Grow Data Skills

Python Interview Questions

Python Basics

EASY:

Q.1 What is Python? List some popular applications of Python in the world of technology.

A. Python is a widely-used general-purpose, high-level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.
It is used for:

- Data Science and Analytics
- Web Development
- Game Development
- Software Development
- Machine Learning and Artificial Intelligence
- Automation and Scripting etc.

Q.2 What are the benefits of using Python language as a tool in the present scenario?

A. The following are the benefits of using Python language:

- Object-Oriented Language
- High-Level Language
- Dynamically Typed language
- Extensive support Libraries
- Presence of third-party modules
- Open source and community development
- Portable and Interactive
- Portable across Operating systems

Q.3 Is Python a compiled language or an interpreted language?

A. Actually, Python is a partially compiled language and partially interpreted language. The compilation part is done first when we execute our code and this will generate byte code internally. This byte code gets converted by the Python virtual machine(p.v.m) according to the underlying platform(machine+operating system).

Q.4 What does the ‘#’ Symbol in Python?

A. ‘#’ is used to comment on everything that comes after on the line.

Q.5 What is the difference between a Mutable data type and an Immutable data type?

A. Mutable data types can be edited i.e., they can change at runtime. Eg – List, Dictionary, etc. Immutable data types can not be edited i.e., they can not change at runtime. Eg – String, Tuple, etc.

Q.6 How are arguments passed by value or by reference in Python?

A. Everything in Python is an object and all variables hold references to the objects. The reference values are according to the functions; as a result, you cannot change the value of the references. However, you can change the objects if it is mutable.

Q.7 What is the difference between a Set and a Dictionary?

A. The set is an unordered collection of data types that is iterable, mutable and has no duplicate elements.
A dictionary in Python is an unordered collection of data values, used to store data values like a map.

Q.8 What is List Comprehension? Give an Example.

A. List comprehension is a syntax construction to ease the creation of a list based on existing iterable.

For Example: my_list = [i for i in range(1, 10)]

Q.9 What is a lambda function?

A. lambda function is an anonymous function. This function can have any number of parameters but can have just one statement.

For Example:

```
a = lambda x, y : x*y

print(a(7, 19))
```

Q.10 What is a pass in Python?

A. Pass means performing no operation or in other words, it is a placeholder in the compound statement, where there should be a blank left and nothing has to be written there.

Q.11 What is the difference between / and // in Python?

A. // represents floor division whereas / represents precise division. For Example:

5//2 = 2

5/2 = 2.5

Q.12 How is python a Dynamically Typed language?

A. Python is considered a dynamically typed language because the type of a variable is determined and checked at runtime, rather than being explicitly declared during variable creation or at compile-time. In a dynamically typed language like Python, you don't need to specify the data type of a variable when you declare it. Instead, the data type of a variable is inferred based on the value assigned to it, and this type can change during the program's execution.

Here's why Python is dynamically typed:

1. Type Inference: When you assign a value to a variable, Python automatically determines the data type of that value and associates it with the variable. For example:

   ```
   x = 5 # x is an integer

   y = "Hello" # y is a string
   ```

2. Type Changes: You can change the type of a variable by assigning a new value of a different type to it. Python allows this flexibility:

x = 5 # x is an integer

x = "Hello" # x is now a string

3. No Explicit Type Declarations: Unlike statically typed languages (e.g., C++ or Java), you don't need to explicitly declare the data type of a variable before using it. This makes Python code more concise and easier to write.
4. Dynamic Type Checking: Python performs type checking at runtime, meaning it checks the compatibility of operations and values when they are executed, not during compilation. This can lead to errors being discovered at runtime rather than compile-time.

   For example:

   x = 5

   y = "Hello"

   z = x + y

   # This would result in a TypeError because you can't add an integer and a string directly.

   While dynamic typing offers flexibility and ease of use, it also requires careful coding to avoid type-related errors at runtime. Static typing languages, on the other hand, require explicit type declarations and perform type checking at compile-time, which can help catch certain types of errors before the program runs.

Q.13 What is a swapcase function in Python?

A. It is a string’s function that converts all uppercase characters into lowercase and vice versa. It is used to alter the existing case of the string. This method creates a copy of the string which contains all the characters in the swap case. For Example:

string = "GeeksforGeeks"

string.swapcase() ---> "gEEKSFORgEEKS"

Q.14 Difference between for loop and while loop in Python

A.The “for” Loop is generally used to iterate through the elements of various collection types such as List, tuple, set and dictionary. Developers use a “for” loop where they have both the conditions start and the end. Whereas, the “while” loop is the actual looping feature that is used in any other programming language. Programmers use a Python while loop where they just have the end conditions.

Q.15 Can we Pass a function as an argument in Python?

A. Yes, several arguments can be passed to a function, including objects, variables (of the same or distinct data types), and functions. Functions can be passed as parameters to other functions because they are objects. Higher-order functions are functions that can take other functions as arguments.

Q.16 What are *args and *kwargs?

A. To pass a variable number of arguments to a function in Python, use the special syntax *args and *kwargs in the function specification. It is used to pass a variable-length, keyword-free argument list. By using the *, the variable we associate with the * becomes iterable, allowing you to do operations on it such as iterating over it and using higher-order operations like map and filter.

Q.17 Is Indentation Required in Python?

A. Yes, indentation is required in Python. A python interpreter can be informed that a group of statements belongs to a specific block of code by using Python indentation. Indentations make the code easy to read for developers in all programming languages but in Python, it is very important to indent the code in a specific order.

Q.18 What is Scope in Python?

A. The location where we can find a variable and also access it if required is called the scope of a variable.

- Python Local variable: Local variables are those that are initialized within a function and are unique to that function. It cannot be accessed outside of the function.
- Python Global variables: Global variables are the ones that are defined and declared outside any function and are not specified to any function.
- Module-level scope: It refers to the global objects of the current module accessible in the program.
- Outermost scope: It refers to any built-in names that the program can call. The name referenced is located last among the objects in this scope.

Q.19 What is docstring in Python?

A. Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

- Declaring Docstrings: The docstrings are declared using ”’triple single quotes”’ or “””triple double quotes””” just below the class, method, or function declaration. All functions should have a docstring.
- Accessing Docstrings: The docstrings can be accessed using the __doc__ method of the object or using the help function.

Q.20 What is a dynamically typed language?

A. Type languages are the languages in which we define the type of data type and it will be known by the machine at the compile-time or at runtime. Typed languages can be classified into two categories:

- Statically typed languages: In this type of language, the data type of a variable is known at the compile time which means the programmer has to specify the data type of a variable at the time of its declaration.
- Dynamically typed languages: These are the languages that do not require any pre-defined data type for any variable as it is interpreted at runtime by the machine itself. In these languages, interpreters assign the data type to a variable at runtime depending on its value.

Q.21 What is a break, continue, and pass in Python?

A. The break statement is used to terminate the loop or statement in which it is present. After that, the control will pass to the statements that are present after the break statement, if available.

Continue is also a loop control statement just like the break statement. The continue statement is opposite to that of the break statement, instead of terminating the loop, it forces the execution of the next iteration of the loop.

Pass means performing no operation or in other words, it is a placeholder in the compound statement, where there should be a blank left and nothing has to be written there.

Q.22 What are Built-in data types in Python?

A. The following are the standard or built-in data types in Python:

- Numeric: The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, a Boolean, or even a complex number.
- Sequence Type: The sequence Data Type in Python is the ordered collection of similar or different data types. There are several sequence types in Python:
    - String
    - List
    - Tuple
    - Range
- Mapping Types: In Python, hashable data can be mapped to random objects using a mapping object. There is currently only one common mapping type, the dictionary, and mapping objects are mutable.
    - Dictionary
- Set Types: In Python, a set is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

Q23. How do you floor a number in Python?

A. The Python math module includes a method that can be used to calculate the floor of a number.

- floor() method in Python returns the floor of x i.e., the largest integer not greater than x.
- Also, The method ceil(x) in Python returns a ceiling value of x i.e., the smallest integer greater than or equal to x.

Q.24 Programs:

- Swap Two Variables:

```
def swap(a, b):
    a, b = b, a
    return a, b
```

- Factorial of a Number:

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

- Check if a Number is Prime:

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

- Calculate the Sum of Digits in a Number:

```
def sum_of_digits(n):
    return sum(int(digit) for digit in str(n))
```

- Print Numbers from 1 to N:

```
def print_numbers(n):
  for i in range(1, n + 1):
    print(i)
```

- Print Multiplication Table:

```
def multiplication_table(n):
  for i in range(1, 11):
    print(f"{n} x {i} = {n * i}")
```

- Print Even Numbers from 1 to N:

```
def print_even_numbers(n):
  for i in range(2, n + 1, 2):
    print(i)
```

- Check if a Number is Positive, Negative, or Zero:

```
def check_number(n):
  if n > 0:
    return "Positive"
  elif n < 0:
    return "Negative"
  else:
    return "Zero"
```

- Determine the Largest of Three Numbers:

```
def largest_of_three(a, b, c):
  if a >= b and a >= c:
    return a
  elif b >= a and b >= c:
        return b
    else:
        return c
```

- Calculate the Area of a Circle:

```
import math
def area_of_circle(radius):
    return math.pi * radius ** 2
```

- Calculate the Power of a Number:

```
def power(base, exponent):
    return base ** exponent
```

- Square of a Number using Lambda:

```
square = lambda x: x ** 2
```

- Sum of Two Numbers using Lambda:

```
add = lambda a, b: a + b
```

- Find the Largest Element in a List:

```
def find_largest(lst):
    return max(lst)
```

- Calculate the Sum of All Elements in a List:

```
def calculate_sum(lst):
    return sum(lst)
```

- Reverse a List:

```
def reverse_list(lst):
    return lst[::-1]
```

- Count the Occurrences of an Element in a List:

```
def count_occurrences(lst, element):
    return lst.count(element)
```

- Remove Duplicates from a List:

```
def remove_duplicates(lst):
    return list(set(lst))
```

- Remove Even Numbers from a List:

```
def remove_even_numbers(lst):
    return [x for x in lst if x % 2 != 0]
```

- Calculate the Product of All Elements in a List:

```
def calculate_product(lst):
    product = 1
    for num in lst:
        product *= num
    return product
```

- Find the Sum of Elements in a Tuple:

```
def tuple_sum(t):
    return sum(t)
```

- Check if a Tuple is Palindrome:

```
def is_tuple_palindrome(t):
    return t == t[::-1]
```

- Count the Occurrences of a Value in a Tuple:

```
def count_occurrences(t, value):
    return t.count(value)
```

- Remove a Key from a Dictionary:

```
def remove_key(d, key):
    if key in d:
        del d[key]
```

MEDIUM:

Q.1 What is the difference between xrange and range functions?

A. range() and xrange() are two functions that could be used to iterate a certain number of times in for loops in Python. In Python 3, there is no xrange, but the range function behaves like xrange in Python 2.

