Core Python Concepts

Q. What are Python's key features?

- **Interpreted:** Executes code line by line, no compilation needed.
- **High-level & dynamically typed:** Simplifies coding with automatic type handling.
- **Object-oriented:** Supports classes and objects for modularity.
- **Cross-platform:** Runs on Windows, macOS, and Linux.
- **Extensive libraries:** Offers built-in and third-party modules for most tasks.

Q. What are Python's data types?

- **Numeric:** int, float, complex — for numbers.
- **Sequence:** str, list, tuple — for ordered data.
- **Set & Mapping:** set, dict — for unique or key-value pairs.
- **Boolean & NoneType:** bool, NoneType — for logical and null values.

Q. What is PEP 8 and why is it important?

- **PEP 8:** Official style guide for Python code.
- **Importance:** Ensures clean, readable, and consistent code across developers.

Q. Explain mutable vs immutable objects.

- **Mutable:** Can change after creation (list, dict, set).
- **Immutable:** Cannot change once created (str, tuple, int).

Q. What are Python's built-in data structures?

- **List:** Ordered, mutable collection.
- **Tuple:** Ordered, immutable collection.
- **Set:** Unordered, unique elements.
- **Dictionary:** Key-value mapping, fast lookups.

Q. Difference between Python 2 and Python 3.

- **Strings:** Python 3 uses Unicode by default.
- **Print:** Function in Python 3 (print() vs statement).
- **Division:** returns float in Python 3.
- **Compatibility:** Python 3 is not backward compatible.

Q. What are Python's memory management features?

- **Private heap:** Stores all Python objects and data structures.
- **Reference counting:** Tracks number of references to each object.
- **Garbage collector:** Frees memory of unused objects automatically.

Q. Explain indentation in Python.

- **Indentation:** Defines code blocks (like braces in C).
- **Mandatory:** Wrong indentation raises an IndentationError.

Q. How is Python interpreted?

- **Bytecode:** Source code compiled to bytecode first.
- **PVM:** Python Virtual Machine executes bytecode line by line.

Q. Explain namespaces.

- **Namespace:** Container mapping names to objects.
- **Types:** Local, global, built-in, and enclosing scopes manage variable access.
- **Local Namespace**: names of the functions and variables declared by a program. This namespace exists as long as the program runs.
- **Global Namespace**: This namespace holds all the names of functions and other variables that are included in the modules being used in the python program. It includes all the names that are part of the Local namespace.

- **Built-in Namespace**: This is the highest level of namespace which is available with default names available as part of the python interpreter that is loaded as the programing environment. It include Global Namespace which in turn include the local namespace.

Q. Why isn't all the memory de-allocated when Python exits?

- When Python quits, some Python modules, especially those with circular references to other objects or objects referenced from global namespaces, are not necessarily freed or deallocated.
- Python would try to de-allocate/destroy all other objects on exit because it has its own efficient cleanup mechanism.
- It is difficult to de-allocate memory that has been reserved by the C library.

Q. "In Python, Functions Are First-class Objects." What Do You Infer from This?

It means that a function can be treated just like an object. You can assign them to variables, or pass them as arguments to other functions. You can even return them from other functions.

Q, What are callables?

- Functions: Regular functions defined with the def keyword or lambda functions created with lambda.
- Methods: Functions defined within classes that can be called on instances of those classes.
- Classes: You can call a class to create an instance of that class, for example, my_instance = MyClass().
- Callable Objects: Some objects define a special method _call_ that allows them to be called as if they were functions. You can create callable instances of a class by defining a _call_ method.
- Built-in Functions: Python's built-in functions like len(), print(), and open() are callables
- Callable Instances: Objects can be made callable by defining the _call_ method, enabling you to create instances that act like functions.

Q, What Does Python Support? - Call By Reference OR Call By Value

❏ Python utilizes a system, which is known as "Call by Object Reference" or "Call by assignment".

❏ In the event that you pass arguments like whole numbers, strings or tuples to a function, the passing is like call-by-value because you cannot change the value of the immutable objects being passed to the function.

❏ Whereas passing mutable objects can be considered as call by reference because when their values are changed inside the function, then it will also be reflected outside the function

Example 1:

```
s1 = "Geeks"
def test(s1):
        s1 = "GeeksforGeeks"
        print("Inside Function:", s1)

test(s1)
print("Outside Function:", s1)
```

Output:
Inside Function: GeeksforGeeks
Outside Function: Geeks

Example 2:

```
def add_more(list):
        list.append(50)
        print("Inside Function", list)

mylist = [10,20,30,40]
add_more(mylist)
print("Outside Function:", mylist)
```

Output:
Inside Function [10, 20, 30, 40, 50]
Outside Function: [10, 20, 30, 40, 50]

Data Structures

Q. List vs Tuple vs dict vs set differences

- **List:** Mutable, slower, uses [], more memory
- **Tuple:** Immutable, faster, uses (), can be used as dict keys, less memory
- **Sets-** Unordered, mutable, unique eliments , {}
- **Dict-** Unordered, key-value pairs, unique keys, {key:value}

Q. How are sets used?

- **Sets:** Store **unique elements** and support **fast membership testing** using hashing.

Q. What are dictionaries?

- **Dictionary:** Unordered collection of **key-value pairs** allowing **O(1)** lookups and updates.

Q. How to merge dictionaries?

- **Method 1:** {**dict1, **dict2} merges copies.
- **Method 2:** dict1.update(dict2) modifies in place.

Q. Remove duplicates from list

- **Using set:** list(set(lst)) removes duplicates but loses order.
- **Using dict:** list(dict.fromkeys(lst)) preserves order.

Q. What are list comprehensions?

- **Definition:** Compact way to create lists from iterables.
- **Example:** [x**2 for x in range(5)] → [0, 1, 4, 9, 16].

Q. Flatten nested list

- **Using sum:** sum(nested, []) joins lists.
- **Using itertools:** itertools.chain(*nested) efficiently flattens.

Q. Shallow vs deep copy

Shallow copy: Copies references only (copy.copy()) , changes in the original object are reflected in the copy.

Deep copy: Recursively copies all objects (copy.deepcopy()). , changes in the original object are not reflected in the copy.

Q. How slicing works?

- **Syntax:** list[start:end:step].
- **Function:** Returns a new list without altering original.

Q. Reverse a list

- **Using slicing:** lst[::-1].
- **Using function:** reversed(lst) returns iterator.

Q. Frozenset in Python

- **Definition:** Immutable version of set.
- **Use:** Can be used as keys in dictionaries or elements in sets.
- It is hashable. We cannot use and update methods from set on frozenset

Q. Difference between is and ==

- **is:** Checks **identity** (same memory object).
- **==:** Checks **equality** (same value).

Q. What Is the Difference Between Del and Remove() on Lists?

del

- del removes all elements of a list within a given range remove()
- Syntax: del list[start:end]

remove

- remove() removes the first occurrence of a particular character
- Syntax: list.remove(element)

Q. Sort dict by value

- **Method:** sorted(d.items(), key=lambda x: x[1]).
- **Returns:** List of tuples sorted by values.

Q. Explain heapq

- **heapq module:** Implements a **min-heap queue** for priority-based operations.
- **Example:** heapq.heappush(), heapq.heappop().

Q. Implement stack & queue

- **Stack:** Use list.append() and list.pop() (LIFO).
- **Queue:** Use collections.deque() with append() and popleft() (FIFO).

Q. Explain Append() And Extend() Property Of List

Append():

- append() adds its argument as a single element to the end of a list.
- The length of the list itself will increase by one.

```
list1.append('AB')
print(list1)
Output: [1, 2, 3, 'AB']
```

Extend():

- extend() iterates over its argument adding each element to the list, extending the list.
- The length of the list will increase by however many elements were in the iterable argument.

```
list1.append('AB')
print(list1)
Output: [1, 2, 3, 'A', 'B']
```

Q. How Do You Use the Split() Function in Python?

The split() function splits a string into several strings based on a specific delimiter.

Syntax - string.split(delimiter, max)

Where: the delimiter is the character based on which the string is split. By default it is space.

max is the maximum number of splits

Example –

```
>>var="Red,Blue,Green,Orange"
>>lst=var.split(",",2)
>>print(lst)
```

Output: ['Red','Blue','Green, Orange']

Here, we have a variable var whose values are to be split with commas. Note that '2' indicates that only the first two values will be split.

Q, What is the improvement in enumerate() function of Python?

In Python, enumerate() function is an improvement over regular iteration. The enumerate() function returns an iterator that gives (0, item[0]).

E.g. >>> thelist=['a','b']

```
>>> for i,j in enumerate(thelist):
        print i,j
```
Output- 0 a 1 b

Q, What is None in Python?

None is a reserved keyword used in Python for null objects. It is neither a null value nor a null pointer. It is an actual object in Python. But there is only one instance of None in a Python environment. We can use None as a default argument in a function. During comparison we have to use "is" operator instead of "==" for None

Q, What is the use of zip() function in Python?

