Python Interview Questions for Freshers:

1. What is `__init__` in Python?
2. What is the difference between Python Arrays and Lists ?
3. What is slicing in Python?
4. What is a docstring in Python?
5. What is the difference between `break`, `continue`, and `pass` in Python?
6. What is the use of `self` in Python?
7. What are modules and packages in Python?
8. What is `pass` in Python?
9. What is the difference between `print()`, `return`, and `yield` in Python?
10. Explain the classifications of Python data types in terms of whether they are ordered, unordered, mutable, or immutable. Provide examples for each category.
11. What are common built-in data types in Python?
12. What are Python functions? Explain their types.
13. What is a Python dictionary? Explain its key properties.
14. Explain list comprehensions with an example.
15. What is the difference between deep copy and shallow copy?
16. Explain Python's `with` statement.
17. What are Python's commonly used libraries?
18. **What is Python's `with` statement, and how is it used for resource management ?**

19. **How can you merge two dictionaries in Python? Provide multiple ways to achieve this.**
20. **What are Python's `any()` and `all()` functions? Explain with examples.**
21. **How does the Python `zip()` function work? What happens if the input iterables have different lengths?**
22. **How can you remove duplicates from a list in Python while maintaining the order?**
23. **How does Python handle negative indexing in sequences like lists or strings? Why is it useful?**
24. **What is the Global Interpreter Lock (GIL) in Python, and how does it affect multi-threading?**
25. **What are Python decorators and how do they work?**
26. **What is a generator in Python, and how does it differ from a regular function?**
27. **What is the difference between `is` and `==` in Python ?**
28. **. What is the difference between mutable and immutable objects in Python?**
29. **. What is Python's `None` type, and how is it used?**

1. What is `__init__` in Python?

The `__init__` method is a constructor in Python. It is called automatically when a new object of a class is created. It is used to initialize the attributes of a class.

Example:

```
python
Copy code
class Student:
    def __init__(self, fname, lname, age, section):
        self.firstname = fname
        self.lastname = lname
        self.age = age
        self.section = section

stu1 = Student("Sara", "Ansh", 22, "A2")
```

2. What is the difference between Python Arrays and Lists?

Feature	Arrays	Lists
Data Type	Homogeneous (only one data type)	Heterogeneous (any data types)
Memory	Consumes less memory	Consumes more memory

Example:

```
python
Copy code
import array
a = array.array('i', [1, 2, 3])  # Array
b = [1, 2, 'string']            # List
```

4. What is slicing in Python?

Slicing is used to extract parts of sequences like strings, lists, or tuples.

Syntax: `[start : stop : step]`

Example:

```
python
Copy code
numbers = [1, 2, 3, 4, 5]
print(numbers[1:4])  # Output: [2, 3, 4]
```

Key Differences:

- **Indexing** accesses a single element at the specified position.
- **Slicing** returns a new sub-sequence (list, string, etc.) from the original sequence.

5. What is a docstring in Python?

A **docstring** is a multiline string used to document a specific segment of code like functions, classes, or modules.

Example:

```
python
Copy code
def example_function():
    """This function demonstrates docstrings."""
    pass
```

---

7. What is the difference between `break`, `continue`, and `pass` in Python?

Keyword	Purpose
break	Terminates the loop immediately.
continue	Skips the current iteration and continues with the next.
pass	Does nothing and is used as a placeholder for empty blocks.

Example:

```
python
Copy code
for i in range(5):
    if i == 2:
        continue
    elif i == 4:
        break
    print(i)  # Output: 0, 1, 3
```

8. What is the use of `self` in Python?

`self` represents the instance of a class and is used to access its attributes and methods.

9. What are global, protected, and private attributes in Python?

Attribute Type	Example	Access Level
Global	`global_var`	Accessible anywhere in the program.
Protected	`_protected`	Accessible within the class and subclasses.
Private	`__private`	Accessible only within the class.

10. What are modules and packages in Python?

- **Modules:** Files containing Python code (e.g., `.py` files).