- *range()* – This returns a list of numbers created using the range() function.
- *xrange()* – This function returns the generator object that can be used to display numbers only by looping. The only particular range is displayed on demand and hence called *lazy evaluation*.

Q.2 What is Dictionary Comprehension? Give an Example

A. Dictionary Comprehension is a syntax construction to ease the creation of a dictionary based on the existing iterable.

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

Q.3 Is Tuple Comprehension? If yes, how, and if not why?

A. (i for i in (1, 2, 3))

Tuple comprehension is not possible in Python because it will end up in a generator, not a tuple comprehension.

Q.4 Differentiate between List and Tuple?

A. Let's analyze the differences between List and Tuple:

List

- Lists are Mutable data types.
- Lists consume more memory
- The list is better for performing operations, such as insertion and deletion.
- The implication of iterations is Time-consuming

Tuple

- Tuples are Immutable data type.

- Tuple consumes less memory as compared to the list
- A Tuple data type is appropriate for accessing the elements
- The implication of iterations is comparatively Faster

Q.5 What is the difference between a shallow copy and a deep copy?

A.Shallow copy is used when a new instance type gets created and it keeps values that are copied whereas deep copy stores values that are already copied.

A shallow copy has faster program execution whereas a deep copy makes it slow.

Q.6 What is the difference between sort and sorted ?

A. In Python, both sort and sorted are used for sorting elements in a list, but they have some key differences:

1. sort Method:
    - sort is a method that is available directly on a list object.
    - It sorts the elements of the list in place, meaning it modifies the original list and does not create a new list.
    - The sort method does not return a new list. It returns None.
    - Example:

      ```
      my_list = [3, 1, 2]
      my_list.sort()
      print(my_list) # Output: [1, 2, 3]
      ```
2. sorted Function:
    - sorted is a built-in function that takes an iterable (e.g., list, tuple, string) as an argument and returns a new sorted list.
    - It does not modify the original list; instead, it creates a new sorted list.
    - The sorted function allows you to sort any iterable, not just lists.
    - Example:

      ```
      original_list = [3, 1, 2]
      ```

```
sorted_list = sorted(original_list)

print(sorted_list) # Output: [1, 2, 3]

print(original_list) # Output: [3, 1, 2]
```

In summary, the main difference is that sort sorts the list in place and does not return a new list, while sorted creates a new sorted list without modifying the original. The choice between them depends on whether you want to sort in place or create a new sorted list.

Q.7 What are Decorators?

A.Decorators are a very powerful and useful tool in Python as they are the specific change that we make in Python syntax to alter functions easily.

Q.8 How do you debug a Python program?

A. By using this command we can debug a Python program:

```
$ python -m pdb python-script.py
```

Q.9 What are Iterators in Python?

A. In Python, iterators are used to iterate a group of elements, containers like a list. Iterators are collections of items, and they can be a list, tuples, or a dictionary. Python iterator implements __itr__ and the next() method to iterate the stored elements. We generally use loops to iterate over the collections (list, tuple) in Python.

Q.10 What are Generators in Python?

A. In Python, the generator is a way that specifies how to implement iterators. It is a normal function except that it yields expression in the function. It does not implement __itr__ and next() method and reduces other overheads as well.

If a function contains at least a yield statement, it becomes a generator. The yield keyword pauses the current execution by saving its states and then resumes from the same when required.

Q.11 Does Python support multiple Inheritance?

A. Python does support multiple inheritance, unlike Java. Multiple inheritances mean that a class can be derived from more than one parent class.

Q.12 How is memory management done in Python?

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

Q.13 How to delete a file using Python?

A. We can delete a file using Python by following approaches:

- os.remove()
- os.unlink()

Q.14 What is slicing in Python?

A. Python slicing is a string operation for extracting a part of the string, or some part of a list. With this operator, one can specify where to start the slicing, where to end, and specify the step. List slicing returns a new list from the existing list.

Syntax: Lst[Initial : End : Step]

Q.15 What is the namespace in Python?

A. A namespace is a naming system used to make sure that names are unique to avoid naming conflicts.

Q.16 Programs:

- Generate Fibonacci Sequence:

```
def fibonacci(n):
    fib_sequence = [0, 1]
    while len(fib_sequence) < n:
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
    return fib_sequence
```

- Calculate Factorial using a Loop:

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

- Print Triangle of Stars:

```
def print_triangle(rows):
    for i in range(1, rows + 1):
        print("*" * i)
```

- Check if a Year is a Leap Year:

```
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    return False
```

- Assign Grade Based on Percentage:

```
def assign_grade(percentage):
    if percentage >= 90:
        return "A"
    elif percentage >= 80:
        return "B"
    elif percentage >= 70:
        return "C"
    elif percentage >= 60:
        return "D"
    else:
        return "F"
```

- Check if a Character is Vowel or Consonant:

```
def check_vowel_consonant(char):
    vowels = "AEIOUaeiou"
    if char in vowels:
        return "Vowel"
    else:
        return "Consonant"
```

- Calculate Simple Interest:

```
def simple_interest(principal, rate, time):
    return (principal * rate * time) / 100
```

- Calculate Greatest Common Divisor (GCD):

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

- Check if a Number is Armstrong Number:

```
def is_armstrong(n):
    num_str = str(n)
    num_digits = len(num_str)
    total = sum(int(digit) ** num_digits for digit in num_str)
    return n == total
```

- Check if a Number is Even using Lambda:

```
is_even = lambda x: x % 2 == 0
```

- Calculate the Difference between Two Numbers using Lambda:

```
subtract = lambda a, b: a – b
```

- Find the Maximum of Three Numbers using Lambda:

```
max_of_three = lambda a, b, c: max(a, b, c)
```

- Check if a List is Palindrome:

```
def is_palindrome(lst):
    return lst == lst[::-1]
```

- Sort a List of Strings by Length:

```
def sort_by_length(lst):
    return sorted(lst, key=len)
```

- Find the Second Largest Element in a List:

```
def find_second_largest(lst):
    return sorted(set(lst))[-2]
```

- Merge Two Sorted Lists:

```
def merge_sorted_lists(list1, list2):
    return sorted(list1 + list2)
```

- Find the Intersection of Two Lists:

```
def find_intersection(list1, list2):
    return list(set(list1) & set(list2))
```

- Count the Number of Words in a Sentence:

```
def count_words(sentence):
    return len(sentence.split())
```

- Convert a String to Title Case:

```
def to_title_case(s):
    return s.title()
```

- Check if a String Contains Only Digits:

```
def contains_only_digits(s):
    return s.isdigit()
```

- Remove Duplicates from a List using a Set:

```
def remove_duplicates_with_set(lst):
    return list(set(lst))
```

- Check if Two Sets are Disjoint:

```
def are_disjoint(set1, set2):
    return set1.isdisjoint(set2)
```

- Find the Intersection of Two Sets:

```
def set_intersection(set1, set2):
    return set1.intersection(set2)
```

- Find the Union of Two Sets:

```
def set_union(set1, set2):
    return set1.union(set2)
```

- Check if a Key Exists in a Dictionary:

```
def key_exists(d, key):
    return key in d
```

HARD:

Q.1 What is main() in python?

A. In Python, main() is not a built-in or reserved function like it is in some other programming languages. However, the term "main()" is often used conventionally to refer to the entry point of a Python program, where the execution of the program starts.

In many programming languages like C, C++, and Java, there is a special function called main() that serves as the starting point for the program's execution. In Python, there is no strict requirement to use a specific function name like main(), but the idea of having a designated entry point is still applicable.

Conventionally, Python programmers often use the following construct to define the entry point of their script:

```
def main():
    # Your main program logic goes here
    print("Hello, world!")
if __name__ == "__main__":
    main()
```

In this example, main() is just a function name, and you can choose any other name that makes sense to you. The important part is the if __name__ == "__main__": block. This block ensures that the code inside it is only executed if the script is run directly, not if it is imported as a module into another script.

So, while main() is not a special or predefined function in Python, it is a common naming convention for the entry point of a Python script

Q.2 What is the use of ‘**’ in Python?

A. In Python, the double asterisk (**) is used as an exponentiation operator and also as a syntax to unpack dictionaries. The specific behavior of the ** operator depends on the context in which it is used.

Exponentiation Operator:

In mathematical operations, ** is used to raise a number to a power. For example:

```
result = 2 ** 3  # 2 raised to the power of 3, result = 8
```

Dictionary Unpacking:

When used in a function call or in dictionary construction, ** is used to unpack the contents of a dictionary. This is often used to pass multiple keyword arguments to a function or to merge dictionaries. For example:

```
def example_function(a, b):
    print(a, b)
kwargs = {'a': 10, 'b': 20}
example_function(**kwargs)  # Unpacks dictionary as keyword arguments
dict1 = {'x': 1, 'y': 2}
dict2 = {'y': 3, 'z': 4}
merged_dict = {**dict1, **dict2}  # Merge dictionaries using unpacking
```

Keep in mind that the usage of ** in different contexts might have different meanings and behaviors. The specific behavior is determined by the Python syntax rules and the context in which it appears.

Q.3 What is PIP?

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

Q.4 What is a zip function?

A. Python zip() function returns a zip object, which maps a similar index of multiple containers. It takes an iterable, converts it into an iterator and aggregates the elements based on iterables passed. It returns an iterator of tuples.

Q.5 What is __init__() in Python?

A. Equivalent to constructors in OOP terminology, __init__ is a reserved method in Python classes. The __init__ method is called automatically whenever a new object is initiated. This method allocates memory to the new object as soon as it is created. This method can also be used to initialize variables.

Q.6 Write a code to display the current time?

A. currenttime= time.localtime(time.time())

print (“Current time is”, currenttime)

Q.7 What are Access Specifiers in Python?

A. Python uses the ‘_’ symbol to determine the access control for a specific data member or a member function of a class. A Class in Python has three types of python access modifiers:

- Public Access Modifier: The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.
- Protected Access Modifier: The members of a class that are declared protected are only accessible to a class derived from it. All data members of a class are declared protected by adding a single underscore ‘_’ symbol before the data members of that class.
- Private Access Modifier: The members of a class that are declared private are accessible within the class only, the private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore ‘__’ symbol before the data member of that class.

Q.8 What is Python Switch Statement

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

Note: Before Python 3.10 Python didn't support match Statements.

- Python3

```
match term:
  case pattern-1:
  action-1
  case pattern-2:
  action-2
  case pattern-3:
  action-3
  case _:
  action-default
```

Q.9 Discuss the Global NameSpace and Local Name Space in Python. How does the LEGB (Local, Enclosing, Global, Built-in) rule work in resolving variable names?

A. Python uses a series of nested namespaces to organize and resolve variable names. The LEGB rule defines the order in which namespaces are searched: Local, Enclosing, Global, and Built-in. When you reference a variable, Python searches through these namespaces in order until it finds the variable or reaches the built-in namespace.

Q.10 Explain the differences between a generator and an iterator in Python. How does the yield keyword contribute to the behavior of generators?