In Python, we have a built-in function zip() that can be used to aggregate all the Iterable objects of an Iterator. We can use it to aggregate Iterable objects from two iterators as well.

E.g. list_1 = ['a', 'b', 'c'] list_2 = ['1', '2', '3']

```
for a, b in zip(list_1, list_2):
    print a, b
```
Output: a1 b2 c3

By using zip() function we can divide our input data from different sources into fixed number of sets

Q, Functions Of List

Concept / Function	Explanation	Example	Output / Notes
Definition	List stores multiple items in a single variable. Ordered, mutable, allows duplicates.	L = [1, 2, 3, 4]	[1, 2, 3, 4]
Heterogeneous List	Lists can contain different data types.	L = [1, "Hi", 3.5, True]	[1, 'Hi', 3.5, True]
Nested List	A list inside another list.	L = [1, [2, 3], 4]	Access nested: L[1][0] → 2
List from String	Convert string into list of characters.	list("hello")	['h','e','l','l','o']
List Comprehension (Basic)	Short syntax to create list.	[i for i in range(5)]	[0,1,2,3,4]
List Comprehension (Condition)	Add conditions to comprehension.	[i for i in range(10) if i%2==0]	[0,2,4,6,8]
Nested List Comprehension	Generate tables or combinations.	[[i*j for j in range(1,4)] for i in range(1,4)]	[[1,2,3],[2,4,6],[3,6,9]]
Access by Index	Retrieve item by position.	L = [10,20,30]; L[1]	20
Negative Indexing	Access from end of list.	L[-1]	Last element
Slicing	Extract sublist with L[start:end:step].	L[1:4], L[::-1]	[20,30,40], reversed list
append(x)	Adds one element to the end.	L=[1,2]; L.append(3)	[1,2,3]

extend(iterable)	Adds multiple elements.	L=[1,2]; L.extend([3,4])	[1,2,3,4]
insert(pos, x)	Insert element at specific index.	L=[1,2,3]; L.insert(1,100)	[1,100,2,3]
remove(x)	Removes first occurrence of value.	L=[1,2,3,2]; L.remove(2)	[1,3,2]
pop() / pop(i)	Removes and returns last or specific element.	L.pop() / L.pop(1)	Returns value; modifies list.
del L[i] / del L[a:b]	Deletes element or slice.	del L[0]	Removes first item.
clear()	Removes all elements.	L.clear()	[]
update by index	Replace item at a position.	L[1]=99	[1,99,3]
update by slice	Replace multiple items.	L[1:3]=[100,200]	[1,100,200,4]
concatenation (+)	Joins two lists.	[1,2]+[3,4]	[1,2,3,4]
repetition (*)	Repeats list elements.	[1,2]*3	[1,2,1,2,1,2]
Membership	Test if element exists.	5 in [1,2,3]	False
Iteration	Loop through elements.	for i in L: print(i)	Prints all elements
len(L)	Returns number of elements.	len([1,2,3])	3
min(L), max(L)	Finds smallest/largest element.	min([3,1,2])	1
sum(L)	Sums numeric elements.	sum([1,2,3])	6
sorted(L)	Returns new sorted list.	sorted([3,1,2])	[1,2,3]
sort()	Sorts list in place.	L.sort(reverse=True)	[3,2,1]
reverse()	Reverses list in place.	L.reverse()	[3,2,1]
count(x)	Counts occurrences of value.	[1,1,2,3].count(1)	2
index(x)	Returns index of first match.	[10,20,30].index(20)	1
copy()	Creates shallow copy.	a=[1,2]; b=a.copy()	id(a) != id(b)
Mutability	Lists can be modified after creation.	L[0]=99	Changes first element.
zip()	Combine multiple lists element-wise.	list(zip([1,2],[3,4]))	[(1,3),(2,4)]
unzip	Separate zipped list.	list(zip(*[(1,3),(2,4)]))	[(1,2),(3,4)]
Nested Loops Example	Use loops with lists.	for i in [1,2]: for j in [3,4]: print(i,j)	(1,3),(1,4),(2,3),(2,4)
Cartesian Product	Multiply combinations of two lists.	[i*j for i in [1,2] for j in [3,4]]	[3,4,6,8]
filter with list comp	Filter based on condition.	[x for x in [1,2,3,4,5] if x%2==0]	[2,4]
Nested Operations	Operations inside comprehensions.	[i**2 for i in range(5)]	[0,1,4,9,16]
Shallow Copy Behavior	Changes in one don't affect the other.	a=[1,2]; b=a.copy(); a[0]=100	a=[100,2]; b=[1,2]

PYTHON INTERVIEW QUESTIONS BY VANSHIKA MISHRA

List with Functions	Can store callable objects.	[1, print, type, input]	Can call: L[1]("Hi") → Hi
List as Matrix	2D list representation.	M=[[1,2,3],[4,5,6]]	M[1][2] → 6
Access by index	`a = [10,20,30]; a[0]`	`10`	Indexing starts at **0**.
Access last element	`a[-1]`	`30`	Negative index starts from end (`-1` is last).
Slice: start:end	`a[0:2]`	`[10,20]`	End index is **excluded**.
Slice: start only	`a[1:]`	`[20,30]`	Goes from index 1 to end.
Slice: end only	`a[:2]`	`[10,20]`	From start to index 2 (excluded).
Slice with step	`a[::2]`	`[10,30]`	Takes every 2nd element.
Reverse list	`a[::-1]`	`[30,20,10]`	Step `-1` reverses the list.
Update element	`a[1] = 99`	`[10,99,30]`	Replace element at index 1.
Delete element	`del a[0]`	`[20,30]`	Removes index 0 element.
Check index exists (safe)	`a[5] if 5 < len(a) else "No"`	`"No"`	Avoids IndexError.

Q, Functions Of Tuple

Concept / Function	Explanation	Example (Code)	Output / Note
Definition	Tuple is an **immutable**, **ordered** collection that allows **duplicates**.	t = (1, 2, 3)	Once created, elements **cannot be modified**.
Characteristics	Ordered, Unchangeable, Allows duplicates	—	—
Empty Tuple	Tuple with no elements.	t1 = ()	()
Single Element Tuple	Needs a comma to differentiate from normal value.	t2 = ('hello',)	('hello',)
Homogeneous Tuple	All elements of same type.	t3 = (1, 2, 3, 4)	(1, 2, 3, 4)
Heterogeneous Tuple	Elements of different types.	t4 = (1, 2.5, True, [1,2,3])	(1, 2.5, True, [1, 2, 3])
Nested Tuple	Tuple inside another tuple.	t5 = (1, 2, 3, (4,5))	(1, 2, 3, (4,5))
Type Conversion	Convert iterable to tuple.	tuple('hello')	('h','e','l','l','o')
Access by Index	Retrieve element using positive/negative index.	t3[0], t3[-1]	1, 4
Nested Access	Access inner tuple element.	t5[-1][0]	4
Slicing	Extract sub-tuple.	t = (1,2,3,4,5); t[-1:-4:-1]	(5,4,3)
Immutability	Elements can't be changed.	t3[0] = 100	❌ TypeError
Adding Items	Not allowed (immutable).	—	❌ TypeError
Deleting Items	Only full tuple deletion possible.	del t3	Tuple deleted; variable undefined.

del t[i]	Deleting element not allowed.	del t5[-1]	❌ TypeError
Concatenation (+)	Combine two tuples.	(1,2)+(3,4)	(1,2,3,4)
Repetition (*)	Repeat elements multiple times.	(1,2)*3	(1,2,1,2,1,2)
Membership Test	Check if value exists.	1 in (1,2,3)	True
Iteration	Loop through items.	for i in (1,2,3): print(i)	Prints 1, 2, 3
len()	Returns tuple length.	len((1,2,3,4))	4
sum()	Adds numeric elements.	sum((1,2,3,4))	10
min(), max()	Returns smallest/largest element.	min((1,2,3,4))	1
sorted()	Returns new sorted list (not tuple).	sorted((4,3,2,1))	[1,2,3,4]
count(x)	Counts number of occurrences.	(1,2,2,3).count(2)	2
index(x)	Returns first index of item.	(1,2,3).index(2)	1
Difference: List vs Tuple	Lists are mutable, slower, larger in memory; Tuples are immutable, faster, smaller.	—	Tuple faster and memory-efficient.
Speed Test	Tuple iteration is faster.	for i in t: i*5	Tuple took less time than list.
Memory Usage	Tuples take less memory.	sys.getsizeof(list(range(1000)))	List > Tuple size
Mutability Difference	Lists share reference; tuples don't.	a=[1,2]; b=a; a.append(3)	Both updated.
	For tuples, new object created on update.	a=(1,2); b=a; a=a+(3,)	a=(1,2,3), b=(1,2)
Tuple Unpacking	Assign multiple variables from tuple.	a,b,c=(1,2,3)	a=1, b=2, c=3
Unpacking Error	Fewer/more vars cause error.	a,b=(1,2,3)	❌ ValueError
Variable Swap	Swap without temp var.	a,b=b,a	a and b values swapped.
Extended Unpacking	Use * to gather remaining items.	a,b,*others=(1,2,3,4)	a=1, b=2, others=[3,4]
Zip Tuples	Combine tuples element-wise.	a=(1,2,3); b=(4,5,6); tuple(zip(a,b))	((1,4),(2,5),(3,6))

