* Python is a high-level language as the translation of Python code takes place into machine language, using an interpreter.
* Python is an interpreted programming language.
* Special programs known as interpreters check the entire Python code and all the errors in the entire Python code is reported at once.
* Python programs are usually slower than C programs as they are interpreted and also since it determines the data type at run time as it is dynamically typed**.**
* In Python, variables are untyped, that is, there is no need to define the data type of a variable while declaring it.
* A given variable in Python can store values of different data types in different parts of the Python code.
* Memory management is automatically handled in Python by the Garbage Collector provided by it.
* Python is a more robust programming language compared to C as it has strong memory management schemes.
* The scope of the variables used in Python is not limited to blocks or loops. The variables are accessible even outside the curly braces.

Reading input from the testcases:

From sys import stdin

T=int(stdin.readline().rstrip())

n = int(stdin.readline().rstrip())

arr = list(map(int, stdin.readline().rstrip().split(" ")))

**Lamda functions:**

* sum=lambda x,y,z:x+y+z

print(sum(2,3,4))

* Takes less time than def func() method.
* L2=list(map(lambda x:x/2, l1) where l1 is a list.

**Map functions:**

* map(function, iterable)- used for calling function on a list, tuple, iterable.
* Can be easily applied on list of numbers, etc than running a loop.
* l=(1,2,3)

a=lambda x:x+1

print(list(map(a,l))[0])

**Arguments- positional and keyword:**

* def details(name, age=29):

print(name, age)

details(‘Ramesh’)

name- positional arg(necessary to provide),

age= keyword arg.(not necessary to provide, default value is given,29).

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

\*args- give as many arguments.

**Iterator- Iter(list)-** it creates a memory for the list but does not create memory for all the elements of list. Only when the elements are called one by one through next(list) or for loop the memory is allocated to each element. Saves memory when list is huge. Acts like dynamic allocation of memory for each element of any iterable only when called upon.

L=[1,2,3,4 ]

Iterator=iter(L)

Print(next)

For I in iter(L):

Print(i)

**Generator:**

For i in range(8):

yield i

* Its like creating an iterator. It saves the local variable value.
* Generator is must faster than iterator.

Decorator:

Adds additional functionality to exsisting code, extends functionality. Used for logging etc .

def my\_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func()

print("Something is happening after the function is called.")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

# Calling the decorated function

say\_hello()

**Custom exception handling**- create custom exception class by inheriting Exception class.

## Created error and bioterror classes with inheriting Exception class in it.

def error(Exception):

Pass

def bioterror(error):

pass

int a=12

try:

if a>12:

print(‘okay’)

else:

raise bioterror

except bioterror:

print(“number not greater than 12”)

**assert statement:** check if logical statmenet is true or false.

num=10

assert num<10

assert sum([1,2,3])==6

If False, it throws attribute error.

**Deep and shallow copy**

**Shallow copy:**

L1=[1,2,3]

L2=L1.copy()

L2[1]=12

This will not change the value of l1[1].

L1=[[1,2,4],[2,3,4]]

L2=L1.copy()

L1[1][0]=23

This will change the value in both l1 and l2. This happens for nested list in shallow copy.

Both the variables are referring to the same location inside the list.

**Deep copy:**

For normal list, shallow copy==deep copy

For nested list, they are not equal, since both lists are creating different locations of each list and storing values.

**Class variables updation: classmethod**

To update class variables:

Class car():

Base\_price=1000

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

….

….

……

@classmethod

def change\_price(cls,discount):

base\_price= base\_price- base\_price\*discount

**car.**change**\_price(**0.20)

* Here base\_price is a class variable.
* If we want to update it, use @classmethod.
* Use class name and call the method.
* Use cls inplace of self.

**Static class**

Class car():

.

.

.

.

@staticmethod

Def check\_year():

.

.

return xyz

* static methods are very fast.
* They are initialised first when a class is initialised.
* They do not take any arguments.
* Can be used for generic purpose methods.

**Sort**

* Sorting in dict: if sorting by value and two values are same then order will be whoever is encountered first in dict.
* Sort by decreasing order of key/value: d=dict(sorted(d.items(),key=lambda x:x[0])). – ascending order of keys.
* d=dict(sorted(d.items(),key=lambda x:x[1], reverse=True)). – descending order of values.

**Garbage collector**

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.

The garbage collection can be invoked manually in the following way:

import gc

collected = gc.collect()

**Interview Questions**

* Can you tell us if Python is object-oriented or functional programming?

Python is considered to be a multi-paradigm language, which means it supports multiple programming techniques including object-oriented and functional programming.

* Enlist some of the benefits of using python language.

Python is a high-level, object-oriented programming language that enhances user interaction through objects, modules, and automatic memory. The language finds widespread use in data science, artificial intelligence, and machine learning because of its in-built data structures. Despite being a high-level language, the simplicity of its syntax makes Python a very easy language to grasp. Moreover, because Python supports various modules and packages, making applications using Python becomes extremely easy as less code is required.

* What is init method in python?

It is a constructor method automatically called to allocate memory when a new instance or object is created, and classes in Python have an init associated with them which initializes attributes declared in the class when an object of that class is created.

* Difference between arrays and lists in python

In order to use arrays in Python, one must import either an array module or a NumPy package, whereas lists are already built into the language and do not require declaration.

* What are first class objects ?

In Python, functions are first class objects 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.
* You can store the function in a variable.
* You can pass the function as a parameter to another function.
* You can return the function from a function.
* You can store them in data structures such as hash tables, lists.
* What are decorators in python?