A. A generator is a special type of iterator that produces values on-the-fly using the yield keyword, allowing you to iterate over large sequences without storing them in memory. An iterator is an object that implements the methods __iter__() and __next__(), allowing you to iterate over its elements. Generators are more memory-efficient compared to iterators.

Q.11 What is the difference between methods and functions?

A. In Python, both methods and functions are blocks of code that perform a specific task. However, there is a distinction between the two based on how they are defined and used:

Functions: Functions are blocks of code that are defined outside of classes.

They can be thought of as standalone units of code that take some input (arguments), perform a task, and return an output (return value).

Functions are defined using the def keyword followed by a function name, parentheses for arguments, and a colon. The function body is indented below.

Example:

```
def add(a, b):
    return a + b
```

Methods: Methods are functions that are associated with objects and are defined within classes.

They operate on the data that belongs to the object they are called on.

Methods are defined similarly to functions, but they have an additional parameter called self, which refers to the instance of the object on which the method is called.

Methods are accessed using dot notation: object.method().

Example:

```
class Circle:
    def init(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius ** 2


my_circle = Circle(5)
print(my_circle.area())  # Calling the method on the Circle object
```

In summary, the key difference between methods and functions lies in their association with classes and objects. Functions are standalone and can be used anywhere in your code, while methods are tied to objects and operate on their specific data.

Q.12 Why do we need lambda functions if the same can be achieved using loops?

A. While lambda functions are one-liners and powerful tools, they are not suitable for all situations. Loops are essential for tasks that involve iterating, accumulating, searching, filtering, and other complex operations. Choosing the right tool for the job ensures that your code is both effective and maintainable.

Here are some examples where you would need looping:

- Iterating: When you have a collection of items (e.g., a list, tuple, dictionary, etc.) and you need to perform an operation on each item, you would typically use a loop. Lambda functions are not well-suited for this scenario because they lack the necessary structure to handle multiple iterations.
- Accumulation and Aggregation: If you need to accumulate or aggregate values from a collection, like summing up the elements of a list or finding the maximum value, loops are essential.
- Searching and Filtering: When you want to find specific elements in a collection that satisfy certain conditions, you often need to loop through the collection and apply a filtering function. Lambda functions can be used for simple cases, but more complex filtering and searching tasks usually require loops to manage the logic effectively.
- Multidimensional Data: When dealing with multi-dimensional data structures (e.g., matrices), loops are necessary to traverse the different dimensions and apply operations or logic at each level.

Q.13 Difference between iterator and iterable in Python?

A. In Python, both iterators and iterable are concepts related to handling collections of items, but they serve different purposes. Let's break down the differences between them with examples:

Iterable:

An iterable is an object that can be looped over (iterated) to retrieve its elements one by one. It needs to implement the iter() method, which returns an iterator object. Common examples of iterables are lists, tuples, strings, dictionaries, sets, etc.

Example:

```
my_list = [1, 2, 3, 4, 5]

for item in my_list:

  print(item)
```

Iterator:

An iterator is an object that provides the actual iteration process. It needs to implement two methods: iter() (which returns itself) and next() (which returns the next item in the sequence). Iterators maintain their internal state, remembering where they are during iteration.

Example:

```
my_iterable = [1, 2, 3, 4, 5]

my_iterator = iter(my_iterable)

print(next(my_iterator))  # Output: 1

print(next(my_iterator))  # Output: 2

print(next(my_iterator))  # Output: 3
```

An important thing to note is that an iterable can be converted to an iterator using the iter() function. When you use a for loop to iterate over an iterable, Python automatically creates an iterator for that iterable behind the scenes.

```
my_list = [1, 2, 3, 4, 5]

for item in my_list:

   print(item)
```

In this loop, Python converts my_list into an iterator and then uses that iterator to loop through the items.

In summary:

An iterable is an object that can be looped over and provides an iterator when requested.

An iterator is an object that performs the actual iteration and maintains its internal state.

Both concepts work together to allow you to efficiently and effectively work with collections of data in Python.

Q.14 Difference between printing variable name and using print(variable name) ,why does ' ' appear on calling string variable name?

A. When you simply type the variable name in a Python interpreter or script without using the print() function, you're actually relying on the interpreter's default behavior of displaying the result of the last expression in the interactive session. This behavior varies depending on where you're running your Python code: it's more common in interactive environments like the standard Python interpreter, Jupyter notebooks, and some integrated development environments (IDEs).

When you use the print() function, you explicitly instruct Python to output the contents of the variable as text to the console or output window. This is the recommended way to display variable values when writing scripts or programs, especially when you want more control over the formatting and when you want to see the value of a variable at a specific point in your code.

Regarding the ' ' appearing when just calling the variable name, this typically happens when the variable contains a string. When you type the variable name directly, Python displays the string representation of the variable, which includes the single quotes ' ' to indicate that it's a string. For example:

```
variable = "Hello, world!"

variable  # This will display 'Hello, world!' (including the single quotes)
```

On the other hand, using print(variable) will print the actual contents of the string without the enclosing single quotes:

```
variable = "Hello, world!"

print(variable)  # This will print: Hello, world!
```

So, the difference between just typing the variable name and using print(variable) is that the former displays the string representation of the variable (including quotes for strings), while the latter displays the actual contents of the variable.

Q.15 Programs:

- Write a Python function to find the kth smallest element in a list.

```
def kth_smallest_el(lst, k):
   lst.sort()
   return lst[k-1]
nums = [1,2,4,3,5,4,6,9,2,1]
print("Original list:")
print(nums)
k = 1
for i in range(1, 11):
   print("kth smallest element in the said list, when k = ",k)
   print(kth_smallest_el(nums, k))
   k=k+1
```

- Write a Python program to find all the pairs in a list whose sum is equal to a given value.

```
def pairs_with_sum(lst, g_sum):
    complement_dict = {}
    pairs = []
    for num in lst:
        if g_sum - num in complement_dict:
            pairs.append((num, g_sum - num))
        else:
            complement_dict[num] = g_sum - num
    return pairs
```

- Find the Most Common Character in a String:

```
from collections import Counter

def most_common_char(s):

    char_count = Counter(s)

    most_common = char_count.most_common(1)

    return most_common[0][0] if most_common else None
```

- Write a Python program to convert a list into a nested dictionary of keys.

```
num_list = [1, 2, 3, 4]
new_dict = current = {}
for name in num_list:
    current[name] = {}
    current = current[name]
print(new_dict)
```

- Capitalize the First Letter of Each Word in a Sentence:

```
def capitalize_words(s):

    return ' '.join(word.capitalize() for word in s.split())
```

- Reverse Words in a Sentence While Maintaining Word Order:

```
def reverse_words_order(s):
    words = s.split()
    return ' '.join(words[::-1])
```

- Remove Special Characters from a String:

```
import re
def remove_special_characters(s):
    return re.sub(r'[^\w\s]', '', s)
```

- Find the Length of the Longest Substring Without Repeating Characters:

```
def longest_substring_without_repeating(s):
    seen = set()
    max_length = start = 0
    for end, char in enumerate(s):
        while char in seen:
            seen.remove(s[start])
            start += 1
        seen.add(char)
        max_length = max(max_length, end - start + 1)
    return max_length
```

- Check if a String is a Palindrome (Ignoring Non-Alphanumeric Characters):

```
def is_palindrome_ignore_nonalphanumeric(s):
    cleaned = ''.join(filter(str.isalnum, s))
    return cleaned == cleaned[::-1]
```

- Find the Longest Common Prefix Among Strings in a List:

```
def longest_common_prefix(strings):
    if not strings:
        return ""
    shortest = min(strings, key=len)
    for i, char in enumerate(shortest):
        if any(s[i] != char for s in strings):
            return shortest[:i]
```

- Combine Two Tuples Element-Wise:

```
def combine_tuples(t1, t2):
    return tuple(x + y for x, y in zip(t1, t2))
```

- Find Common Elements in Two Tuples:

```
def common_elements(t1, t2):
    return tuple(set(t1) & set(t2))
```

- Remove Common Elements from Two Sets:

```
def remove_common_elements(set1, set2):
    return set1 - set2
```

- Merge Two Dictionaries (Python 3.9+):

```
def merge_dicts(dict1, dict2):
    return {**dict1, **dict2}
```

- Find the Key with the Maximum Value in a Dictionary:

```
def key_with_max_value(d):
    return max(d, key=d.get)
```

- Count the Frequency of Characters in a String using a Dictionary:

```
def char_frequency(s):
    freq_dict = {}
    for char in s:
        freq_dict[char] = freq_dict.get(char, 0) + 1
    return freq_dict
```

NUMPY

EASY:

Q.1 What is NumPy, and why is it used in data analysis?

A. NumPy is a Python library for numerical computations, particularly array operations. It's essential for handling large datasets efficiently and performing mathematical operations.

Q.2 How are NumPy arrays better than Python's lists?

A:

- Python lists support storing heterogeneous data types whereas NumPy arrays can store data types of one nature itself. NumPy provides extra functional capabilities that make operating on its arrays easier which makes NumPy arrays advantageous in comparison to Python lists as those functions cannot be operated on heterogeneous data.
- NumPy arrays are treated as objects which results in minimal memory usage. Since Python keeps track of objects by creating or deleting them based on the requirements, NumPy objects are also treated the same way. This results in lesser memory wastage.
- NumPy arrays support multi-dimensional arrays.
- NumPy provides various powerful and efficient functions for complex computations on the arrays.

- NumPy also provides a variety of functions for BitWise Operations, String Operations, Linear Algebraic operations, Arithmetic operations etc. These are not provided on Python's default lists.

Q.3 What are ndarrays in NumPy?

A. ndarray object is the core of the NumPy package. It consists of n-dimensional arrays storing elements of the same data types and also has many operations that are done in compiled code for optimised performance. These arrays have fixed sizes defined at the time of creation. Following are some of the properties of ndarrays:

- When the size of ndarrays is changed, it results in a new array and the original array is deleted.
- The ndarrays are bound to store homogeneous data.
- They provide functions to perform advanced mathematical operations in an efficient manner.

Q.4 What are the ways of creating 1D, 2D and 3D arrays in NumPy?