Q, Functions Of Dictionary

Concept / Function	Explanation	Example & Output / Notes
Definition	Dictionary = Key–Value pairs (like Map or Associative Array).	d = {'name':'vanshika','age':33}

PYTHON INTERVIEW QUESTIONS BY VANSHIKA MISHRA

Characteristics	✅ Mutable ❌ Indexing (order not fixed) ❌ Duplicate keys ❌ Mutable keys not allowed	{'a':1,'a':2} → keeps last → {'a':2}
Creating Dictionaries	Use {} or dict()	d1={'name':'vanshika','gender':'female'} d2=dict([('name',vinay'),('age',42)])
Mixed Keys	Keys can be of any immutable type.	{(1,2,3):1,'hello':'world'} ✅
Nested (2D) Dictionary	Supports hierarchy like JSON.	s={'name':'vanshika','subjects':{'dsa':50,'maths':67}}
Mutable Keys Error	Lists can't be keys.	{[1,2]:3} ❌ → TypeError: unhashable type
Accessing Values	Access by key using [] or .get()	my_dict['age'] → 26 my_dict.get('age') → 26
Access Nested Data	Use multiple key layers.	s['subjects']['maths'] → 67
Adding / Updating Items	Add or modify key-value pairs.	d['gender']='male' s['subjects']['ds']=75
Editing Values	Change existing data.	s['subjects']['dsa']=80
pop(key)	Removes key, returns its value.	d.pop('age')
popitem()	Removes last inserted key-value pair.	d.popitem()
del dict[key]	Deletes specific item.	del d['name']
clear()	Removes all items.	d.clear() → {}
Membership Test	Checks if key exists.	'name' in d → True
Iteration	Loops over keys (default), or items.	for k in d: print(k, d[k])
len(d)	Number of key-value pairs.	len({'a':1,'b':2}) → 2
sorted(d)	Returns sorted list of keys.	sorted({'b':2,'a':1}) → ['a','b']
max(d) / min(d)	Based on key order (lexical).	max({'a':1,'b':2}) → 'b'
.items()	Returns list-like view of (key, value) pairs.	d.items() → dict_items([('name','vanshika')])
.keys()	Returns all keys.	d.keys() → dict_keys(['name','age'])
.values()	Returns all values.	d.values() → dict_values(['vanshika',33])
.update(other_dict)	Merges dictionaries (overwrites duplicates).	{1:2,3:4}.update({3:7,5:8}) → {1:2,3:7,5:8}
Dictionary Comprehension	Create dictionaries dynamically.	{i:i**2 for i in range(1,6)} → {1:1,2:4,3:9,4:16,5:25}
Modify Existing Dict via Comprehension	Perform operations on existing items.	{k:v*0.62 for (k,v) in {'delhi':1000}.items()} → {'delhi':620.0}
Using zip()	Combine two lists into a dictionary.	python days=['Mon','Tue']; temp=[32,33]; dict(zip(days,temp)) → {'Mon':32,'Tue':33}
Comprehension with Condition	Filter based on values.	{k:v for k,v in {'phone':10,'laptop':0}.items() if v>0} → {'phone':10}
Nested Comprehension	Dictionary inside dictionary (e.g., multiplication tables).	python {i:{j:i*j for j in range(1,4)} for i in range(2,5)} → {2:{1:2,2:4,3:6},3:{1:3,...}}

Q, Functions Of Set

Concept / Function	Explanation	Example & Output / Notes
Definition	Unordered collection of unique, immutable items; set itself is mutable.	s = {1,2,3}
Characteristics	✅ Unordered ✅ Mutable ✅ No duplicates ❌ Can't contain mutable elements (like lists).	{1,2,3} valid, {1,[2,3]} ❌ → TypeError
Creating Sets	Use {} or set() constructor.	s=set() → set() s1={1,2,3} s2=set([1,2,3])
Duplicate Removal	Duplicates auto-removed.	{1,1,2,2,3} → {1,2,3}
Heterogeneous Sets	Can contain mixed immutable types.	{1,'hi',3.14,(1,2)}
Equality Check	Order doesn't matter in sets.	{1,2,3} == {3,2,1} → True
Accessing Items	Not indexable or sliceable.	s1[0] ❌ → TypeError
Editing Items	Direct item assignment not allowed.	s1[0]=10 ❌ → TypeError
Adding Items	Add single or multiple elements.	s.add(5) → adds one s.update([6,7]) → adds many
Deleting Items	Various deletion methods:	
- del	Deletes entire set.	del s
- discard(x)	Removes element if present; no error if absent.	s.discard(5)
- remove(x)	Removes element; error if not present.	s.remove(5)
- pop()	Removes random element.	s.pop()
- clear()	Removes all items.	s.clear() → set()
Union (`	or.union()`)	Combines all unique elements.
Intersection (& or .intersection())	Common elements only.	{1,2,3} & {2,3,4} → {2,3}
Difference (- or .difference())	Elements in first not in second.	{1,2,3}-{2,3,4} → {1}
Symmetric Difference (^ or .symmetric_difference())	Elements not common to both.	{1,2,3}^{3,4} → {1,2,4}
Membership Test	Check if element present.	3 in s1, 5 not in s1
Iteration	Loop through set items.	for i in s1: print(i)
Built-in Functions	Common aggregation or inspection tools.	
len(s)	Count of elements.	{1,2,3} → 3
sum(s)	Sum of numeric elements.	{1,2,3} → 6
min(s) / max(s)	Smallest / largest item.	{3,1,2} → min=1, max=3
sorted(s, reverse=True)	Returns sorted list of elements.	{3,1,2} → [3,2,1]
Set Update Methods	Modify first set directly.	
.update(s2)	Adds all from s2 into s1.	s1={1,2}; s2={2,3}; s1.update(s2) → {1,2,3}

.intersection_update(s2)	Keeps only common elements.	{1,2,3}.intersection_update({2,3,4}) → {2,3}
.difference_update(s2)	Removes common elements.	{1,2,3}.difference_update({2,3}) → {1}
.symmetric_difference_update(s2)	Keeps uncommon elements only.	{1,2,3}.symmetric_difference_update({3,4}) → {1,2,4}
Relation Checks	Compare sets structurally.	
.isdisjoint(s2)	True if no common elements.	{1,2}.isdisjoint({3,4}) → True
.issubset(s2)	True if all elements of s1 in s2.	{1,2}.issubset({1,2,3}) → True
.issuperset(s2)	True if s1 contains all elements of s2.	{1,2,3}.issuperset({2,3}) → True
Copying	Create shallow copy.	s2 = s1.copy()
Frozenset	Immutable set version — allows nesting.	
Creation	fs = frozenset([1,2,3])	
Allowed Operations	All **read-only** operations (union, intersection, etc.).	`fs1
Disallowed Operations	Any modification (add, remove, etc.).	fs.add(4) ❌
Usage	Useful for 2D sets / dictionary keys.	fs=frozenset([1,2,frozenset([3,4])])
Set Comprehension	Build sets dynamically with conditions.	{i**2 for i in range(1,11) if i>5} → {36,49,64,81,100}

