*. It *stores data in a tabular form*. Its three main components are *data, rows, and columns.*

###### Series

*Creat*ing a *Series from an* *Array/List*

# import pandas and numpy

import pandas as pd

import numpy as np

# simple array

data = np.array(['g', 'e', 'e', 'k', 's'])

# convert array to Series

print(**pd.Series**(data))

0 g

1 e

2 e

3 k

4 s

dtype: object

Creating a Series from an Array with a custom Index

# import pandas and numpy

import pandas as pd

import numpy as np

# simple array

data = np.array(['g', 'e', 'e', 'k', 's'])

# providing an index

ser = **pd.Series**(data, **index**=[10, 11, 12, 13, 14])

print(ser)

10 g

11 e

12 e

13 k

14 s

dtype: object

Creating a *Series from Dictionary*

# import pandas

import pandas as pd

# a simple dictionary

dict = {'Geeks': 10,

'for': 20,

'geeks': 30}

# create series from dictionary

print(**pd.Series**(dict))

Geeks 10

for 20

geeks 30

dtype: int64

Creating a *Series from Scalar Value*

ser = pd.Series(10, index=[0, 1, 2, 3, 4, 5])

print(ser)

0 10

1 10

2 10

3 10

4 10

5 10

dtype: int64

###### Dataframes

*Creat*ing a *DataFrame using a List*

# import pandas as pd

import pandas as pd

# list of strings

lst = ['Geeks', 'For', 'Geeks', 'is', 'portal', 'for', 'Geeks']

# Calling DataFrame constructor on list

print(**pd.DataFrame**(lst))

0

0 Geeks

1 For

2 Geeks

3 is

4 portal

5 for

6 Geeks

Creating a *DataFrame using* a *List of Lists*

# import pandas as pd

import pandas as pd

# list of strings

lst = [[1, 'Geeks'], [2, 'For'], [3, 'Geeks']]

# Calling DataFrame constructor

# on list with column names

print(**pd.DataFrame**(lst, **columns**=['Id', 'Data']))

Creating a *DataFrame using a Dictionary*

import pandas as pd

# initialise data of lists.

data = {'Name':['Tom', 'nick', 'krish', 'jack'], 'Age':[20, 21, 19, 18]}

# Print the data frame created

print(pd.DataFrame(data))

Name Age

0 Tom 20

1 nick 21

2 krish 19

3 jack 18

<https://www.geeksforgeeks.org/python-interview-questions/>

##### How did you perform the Unit test?/What are unit tests in Python?

“CHATGPT”

Unit tests in Python are a way to verify that individual units of code (like functions or classes) work as expected. You write test cases to check different scenarios and edge cases. Python has built-in libraries like unittest and pytest for organizing and running unit tests. To perform unit tests, you typically:

* Write test cases: Define scenarios to test different behaviors of your code.
* Execute tests: Run the tests using testing frameworks.
* Verify results: Check if the actual output matches the expected outcome.