**Python**

**Vanilla python:**

For binary: try to calculate with 2’s power then sees the sum of it, how much you are getting to that decides the number of bits, and likewise you can then decide the 1 and 0’s.

Bin of (111) = 1 \* 2^2 + 1 \* 2^1 + 1 \* 2\*0

Hashable: objects which can be hashed, int, str, tuple (with unique values)

What is hash= unique and fixed (no change), immutable objects are hashable, whereas mutable are not hashable.

Mutable in python: List, dict, sets etc.

Immutable: int, float, str, tuple, frozen set etc.

* How to get the smallest string from the list of strings, **min (l, key = Len)**

Comprehensions: Runs the code from left to right, i.e. the left one is outer one and right one is inner one and likewise the hierarchy in the loops. Left to right, same is the case for the conditions, left to right condition after the loop is condition for that loop. To simply the evaluation just tries the logic with left loop as outer loop and first immediate condition as outer condition right loop as in inner one and right condition after that is the inner loops condition.

What is hexadecimal representation?

* It is the representation with 0-9 and then A-F, use int(‘FF’,16) for int conversion like you do for the int(binary,2)

Lexicographical sorting: sorting based of the ascii values.

Sorted ([‘Harshal’, ‘Nayan’]), sorting start based off comparing from left, if unmatched found then it stops there itself. Same for the numbers.

Same for the comparison, i.e. “Harshal” < “Nayan”, same lexicographical comparison, start from the left then stops as soon as the inequality is matched.

**Module, package and library?**

A **module** is a single file containing functions, classes, and variables; a **package** is a directory with multiple related modules and an \_\_init\_\_.py file to organize them; and a **library** is a collection of packages and modules bundled together to provide broader reusable functionality across projects. Modules form the basic building blocks, packages group these modules, and libraries encompass multiple packages for larger-scale use.

**Modules:** math, random, os, sys, re, time, Json, shutil, queue, pprint

**Packages:**  
flask, django, numpy, pandas, scipy, matplotlib, sklearn, urllib, xml, tkinter

**Libraries:**  
NumPy, Pandas, Matplotlib, Scikit-learn, TensorFlow, Requests, Flask, Django, Beautiful Soup, Pygame, SciPy, Seaborn

e.g. from itertools import combinations

from itertools import groupby

or itertools. combinations () or itertools.groupby()

both ways work, meaning you can access the classes and function in the module just like the methods of any object instance.

**OOPS:**

Special methods (Dunder methods):

Methods which have special ability, which gets externally called based off the conditions specified mostly externally, the logic is written for the special condition in terms of new method, and as the condition is mate the same method is called.

Example, add, write the special method inside the class with \_\_add\_\_ where it takes one set of input from the current (object) from the future, and the other input from the other object where the other object must be passed to the special method like the self. The special condition here would be the use of + in the object1 and object2. Likewise other methods are defined many are specific to single object whereas the other are interactive. E.g. \_\_len\_\_, \_\_str\_\_ etc. here \_\_str\_\_ get called on the condition of the using str or print on the object.

Whenever you create attributes from outside those just get sorted as it is, without any self and not in the constructor or anything.

**In classes, in any methods I can either use variable = or self. attribute to create the attribute just the difference is self. attribute is accessible throughout the instance whereas only attribute is accessible in that methods execution.**

**self.pin is an instance attribute, while pin is a separate local variable; they are two different variables even if their values are the same.**

Self is not specifically to self; we can use any other word.

Special methods logic, special def name is given to methods and this same method is called by special object interaction or operation, like +, -, / and print, where on using this operation on object return of these special methods are returned. In case where two object interaction are needed then those methods are passed with other object as arguments as well like other and other is treated as second object while writing the logic in the method.

**Pass by reference:** Function receives the actual reference to the original variable, so any modification changes made to the reference variable affects the original variable.

**Pass by Value:** Whereas in pass by value, function receives the value of the variable or copy of the variable as an argument to the function so, any changes made are not reflected back to the original object/variable.

In python everything is a pass by reference, but the immutable object behaves like pass by value.