Q, Explain string slicing

Topic	Concept / Function	Example / Output / Notes
Basics	Strings are **sequence of Unicode characters**.	'hello', "hello", '''hello''', """hello""", str('hello') → 'hello'
Accessing Strings	**Positive Indexing:** starts from 0	s='hello world'; s[4] → 'o'
	Negative Indexing: starts from -1 (end)	s[-3] → 'r'
Slicing	s[start:stop:step] extracts substring	s[6:0:-2] → 'wol'
	Reverse a string	s[::-1] → 'dlrow olleh'
	Partial reverse	s[-1:-6:-1] → 'dlrow'
Immutability	Strings **cannot** be changed in place	s[0]='H' → ❌ TypeError
Deleting Strings	Entire string can be deleted, not parts	del s → removes variable name
Arithmetic Ops	Concatenation	'delhi' + ' ' + 'mumbai' → 'delhi mumbai'
	Repetition	'delhi'*3 → 'delhidelhidelhi'
Relational Ops	Lexicographic comparison	'mumbai' > 'pune' → False; 'Pune' > 'pune' → False
Logical Ops	and, or, not on strings	'hello' and 'world' → 'world'; 'hello' or '' → 'hello'; not 'hello' → False
Loops	Iterate through characters	for i in 'hello': print(i) prints each letter
Membership	Check if substring exists	'D' in 'delhi' → False
len()	Returns length	len('hello world') → 11
max()	Returns highest Unicode char	max('hello') → 'o'

min()	Returns lowest Unicode char	min('hello') → 'e'
sorted()	Returns sorted list of chars	sorted('hello', reverse=True) → ['o','l','l','h','e']
capitalize()	Capitalizes 1st letter	'hello world'.capitalize() → 'Hello world'
title()	Capitalizes each word	'hi there'.title() → 'Hi There'
upper()	Converts to uppercase	'hello'.upper() → 'HELLO'
lower()	Converts to lowercase	'HELLO'.lower() → 'hello'
swapcase()	Swaps upper/lower case	'HeLLo'.swapcase() → 'hEllO'
count()	Count occurrences of substring	'banana'.count('a') → 3
find()	Returns first index of substring or -1	'apple'.find('p') → 1; 'apple'.find('z') → -1
index()	Same as find() but errors if not found	'apple'.index('l') → 3
startswith()	Checks prefix	'hello'.startswith('he') → True
endswith()	Checks suffix	'hello'.endswith('lo') → True
format()	String formatting with placeholders	'My name is {1}, I am {0}'.format('female','Vanshika') → 'My name is Vanshika, I am female'
f-string	Modern formatting	f'My age is {21}' → 'My age is 21'
isalnum()	True if all alphanumeric	'abc123'.isalnum() → True
isalpha()	True if all alphabets	'abc'.isalpha() → True
isdigit()	True if all digits	'123'.isdigit() → True
istitle()	True if title-cased	'Hello World'.istitle() → True
isupper()	True if all uppercase	'HELLO'.isupper() → True
islower()	True if all lowercase	'hello'.islower() → True
isspace()	True if only spaces	' '.isspace() → True
split()	Splits string into list by spaces	'hi my name'.split() → ['hi','my','name']
join()	Joins list into string	" ".join(['hi','my','name']) → 'hi my name'
replace()	Replaces substring	'hello world'.replace('world','Python') → 'hello Python'
strip()	Removes leading/trailing spaces	' test '.strip() → 'test'

Functions

Q. Function vs Method

- **Function:** Independent block of code defined with def.
- **Method:** Function associated with an object and called via that object.

Q. *args and kwargs

- ***args:** Handles variable number of positional arguments as a tuple.
- ****kwargs:** Handles variable keyword arguments as a dictionary.

***args Python Example:**

```
def sum(*args):
        total = 0
        for a in args:
                total = total + a
        print(total)

sum(1,2,3,4,5)
```

Output:
15

****Kwargs Python Example**

```
def show(**kwargs):
        print(kwargs)

show(A=1,B=2,C=3)
```

Output:
{'A': 1, 'B': 2, 'C': 3}

Q. Recursion example

- **Recursion**: Function calling itself until base condition is met.
- Example:
- def fact(n): return 1 if n==0 else n*fact(n-1)

Q. What are decorators?

- **Decorators**: Functions that wrap another function to modify or extend its behavior.
- **Example:** @decorator_name above a function.
- All of this without altering the source code of the original function that you passed in

Example:-

```
def double(func):
    def wrap(a, b):
        return func(a, b) * 2
    return wrap

@double
def add(a, b):
    return a + b

print(add(3, 5))  # Output: 16
```

The below simple program is equivalent to the above decorator example. Here we are changing the function call.

```
def decorateFun(func):
    def sumOfSquare(x, y):
        return func(x**2, y**2)
    return sumOfSquare

def addTwoNumbers(a, b):
    c = a+b
    return c

obj=decorateFun(addTwoNumbers)
c=obj(4,5)
print("Addition of square of two numbers=", c)
#Addition of square of two numbers=41
```

Q. What are generators?

- **Generators**: Functions using yield to return values one at a time.
- **Benefit:** Memory-efficient, used for large data streams.
- Generators are iterators which can execute only once.
- Every generator is an iterator.
- Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop

EXAMPLE:

```
def sqr(n):
```

```
for i in range(1, n+1):
        yield i*i
a = sqr(3)
print(next(a))
print(next(a))
print(next(a))
```

Output: 1 4 9

Q. Iterator vs generator

Feature	Iterator	Generator
Creation	Created using iter() on an iterable or by defining a class with __iter__() and __next__() methods	Created using a **function** that contains the yield keyword
Memory Usage	Stores **all elements** in memory	Produces **one value at a time** → very memory efficient
Execution Type	Fetches existing elements from an iterable	Generates elements **on the fly (lazy evaluation)**
Ease of Implementation	Requires more code (custom class implementation)	Very easy — just use yield
Use of next()	Used to get next element from the iterable	Used to get next value generated by yield
Example	it = iter([1,2,3])	def gen(): yield 1; yield 2; yield 3
When to Use	When data already exists and you need to iterate over it	When generating data dynamically or working with large datasets

Q, Explain global variables and local variables in Python.

Local Variables: A local variable is any variable declared within a function. This variable exists only in local space, not in global space.

```
x = 10  # Global variable

def my_function():
    x = 5  # Local variable
    print("Inside function:", x)

my_function()
print("Outside function:", x)

# Output:
# Inside function: 5
# Outside function: 10
```

Global Variables: Global variables are variables declared outside of a function or in a global space. Any function in the program can access these variables.

```
x = 10

def modify():
    global x
    x = 20

modify()
print(x)  # Output: 20
```

Q, Difference between Global and Local variable

Local Variable	Global Variable
It is declared inside a function.	It is declared outside the function.
If it is not initialized, a garbage value is stored.	If it is not initialized, zero is stored as default.
It is created when the function starts execution and lost when the function terminates.	It is created before the program's global execution starts and lost when the program terminates.
Data sharing is not possible as data of the local variable can be accessed by only one function.	Data sharing is possible as multiple functions can access the same global variable.
Parameter passing is required for local variables to access the value in another function.	Parameter passing is not necessary for a global variable as it is visible throughout the program.
When the value of the local variable is modified in one function, the changes are not visible in another function.	When the value of the global variable is modified in one function, changes are visible in the rest of the program.
Local variables can be accessed with the help of statements inside a function in which they are declared.	You can access global variables by any statement in the program.
It is stored on the stack unless specified.	It is stored on a fixed location decided by the compiler.

Q. With statement

- Purpose: Manages resources automatically (e.g., closing files).
- Example:
  with open('file.txt') as f:
    data = f.read()

Q. What are closures?

- Closure: Function capturing variables from its enclosing scope.
- Use: Maintain state without global variables.

Q. Lambda functions

- Lambda: Small anonymous function defined inline.
- Syntax: lambda x: x*2.

Q.How to use lambda with map, reduce, filter?

Map

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)
# Output: [1, 4, 9, 16, 25]
```

Reduce

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
sum_all = reduce(lambda a, b: a + b, numbers)
print(sum_all)
# Output: 15
```

Filter

```
numbers = [10, 15, 21, 30, 45]
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)
# Output: [15, 21, 45]
```

Q. @staticmethod vs @classmethod

- **@staticmethod:** No access to class or instance, used for utility functions.
- **@classmethod:** Takes class (cls) as first argument, modifies class state.

Q. Magic methods

- **Magic methods:** Special methods with double underscores (dunder methods).
- These methods have special meanings and purposes, allowing you to customize the behavior of your classes and make them work seamlessly with Python's built in functions and operators.
- _init_(self, ...): Initializes object attributes when an object is created.
- _str_(self): Provides a human-readable string representation of the object, used by str() and print().
- _repr_(self): Returns an unambiguous string representation of the object, helpful for debugging.
- _add_(self, other): Defines behavior for the + operator when applied to objects of the class.
- _eq_(self, other): Specifies how objects of the class are compared using the == operator.
- _len_(self): Defines the behavior of the len() function when applied to objects of the class.

Q. Monkey patching

- **Definition:** Dynamically modifying class or module behavior at runtime.
- **Example:** Reassigning a class method after definition.

Q, How To Read Multiple Values From Single Input?

By using split()

```
x = list(map(int, input("Enter a multiple value: ").split()))
 print("List of Values: ", x)
x = [int(x) for x in input("Enter multiple value: ").split()]
 print("Number of list is: ", x)
x = [int(x) for x in input("Enter multiple value: ").split(",")]
print("Number of list is: ", x)
```

Q, Difference Between Anonymous and Lambda Function

Lambda function:

❏ It can have any number of arguments but only one expression.

❏ The expression is evaluated and returned.