- **Packages:** A collection of modules organized in directories with an `__init__.py` file.**Example:**
- **Module**: A single Python file containing code (functions, variables, etc.).
- **Package**: A collection of modules in a directory with an `__init__.py` file.
- **Library**: A collection of modules and packages that serve a specific purpose (e.g., NumPy, Pandas).

```
1. Python Module:
A module in Python is a single file that contains Python definit
```

```


# Module usage
import math
print(math.sqrt(16))  # Output: 4.0

```

---

2. Python Package:

A **package** is a collection of Python modules. It is a directory that contains multiple Python files along with an `__init__.py` file. The `__init__.py` file tells Python that the directory should be treated as a package.

3. Python Library:

A **library** is a collection of modules and packages that provide functionalities for a specific purpose. A library can contain hundreds or thousands of modules and packages. Libraries are typically distributed via package managers like `pip`.

11. What is `pass` in Python?

The `pass` keyword is used as a placeholder for empty code blocks to prevent syntax errors.

Example:

```
def my_function():
    pass  # Placeholder for future code
```

12. What are common built-in data types in Python?

Category	Data Types	Description
Numeric	`int`, `float`, `complex`	Numbers including integers, decimals, etc.
Sequence	`list`, `tuple`, `range`, `str`	Ordered collections of items.
Mapping	`dict`	Key-value pairs.
Set Types	`set`, `frozenset`	Unordered collections of unique items.
Boolean	`bool`	`True` or `False`.
None Type	`None`	Represents a null value.

Example:

```
python
Copy code
a = [1, 2, 3]  # List
b = {'name': 'Alice'}  # Dictionary
c = {1, 2, 3}  # Set
```

13. What are Python functions? Explain their types.

Python functions are blocks of reusable code used to perform specific tasks. A function is defined using the `def` keyword, followed by its name and parameters.

```
python
Copy code
def greet(name):
    return f"Hello, {name}!"
print(greet("Alice"))  # Output: Hello, Alice!
```

Types of functions in Python:

1. **Built-in Functions:** Predefined functions like `print()`, `len()`, `type()`, etc.
2. **User-Defined Functions:** Functions created by the user.
3. **Lambda Functions:** Anonymous functions defined using the `lambda` keyword.
4. **Recursive Functions:** Functions that call themselves to solve smaller instances of a problem.
5. **Higher-Order Functions:** Functions that take other functions as arguments (e.g., `map()`, `filter()`).

---

14 . What is the difference between `print()`, `return`, and `yield` in Python?

Answer:

Here's a detailed breakdown of the differences between `print()`, `return`, and `yield`:

1. `print()`:

- **Purpose**: The `print()` function outputs data to the console or standard output. It does not affect the flow of the program.

- **Behavior**: It prints values, expressions, or strings, and then continues executing the rest of the program.
- **Use case**: Use `print()` when you want to display information, typically for debugging or user interaction.

Example:

```
python
Copy code
def greet():
    print("Hello, World!")
greet()
```

Output:

```
Copy code
Hello, World!
```

2. `return`:

- **Purpose**: The `return` statement is used in a function to send a result back to the caller and exit the function. It can return any type of value (including a list, string, integer, etc.).
- **Behavior**: When `return` is executed, the function terminates, and the value is passed back to the calling code.
- **Use case**: Use `return` when you want a function to compute and return a value.

Example:

```
python
Copy code
def add(a, b):
    return a + b
```

```
result = add(5, 3)
print(result)  # Output: 8
```

3. `yield`:

- **Purpose**: The `yield` keyword is used to produce a value from a generator function and pauses the function's state, which can later be resumed.
- **Behavior**: Unlike `return`, `yield` doesn't terminate the function. Instead, it "yields" a value to the caller and allows the function to resume from where it left off when called again.
- **Use case**: Use `yield` when dealing with large datasets or when you want to generate a sequence of values lazily (i.e., on-demand) without loading everything into memory at once.

Example:

```
python
Copy code
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)
```

Output:

```
Copy code
5
4
3
2
```

```
1
```

Key Differences:

Feature	`print()`	`return`	`yield`
Purpose	Output data to the console	Send a result back to the caller	Generate values lazily in a generator
Execution Flow	Continues execution after printing	Exits the function and returns a value	Pauses function, resuming on next call
Use Case	Debugging or displaying info	Returning values from a function	Handling large data or sequences lazily

Let me know if you'd like further examples or details!

Quick Summary:

- **Ordered**: `int`, `float`, `complex`, `list`, `tuple`, `range`, `str`, `bytes`, `memoryview`.
- **Unordered**: `dict`, `set`, `frozenset`.
- **Mutable**: `list`, `dict`, `set`, `bytearray`.
- **Immutable**: `int`, `float`, `complex`, `tuple`, `str`, `bytes`, `frozenset`, `range`, `memoryview`.