**Encapsulation:** hiding things, it **works similarly** on attributes as well as **on methods.**

There is one more thing is which is **“\_”** Using a single underscore act as a hint to other programmers that the attribute is "protected" or non-public, but it does not make it private or hidden—unlike double leading underscores, which trigger name mangling for stronger protection.

In Python, **inheritance** ("is-a" relationship) means one class derives from another, inheriting its attributes and methods, like class Dog (Animal). This allows the subclass to reuse and extend the superclass behaviour. **Aggregation** ("has-a" relationship) means one class contains or uses an instance of another class, such as a Car having an Engine object, where both can exist independently. Inheritance models strong hierarchical connections, while aggregation models

* **Static Variable (Class Variable):**A variable defined inside a class but outside any method. It is shared by all instances of the class, so every object accesses the same variable. Changes to it affect all instances unless shadowed by an instance variable. Can be accessed by **class. attribute**
* **Static Method:**A method defined inside a class with the @staticmethod decorator. It does not receive any automatic reference to the instance (self) or class (cls) and cannot access or modify instance or class data. Static methods behave like regular functions grouped inside the class for logical organization.

normally when the child does not naturally access any of the methods in that case to have the access of those methods **super** is use. Mainly the parents’ constructor can be called in child’s constructor as child’s constructor if defined won’t have access to parents’ constructor, so it is called in child’s constructor with super (). \_\_init\_\_ (). Or any other method, super is mainly used to call methods in Childs.

The **super ()** function in Python is used to explicitly call a parent class’s method from a child class, especially when the child overrides that method. Using super (), you can "force" the parent method to be executed within the child’s overridden method. This is useful when you want to extend or modify the parent’s behaviour rather than completely replace it.

**Exception handling:**

Normally when any execution error is thrown, in that case, first display the error type and then an error message.

**e.g. ZeroDivisionError: division by zero**

Try block has the main code, depending on the return of the try block, except block is triggered.

When I do normally

except Exception as e:

print(e) ## here in this block e is available which holds the error message against that error type.

This is the scenario for the general exceptions. For specific ones, we can have

except ZeroDivisionError:

print (“can’t divide by zero”)

raise does what? so raise your own error which are then catch by except block.

I can either raise specific error or in general exception with any message which you want which is basically creating your own type of error

e.g. raise ValueError (“This value is not allowed”)

e.g. raise raise Exception("nope")

and at the same time, we can have custom exception defined by inheriting Exception class.

try:

raise Exception("nope")

except Exception as e:

print(e) # Output: nope

**dir** in python is to list all the methods and attributes of the python object.

**Decorators:**

Takes one function as an input wraps it inside another function and then returns the wrapped functions with extra new behaviour.

def decorator(func):

def wrapper ():

logic with func

return wrapper (i.e. same func with something wrapped around it)

def func(a):

return a

apart from manually calling the process step by step we use below @ decorator while defining this func only and then normally call func, then decorated func gets executed. And we get the wrapped func.

@decorator

def func(a):

return a

what happens logically step by step:

* The moment you put @decorator at the top of func, func is passed to the decorator and return of the decorator is stored in name same variable func.
* I.e. func = decorator(func)
* And when func is called this updated already wrapped func is called which was at the moment a @decorator was written at the top of it.

We use decorators in Python mainly to add extra functionality to functions or methods without modifying their original code. Decorators provide a clean, reusable, and modular way to "wrap" functions, allowing you to extend or change their behavior, such as adding logging, input validation, timing, caching, or authentication. This helps keep your code cleaner and less cluttered by separating cross-cutting concerns (like debugging or access control) from core logic.

**range (1,10)**

In Python, range () is a **lazy iterable** that represents a sequence of numbers without generating them all at once, which makes it memory-efficient—like **generators**, which are **lazy iterators** created using yield or generator expressions. Unlike lists (which are **eager** and store all values immediately), both range and generators compute values **on demand**, making them ideal for large or infinite sequences. For example, to see all the numbers in a range, you can convert it to a list using list (**range (1, 10**)), which forces Python to generate and store the numbers from 1 to 9. While range is reusable and behaves like a sequence, generators are true iterators that can only be consumed once. This lazy behaviour supports efficient looping, better performance, and reduced memory usage in Python programs.