❑ Lambda functions can be used wherever function objects are required.
square = lambda x : x * x square(5) #25
Anonymous function:
❑ In Python, Anonymous function is a function that is defined without a name.
❑ While normal functions are defined using the def keyword, Anonymous functions are defined using the lambda keyword.
❑ Hence, anonymous functions are also called lambda functions.
print((lambda x: x*x)(5))

Q, What are first class functions?
Functions treated as objects- can be assigned to variables, passed as argument, returned from other functions. Enables- high order functions, decorators, callbacks

Object-Oriented Programming (OOP)
Q. What is OOP in Python?

- **Concept:** Organizing code into **classes and objects**.
- **Benefit:** Promotes reusability, modularity, and easier maintenance.

Q. What is a Class?

- **Definition:** Blueprint for creating objects.
- **Example:**
- class Car:
- pass

Q. What is an Object?

- **Definition:** Instance of a class containing data (attributes) and behavior (methods).
- **Example:** car1 = Car()

Q, What does an object() do?
It produces a featureless object that serves as the foundation for all classes. It also does not accept any parameters.

Q. What is __init__() Method?

- **Purpose:** Constructor used to initialize object attributes automatically.
- **Example:**
- def __init__(self, name):
- self.name = name

Q. Difference Between Class and Object

- **Class:** Template/blueprint.
- **Object:** Instance created from that blueprint.

Q. What is Method Overriding?

- **Definition:** Redefining parent class method in the child class.
- **Use:** To modify or extend parent behavior.

Q. What is Method Overloading?

- **Definition:** Same method name with different parameters (not directly supported in Python).
- **Simulated by:** Default or variable arguments.

Q. What are Class Variables and Instance Variables?

- **Class Variable:** Shared among all objects.
- **Instance Variable:** Unique to each object.

Q. What is self in Python?

- **Meaning:** Refers to the current instance of the class.
- **Use:** Access variables and methods of the class.

```
class Person:
def __init__(self, name, age):
        self.name = name
        self.age = age
def info(self):
        print(f"My name is {self.name}. I am {self.age} years old.")
c = Person("Vanshika",23)
c.info()
```

Output: My name is Vanshika. I am 23 years old.

Q, What are global, protected, and private attributes in Python?

The attributes of a class are also called variables. There are three access modifiers in Python for variables, namely

a. public – The variables declared as public are accessible everywhere, inside or outside the class.

b. private – The variables declared as private are accessible only within the current class.

c. protected – The variables declared as protected are accessible only within the current package.

Attributes are also classified as:

– Local attributes are defined within a code-block/method and can be accessed only within that code-block/method.

– Global attributes are defined outside the code-block/method and can be accessible everywhere.

Q. What is Multiple Inheritance?

- **Definition:** A class inheriting from multiple parent classes.
- **Example:**

```
class C(A, B):
    pass
```

Q, What is a metaclass in Python?

A metaclass in Python is also known as class of a class. A class defines the behavior of an instance. A metaclass defines the behavior of a class. One of the most common metaclass in Python is type. We can subclass type to create our own metaclass. We can use metaclass as a class-factory to create different types of classes

Q. Method Resolution Order (MRO)

- **MRO:** Determines the order in which base classes are searched when invoking a method.
- **Uses:** ClassName.mro() or help(ClassName).
- In Python, the MRO is from bottom to top and left to right.

- This means that, first, the method is searched in the class of the object. If it's not found, it is searched in the immediate super class.
- In the case of multiple super classes, it is searched left to right, in the order by which was declared by the developer.
- For example: def class C(B,A): In this case, the MRO would be C -> B -> A. Since B was mentioned first in class declaration, it will be searched first while resolving a method

Example 3:

```
class A:
 def method(self):
  print("A.method() called")

class B:
 def method(self):
  print("B.method() called")

class C(A, B):
 pass

class D(C, B):
 pass

d = D()
d.method()
```

The MRO for this can be a bit tricky.
The immediate superclass for D is C, so if the method is not found in D, it is searched for in C.
However, if it is not found in C, then you have to decide if you should check A (declared first in the list of C's super classes) or check B (declared in D's list of super classes after C).
In Python 3 onwards, this is resolved as first checking A.

So, the MRO becomes:

D -> C -> A -> B

Q, Does Python Support Multiple Inheritance. (Diamond Problem)

Yes, Python Supports Multiple Inheritance.

What Is Diamond Problem? What Java does not allow is multiple inheritance where one class can inherit properties from more than one class. It is known as the diamond problem.

Multiple Inheritance In Python:

```
class A:
        def abc(self):
                print("a")

class B(A):
        def abc(self):
                print("b")

class C(A):
        def abc(self):
                print("c")

class D(B,C):
        pass

d = D()
d.abc()
```

Output:
b

Q. What is init?

- **init:** Constructor method automatically called when object is created.
- **Use:** Initialize attributes of the class.
- The _init_ method is a special method that serves as a constructor for Python objects.
- It is automatically called when an object is created from a class.
- The self parameter in the _init_ method refers to the instance being created and al lows you to set the initial state of object attributes.
- Additional parameters in the _init_ method can be used to pass values when creating objects, allowing you to customize the initial state.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years
old."

person1 = Person("Alice", 30)
print(person1.greet())
# Output: "Hello, my name is Alice and I am 30 years old."
```

Q. Inheritance in Python and its types

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class

```
class A:
  def display(self):
    print("A Display")

class B(A):
  def show(self):
    print("B Show")
d = B()
d.show()
d.display()


Output:
B Show
A Display
```

- Single inheritance: The members of a single superclass are acquired by a derived class A→ B
- Multiple inheritance: More than one base class is inherited by a derived class. A→ C B→ C
- Muti-level inheritance: D1 is a derived class inherited from base1 while D2 is inherited from base2. A→B→C
- Hierarchical Inheritance: You can inherit any number of child classes from a single base class. A→B A→C

Q. Multiple vs Multilevel Inheritance

- **Multiple:** Class inherits from multiple parents (class C(A, B)).
- **Multilevel:** One class inherits from another forming a chain (A → B → C).

Q, What Is _a, __a, __a__ in Python?

_a

❏ Python doesn't have real private methods, so one underline in the beginning of a variable/function/method name means it's a private variable/function/method and It is for internal use only

❏ We also call it weak Private

__a

❏ Leading double underscore tell python interpreter to rewrite name in order to avoid conflict in subclass.

❏ Interpreter changes variable name with class extension and that feature known as the Mangling.

❏ In Mangling python interpreter modify variable name with __.
❏ So Multiple time It use as the Private member because another class can not access that variable directly.
❏ Main purpose for __ is to use variable/method in class only If you want to use it outside of the class you can make public api.
__a__
❏ Name with start with __ and ends with same considers special methods in Python.
❏ Python provide this methods to use it as the operator overloading depending on the user.
❏ Python provides this convention to differentiate between the user defined function with the module's function

Q. What is Polymorphism?

- **Definition:** Same method name behaves differently based on object type.
- **Example:** len() works on list, tuple, or string.
- For example, if the parent class has a method named ABC, the child class can likewise have a method named ABC with its own parameters and variables. Python makes polymorphism possible.

Q. Abstraction

- **Definition:** Hides complex implementation, exposes only necessary details.
- **Implemented via:** Abstract classes (from abc module).
- A class with one or more **abstract methods** is called an **abstract class**.
- **Abstract methods** have no implementation — only a declaration.
- **Subclasses** inherit the abstract class and **define** its abstract methods.
- Acts as a **blueprint** for other classes.
- Useful for **large programs** to ensure a consistent interface across components.
- Python uses the **abc module** to implement abstraction.
- **Syntax example:**

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
        def area(self):
                pass
```

Q. Encapsulation

- **Definition:** Binding data and methods together.
- **Use:** Restricts direct access using private (__var) attributes.
- Achieved using **private (__var)** and **protected (_var)** access modifiers.
- Promotes **data hiding** and **modularity** in programs.
- Access or modify data using **getter and setter methods**.

```
Example
class Student:
    def __init__(self, name, grade, percentage):
        self.name = name
        self.grade = grade
        self.__percentage = percentage  # Private attribute
(hidden)
    def get_percentage(self):  # Public method to access the
private attribute
        return self.__percentage

# Creating a student object
student1 = Student("Madhav", 10, 98)

# Accessing the private attribute using the public method
print(f"{student1.name}'s percentage is
{student1.get_percentage()}%.")
print(student1.__percentage) # error
```

Q, How will you check in Python, if a class is subclass of another class Python provides a useful method issubclass(a,b) to check whether class a is a subclass of b. E.g. int is not a subclass of long
>>> issubclass(int,long) False bool is a subclass of int
>>> issubclass(bool,int) True

Q, Class Method vs Static Method

Class Method	Static Method
The class method takes cls (class) as first argument.	The static method does not take any specific parameter.
Class method can access and modify the class state.	Static Method cannot access or modify the class state.
The class method takes the class as parameter to know about the state of that class.	Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters.
@classmethod decorator is used here.	@staticmethod decorator is used here.