Decorators are used to modify the behaviour of function or class. Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.

OOPS Concepts:

1. Encapsulation:

* It describes the idea of wrapping data and the methods that work on data within one unit.
* This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
* To prevent the accidental modification of data, an object’s variable can only be changed by an object’s method. Those types of variables are known as **private variables.**

1. Abstraction: hiding internal information or functionality from the user and only showing the relevant information.

Using Access modifiers:

* **Private variables in python are denoted by \_\_ in python.**
* **class** Base:
* **def** \_\_init\_\_(self):
* self.a **=** "GeeksforGeeks"
* self.\_\_c **=** "GeeksforGeeks"
* # Creating a derived class
* **class** Derived(Base):
* **def** \_\_init\_\_(self):
* # Calling constructor of
* # Base class
* Base.\_\_init\_\_(self)
* print("Calling private member of base class: ")
* print(self.\_\_c)
* # Driver code
* obj1 **=** Base()
* print(obj1.a)
* Protected members are denoted by \_ in Python.

**class** Base:

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

        # Protected member

        self.\_a **=** 2

# Creating a derived class

**class** Derived(Base):

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

        # Calling constructor of

        # Base class

        Base.\_\_init\_\_(self)

**print**("Calling protected member of base class: ",

              self.\_a)

        # Modify the protected variable:

        self.\_a **=** 3

**print**("Calling modified protected member outside class: ",

              self.\_a)

obj1 **=** Derived()

obj2 **=** Base()

# Calling protected member

# Can be accessed but should not be done due to convention

print("Accessing protected member of obj1: ", obj1.\_a)

# Accessing the protected variable outside

print("Accessing protected member of obj2: ", obj2.\_a)

* There are two types of abstraction:
* **Data abstraction**: the original data entity is hidden via a data structure that can internally work through the hidden data entities.
* **Process abstraction:** It refers to hiding the underlying implementation details of a process.
* A class is said to be an abstract class if it cannot be instantiated, that is you can have an object of an abstract class but you can use it as a base or parent class for constructing other classes.
* To form an abstract class in Python, it must inherit ABC class that is defined in the abc module. This module is available in Python's standard library.
* The abstract class must have at least one abstract method.
* An abstract method is the one with no code logic and only ‘pass’ and which cannot be called, but can be overridden. You need to decorate it with @abstractmethod decorator.
* Car is an abstract class derived from ABC. This class has an abstract method printDetails and other concrete methods where some functionality is written.
* It has derived classes hatchbatck and SUV.
* An object of Hatchback is created and car methods are also called from it.

# Import required modules

**from** abc **import** ABC, abstractmethod

# Create Abstract base class

**class** Car(ABC):

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

    self.brand **=** brand

    self.model **=** model

    self.year **=** year

  # Create abstract method

  @abstractmethod

**def** printDetails(self):

**pass**

  # Create concrete method

**def** accelerate(self):

**print**("speed up ...")

**def** break\_applied(self):

    print("Car stop")

# Create a child class

**class** Hatchback(Car):

**def** printDetails(self):

**print**("Brand:", self.brand);

    print("Model:", self.model);

    print("Year:", self.year);

**def** Sunroof(self):

**print**("Not having this feature")

# Create a child class

**class** Suv(Car):

**def** printDetails(self):

    print("Brand:", self.brand);

**print**("Model:", self.model);

    print("Year:", self.year);

**def** Sunroof(self):

**print**("Available")

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

car1.printDetails()

car1.accelerate()

* If you try to declare an object of demo class, Python raises TypeError: Can't instantiate abstract class Car with abstract method “printDetails
* The **demo** class here may be used as parent for another class. However, the child class must override the abstract method in parent class. If not, Python throws this error − Can't instantiate abstract class SUV with abstract method printDetails.

1. **Inheritance:**

* class Student():
* def \_\_init\_\_(self,name,age,clas):
* self.name=name
* self.age=age
* self.clas=clas
* def display(self):
* print('My name is {}. I am {} years old and in class {}'.format (self.name, self.age, self.clas))
* class Physicsstudent(Student):
* def \_\_init\_\_(self,name,age,clas,marks):
* super().\_\_init\_\_(name,age,clas) #using base class constructor
* self.marks=marks
* def marksdisplay(self):
* print("MArks are:",self.marks)
* s2=Physicsstudent("Varun",24,6,23)
* s2.display()
* s2.marksdisplay()
* s2.clas
* s2.name
* super().\_\_init\_\_()

1. **Polymorphism**: refer to the occurrence of something in multiple forms. As part of polymorphism, a Python child class can have methods with the same name as a parent class method.

**Using functions and Objects**:

class Dog():

     def animal\_kingdom(self):

       print("Mammal")

     def legs(self):

       print("Four")

class Lizard():

     def animal\_kingdom(self):

       print("Mammal")

     def legs(self):

       print("Four")

def function1(obj):

       obj.animal\_kingdom()

       obj.legs()

obj\_dog = Dog()

obj\_lizard = Lizard()

function1(obj\_dog)

function1(obj\_lizard)

**Method overriding**:

* A way of polymorphism of methods which are re-defined in child class. Method overriding is a technique that lets us change the function implementation of a parent class in the child class.
* Conditions of Method Overriding:
* There should be inheritance. Within a class, function overriding is not possible, therefore a child class is derived from a parent class.
* A function of the child class should have the same number of parameters as that of the parent class.

Example:

class Vehicle:

     def desc(self):

       print("So many categories of vehicle")

     def wheels(self):

       print("Differs according to the vehicle category")

class car(Vehicle):

     def wheels(self):

       print(4)