14. What is a Python dictionary? Explain its key properties.

A Python dictionary is a collection of key-value pairs. It is unordered, mutable, and indexed by keys.

Key Properties:

- Keys must be unique and immutable (e.g., strings, numbers, or tuples).
- Values can be of any data type.
- Dictionaries are defined using `{}`.

Example:

```
python
Copy code
my_dict = {"name": "John", "age": 25}
print(my_dict["name"])  # Output: John
my_dict["age"] = 30  # Updating value
```

Key Properties of a Python Dictionary:

1. **Key-Value Pair Structure**:
    - A dictionary consists of keys and their associated values.
    - Example:

      ```
      python
      Copy code
      student = {"name": "John", "age": 21, "grade": "A"}
      ```

        - Here, `name`, `age`, and `grade` are **keys**, and `"John"`, `21`, and `"A"` are the **values**.
2. **Keys Must Be Unique**:
    - Keys in a dictionary must be **unique**. If a key is repeated, the most recent value will overwrite the previous one.
    - Example:

      ```
      python
      Copy code
      data = {"a": 1, "b": 2, "a": 3}
      print(data)  # Output: {'a': 3, 'b': 2}
      ```
3. **Keys Must Be Immutable**:

- Keys can only be **immutable data types** like strings, numbers, or tuples.
- Mutable types like lists or other dictionaries cannot be used as keys.
- Example (invalid key):

```
python
Copy code
invalid_dict = {[1, 2]: "value"}  # Raises a TypeError
```

4. **Values Can Be Any Data Type**:
   - Values can be of any data type, including lists, dictionaries, or even functions.
   - Example:

```
python
Copy code
sample_dict = {"numbers": [1, 2, 3], "nested": {"key":
"value"}}
```

5.

15. Explain list comprehensions with an example.

List comprehension is a concise way to create lists in Python.

Syntax:

```
python
Copy code
[expression for item in iterable if condition]
```

Example:

```
python
Copy code
squares = [x**2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

This creates a list of squares from 1 to 5.

16. What is the difference between deep copy and shallow copy?

- **Shallow Copy:** Creates a new object but references the same elements in the original object. Changes in nested objects reflect in the copied object.

  Example:

  ```
  python
  Copy code
  import copy
  list1 = [[1, 2], [3, 4]]
  shallow_copy = copy.copy(list1)
  shallow_copy[0][0] = 99
  print(list1)  # Output: [[99, 2], [3, 4]]
  ```

- **Deep Copy:** Creates a completely independent copy of the original object, including nested objects. Changes do not affect the original object.

  Example:

  ```
  python
  Copy code
  deep_copy = copy.deepcopy(list1)
  deep_copy[0][0] = 100
  print(list1)  # Output: [[99, 2], [3, 4]]
  ```

17. Explain Python's `with` statement.

The `with` statement is used for resource management. It ensures that resources like files or database connections are properly closed or released after use, even if exceptions occur.

Example:

```
python
Copy code
with open("example.txt", "r") as file:
    content = file.read()
# File is automatically closed after exiting the block.
```

18. What are Python's commonly used libraries?

Some commonly used Python libraries include:

- **NumPy:** For numerical computing.
- **Pandas:** For data manipulation and analysis.
- **Matplotlib/Seaborn:** For data visualization.
- **Scikit-learn:** For machine learning.
- **TensorFlow/PyTorch:** For deep learning.
- **Flask/Django:** For web development.
- **Requests:** For handling HTTP requests.
- **BeautifulSoup:** For web scraping.
- **OS and Sys:** For interacting with the operating system.

23. How can you merge two dictionaries in Python? Provide multiple ways to achieve this.

- **Using `update()`:**

  ```
  python
  Copy code
  dict1 = {"a": 1, "b": 2}
  dict2 = {"c": 3, "d": 4}
  dict1.update(dict2)
  print(dict1)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
  ```

---

25. What are Python's `any()` and `all()` functions? Explain with examples.

- `any()`: Returns `True` if at least one element of an iterable is true.
  - **Example**:

    ```
    any([0, False, True])  # Output: True
    ```

- `all()`: Returns `True` if all elements of an iterable are true.
  - **Example**:

    ```
    all([1, 2, 3])  # Output: True
    ```

---

26. How does the Python `zip()` function work? What happens if the input iterables have different lengths?

- **Purpose**: Combines multiple iterables element-wise into tuples.
  - **Example**:

```
python
Copy code
a = [1, 2, 3]
b = ['a', 'b', 'c']
print(list(zip(a, b)))  # Output: [(1, 'a'), (2, 'b'),
(3, 'c')]
```

- **Different Lengths**: `zip()` stops when the shortest iterable is exhausted.
  - **Example**:

```
python
Copy code
a = [1, 2, 3]
b = ['a', 'b']
print(list(zip(a, b)))  # Output: [(1, 'a'), (2, 'b')]
```

---

28. How can you remove duplicates from a list in Python while maintaining the order?

- **Using `dict.fromkeys()`:**

```
my_list = [1, 2, 2, 3, 3, 4]
unique_list = list(dict.fromkeys(my_list))
print(unique_list)  # Output: [1, 2, 3, 4]
```

- **Using a loop**:

  ```
  python
  Copy code
  my_list = [1, 2, 2, 3, 3, 4]
  unique_list = []
  for item in my_list:
      if item not in unique_list:
          unique_list.append(item)
  print(unique_list)  # Output: [1, 2, 3, 4]
  ```

---

29. How does Python handle negative indexing in sequences like lists or strings? Why is it useful?

- **Negative indexing** starts from the end of the sequence. `1` refers to the last element, `2` to the second last, and so on.
  - **Example**:

    ```
    python
    Copy code
    my_list = [1, 2, 3, 4]
    print(my_list[-1])  # Output: 4
    print(my_list[-2])  # Output: 3
    ```

- **Usefulness**: It allows easy access to the last elements without needing to know the length of the sequence.

---

30. What is the Global Interpreter Lock (GIL) in Python, and how does it affect multi-threading?

- **GIL**: A mutex that protects access to Python objects, limiting execution to one thread at a time in a single process.
- **Effect on multi-threading**:

- It makes Python threads execute one at a time, preventing true parallelism.
- Affects CPU-bound tasks but not I/O-bound tasks, where threads can run concurrently.
- **Example:** Python's multithreading is useful for I/O-bound tasks, but not for CPU-bound tasks like computations.

What is a generator in Python, and how does it differ from a regular function?

- **Answer**:
  A
  generator is a function that yields a sequence of values instead of returning a single value. When a generator function is called, it returns an iterator object, which can be iterated over using a loop.

  Unlike a regular function, which returns a single value and exits, a generator uses the `yield` keyword to return multiple values one by one, maintaining the state between function calls.

  Example:

  ```
  python
  Copy code
  def my_generator():
      yield 1
      yield 2
      yield 3

  gen = my_generator()
  for val in gen:
      print(val)  # Output: 1 2 3
  ```

4. What are Python decorators and how do they work?

- **Answer**:
  A
  decorator is a function that wraps another function or method, adding additional functionality to it. It allows you to modify or extend the behavior of the original function without changing its code directly.

  Syntax: You apply a decorator using the `@` symbol above the function definition.

  Example:

```
python
Copy code
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@my_decorator
def greet():
    print("Hello!")

greet()
# Output:
# Before function call
# Hello!
# After function call
```

What is the difference between `is` and `==` in Python?

- **Answer**:
  - `is`: Checks whether two objects refer to the same memory location (identity comparison).

    - `==`: Checks whether the values of two objects are equal (value comparison).

Example:

```
python
Copy code
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a == b)  # True (values are equal)
print(a is b)  # False (different memory locations)
print(a is c)  # True (same memory location)
```

The operators `is` and `==` are both used for comparisons in Python, but they serve **different purposes** and behave differently depending on the context.

Key Differences Between `is` and `==`:

Feature	`is` (Identity Operator)	`==` (Equality Operator)
Purpose	Checks **identity**: Whether two objects refer to the same memory location.	Checks **equality**: Whether the values of two objects are the same.
Comparison	Compares the **memory address** (object identity).	Compares the **values** of objects.
Use Cases	Used for checking if two variables point to the **same object** in memory.	Used for comparing if two variables have **equal values**.
Types	Works on any object but only checks identity.	Works on any object and compares values.

1. What is the difference between mutable and immutable objects in Python?

- **Answer**:

- **Mutable objects** can be changed after they are created. Examples include lists, dictionaries, and sets.
- **Immutable objects** cannot be changed after they are created. Examples include strings, tuples, and numbers.

Example:

```
python
Copy code
# Mutable Example
my_list = [1, 2, 3]
my_list[0] = 10
print(my_list)  # Output: [10, 2, 3]

# Immutable Example
my_string = "hello"
try:
    my_string[0] = "H"  # Throws an error
except TypeError as e:
    print(e)  # 'str' object does not support item assignment
```

---

2. What is Python's `None` type, and how is it used?

- **Answer**:
  - `None` is a special data type in Python that represents the absence of a value or a null value.
  - It is used as a placeholder for variables that are declared but not yet assigned a value, or as a return value for functions that do not explicitly return anything.

Example:

```
python
Copy code
```

```
# Example of None as a placeholder
result = None
print(result)  # Output: None

# Example of a function returning None
def greet(name):
    print(f"Hello, {name}!")

x = greet("Alice")  # The function prints a message but retur
ns None
print(x)  # Output: None
```

. What is a lambda function? How is it different from a regular Python function?

Answer:

- A **lambda function** is an anonymous, inline function defined using the `lambda` keyword. It can take any number of arguments but can only have one expression.
- Lambda functions are often used for short-lived purposes where defining a full function isn't necessary.

Syntax:

```
python
Copy code
lambda arguments: expression
```

Example:

```
python
Copy code
# Regular function
def add(x, y):
    return x + y

# Lambda function
add_lambda = lambda x, y: x + y

print(add(3, 5))         # Output: 8
print(add_lambda(3, 5))  # Output: 8
```

Difference:

- A regular function can have multiple statements, while a lambda function is limited to a single expression.
- Lambda functions are typically used for concise operations like in `map`, `filter`, or `reduce`.

How do you use `lambda` with `map`?

Answer:
The
`map()` function applies a lambda (or any function) to each item in an iterable (like a list) and returns a map object.

Example:

```
python
Copy code
nums = [1, 2, 3, 4, 5]
```

```
squared = map(lambda x: x ** 2, nums)
print(list(squared))  # Output: [1, 4, 9, 16, 25]
```

---

3. What is the `filter` function and how does it work with `lambda`?

Answer:
The
`filter()` function filters elements of an iterable based on a function (or lambda) that returns `True` or `False`.

Example:

```
python
Copy code
nums = [1, 2, 3, 4, 5, 6]
even_nums = filter(lambda x: x % 2 == 0, nums)
print(list(even_nums))  # Output: [2, 4, 6]
```

---

4. What is the `reduce` function? How is it different from `map` and `filter`?

Answer:

- The `reduce()` function (from the `functools` module) applies a rolling computation to the elements of an iterable.
- Unlike `map` and `filter`, which operate on individual elements, `reduce` operates on pairs of elements to reduce the iterable to a single cumulative value.

Syntax:

```
python
Copy code
```

```
reduce(function, iterable, [initial])
```

Example:

```
python
Copy code
from functools import reduce

nums = [1, 2, 3, 4, 5]
# Sum of all numbers
result = reduce(lambda x, y: x + y, nums)
print(result)  # Output: 15
```

---

5. What are some practical use cases of lambda, map, filter, and reduce?

Use Case 1: Find squares of numbers using `map`.

```
python
Copy code
nums = [1, 2, 3, 4]
squares = map(lambda x: x ** 2, nums)
print(list(squares))  # Output: [1, 4, 9, 16]
```

Use Case 2: Filter out odd numbers using `filter`.

```
python
Copy code
nums = [10, 15, 20, 25]
evens = filter(lambda x: x % 2 == 0, nums)
```

```
print(list(evens))  # Output: [10, 20]
```

Use Case 3: Find the product of a list of numbers using `reduce`.

```
python
Copy code
from functools import reduce

nums = [2, 3, 4]
product = reduce(lambda x, y: x * y, nums)
print(product)  # Output: 24
```

Use Case 4: Sort tuples based on the second value using `lambda`.

```
python
Copy code
data = [(1, 'b'), (3, 'a'), (2, 'c')]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)  # Output: [(3, 'a'), (1, 'b'), (2, 'c')]
```