A. Consider you have a normal python list. From this, we can create NumPy arrays by making use of the array function as follows:

- One-Dimensional array

  ```
  arr = [1,2,3,4]      #python list
  numpy_arr = np.array(arr)   #numpy array
  ```

- Two-Dimensional array

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

- Three-Dimensional array

  ```
  arr = [[[1,2,3,4],[4,5,6,7],[7,8,9,10]]]
  numpy_arr = np.array(arr)
  ```

Using the np.array() function, we can create NumPy arrays of any dimensions

Q.5 Programs:

- Write a program for creating an integer array with values belonging to the range 10 and 60

```
import numpy as np

arr = np.arange(10, 60)

print(arr)
```

- Write a NumPy program to create a 2-dimensional array of size 2 x 3 (composed of 4-byte integer elements), also print the shape, type and data type of the array.

```
import numpy as np

x = np.array([[2, 4, 6], [6, 8, 10]], np.int32)

print(type(x))

print(x.shape)

print(x.dtype)
```

- Write a NumPy program to create a new array of 3*5, filled with 2.

```
import numpy as np
#using no.full
x = np.full((3, 5), 2, dtype=np.uint)
print(x)
#using no.ones
y = np.ones([3, 5], dtype=np.uint) *2
print(y)
```

- Write a NumPy program to create an array of (3, 4) shapes, multiply every element value by 3 and display the result array.

```
import numpy as np
x= np.arange(12).reshape(3, 4)
print("Original array elements:")
print(x)
for a in np.nditer(x, op_flags=['readwrite']):
    a[...] = 3 * a
print("New array elements:")
print(x)
```

- Create an Identity Matrix:

```
import numpy as np

def create_identity_matrix(n):

    return np.eye(n)
```

- Generate an Array of Random Integers:

```
def random_integers_array(low, high, size):

    return np.random.randint(low, high, size)
```

- Calculate the Mean and Median of an Array:

```
def mean_and_median(arr):

    mean = np.mean(arr)

    median = np.median(arr)

    return mean, median
```

- Reshape an Array:

```
def reshape_array(arr, rows, cols):

    return arr.reshape(rows, cols)
```

MEDIUM:

Q.1 Explain the concept of data normalization and its significance in data analysis.

A. Data normalization is the process of scaling features to have similar ranges, which helps algorithms converge faster during training and prevents one feature from dominating others.

Q.2 How is np.mean() different from np.average() in NumPy?

A. np.mean() method calculates the arithmetic mean and provides additional options for input and results. For example, it has the option to specify what data types have to be taken, where the result has to be placed etc.

np.average() computes the weighted average if the weights parameter is specified. In the case of weighted average, instead of considering that each data point is contributing equally to the final average, it considers that some data points have more weightage than the others (unequal contribution).

Q.3 How do you multiply 2 NumPy array matrices?

A. We can make use of the dot() for multiplying matrices represented as NumPy arrays. This is represented in the code snippet below:

```
import numpy as np

# NumPy matrices

A = np.arange(15,24).reshape(3,3)

B = np.arange(20,29).reshape(3,3)

print("A: ",A)

print("B: ",B)


# Multiply A and B

result = A.dot(B)

print("Result: ", result)
```

Output

```
A:  [[15 16 17]
 [18 19 20]
 [21 22 23]]
B:  [[20 21 22]
 [23 24 25]
 [26 27 28]]
Result:  [[1110 1158 1206]
 [1317 1374 1431]
 [1524 1590 1656]]
```

Q.4 How is arr[:,0] different from arr[:,[0]]

A. arr[:,0] - Returns 0th index elements of all rows. In other words, return the first column elements.

```
 import numpy as np
arr = np.array([[1,2,3,4],[5,6,7,8]])
new_arr =arr[:,0]
print(new_arr)
```

Output:

```
 [1 5]
```

arr[:,[0]] - This returns the elements of the first column by adding extra dimension to it.

```
import numpy as np
arr = np.array([[1,2,3,4],[5,6,7,8]])
new_arr =arr[:,[0]]
print(new_arr)
```

Output:

```
[[1]
[5]]
```

Q.5 Explain broadcasting in NumPy with an example.

A. Broadcasting is a feature in NumPy that allows element-wise operations on arrays of different shapes. It automatically adjusts the shape of smaller arrays to match the shape of larger arrays during arithmetic operations.

Example:

```
a = np.array([1, 2, 3])
print("Array a:")
print(a)
# Adding a scalar to an array
scalar = 2
result = a + scalar
print("\nAdding a scalar to an array:")
print(result)
```

Output:

```
Array a:
[1 2 3]
Adding a scalar to an array:
[3 4 5]
```

In this example, broadcasting allows us to add a scalar value (2) to each element of the array a without explicitly looping through the array. The scalar is automatically "broadcasted" to match the shape of the array, and the addition is performed element-wise.

Q.6 Programs:

- Write a NumPy program to test whether any array element along a given axis evaluates to True.Note: 0 evaluates to False in NumPy.

```
import numpy as np

print(np.any([[False,False],[False,False]]))

print(np.any([[True,True],[True,True]]))

print(np.any([10, 20, 0, -50]))

print(np.any([10, 20, -50]))
```

- Write a NumPy program to find the indices of the maximum and minimum values along the given axis of an array.
  Original array: [1 2 3 4 5 6]
  Maximum Values: 5
  Minimum Values: 0

  Answer:

```
import numpy as np
x = np.array([1, 2, 3, 4, 5, 6])
print("Original array: ",x)
print("Maximum Values: ",np.argmax(x))
print("Minimum Values: ",np.argmin(x))
```

- Write a NumPy program to save a NumPy array to a text file.

```
import numpy as np
a = np.arange(1.0, 2.0, 36.2)
np.savetxt('file.out', a, delimiter=',')
```

- Write a NumPy program to find the memory size of a NumPy array.

```
import numpy as np
n = np.zeros((4,4))
print("%d bytes" % (n.size * n.itemsize))
```

- Write a NumPy program to create a contiguous flattened array.

  Original array:

```
[[10 20 30]
[20 40 50]]
New flattened array:
[10 20 30 20 40 50]
```

Answer:

```
import numpy as np
x = np.array([[10, 20, 30], [20, 40, 50]])
print("Original array:")
print(x)
y = np.ravel(x)
print("New flattened array:")
print(y)
```

- Calculate Dot Product of Two Matrices:

```
def dot_product(matrix1, matrix2):
    return np.dot(matrix1, matrix2)
```

- Find Unique Elements and Their Counts:

```
def unique_elements_and_counts(arr):
    unique, counts = np.unique(arr, return_counts=True)
    return dict(zip(unique, counts))
```

- Calculate the Element-Wise Square Root of an Array:

```
def elementwise_sqrt(arr):
    return np.sqrt(arr)
```

- Find the Index of the Maximum Value in an Array:

```
def index_of_max_value(arr):
    return np.argmax(arr)
```

- Create a Diagonal Matrix:

```
def diagonal_matrix(diagonal_values):
    return np.diag(diagonal_values)
```

- Calculate the Standard Deviation of an Array:

```
def array_std_deviation(arr):
    return np.std(arr)
```

- Find missing data in a given array.

```
import numpy as np
nums = np.array([[3, 2, np.nan, 1],
              [10, 12, 10, 9],
              [5, np.nan, 1, np.nan]])

print("Original array:")
print(nums)
print("\nFind the missing data of the said array:")
print(np.isnan(nums))
```

HARD:

Q.1 What do you understand about Vectorization in NumPy?

A. Vectorization in NumPy refers to the practice of performing element-wise operations on entire arrays or matrices without the need for explicit looping. It is a fundamental concept in NumPy and is a key reason why NumPy is widely used for numerical and scientific computations in Python.

When you perform operations on individual elements of NumPy arrays, those operations are automatically applied to all elements in parallel, taking advantage of low-level optimizations and efficient memory usage. This leads to more concise and efficient code compared to traditional Python loops.

Benefits of vectorization in NumPy:

- Performance: Vectorized operations are implemented using highly optimized C and Fortran code, making them much faster than equivalent operations using native Python loops.
- Readability: Vectorized code is often more concise and easier to read than explicit loops, making the codebase more maintainable.
- Efficiency: NumPy's vectorized operations are optimized for efficient memory usage, allowing you to process large datasets without consuming excessive memory.

Q.2 How is vstack() different from hstack() in NumPy?

A.Both methods are used for combining the NumPy arrays. The main difference is that the hstack method combines arrays horizontally whereas the vstack method combines arrays vertically.

For example, consider the below code.

```
import numpy as np

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

b = np.array([4,5,6])


# vstack arrays

c = np.vstack((a,b))

print("After vstack: \n",c)
```

# hstack arrays

d = np.hstack((a,b))

print("After hstack: \n",d)

The output of this code would be:

After vstack:

[[1 2 3]

[4 5 6]]

After hstack:

[1 2 3 4 5 6]

Notice how after the vstack method, the arrays were combined vertically along the column and how after the hstack method, the arrays were combined horizontally along the row.

Q.3 How is Vectorization related to Broadcasting in NumPy?

A. Vectorization involves delegating NumPy operations internally to optimized C language functions to result in faster Python code. Broadcasting refers to the methods that allow NumPy to perform array-related arithmetic operations. The size or shape of the arrays does not matter in this case. Broadcasting solves the problem of mismatched shaped arrays by replicating the smaller array along the larger array to ensure both arrays are having compatible shapes for NumPy operations. Performing Broadcasting before Vectorization helps to vectorize operations which support arrays of different dimensions.

- Write a NumPy program to swap rows and columns of a given array in reverse order.

```
import numpy as np
nums = np.array([[[1, 2, 3, 4],
        [0, 1, 3, 4],
        [90, 91, 93, 94],
        [5, 0, 3, 2]]])
print("Original array:")
print(nums)
print("\nSwap rows and columns of the said array in reverse order:")
new_nums = print(nums[::-1, ::-1])
print(new_nums)
```

- Write a NumPy program to multiply two given arrays of the same size element-by-element.

```
nums1 = np.array([[2, 5, 2],
          [1, 5, 5]])
nums2 = np.array([[5, 3, 4],
          [3, 2, 5]])
print("Array1:")
print(nums1)
print("Array2:")
print(nums2)
print("\nMultiply said arrays of same size element-by-element:")
print(np.multiply(nums1, nums2))
```

- Write a NumPy program to repeat all the elements three times of a given array of string

```
Original Array:
['Python' 'PHP' 'Java' 'C++']
New array:
['PythonPythonPython' 'PHPPHPPHP' 'JavaJavaJava' 'C++C++C++']
```

Answer:

```
x1 = np.array(['Python', 'PHP', 'Java', 'C++'], dtype=np.str)

print("Original Array:")

print(x1)

new_array = np.char.multiply(x1, 3)

print("New array:")

print(new_array)
```

- Write a NumPy program to remove the leading whitespaces of all the elements of a given array.

```
x = np.array([' python exercises ', ' PHP  ', ' java  ', '  C++'], dtype=np.str)

print("Original Array:")

print(x)

lstripped_char = np.char.lstrip(x)
```

```
print("\nRemove the leading whitespaces : ", lstripped_char)
```

- Write a NumPy program to replace "PHP" with "Python" in the element of a given array.

```
import numpy as np
x = np.array(['PHP Exercises, Practice, Solution'], dtype=np.str)
print("\nOriginal Array:")
print(x)
r = np.char.replace(x, "PHP", "Python")
print("\nNew array:")
print(r)
```

- Write a NumPy program to count a given word in each row of a given array of string values.

```
import numpy as np
str1 = np.array([['Python','NumPy','Exercises'],
            ['Python','Pandas','Exercises'],
            ['Python','Machine learning','Python']])
print("Original array of string values:")
print(str1)
print("\nCount 'Python' row wise in the above array of string values:")
print(np.char.count(str1, 'Python'))
```

- Write a NumPy program to split a given text into lines and split the single line into array values.