Exception Handling

Q. Errors vs Exceptions

- **Errors:** Syntax or logical issues that stop execution (e.g., SyntaxError).
- **Exceptions:** Runtime events handled using try-except.

Exception	Description
ZeroDivisionError	Raised when dividing by zero
ValueError	Raised when a function gets an incorrect argument
TypeError	Raised when an operation is performed on an invalid type
IndexError	Raised when trying to access an out-of-range index in a list
KeyError	Raised when a dictionary key is not found
FileNotFoundError	Raised when attempting to open a non-existent file

Q. Try-Except-Finally

- **try:** Block of code that might raise an error.
- **except:** Catches and handles the error.
- **finally:** Executes regardless of whether an error occurred.

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("Closing file...")
    file.close()  # Ensuring the file is closed
```

Q. Raise vs Assert

- **raise:** Manually trigger an exception using raise Exception("msg").
- **assert:** Test condition; raises AssertionError if false.

Q. Custom Exceptions

- **Definition:** User-defined error classes by inheriting from Exception.
- **Example:**
  class MyError(Exception): pass

Q. Exception Chaining

- **Concept:** Raise a new exception while preserving the original cause.
- **Syntax:** raise NewError("msg") from OriginalError.

Q. Catch Multiple Exceptions

- **Syntax:** except (TypeError, ValueError):.
- **Use:** Handles multiple error types in one block.

Q. Use of else in Exception Handling

- **else block:** Runs only if **no exception** occurs in try.
- **Example:**
- try: ...
- except: ...
- else: print("No Error")

Q. If Exception Not Handled

- **Outcome:** Program stops abruptly and shows a traceback error message.

Q. Pass Continue Break

- Continue:- When a specified condition is met, the control is moved to the beginning of the loop, allowing some parts of the loop to be transferred.
- Break:- When a condition is met, the loop is terminated and control is passed to the next statement.
- Pass:- When you need a piece of code syntactically but don't want to execute it, use this. This is a null operation.

Q. What is Exception Handling?

- **Definition:** Process of handling runtime errors to prevent program crash.
- **Use:** Ensures smooth program execution.

Q. Keywords Used in Exception Handling

- **Main Keywords:** try, except, else, finally, raise.
- **Purpose:** Used to detect and manage errors gracefully.

Q. Syntax of Try-Except Block

- **Example:**
- try:
- x = 1/0
- except ZeroDivisionError:
- print("Cannot divide by zero")

Q. What is else in Exception Handling?

- **Use:** Executes code if **no exception** occurs in the try block.

Q95. What is finally Block?

- **Use:** Executes code **always**, whether exception occurs or not.
- **Example:** Close files or release resources.

Q. Raising Exceptions

- **Keyword:** raise
- **Example:**
- raise ValueError("Invalid input")

Q. Catching Multiple Exceptions

- **Syntax:**
- except (ValueError, TypeError):
- print("Error occurred")
- **Use:** Handles multiple error types in one block.

Q. Nested Try Blocks

- **Definition:** Try block inside another try block.
- **Use:** Handle specific exceptions separately.

Q. Custom Exceptions

- **Created by:** Defining a new class inheriting from Exception.
- **Example:**
- class MyError(Exception):
- pass

Q. Difference Between Error and Exception

- **Error:** Occurs at compile time and cannot be handled (e.g., syntax error).
- **Exception:** Occurs at runtime and can be handled using try-except.

Modules & Packages

Q. Module vs Package

- **Module:** A single .py file containing code (functions, classes).
- **Package:** Collection of modules in a directory with __init__.py.

Q. Role of __init__.py

- **Marks directory as a package** so it can be imported.
- Can also execute initialization code for the package.

Q. Install Packages

- **Command:** pip install package-name.
- **Use:** Installs external libraries from the Python Package Index (PyPI).

Q. sys Module Use

- **Purpose:** Provides access to system-specific parameters and functions.
- **Examples:** sys.argv (arguments), sys.path (module search path).

Q. Virtual Environments

- **Definition:** Isolated environments for separate project dependencies.
- **Created by:** python -m venv env_name.

Q. Import Dynamically

- **Use:** Import module at runtime.
- **Method:** importlib.import_module('module_name').

Q. What is pip?

- **pip:** Python's package manager for installing, upgrading, or removing packages.
- **Example:** pip install numpy.

Q. Import Mechanism

- **Process:** Searches sys.path → loads module → caches it in sys.modules.
- **Ensures:** Module is loaded only once per session.

Q. Absolute vs Relative Imports

- **Absolute:** Full path from project root (from package.module import func).
- **Relative:** Based on current module (from .submodule import func).

Q. dir() Function

- **Purpose:** Lists attributes and methods of an object or module.
- **Example:** dir(math) shows all functions in math module.

Q. What is a Module in Python?

- **Definition:** A file containing Python code (functions, variables, classes).
- **Use:** Reusability and better organization.

Q. How to Create a Module?

- **Method:** Save Python code in a .py file (e.g., math_utils.py).

Q. How to Import a Module?

- **Syntax:** import module_name or from module_name import function.

Q. Built-in Modules Example

- **Examples:** math, os, random, datetime, sys.

Q. What is the __name__ Variable?

- **Use:** Distinguishes if a module is run directly or imported.
- **Example:**
- if __name__ == "__main__":
- main()

Q. What is a Package in Python?

- **Definition:** Collection of modules in a directory with an __init__.py file.
- **Use:** Organize related modules.

Q. Import from a Package

- **Syntax:**
- from package.module import func
- **Example:** from math import sqrt

Q. How to List Installed Packages?

- **Command:** pip list
- **Use:** Displays all installed Python packages.

Modern & Practical Python (Add-on Section)

Q. What are Type Hints in Python?

- **Definition:** Used to specify expected data types of variables and function arguments.
- **Example:**
- def add(x: int, y: int) -> int:
- return x + y

Q. What are Dataclasses?

- **Definition:** Introduced in Python 3.7 to reduce boilerplate in classes.
- **Example:**
- from dataclasses import dataclass
- @dataclass
- class Point:
- x: int
- y: int

Q. Explain Enumerations (Enum) in Python.

- **Definition:** Used to define symbolic names for constant values.
- **Example:**
- from enum import Enum
- class Color(Enum):
- RED = 1

- BLUE = 2

Q. What is the Walrus Operator (:=)?

- **Definition:** Allows assignment and expression in one line (Python 3.8+).
- **Example:**
- if (n := len(items)) > 5:
- print(n)

Q. Pattern Matching in Python (match-case)?

- **Definition:** Introduced in Python 3.10 for structural pattern matching.
- **Example:**
- match command:
- case "start":
- print("Go")

Q. What is the difference between async and await?

- **Definition:**
  - async defines a coroutine.
  - await pauses execution until the coroutine completes.

Q. Explain Context Managers and Custom Context Managers.

- **Definition:** Manage setup and cleanup (e.g., files).
- **Custom Example:**
- class Demo:
- def __enter__(self): ...
- def __exit__(self, *args): ...

Q. Explain Property Decorator in Python.

- **Definition:** Turns class methods into attributes for getter/setter behavior.
- **Example:**
- @property
- def area(self):
- return self.length * self.width

Q. What are F-strings?

- **Definition:** Fast, readable string formatting (Python 3.6+).

#How To Use f-string

```
name = 'Vanshika'
role = 'Python Developer'
print(f'Hello, My name is {name} and I'm {role}")
```

Output: Hello, My name is Vanshika and I'm Python Developer

#How To Use format Operator