**NumPy**

* **Mesh grid**takes two 1D arrays x and y and produces two structured 2D arrays XX and YY that represent the Cartesian product of all possible (x, y) coordinate pairs. Each corresponding element XX [i,j] and YY [i,j] forms one unique coordinate, and the total number of elements in either grid equals len(x) \* len(y). The structured grid format is especially useful for vectorized computations, visualizations (surface plots, contour maps, heatmaps), and scientific simulations, as it allows evaluating functions or applying operations to every point in the grid without explicit loops. While this representation contains all combinations of x and y, its 2D structure makes it directly compatible with plotting libraries and numerical methods; if needed, it can be flattened to get a simple list of coordinate pairs.
* **Seed** is a starting value for a pseudo‑random number generator that ensures reproducibility — using the same seed will always produce the same sequence of “random” numbers. This is a general concept used across programming languages and libraries, not just NumPy. In Python, the built‑in random. seed () seeds Python’s own random generator, while NumPy has its own implementation numpy.random.seed() for its legacy random number generator, and the modern numpy.random.default\_rng(seed) for improved control. Many libraries like pandas and scikit‑learn include a random\_state parameter, which internally works like setting a seed so that operations involving randomness (e.g., sampling, data splitting) give consistent, repeatable results across runs.
* **Structured array** in NumPy is a special type of ndarray where each element can contain multiple named fields, each with its own data type, like a row in a database table or a struct in C. This array allows you to group heterogeneous data—different types and sizes—together in one array. For example, one element can have a string field for a name, an integer field for age, and a float for weight. All elements share the same field names and structure. You can access or modify data in these fields using their names, either with dictionary-style indexing or dot notation in record arrays. Structured arrays are useful for handling tabular data efficiently in scientific computing, enabling organized and memory-efficient representation of complex datasets.
* **Mode:** There is no mode in NumPy, how to get the most frequent element then? Using elements, counts = np.unique(a, return\_counts=True), returns two arrays, one with elements and second one being counts, get the argmax of counts and index elements based of it.
* **NumPy does not have apply,** we can not apply function, but broadcasting can be used while applying custom functions logic, like use,

def func(arr):

return np.exp(arr) – np.exp(-arr)

Sorting index, classical mistake is putting the new index of the corresponding current value, which is wrong, rather put sort the values, then see which one is first and see what its current position is and then put that position at the first place, and then do the same one by one, putting elements one by one like first, second, third.

**Pandas:**

**Apply** functions: axis = 0 which is by default, function takes one column as an input at a time sequentially, whereas for axis=1 i.e. along axis a row is send to the function at a time, so this is how backend iterative operation happens in apply.

**Apply is an iterator** which iterate on either rows or columns, depending on axis = 1,0 which returns usually a series, but it is totally depends on the function logic returns.

Apply on group by object, (split, apply, combine) works similarly as it works on the data frame just the input in this case to the function is entire sub data frame, then it depends how you handle this. group\_keys = True while doing grouping a data frame and group\_includes = True while apply.

**Concat:** columns which are same gets, matched well everything else is outer join i.e. new unique column gets created.

**Python Problem solving**

* Sort the list with list elements based of the sub list elements values.

sorted (array, key = lambda x:x[k])

* Do the same with the loops, then you must remove the elements of the list and continue iterating, but while removing the elements and iterating on the same list gets messy with the indexing, so better to break the sub iteration there and then basically the outer iteration would be on the revised list.

**NumPy:**

Np.round (a,2) where second parameter describes up to what digit to round off to.

Masking and indexing, both are used for accessing the specific elements of the array, indexing is based off the index values whereas the masking is based off the Boolean conditions (True/False) of the similar shape.

Np.tile – repeats the array in the specific directions.

Np.repeats- repeats the individual values multiple times.