Sample output:

Original text:

```
01 V Debby Pramod
02 V Artemiy Ellie
```

03 V Baptist Kamal

04 V Lavanya Davide

05 V Fulton Antwan

06 V Euanthe Sandeep

07 V Endzela Sanda

08 V Victoire Waman

09 V Briar Nur

10 V Rose Lykos

Array from the said text:

```
[['01' 'V' 'Debby Pramod']
['02' 'V' 'Artemiy Ellie']
['03' 'V' 'Baptist Kamal']
['04' 'V' 'Lavanya Davide']
['05' 'V' 'Fulton Antwan']
['06' 'V' 'Euanthe Sandeep']
['07' 'V' 'Endzela Sanda']
['08' 'V' 'Victoire Waman']
['09' 'V' 'Briar Nur']
['10' 'V' 'Rose Lykos']]
```

Answer:

```
import numpy as np
student = """01	V	Debby Pramod
02	V	Artemiy Ellie
03	V	Baptist Kamal
04	V	Lavanya Davide
05	V	Fulton Antwan
06	V	Euanthe Sandeep
07	V	Endzela Sanda
08	V	Victoire Waman
09	V	Briar Nur
```

```
10	V	Rose Lykos"""

print("Original text:")
print(student)
text_lines = student.splitlines()
text_lines = [r.split('\t') for r in text_lines]
result = np.array(text_lines, dtype=np.str)
print("\nArray from the said text:")
print(result)
```

- Write a program to convert a string element to uppercase, lowercase, capitalise the first letter, title-case and swapcase of a given NumPy array.

```
import numpy as np

# Create Sample NumPy array

arr = np.array(['i', 'love', 'NumPy', 'AND', 'interviewbit'], dtype=str)

upper_case_arr = np.char.upper(arr)

lower_case_arr = np.char.lower(arr)

capitalize_case_arr = np.char.capitalize(arr)

titlecase_arr = np.char.title(arr)

swapcase_arr = np.char.swapcase(arr)
```

- Write a program to transform elements of a given string to a numeric string of 10 digits by making all the elements of a given string to a numeric string of 8 digits with zeros on the left.

```
import numpy as np

# Create Sample NumPy array

arr = np.array(['22', '9', '1234', '567', '89102'], dtype=str)

zeroes_filled_arr = np.char.zfill(arr, 8)

print("Transformed array: ")

print(zeroes_filled_arr)
```

- Write a program for inserting space between characters of all elements in a NumPy array.

```
import numpy as np

# Create Sample NumPy Array

arr = np.array(['i', 'love', 'NumPy', 'AND', 'interviewbit'], dtype=str)

transformed_arr = np.char.join(" ", arr)

print("Transformed Array: ")

print(transformed_arr)
```

- Write a program to add a border of zeros around the existing array.

```
import numpy as np

# Create NumPy arrays filled with ones
ones_arr = np.ones((4,4))

print("Transformed array:")
transformed_array = np.pad(ones_arr, pad_width=1, mode='constant',
constant_values=0)
print(transformed_array)
```

PANDAS

EASY:

Q.1 What are Pandas in Python?

A.Pandas is an open-source Python package that is most commonly used for data science, data analysis, and machine learning tasks. It is built on top of another library named Numpy. It provides various data structures and operations for manipulating numerical data and time series and is very efficient in performing various functions like data visualization, data manipulation, data analysis, etc.

Q.2 Mention the different types of Data Structures in Pandas?

A. Pandas have three different types of data structures. It is due to these simple and flexible data structures that it is fast and efficient.

- Series - It is a one-dimensional array-like structure with homogeneous data which means data of different data types cannot be a part of the same series. It can hold any data type such as integers, floats, and strings and its values are mutable i.e. it can be changed but the size of the series is immutable i.e. it cannot be changed.
- DataFrame - It is a two-dimensional array-like structure with heterogeneous data. It can contain data of different data types and the data is aligned in a tabular manner. Both size and values of DataFrame are mutable.
- Panel - The Pandas have a third type of data structure known as Panel, which is a 3D data structure capable of storing heterogeneous data but it isn't that widely used.

Q.3 What are the significant features of the pandas Library?

A. Pandas library is known for its efficient data analysis and state-of-the-art data visualization.

The key features of the panda's library are as follows:

- Fast and efficient DataFrame object with default and customized indexing.
- High-performance merging and joining of data.
- Data alignment and integrated handling of missing data.
- Label-based slicing, indexing, and subsetting of large data sets.
- Reshaping and pivoting of data sets.
- Tools for loading data into in-memory data objects from different file formats.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- Time Series functionality.

Q.4 Define Series in Pandas?

A. It is a one-dimensional array-like structure with homogeneous data which means data of different data types cannot be a part of the same series. It can hold any data type such as integers, floats, and strings and its values are mutable i.e. it can be changed but the size of the series is immutable i.e. it cannot be changed. By using a 'series' method, we can easily convert the list, tuple, and dictionary into a series. A Series cannot contain multiple columns.

Q.5 Define DataFrame in Pandas?

A. It is a two-dimensional array-like structure with heterogeneous data. It can contain data of different data types and the data is aligned in a tabular manner i.e. in rows and columns and the indexes with respect to these are called row index and column index respectively. Both size and values of DataFrame are mutable. The columns can be heterogeneous types like int and bool. It can also be defined as a dictionary of Series.

The syntax for creating a dataframe:

import pandas as pd

dataframe = pd.DataFrame(data, index, columns, dtype)

Here:

- data - It represents various forms like series, map, ndarray, lists, dict, etc.
- index - It is an optional argument that represents an index to row labels.
- columns - Optional argument for column labels.
- Dtype - It represents the data type of each column. It is an optional parameter.

Q.6 What are the different ways in which a series can be created?

A. There are different ways of creating a series in Pandas.

- Creating an empty Series: The simplest series that can be created is an empty series. The Series() function of Pandas is used to create a series of any kind.
- Creating a series from an array: Pandas is built on top of the Numpy library. In order to create a series from the NumPy array, we have to import the NumPy module and have to use numpy.array() the function.
- Creating a series from the array with an index: In order to create a series by exclusively providing an index instead of the default value we need to provide a list of elements to the index parameter with the same number of elements as given in the array.

- Creating a series from Lists: In order to create a series from a list, the first step is to create a list, and then we need to create a series from the given list.
- Creating a series from Dictionary: In order to create a series from the dictionary, the first step is to create a dictionary, and only then can we create a series using. The dictionary keys serve as indexes for the Series.
- Creating a series from Scalar value: In order to create a series from scalar value, an index must be provided. The value repeats itself to fit the length of the series or index given in general.

Q.7 What are the different ways in which a dataframe can be created?

A.

- Creating an empty dataframe: A basic DataFrame, which can be created is an Empty Dataframe. An Empty Dataframe is created just by calling a pandas.DataFrame() constructor.
- Creating a dataframe using List: DataFrame can be created using a single list or by using a list of lists.
- Creating DataFrame from dict of ndarray/lists: To create a DataFrame from dict of narray/list there are a few conditions to be met.
    - First, all the arrays must be of the same length.
    - Second, if the index is passed then the length index should be equal to the length of arrays.
    - Third, if no index is passed, then by default, the index will be in the range(n) where n is the length of the array.
- Create pandas dataframe from lists using a dictionary: Creating pandas DataFrame from lists using a dictionary can be achieved in multiple ways. We can create pandas DataFrame from lists using a dictionary by using pandas.DataFrame().
- Creating dataframe from series: In order to create a dataframe using series the argument to be passed in a DataFrame() function has to be a Series.
- Creating DataFrame from Dictionary of series: To create a DataFrame from Dict of series, a dictionary needs to be passed as an argument to form a DataFrame. The resultant index is the union of all the series of passed indexed.

Q.8 Explain the difference between pandas and NumPy.

A. Pandas is a library built on top of NumPy that provides data structures like DataFrame and Series, which are more suited for data analysis and manipulation.

Q.9 Programs:

- Read and Display Data from a CSV File:

```
import pandas as pd

def read_csv_file(filename):

    return pd.read_csv(filename)
```

- Filter Rows Based on a Condition:

```
def filter_rows(df, condition_column, condition_value):

    return df[df[condition_column] == condition_value]
```

- Calculate Summary Statistics of a DataFrame:

```
def summary_statistics(df):

    return df.describe()
```

- Write a Pandas program to create and display a one-dimensional array-like object containing an array of data using Pandas module.

```
import pandas as pd

ds = pd.Series([2, 4, 6, 8, 10])

print(ds)
```

- Write a Pandas program to convert a Panda module Series to Python list and it's type.

```
import pandas as pd
ds = pd.Series([2, 4, 6, 8, 10])
print("Pandas Series and type")
print(ds)
print(type(ds))
print("Convert Pandas Series to Python list")
print(ds.tolist())
print(type(ds.tolist()))
```

- Write a Pandas program to add, subtract, multiple and divide two Pandas Series.
  Sample Series: [2, 4, 6, 8, 10], [1, 3, 5, 7, 9]

  ```
  import pandas as pd
  ds1 = pd.Series([2, 4, 6, 8, 10])
  ds2 = pd.Series([1, 3, 5, 7, 9])
  ds = ds1 + ds2
  print("Add two Series:")
  print(ds)
  print("Subtract two Series:")
  ds = ds1 - ds2
  print(ds)
  print("Multiply two Series:")
  ds = ds1 * ds2
  print(ds)
  print("Divide Series1 by Series2:")
  ds = ds1 / ds2
  print(ds)
  ```

- Write a Pandas program to compare the elements of the two Pandas Series.
  Sample Series: [2, 4, 6, 8, 10], [1, 3, 5, 7, 10]

  ```
  import pandas as pd
  ds1 = pd.Series([2, 4, 6, 8, 10])
  ds2 = pd.Series([1, 3, 5, 7, 10])
  print("Series1:")
  print(ds1)
  print("Series2:")
  print(ds2)
  print("Compare the elements of the said Series:")
  print("Equals:")
  print(ds1 == ds2)
  print("Greater than:")
  print(ds1 > ds2)
  print("Less than:")
  print(ds1 < ds2)
  ```

MEDIUM:

Q.1 How can we create a copy of the series in Pandas?

A. We can create a copy of the series by using the following syntax: Series.copy(deep=True)

The default value for the deep parameter is set to True.

When the value ofdeep=True, the creation of a new object with a copy of the calling object's data and indices takes place. Modifications to the data or indices of the copy will not be reflected in the original object whereas when the value of deep=False, the creation of a new object will take place without copying the calling object's data or index i.e. only the references to the data and index will be copied. Any changes made to the data of the original object will be reflected in the shallow copy and vice versa.

Q.2 Explain Categorical data in Pandas?

A. Categorical data is a discrete set of values for a particular outcome and has a fixed range. Also, the data in the category need not be numerical, it can be textual in nature. Examples are gender, social class, blood type, country affiliation, observation time, etc. There is no hard and fast rule for how many values a categorical value should have. One should apply one's domain knowledge to make that determination on the data sets.