```
name = 'Vanshika'
role = 'Python Developer'
 print(("Hello, My name is {} and I'm {}").format(name,role))
```

Output: Hello, My name is Vanshika and I'm Python Developer

Q, Explain the split(), sub(), and subn() methods of the Python "re" module.

- split (): a regex pattern is used to "separate" a string into a list
- subn(): It works similarly to sub(), returning the new string as well as the number of replacements.
- sub(): identifies all substrings that match the regex pattern and replaces them with a new string

Q, How can we retrieve data from a MySQL database in a Python script?

For MySQL database, we import MySQLdb module in our Python script. We have to first connect to a specific database by passing URL, username, password and the name of database. Once we establish the connection, we can open a cursor with cursor() function. On an open cursor, we can run fetch() function to execute queries and retrieve data from the database tables

Q, What is GIL?

❏ The Global Interpreter Lock (GIL) of Python allows only one thread to be executed at a time. It is often a hurdle, as it does not allow multi-threading in python to save time

❏ The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

❏ This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

❏ Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an "infamous" feature of Python.

❏ Basically, GIL in Python doesn't allow multi-threading which can sometimes be considered as a disadvantage.

Q, How to achieve Multiprocessing and Multithreading in Python?

Multithreading:

❏ It is a technique where multiple threads are spawned by a process to do different tasks, at about the same time, just one after the other.

❏ This gives you the illusion that the threads are running in parallel, but they are actually run in a concurrent manner.

❏ In Python, the Global Interpreter Lock (GIL) prevents the threads from running simultaneously.

Multiprocessing:

❏ It is a technique where parallelism in its truest form is achieved.

❏ Multiple processes are run across multiple CPU cores, which do not share the resources among them.

❏ Each process can have many threads running in its own memory space.

❏ In Python, each process has its own instance of Python interpreter doing the job of executing the instructions.

File Handling

Q. Open a File in Python

- **Syntax:** open(filename, mode)
- **Example:** f = open('data.txt', 'r') opens file for reading.

Q. File Modes

- **Common Modes:**

❏ Read Only ('r') : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.

❏ Read and Write ('r+') : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.

❏ Write Only ('w') : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists

❏ Write and Read ('w+') : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.

❏ Append Only ('a') : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

❏ Append and Read ('a+') : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

❏ Text mode ('t'): meaning \n characters will be translated to the host OS line endings when writing to a file, and back again when reading.

❏ Exclusive creation ('x'): File is created and opened for writing – but only if it doesn't already exist. Otherwise you get a FileExistsError. ❏ Binary mode ('b'): appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'.

Q. Read File Content

- **Methods:**
  - read() → entire file
  - readline() → one line
  - readlines() → list of lines

Q. Write to File

- **Method:** write() or writelines()
- **Example:**
- f.write("Hello World")

Q. Close a File

- **Method:** f.close()
- **Purpose:** Frees system resources and ensures data is written.

Q. Use of with Statement

- **Syntax:**
- with open('file.txt', 'r') as f:
- data = f.read()
- **Benefit:** Automatically closes file after use.

Q. File Exception Handling

- **Handled by:** try...except
- **Example:**
- try:
- open('file.txt')
- except FileNotFoundError:
- print("File not found")

Q. Tell and Seek

- **tell()** → Returns current cursor position.
- **seek(offset)** → Moves cursor to a specific position.

Q. Delete a File

- **Module:** os
- **Command:** os.remove('file.txt') deletes file from directory.

Q. Check File Exists

- **Method:** os.path.exists('file.txt')
- **Returns:** True if file exists, else False.

Q, What is pickling and un-pickling?

Pickling: In python, the pickle module accepts any Python object, transforms it into a string representation, and dumps it into a file by using the dump function. This process is known as pickling. The function used for this process is pickle.dump()

Unpickling: The process of retrieving the original python object from the stored string representation is called unpickling. The function used for this process is pickle.load()

Coding questions

1. Question: Reverse a string

```
s = 'hello'
print(s[::-1])
```

2. Question: Check if a number is prime

```
n = 7
print(all(n % i != 0 for i in range(2, int(n**0.5) + 1)))
```

3. Question: Find factorial using recursion

```
def fact(n):
if n == 0 return 1
else n * fact(n-1)
```

4. Question: Check palindrome string

```
s = 'madam'
print(s == s[::-1])
```

5. Question: Find largest number in a list

```
nums = [1, 5, 2]
print(max(nums))
```

6. Question: Swap two variables

```
a, b = 5, 10
a, b = b, a
```

7. Question: Count vowels in a string

```
count = 0
for ch in s:
    if ch.lower() in 'aeiou':
        count += 1
```

8. Question: Check Armstrong number

```
p sum = 0
temp = n
while temp > 0:
    digit = temp % 10
    sum += digit ** len(str(n))
    temp //= 10
print("Armstrong" if sum == n else "Not Armstrong")
```

9. Question: Generate Fibonacci series

```
a, b = 0, 1
for _ in range(5):
print(a)
a, b = b, a + b
```

10. Question: Find GCD of two numbers

```
import math
print(math.gcd(12, 15))
```

11. Question: Check anagram

```
s1, s2 = 'listen', 'silent'
print(sorted(s1) == sorted(s2))
```

12. Question: Reverse words in a sentence

```
s = 'hello world'
print(' '.join(s.split()[::-1]))
```

13. Question: Find duplicates in a list

```
lst = [1, 2, 3, 2, 4, 3]
print("Duplicates:", list(set([x for x in lst if lst.count(x) > 1])))
```

14. Question: Find second largest number in a list

```
nums = [1, 2, 3, 4]
print(sorted(set(nums))[-2])
```

15. Question: Check if string contains only digits

```
print('123'.isdigit())
```

16. Question: Flatten nested list

```
nested = [[1, 2], [3, 4]]
flat = [x for sub in nested for x in sub]
print(flat)
```

17. Question: Sort dictionary by values

```
d = {'a': 2, 'b': 1}
print(dict(sorted(d.items(), key=lambda x: x[1])))
```

18. Question: Count words in a string

```
s = 'this is test'
print(len(s.split()))
```

19. Question: Check leap year

```
y = 2024
print(y % 4 == 0 and (y % 100 != 0 or y %
400 == 0))
```

20. Question: Merge two dictionaries

```
d1 = {'a': 1}
d2 = {'b': 2}
d1.update(d2)
```

21. Question: Group words by first letter

```
from collections import defaultdict
words = ["apple","ant","banana","ball","cat","car"]
grouped = defaultdict(list)
for word in words:
        grouped[word[0]].append(word)
print(dict(grouped))
```

22. Question: Find all pairs in list whose sum equals target

```
lst = [1, 2, 3, 4, 5, 6]
target = 7
for i in range(len(lst)):
   for j in range(i + 1, len(lst)):
      if lst[i] + lst[j] == target:
         print(lst[i], lst[j])
```

23. Question: Convert list of tuples into dictionary

```
pairs = [("a",1),("b",2),("c",3)]
print(dict(pairs))
```

24. Question: Find top 3 most frequent elements in a list

```
from collections import Counter
nums = [1,2,2,3,3,3,4,4,4,4,5]
print(Counter(nums).most_common(3))
```

25. Question: Transpose a matrix

```
matrix = [[1, 2, 3],
          [4, 5, 6]]
rows = len(matrix)
cols = len(matrix[0])
transpose = []
for i in range(cols):
    new_row = []
    for j in range(rows):
        new_row.append(matrix[j][i])
    transpose.append(new_row)
```

26. Question: Implement stack using Python list

```
stack = []
stack.append(10)
stack.append(20)
print(stack.pop())
```

27. Question: Implement queue using collections.deque

```
from collections import deque
queue = deque()
queue.append(10)
queue.append(20)
print(queue.popleft())
```

28. Question: Generate random numbers without random module

```
import time
def pseudo_random(seed=1):
            seed = (seed*9301+49297) % 233280
            return seed/233280.0
print(pseudo_random(int(time.time())))
```

29. Question: Convert string to title case

```
text = "data science with python"
print(text.title())
```

30. Question: Remove punctuation from string

```
import string
text = "Hello, World! Data@Science."
clean = "".join(c for c in text if c not in string.punctuation)
print(clean)
```

31. Question: Load CSV in pandas and display first 10 rows

```
import pandas as pd
df = pd.read_csv("data.csv")
print(df.head(10))
```

32. Question: Select rows where column value > 100
```
print(df[df["Sales"] > 100])
```

33. Question: Drop rows with missing values
```
print(df.dropna())
```

34. Question: Fill missing values with column mean
```
df["Sales"].fillna(df["Sales"].mean(), inplace=True)
```

35. Question: Merge two DataFrames on a column
```
merged = pd.merge(df1,df2,on="ID")
print(merged)
```

36. Question: Group data by column and calculate mean
```
print(df.groupby("Category") ["Sales"].mean())
```

37. Question: Sort DataFrame by multiple columns
```
print(df.sort_values(by= ["Category","Sales"], ascending=[True,False]))
```

38. Question: Apply custom function to column
```
df["Discounted"] = df["Sales"].apply(lambda x: x*0.9)
```

39. Question: Find number of unique values in a column
```
print(df["CustomerID"].nunique())
```

40. Question: Rename multiple columns in DataFrame
```
df.rename(columns= {"Sales":"Total_Sales","Date":"Order_Date"}, inplace=True)
```

41. Question: Create a NumPy array of zeros
```
import numpy as np
arr = np.zeros((3,3))
```

42. Question: Create a NumPy array from 1 to 10
```
arr = np.arange(1,11)
print(arr)
```

43. Question: Reshape 1D NumPy array to 2D
```
arr = np.arange(1,7).reshape(2,3)
print(arr)
```

44. Question: Find max and min in NumPy array
```
arr = np.array([3,7,1,9,2])
print(arr.max(), arr.min())
```

45. Question: Compute mean, median, std of array
```
arr = np.array([1,2,3,4,5,6])
print(arr.mean(), np.median(arr), arr.std())
```

46. Question: Slice first 3 elements of array
```
arr = np.array([10,20,30,40,50])
print(arr[:3])
```

47. Question: Find index of max in NumPy array
```
arr = np.array([4,8,2,9,6])
print(arr.argmax())
```

48. Question: Create diagonal matrix in NumPy
```
arr = np.diag([1,2,3])
print(arr)
```

49. Question: Multiply two NumPy arrays element-wise
```
a = np.array([1,2,3])
b = np.array([4,5,6])
print(a*b)
```

50. Question: Perform matrix multiplication
```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
print(np.dot(a,b))
```

51. Question: Create random NumPy array of shape 2x3
```
arr = np.random.rand(2,3)
print(arr)
```

52. Question: Normalize a NumPy array
```
arr = np.array([10,20,30])
norm = (arr-arr.min())/(arr.max()-arr.min())
print(norm)
```

53. Question: Get unique values from NumPy array
```
arr = np.array([1,2,2,3,3,3,4])
print(np.unique(arr))
```

54. Question: Replace negative values with zero
```
arr = np.array([-2,5,-7,8])
arr[arr < 0] = 0
print(arr)
```

55. Question: Check for NaN in NumPy array
```
arr = np.array([1,np.nan,3])
print(np.isnan(arr))
```

56. Question: Convert NumPy array to Python list
```
arr = np.array([1,2,3])
print(arr.tolist())
```

57. Question: Filter rows in pandas with multiple conditions
```
df = pd.DataFrame({"A":[1,2,3,4],"B": [10,20,30,40]})
print(df[(df["A"] > 2) & (df["B"] < 40)])
```

58. Question: Create pivot table in pandas
```
df = pd.DataFrame({"Category": ["A","A","B","B"],"Sales":[100,200,300,400]})
pivot = df.pivot_table(values="Sales", index="Category", aggfunc="sum")
print(pivot)
```

59. Question: Get correlation matrix
```
print(df.corr())
```

60. Question: Save DataFrame to CSV
```
df.to_csv("output.csv", index=False)
```

61. Question: Select specific columns
```
df = pd.DataFrame({"A":[1,2,3],"B": [4,5,6],"C":[7,8,9]})
print(df[["A","C"]])
```

62. Question: Sort DataFrame by column descending
```
print(df.sort_values("B", ascending=False))
```

63. Question: Rename columns
```
df = df.rename(columns= {"A":"Col1","B":"Col2"})
print(df)
```

64. Question: Drop rows with missing values
```
df = pd.DataFrame({"A":[1,None,3],"B": [4,5,None]})
print(df.dropna())
```

65. Question: Fill missing values with mean
```
df["A"].fillna(df["A"].mean(), inplace=True)
print(df)
```

66. Question: Convert column to datetime
```
df = pd.DataFrame({"Date":["2023-01-01","2023-02-01"]})
df["Date"] = pd.to_datetime(df["Date"])
print(df.dtypes)
```

67. Question: Extract year and month from datetime
```
df["Year"] = df["Date"].dt.year
df["Month"] = df["Date"].dt.month
print(df)
```

68. Question: Remove duplicate rows
```
df = pd.DataFrame({"A":[1,2,2,3],"B": [4,5,5,6]})
print(df.drop_duplicates())
```

69. Question: Group by column and sum
```
df = pd.DataFrame({"Category": ["A","A","B"],"Sales":[100,200,300]})
print(df.groupby("Category")
["Sales"].sum())
```

70. Question: Apply custom function to column
```
df = pd.DataFrame({"A":[1,2,3]})
df["Square"] = df["A"].apply(lambda x:x**2)
print(df)
```

71. Question: Merge two DataFrames on key
```
df1 = pd.DataFrame({"ID":[1,2],"Name": ["A","B"]})
df2 = pd.DataFrame({"ID":[1,2],"Age": [25,30]})
print(pd.merge(df1,df2,on="ID"))
```

72. Question: Concatenate two DataFrames vertically
```
df1 = pd.DataFrame({"A":[1,2]})
df2 = pd.DataFrame({"A":[3,4]})
print(pd.concat([df1,df2]))
```

73. Question: Find top 2 rows by column
```
df = pd.DataFrame({"A":[10,40,30,20]})
print(df.nlargest(2,"A"))
```

74. Question: Find bottom 2 rows by column
```
print(df.nsmallest(2,"A"))
```

75. Question: Count missing values
```
df = pd.DataFrame({"A":[1,None,3],"B": [None,5,6]})
print(df.isnull().sum())
```

76. Question: Replace specific value
```
df = pd.DataFrame({"A":[1,2,2,3]})
df["A"].replace(2, 99, inplace=True)
print(df)
```

77. Question: Convert categorical to dummy variables
```
df = pd.DataFrame({"Fruit": ["Apple","Banana","Apple"]})
print(pd.get_dummies(df, columns= ["Fruit"]))
```

78. Question: Calculate cumulative sum
```
df = pd.DataFrame({"Sales": [100,200,300]})
df["Cumulative"] = df["Sales"].cumsum()
print(df)
```

79. Question: Calculate percentage of total
```
df["Percentage"] = df["Sales"]/df["Sales"].sum()*100
print(df)
```

80. Question: Detect outliers using IQR

```
Q1 = df["Sales"].quantile(0.25)
Q3 = df["Sales"].quantile(0.75)
IQR = Q3 - Q1
outliers = df[(df["Sales"] < Q1 - 1.5*IQR) | (df["Sales"] > Q3 + 1.5*IQR)]
print(outliers)
```

81. Question: Standardize a numeric column using z-score

```
df["Zscore"] = (df["Sales"] -
df["Sales"].mean()) / df["Sales"].std()
print(df)
```

82. Question: Normalize values between 0 and 1

```
df["Normalized"] = (df["Sales"] df["Sales"].min())/(df["Sales"].max() df["Sales"].min())
print(df)
```

83. Question: Split dataset into train and test

```
from sklearn.model_selection import
train_test_split
X = df[["Sales"]]
y = [0,1,0,1]
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_stat e=42)
print(X_train,y_train)
```

84. Question: Train simple linear regression

```
from sklearn.linear_model import
LinearRegression
X = np.array([[1],[2],[3],[4]])
y = np.array([2,4,6,8])
model = LinearRegression()
model.fit(X,y)
print(model.coef_, model.intercept_)
```

85. Question: Make predictions

```
pred = model.predict([[5]])
print(pred)
```

86. Question: Evaluate regression using R2 score

```
from sklearn.metrics import r2_score
y_true = [2,4,6,8]
y_pred = model.predict(X)
print(r2_score(y_true,y_pred))
```

87. Question: Train logistic regression

```
from sklearn.linear_model import
LogisticRegression
X = np.array([[1],[2],[3],[4]])
```

```
y = np.array([0,0,1,1])
clf = LogisticRegression()
clf.fit(X,y)
print(clf.predict([[2.5]]))
```

88. Question: Create confusion matrix

```
from sklearn.metrics import
confusion_matrix
y_true = [0,0,1,1]
y_pred = [0,1,1,1]
print(confusion_matrix(y_true,y_pred))
```

89. Question: Calculate accuracy, precision, recall, f1-score

```
from sklearn.metrics import
classification_report
print(classification_report(y_true,y_pred))
```

90. Question: Perform k-fold cross validation

```
from sklearn.model_selection import
cross_val_score
scores = cross_val_score(clf,X,y,cv=3)
print(scores.mean())
```

91. Question: Train decision tree classifier

```
from sklearn.tree import
DecisionTreeClassifier
clf = DecisionTreeClassifier()
clf.fit(X,y)
print(clf.predict([[2]]))
```

92. Question: Train random forest classifier

```
from sklearn.ensemble import
RandomForestClassifier
clf = RandomForestClassifier()
clf.fit(X,y)
print(clf.predict([[3]]))
```

93. Question: Train support vector machine classifier

```
from sklearn.svm import SVC
clf = SVC()
clf.fit(X,y)
print(clf.predict([[2.5]]))
```

94. Question: Scale features using StandardScaler

```
from sklearn.preprocessing import
StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
print(X_scaled)
```

95. Question: Encode categorical using LabelEncoder

```
from sklearn.preprocessing import
LabelEncoder
fruits = ["Apple","Banana","Apple","Orange"]
encoder = LabelEncoder()
encoded = encoder.fit_transform(fruits)
print(encoded)
```

96. Question: One-hot encode categorical feature

```
from sklearn.preprocessing import
OneHotEncoder
encoder = OneHotEncoder(sparse=False)
data = np.array(fruits).reshape(-1,1)
encoded = encoder.fit_transform(data)
print(encoded)
```

97. Question: Reduce dimensions using PCA

Answer:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X_scaled)
print(X_reduced)
```

98. Question: Save trained model with joblib

```
import joblib
joblib.dump(clf,"model.pkl")
```

99. Question: Load trained model with joblib

```
model = joblib.load("model.pkl")
print(model.predict([[2]]))
```

100. Question: Create pipeline with scaler and classifier

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([("scaler",
StandardScaler()),("clf",
LogisticRegression())])
pipeline.fit(X,y)
print(pipeline.predict([[2.5]]))-
```