Q.3 Explain Reindexing in pandas along with its parameters?

A.Reindexing as the name suggests is used to alter the rows and columns in a DataFrame. It is also defined as the process of conforming a dataframe to a new index with optional filling logic. For missing values in a dataframe, the reindex() method assigns NA/NaN as the value. A new object is returned unless a new index is produced that is equivalent to the current one. The copy value is set to False. This is also used for changing the index of rows and columns in the dataframe.

Q.4 How do you handle missing data in pandas?

A. You can handle missing data in pandas using functions like dropna() to remove NaN values, fillna() to fill in missing values, or using techniques like interpolation.

Q.5 How can you efficiently group and aggregate data in pandas?

A. You can use the groupby() function in pandas to group data based on specific columns and then apply aggregate functions like sum(), mean(), etc.

Q.6 What is the purpose of the apply function in pandas?

A. The apply() function is used to apply a function along an axis of a DataFrame or Series. It's useful for applying custom operations to your data.

Q.7 Describe the differences between .loc[] and .iloc[] in pandas.

A. .loc[] and .iloc[] are both used for indexing and selecting data from a pandas DataFrame, but they have distinct purposes and ways of specifying the rows and columns you want to access:

.loc[] (Label-based Indexing):

- .loc[] is used for selecting data by label (row and column names).
- It accepts label-based indexing for both rows and columns.
- The syntax is df.loc[row_label, column_label], where row_label and column_label can be single labels, slices, or boolean arrays.

Example:

```
data = {'A': [1, 2, 3],
    'B': [4, 5, 6]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])
selected_data = df.loc['row2', 'B']  # Selects data in row 'row2' and column 'B'
print(selected_data)  # Output: 5
```

.iloc[] (Integer-based Indexing):

- .iloc[] is used for selecting data by integer position (0-based index).
- It accepts integer-based indexing for both rows and columns.
- The syntax is df.iloc[row_index, column_index], where row_index and column_index can be single integers, slices, or boolean arrays.

Example:

```
data = {'A': [1, 2, 3],
    'B': [4, 5, 6]}
df = pd.DataFrame(data)
selected_data = df.iloc[1, 1]  # Selects data in the second row and second column
print(selected_data)  # Output: 5
```

Key Differences:

.loc[] uses labels (row and column names) for indexing, while .iloc[] uses integer positions (0-based index).

.loc[] is inclusive of both the start and end indices/slices, while .iloc[] is inclusive of the start index and exclusive of the end index/slice.

When using .loc[], you can select rows and columns using Boolean arrays based on label conditions, while .iloc[] only allows integer-based indexing.

Q.8 How would you handle a situation where a DataFrame has duplicate rows?

A. To handle a situation where a DataFrame has duplicate rows, you can use the drop_duplicates() method. This method removes duplicate rows from the DataFrame, keeping only the first occurrence or a specified subset of columns.

Q.9 Give a brief description of time series in Panda?

A. Time series is an organized collection of data that depicts the evolution of a quantity through time. Pandas have a wide range of capabilities and tools for working with time-series data in all fields.

Supported by pandas:

- Analyzing time-series data from a variety of sources and formats.
- Create time and date sequences with preset frequencies.
- Date and time manipulation and conversion with timezone information.
- A time series is resampled or converted to a specific frequency.
- Calculating dates and times using absolute or relative time increments is one way to.

Q.10 Explain MultiIndexing in Pandas.

A.Multiple indexing is defined as essential indexing because it deals with data analysis and manipulation, especially for working with higher dimensional data. It also enables us to store and manipulate data with an arbitrary number of dimensions in lower-dimensional data structures like Series and DataFrame.

Q.11 How can we convert Series to DataFrame?

A.The conversion of Series to DataFrame is quite a simple process. All we need to do is to use the to_frame() function.

Syntax:

Series.to_frame(name=None)

Parameters:

- name: It accepts data objects as input. It is an optional parameter. The value of the name parameter will be equal to the name of the Series if it has any.
- Return Type: It returns the DataFrame after converting it from Series.

Q.12 How can we convert DataFrame to Numpy Array?

A.In order to convert DataFrame to a Numpy array we need to use DataFrame.to_numpy() method.

Syntax:

DataFrame.to_numpy(dtype=None, copy=False, na_value=_NoDefault.no_default)

Parameters:

- dtype: It accepts string or numpy.dtype. It is an optional parameter.
- copy: It accepts a boolean value whose default is set to False.
- na_value: It is an optional parameter. It specifies the value to use for missing values. The data type will depend on the data type of the column in the dataframe.

Q.13 What is TimeDelta?

A. Timedeltas are differences in times, expressed in different units, e.g. days, hours, minutes, and seconds. They can be both positive and negative.

Q.14 List some statistical functions in Python Pandas?

A. Some of the major statistical functions in Python Pandas are:

- sum() – It returns the sum of the values.
- min() – It returns the minimum value.
- max() – It returns the maximum value.
- abs() – It returns the absolute value.
- mean() – It returns the mean which is the average of the values.
- std() – It returns the standard deviation of the numerical columns.
- prod() – It returns the product of the values.

Q.15 How will you sort a DataFrame?

A.The function used for sorting in pandas is called DataFrame.sort_values(). It is used to sort a DataFrame by its column or row values. The function comes with a lot of parameters, but the most important ones to consider for sort are:

- by: It is used to specify the column/row(s) which are used to determine the sorted order. It is an optional parameter.
- axis: It specifies whether the sorting is to be performed for a row or column and the value is 0 and 1 respectively.
- ascending: It specifies whether to sort the dataframe in ascending or descending order. The default value is set to ascending. If the value is set as ascending=False it will sort in descending order.

Q.16 What's the difference between interpolate() and fillna() in Pandas?

A. fillna(): It fills the NaN values with a given number with which you want to substitute. It gives you the option to fill according to the index of rows of a pd.DataFrame or on the name of the columns in the form of a python dict.

interpolate(): It gives you the flexibility to fill the missing values with many kinds of interpolations between the values like linear, time, etc.

Q.17 Programs:

- Group and Aggregate Data:

```
def group_and_aggregate(df, group_column, aggregation_column, aggregation_function):
    return df.groupby(group_column)[aggregation_column].agg(aggregation_function)
```

- Merge Two DataFrames:

```
def merge_dataframes(df1, df2, merge_column):
   return pd.merge(df1, df2, on=merge_column)
```

- Calculate the Rolling Mean of a Column:

```
def rolling_mean(df, column, window):
   return df[column].rolling(window).mean()
```

- Fill Missing Values with the Mean:

```
def fill_missing_with_mean(df):
   return df.fillna(df.mean())
```

- Sort DataFrame Rows by a Column:

```
def sort_dataframe(df, sort_column):
   return df.sort_values(by=sort_column)
```

- Create a Pivot Table:

```
def create_pivot_table(df, index_column, columns_column, values_column,
aggregation_function):
   return df.pivot_table(index=index_column, columns=columns_column,
values=values_column, aggfunc=aggregation_function)
```

- Convert DataFrame to a Dictionary:

```
def dataframe_to_dict(df):
   return df.to_dict()
```

HARD:

Q.1 What is a pivot table in pandas, and how can you create one?

A. A pivot table in pandas is a data summarization tool that allows you to rearrange, reshape, and aggregate data from a DataFrame. You can create a pivot table using the pivot_table() function, specifying columns for rows, columns, and values, along with an aggregation function if needed.

Example: # Create a DataFrame

```
data = {'Date': ['2023-08-01', '2023-08-01', '2023-08-02', '2023-08-02'],
    'Product': ['A', 'B', 'A', 'B'],
    'Sales': [100, 200, 150, 250]}
df = pd.DataFrame(data)
pivot_table = df.pivot_table(index='Date', columns='Product', values='Sales', aggfunc='sum')
print(pivot_table)
```

Output:

```
Product     A    B
Date
2023-08-01  100  200
2023-08-02  150  250
```

Q.2 In pandas, how can you efficiently handle a large dataset that doesn't fit into memory?

A. You can efficiently handle a large dataset that doesn't fit into memory in pandas by reading and processing the data in smaller chunks using the chunksize parameter of functions like read_csv() or read_sql(). This allows you to work with the data incrementally without loading the entire dataset into memory at once.

Q.3 Explain the concept of method chaining in pandas. Provide an example.

Method chaining involves applying multiple methods in sequence to a DataFrame or Series. It results in more concise and readable code.

Example: # Create a DataFrame

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 22, 28],
        'Salary': [50000, 60000, 45000, 55000]}
df = pd.DataFrame(data)
# Method chaining example: Selecting and filtering data
result = df[df['Age'] > 25].sort_values('Salary', ascending=False).reset_index(drop=True)
print(result)
```

Output:

```
    Name  Age  Salary
0    Bob   30   60000
1  David   28   55000
```

Q.4 What is the purpose of the applymap() function in pandas?

A. The applymap() function in pandas is used to apply a specified function element-wise to every element in a DataFrame. It is particularly useful when you want to perform a custom operation on each individual element of the DataFrame. This function is available on both DataFrames and Series.

Syntax:

```
df.applymap(func)
```

Q.5 Describe the concept of "reshaping" in pandas. How can you reshape data using functions like pivot(), melt(), and stack()?

A. Reshaping in pandas refers to the process of transforming the structure of data within a DataFrame, typically by reorganizing rows and columns. This can help in better understanding and analysis of the data. Three commonly used functions for reshaping data in pandas are pivot(), melt(), and stack().

1. pivot() Function:
    - The pivot() function is used to reshape data by changing the layout of columns and rows.

- It takes columns as index, columns, and values arguments to specify the new arrangement.
- It aggregates data if there are multiple rows with the same index and column values.
- Example:

  pivot_table = df.pivot(index='Date', columns='Product', values='Sales')

2. melt() Function:
   - The melt() function is used to transform a wide DataFrame into a long one by "melting" columns into rows.
   - It gathers columns into two new columns: one for variable names and another for corresponding values.
   - Useful when you have multiple columns representing different categories or time periods.
   - Example:

     melted_df = df.melt(id_vars='Date', value_vars=['ProductA', 'ProductB'], var_name='Product', value_name='Sales')
3. stack() Function:
   - The stack() function is used to reshape data by "stacking" the specified columns into a new index level.
   - It is typically used to transform wide data into a long format, especially when dealing with multi-level index.
   - Example:

     stacked_df = df.set_index('Date').stack()

Q.6 How can you efficiently merge two DataFrames in pandas with a large number of rows and columns?

A. To efficiently merge two large DataFrames in pandas:

1. Sort and Index: Sort DataFrames by merging columns and set them as indexes.
2. Specify Merge Columns: Use on or left_on and right_on parameters.
3. Choose Merge Type: Decide inner, outer, left, or right merge.

4. Reduce Memory: Downcast data types, convert to categorical, remove unnecessary columns.
5. Use merge(): Prefer it over join() for more control.

Q.7 Explain the use of the cut() function in pandas.

A. The cut() function in pandas is used to segment and categorize continuous data into discrete bins or intervals. It's often used to convert a continuous variable into a categorical variable, allowing you to analyze and summarize data more effectively.

Syntax:

pd.cut(x, bins, labels=None)

Q.8 Why do we need groupby in pandas and what is the use of index in it?

A. In pandas, the groupby function is used to group data based on one or more columns in a DataFrame. When you use groupby and then apply an aggregation function (e.g., sum, mean, etc.), by default, pandas will use the grouping columns as the new index for the resulting DataFrame. This can be desirable in some cases, as it provides a clearer representation of the grouped data.

However, in certain scenarios, you might want to keep the original index of the DataFrame even after applying the groupby operation. This is where the index=False parameter comes into play. When you set index=False while performing a groupby and aggregation operation, pandas will reset the index of the resulting DataFrame and use the default integer index instead of the grouping columns as the index.

Q.9 Problems:

- Write a Pandas program to replace the 'qualify' column containing the values 'yes' and 'no' with True and False.

  Sample Python dictionary data and list labels:

  exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

  'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],

  'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1], 'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

Answer:

df = pd.DataFrame(exam_data , index=labels)

print("Original rows:")

print(df)

print("\nReplace the 'qualify' column contains the values 'yes' and 'no' with True and False:")

df['qualify'] = df['qualify'].map({'yes': True, 'no': False})

print(df)

- Write a Pandas program to get list from DataFrame column headers.

  import pandas as pd

  import numpy as np

  exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

  'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],

  'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

  'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

  labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

  df = pd.DataFrame(exam_data , index=labels)

  print(list(df.columns.values))

- Write a Pandas program to change the order of a DataFrame columns.

  import pandas as pd

  import numpy as np

  d = {'col1': [1, 4, 3, 4, 5], 'col2': [4, 5, 6, 7, 8], 'col3': [7, 8, 9, 0, 1]}

  df = pd.DataFrame(data=d)

print("Original DataFrame")

print(df)

print('After altering col1 and col3')

df = df[['col3', 'col2', 'col1']]

print(df)

- Write a Pandas program to count city wise number of people from a given of data set (city, name of the person).

Sample data:

```
      city  Number of people
0   California            4
1     Georgia             2
2  Los Angeles              4
```

Answer:

import pandas as pd

df1 = pd.DataFrame({'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

'city': ['California', 'Los Angeles', 'California', 'California', 'California', 'Los Angeles', 'Los Angeles', 'Georgia', 'Georgia', 'Los Angeles']})

g1 = df1.groupby(["city"]).size().reset_index(name='Number of people')

print(g1)

- Write a Pandas program to delete DataFrame row(s) based on given column value.

  Sample data:

  Original DataFrame

```
  col1  col2  col3
0    1    4    7
1    4    5    8
```

```
2    3    6    9

3    4    7    0

4    5    8    1

New DataFrame

   col1  col2  col3

0    1    4    7

2    3    6    9

3    4    7    0

4    5    8    1
```

Answer:

```
import pandas as pd
import numpy as np
d = {'col1': [1, 4, 3, 4, 5], 'col2': [4, 5, 6, 7, 8], 'col3': [7, 8, 9, 0, 1]}
df = pd.DataFrame(data=d)
print("Original DataFrame")
print(df)
df = df[df.col2 != 5]
print("New DataFrame")
print(df)
```

- Write a Pandas program to replace all the NaN values with Zero's in a column of a dataframe.

```
import pandas as pd

import numpy as np

exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
'Matthew', 'Laura', 'Kevin', 'Jonas'],

        'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],

        'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],

        'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

df = pd.DataFrame(exam_data)

print("Original DataFrame")
print(df)

df =  df.fillna(0)

print("\nNew DataFrame replacing all NaN with 0:")

print(df)
```

- Write a Pandas program to convert indexes in a column of the given dataframe.

```
import pandas as pd
import numpy as np
exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael',
'Matthew', 'Laura', 'Kevin', 'Jonas'],
    'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
    'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
    'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}
df = pd.DataFrame(exam_data)
print("Original DataFrame")
print(df)
print("\nAfter converting index in a column:")
df.reset_index(level=0, inplace=True)
print(df)
print("\nHiding index:")
print( df.to_string(index=False))
```

DATA VISUALIZATION

EASY:

Q.1 What is data visualization in Python?

A: Data visualization in Python refers to the process of creating graphical representations of data using Python programming language libraries such as Matplotlib, Seaborn, Plotly, etc. It allows for the exploration and communication of patterns and relationships within the data.

Q:2 How do you choose the best visualization for your data?

A: The best visualization for your data depends on the type of data you have, the question you are trying to answer, and the audience you are presenting to. Generally, simple visualizations like bar charts and scatter plots work well for basic analysis, while more complex visualizations like heatmaps and network graphs are better for advanced analysis.

Q:3 What are some best practices for designing effective visualizations?

A: Some best practices for designing effective visualizations include keeping it simple, using appropriate colors and fonts, labeling axes and titles clearly, providing context and explanations, and using appropriate scales and axes.

Q:4 What is Matplotlib?

A: Matplotlib is a Python plotting library that provides a wide range of 2D and 3D plots for visualizing data. It is widely used for creating static, interactive, and animated visualizations in Python.

Q.5 What is Seaborn?

A: Seaborn is a Python visualization library built on top of Matplotlib that provides a higher-level interface for creating statistical graphics. It is used for creating visually appealing and informative statistical graphics such as heatmaps, pair plots, etc.

Q:6 How do you install Matplotlib and Seaborn?

A: Matplotlib and Seaborn can be installed using pip, a Python package installer. To install Matplotlib, run "pip install matplotlib" in the command line. To install Seaborn, run "pip install seaborn" in the command line.

MEDIUM:

Q.1 Explain the difference between Matplotlib and Seaborn. When would you use one over the other?

A. Matplotlib and Seaborn are both popular Python libraries used for data visualization, but they have different focuses and design philosophies. Here's an explanation of when you might choose one over the other:

Matplotlib:

- Matplotlib is a versatile and foundational plotting library in Python. It provides a wide range of functionalities for creating static, interactive, and animated visualizations.
- It offers fine-grained control over plot elements, allowing you to customize every aspect of your plot.
- Matplotlib follows a low-level approach, meaning you have to write more code to achieve certain visualizations.
- It's highly customizable and is well-suited for creating complex plots from scratch.
- Matplotlib serves as the foundation for many other data visualization libraries, making it a fundamental tool for any data scientist or analyst.
- Use Matplotlib when you need precise control over plot customization and want to create complex or highly customized visualizations.

Seaborn:

- Seaborn is built on top of Matplotlib and provides a high-level interface for creating attractive and informative statistical graphics.
- It simplifies many common data visualization tasks and automatically applies aesthetically pleasing styles and color palettes.
- Seaborn is particularly well-suited for creating complex statistical plots, such as regression plots, pair plots, and distribution plots.
- It abstracts away much of the low-level customization that Matplotlib requires, making it faster and easier to generate visually appealing plots.
- Seaborn provides functions that directly work with Pandas DataFrames, making data manipulation and visualization seamless.
- Use Seaborn when you want to quickly create visually appealing statistical visualizations without delving into the details of plot customization.

In summary, Matplotlib is a powerful and flexible library that gives you granular control over plot creation, while Seaborn is focused on simplifying the creation of attractive statistical visualizations. Depending on your needs, you might choose Matplotlib when you require extensive customization or Seaborn when you want to create visually pleasing and informative statistical graphics more quickly. Additionally, you can also use both libraries together, leveraging Matplotlib for fine-tuned customization and Seaborn for streamlined statistical visualization.

Q:2 How do you create a scatter plot in Matplotlib?

A: A scatter plot can be created in Matplotlib using the "scatter" function. For example, the following code will create a scatter plot of x and y coordinates:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]

y = [2, 4, 1, 3, 5]

plt.scatter(x, y)

plt.show()
```

Q:3 How do you create a line plot in Matplotlib?

A: A line plot can be created in Matplotlib using the "plot" function. For example, the following code will create a line plot of x and y coordinates:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]

y = [2, 4, 1, 3, 5]

plt.plot(x, y)

plt.show()
```

Q:4 How do you customize the appearance of a plot in Matplotlib?

A: The appearance of a plot in Matplotlib can be customized using various functions such as "title", "xlabel", "ylabel", "xlim", "ylim", "legend", "grid", etc. For example, the following code will customize the appearance of a scatter plot:

```
import matplotlib.pyplot as plt
```

```
x = [1, 2, 3, 4, 5]

y = [2, 4, 1, 3, 5]

plt.scatter(x, y)

plt.title("Scatter Plot")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.xlim(0, 6)

plt.ylim(0, 6)

plt.legend(["Data"])

plt.grid(True)

plt.show()
```

Q:5 How do you create a bar plot in Matplotlib?

A: A bar plot can be created in Matplotlib using the "bar" function. For example, the following code will create a bar plot of x and y values:

```
import matplotlib.pyplot as plt

x = ["A", "B", "C", "D", "E"]

y = [2, 4, 1, 3, 5]

plt.bar(x, y)

plt.show()
```

Q.6 What is the difference between a histogram and a bar chart?

A. Both histograms and bar charts are graphical tools used to visualize data distributions. However, they are used for different types of data and serve different purposes. Here's a breakdown of the differences between a histogram and a bar chart:

Histogram:

- A histogram is used to visualize the distribution of continuous or quantitative data.
- It divides the data range into intervals (bins) and counts the frequency or number of data points that fall into each interval.

- The x-axis represents the data range (intervals) while the y-axis represents the frequency or count of data points in each interval.
- Histograms provide insight into the underlying distribution of data, showing patterns like skewness, central tendency, and spread.
- There are no gaps between the bars in a histogram, as it represents continuous data.
- Histograms are commonly used for analyzing data such as age distribution, income distribution, exam scores, etc.

Bar Chart:

- A bar chart is used to visualize categorical or qualitative data.
- It displays the frequency, count, or proportion of different categories or groups.
- The x-axis represents the categories or groups, while the y-axis represents the frequency, count, or proportion associated with each category.
- Bar charts are often used to compare different categories and identify patterns or trends among them.
- There are gaps between the bars in a bar chart, as it represents distinct categories or groups.
- Bar charts are commonly used for comparing items like sales by product, population by region, survey responses, etc.

In summary, the key distinction between a histogram and a bar chart lies in the type of data they are used to visualize. Histograms are used for continuous data distributions, while bar charts are used for categorical data comparisons

Q.7 How do you decide whether to use a bar chart or a line chart?

A: A bar chart is used to compare categorical or discrete data, while a line chart is used to show trends or changes over time. When deciding which to use, consider the type of data you have and the question you are trying to answer.

Q.8 Programs:

- Create a line plot using Matplotlib or Seaborn to visualize the trend of monthly sales for a year. Use random data to simulate the sales values.

  Sample answer:
  #Line Plot - Visualizing Monthly Sales Trend:

  ```
  import matplotlib.pyplot as plt
  import numpy as np
  ```

```
# Simulate random monthly sales data for a year
months = np.arange(1, 13)
sales = np.random.randint(50000, 150000, size=12)

# Create a line plot
plt.figure(figsize=(8, 5))
plt.plot(months, sales, marker='o')
plt.title('Monthly Sales Trend')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.xticks(months)
plt.grid(True)
plt.show()
```

- Generate a scatter plot using Matplotlib or Seaborn to show the relationship between two continuous variables, such as age and income, from a given dataset.

  Sample Answer:

```
# Scatter Plot - Relationship between Age and Income:
import matplotlib.pyplot as plt
import numpy as np

# Generate random data for age and income
np.random.seed(0)
age = np.random.randint(18, 65, size=100)
income = np.random.randint(20000, 100000, size=100)

# Create a scatter plot
plt.figure(figsize=(8, 5))
plt.scatter(age, income, color='blue', alpha=0.7)
plt.title('Scatter Plot: Age vs. Income')
plt.xlabel('Age')
plt.ylabel('Income')
plt.grid(True)
plt.show()
```

- Create a bar chart using Matplotlib or Seaborn to display the top 10 countries with the highest GDP. Use a dataset containing country names and GDP values.

  Sample Answer:

```
# Bar Chart - Top 10 Countries by GDP:
```

```
import matplotlib.pyplot as plt

# Sample data: Country names and GDP values
countries = ['USA', 'China', 'Japan', 'Germany', 'India', 'UK', 'France', 'Brazil', 'Italy',
'Canada']
gdp_values = [21393781, 14342903, 5081779, 3845714, 2712052, 2622434, 2582507,
2061212, 1934241, 1803650]

# Create a bar chart
plt.figure(figsize=(10, 6))
plt.bar(countries, gdp_values, color='green')
plt.title('Top 10 Countries by GDP')
plt.xlabel('Country')
plt.ylabel('GDP ($ Trillion)')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- Generate a histogram using Matplotlib or Seaborn to visualize the distribution of exam scores. Use a dataset containing student names and their scores.

  Sample Answer:
  # . Histogram - Exam Score Distribution:

```
import matplotlib.pyplot as plt

import numpy as np

# Simulate random exam scores for students

np.random.seed(0)

scores = np.random.normal(75, 10, size=200)

# Create a histogram

plt.figure(figsize=(8, 5))

plt.hist(scores, bins=10, edgecolor='black', alpha=0.7)

plt.title('Exam Score Distribution')

plt.xlabel('Score')

plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

- Create a pie chart using Matplotlib or Seaborn to show the percentage distribution of different types of expenses in a monthly budget.

  Sample Answer:
  # Pie Chart - Monthly Expenses Distribution:

```
import matplotlib.pyplot as plt
# Sample data: Expense categories and percentages
categories = ['Rent', 'Food', 'Transport', 'Entertainment', 'Utilities']
percentages = [30, 20, 15, 10, 25]
# Create a pie chart
plt.figure(figsize=(8, 8))
plt.pie(percentages, labels=categories, autopct='%1.1f%%', startangle=140,
colors=plt.cm.Paired.colors)
plt.title('Monthly Expenses Distribution')
plt.axis('equal')  # Equal aspect ratio ensures a circular pie
plt.show()
```

- Generate a box plot using Matplotlib or Seaborn to compare the distribution of heights among different age groups. Use a dataset with age and height values.

  Sample Answer:
  # Box Plot - Height Distribution by Age Group:

```
import matplotlib.pyplot as plt
import numpy as np

# Simulate random height data for different age groups
np.random.seed(0)
heights = np.random.normal(160, 10, size=300)
age_groups = np.repeat(['18-25', '26-35', '36-45'], 100)

# Create a box plot
plt.figure(figsize=(8, 5))
plt.boxplot([heights[age_groups == '18-25'], heights[age_groups == '26-35'],
heights[age_groups == '36-45']],
        labels=['18-25', '26-35', '36-45'], notch=True, patch_artist=True)
```

```
plt.title('Height Distribution by Age Group')
plt.xlabel('Age Group')
plt.ylabel('Height (cm)')
plt.grid(True)
plt.show()
```

- Generate a pair plot using Seaborn to explore the relationships between multiple numerical variables in a dataset. Include color differentiation based on a categorical variable.

  Sample Answer:

```
# Pair Plot - Relationships between Numerical Variables:
import seaborn as sns
import pandas as pd
import numpy as np

# Generate random data with multiple numerical variables and a categorical variable
np.random.seed(0)
data = pd.DataFrame({
    'A': np.random.normal(0, 1, 100),
    'B': np.random.normal(1, 2, 100),
    'C': np.random.normal(2, 3, 100),
    'Category': np.random.choice(['X', 'Y'], size=100)
})

# Create a pair plot with color differentiation based on 'Category'
sns.pairplot(data, hue='Category', diag_kind='kde')
plt.suptitle('Pair Plot of Numerical Variables with Color Differentiation', y=1.02)
plt.show()
```

HARD:

Q.1 Describe the concept of a "figure" and a "subplot" in Matplotlib. How would you create a figure with multiple subplots?

A. In Matplotlib, a "figure" is a top-level container that represents the entire graphic window or canvas. It serves as the overall container for one or more "subplots," which are smaller plots or charts placed within the figure. Subplots allow you to display multiple plots in a single figure, making it easier to compare and analyze different visualizations together.

Here's how you would create a figure with multiple subplots using Matplotlib:

```
import matplotlib.pyplot as plt

# Create a figure with 2 rows and 2 columns of subplots

# The number 221 means 2 rows, 2 columns, and subplot 1 (top-left)

plt.figure(figsize=(10, 6))  # Create a new figure

# Create the first subplot (top-left)

plt.subplot(2, 2, 1)  # 2 rows, 2 columns, subplot 1

plt.plot([0, 1], [0, 1])  # Example plot

# Create the second subplot (top-right)

plt.subplot(2, 2, 2)  # 2 rows, 2 columns, subplot 2

plt.plot([0, 1], [0, 2])  # Example plot

# Create the third subplot (bottom-left)

plt.subplot(2, 2, 3)  # 2 rows, 2 columns, subplot 3

plt.plot([0, 1], [0, 3])  # Example plot

# Create the fourth subplot (bottom-right)

plt.subplot(2, 2, 4)  # 2 rows, 2 columns, subplot 4

plt.plot([0, 1], [0, 4])  # Example plot

# Adjust layout to prevent subplot overlapping

plt.tight_layout()

# Show the figure

plt.show()
```

Q:2 What is fidelity in data visualization?

A: Fidelity refers to the degree of accuracy and precision in a visualization, or how closely the visualization represents the underlying data.

Q.3 Explain the concept of "tidy data" and why it's important in data analysis.

A. Tidy data is a structured format where each variable forms a column, each observation forms a row, and each type of observational unit forms a table. It simplifies data manipulation and analysis.

Q.4 Define the term 'Data Wrangling in Data Analytics.

A. Data Wrangling is the process wherein raw data is cleaned, structured, and enriched into a desired usable format for better decision making. It involves discovering, structuring, cleaning, enriching, validating, and analyzing data. This process can turn and map out large amounts of data extracted from various sources into a more useful format. Techniques such as merging, grouping, concatenating, joining, and sorting are used to analyze the data. Thereafter it gets ready to be used with another dataset.

Q.5 What are the various steps involved in any analytics project?

A. The various steps involved in any common analytics projects are as follows:

- Understanding the Problem: Understand the business problem, define the organizational goals, and plan for a lucrative solution.
- Collecting Data: Gather the right data from various sources and other information based on your priorities.
- Cleaning Data: Clean the data to remove unwanted, redundant, and missing values, and make it ready for analysis.
- Exploring and Analyzing Data: Use data visualization and business intelligence tools to analyze data.
- Interpreting the Results: Interpret the results to find out hidden patterns, future trends, and gain insights.

Q.6 What are the common problems that data analysts encounter during analysis?

A. The common problems steps involved in any analytics project are:

- Handling duplicate
- Collecting the meaningful right data and the right time
- Handling data purging and storage problems
- Making data secure and dealing with compliance issues

Q.7 What is the significance of Exploratory Data Analysis (EDA)?

- Exploratory data analysis (EDA) helps to understand the data better.
- It helps you obtain confidence in your data to a point where you're ready to engage a machine learning algorithm.
- It allows you to refine your selection of feature variables that will be used later for model building.
- You can discover hidden trends and insights from the data.

Q.8 Programs:

- You are given a dataset containing information about monthly sales for different products. Each row represents a sale with columns 'Product', 'Date', and 'Amount'. Write code to:

  a) Calculate the total sales amount for each product.

  b) Transform the data to create a new DataFrame with products as columns and months as index, showing the total sales amount for each month and product.

  c) Create a line plot to visualize the sales trend over months for the top three products.

  Answer:

```
import pandas as pd

import matplotlib.pyplot as plt

# Sample data

data = {

  'Product': ['A', 'B', 'C', 'A', 'B', 'C'],

  'Date': ['2023-01', '2023-01', '2023-01', '2023-02', '2023-02', '2023-02'],

  'Amount': [100, 150, 200, 120, 160, 180]
}

# Create DataFrame

df = pd.DataFrame(data)

# a) Calculate total sales amount for each product

total_sales = df.groupby('Product')['Amount'].sum()

# b) Transform data for monthly sales per product

monthly_sales = df.pivot_table(index='Date', columns='Product', values='Amount',
aggfunc='sum')

# c) Create line plot for top three products

top_products = total_sales.nlargest(3).index

monthly_sales[top_products].plot(kind='line')

plt.title('Monthly Sales Trend for Top Products')

plt.xlabel('Month')

plt.ylabel('Sales Amount')

plt.legend(title='Product')

plt.grid(True)

plt.show()
```

- You have a dataset containing information about customer ratings for different categories. Each row represents a rating with columns 'Category', 'Customer', and 'Rating'. Write code to:

  a) Calculate the average rating for each category.

  b) Transform the data to create a new DataFrame with categories as columns and customers as index, showing the average rating for each category and customer.

  c) Create a grouped box plot to visualize the distribution of ratings across different categories.

  Answer:

```
import pandas as pd
import matplotlib.pyplot as plt

# Sample data

data = {

    'Category': ['A', 'B', 'C', 'A', 'B', 'C'],

    'Customer': ['C1', 'C2', 'C3', 'C1', 'C2', 'C3'],

    'Rating': [4, 5, 3, 2, 4, 5]

}

# Create DataFrame

df = pd.DataFrame(data)


# a) Calculate average rating for each category

average_rating = df.groupby('Category')['Rating'].mean()


# b) Transform data for average ratings per category per customer

avg_ratings = df.pivot_table(index='Customer', columns='Category', values='Rating',
aggfunc='mean')


# c) Create grouped box plot

avg_ratings.boxplot(grid=False)

plt.title('Distribution of Ratings by Category')

plt.ylabel('Rating')

plt.xlabel('Category')

plt.show()
```