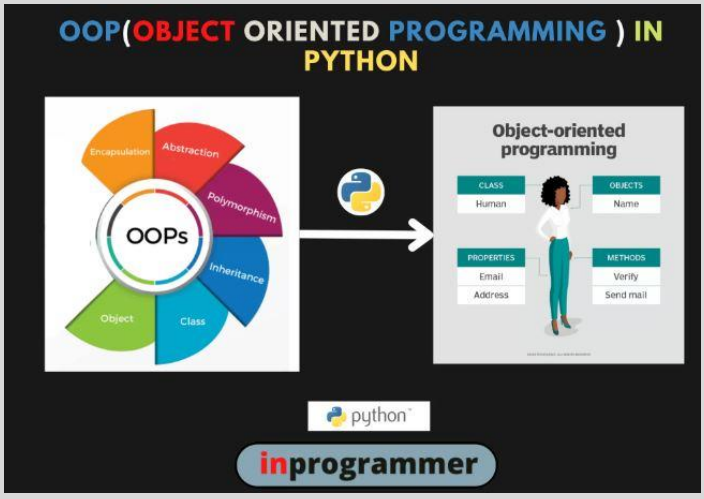
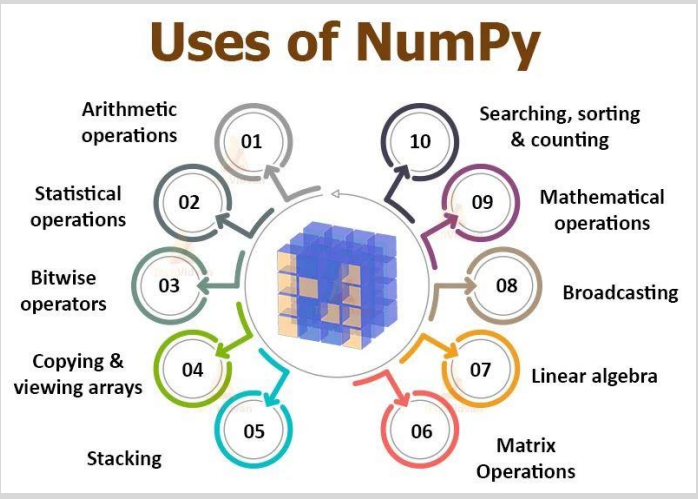
Machine Learning Part 04

01. Matrix Plot in seaborn
02. Sets interview Questions
03. Dictionary Interview questions
04. Tuples Interview Questions
05. List Interview Questions
06. Python Use Cases Interview Questions
07. OOP (Python) Interview Questions
08. Pandas Interview Questions (Part 1)
09. Pandas interview Questions (Part-2)
10. NumPy Interview Questions (Part 1)
11. NumPy interview Questions (Part 2)
12. Matplotlib Interview Questions
13. Python Coding Question
14. Pattern programming Questions in Python (Part 1)
15. Python Programming Questions (Part 2)





Matrix Plot

In Seaborn, both `heatmap` and `clustermap` functions are used for visualizing matrices, but they serve slightly different purposes.

1. **Heatmap:**
   - The `heatmap` function is used to plot rectangular data as a color-encoded matrix. It is essentially a 2D representation of the data where each cell is colored based on its value.
   - Heatmaps are useful for visualizing relationships and patterns in the data, making them suitable for tasks such as correlation matrices or any other situation where you want to visualize the magnitude of a phenomenon.

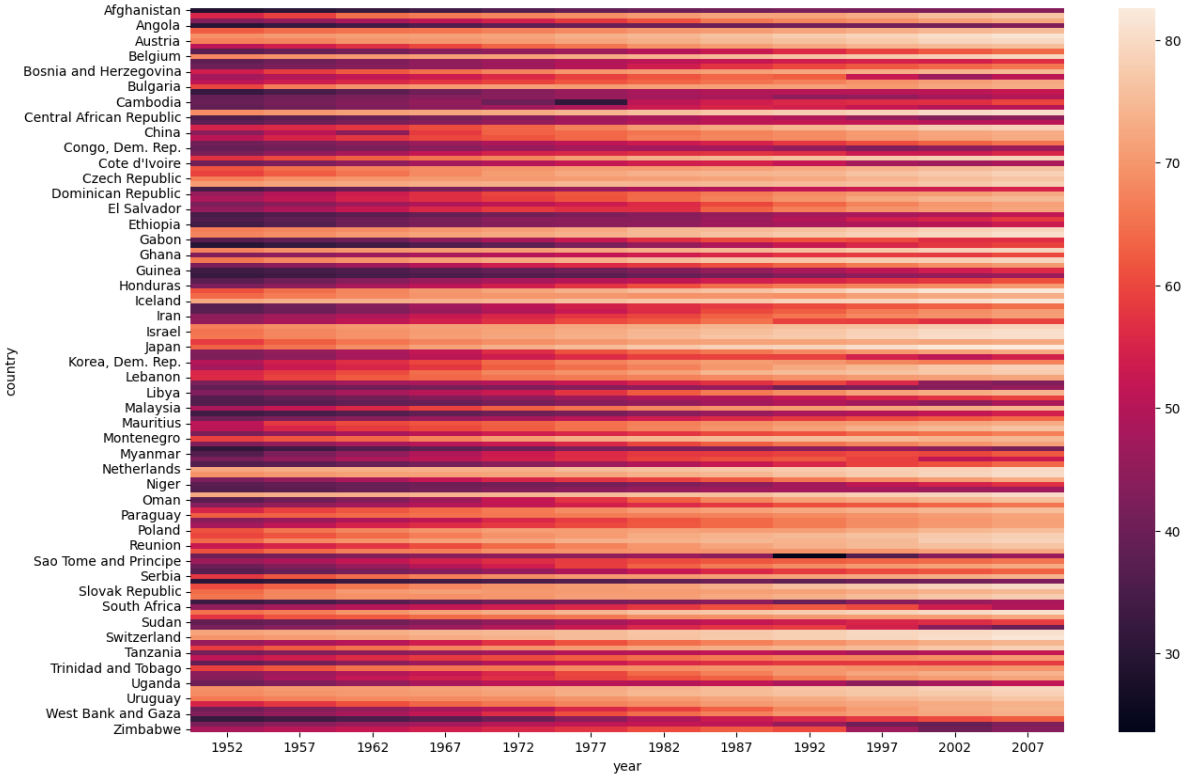
1. **Clustermap:**
   - The `clustermap` function, on the other hand, not only visualizes the matrix but also performs hierarchical clustering on both rows and columns to reorder them based on similarity.
   - It is useful when you want to identify patterns not only in the individual values of the matrix but also in the relationships between rows and columns.

In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
```

In [ ]:
```
gap = px.data.gapminder()
```

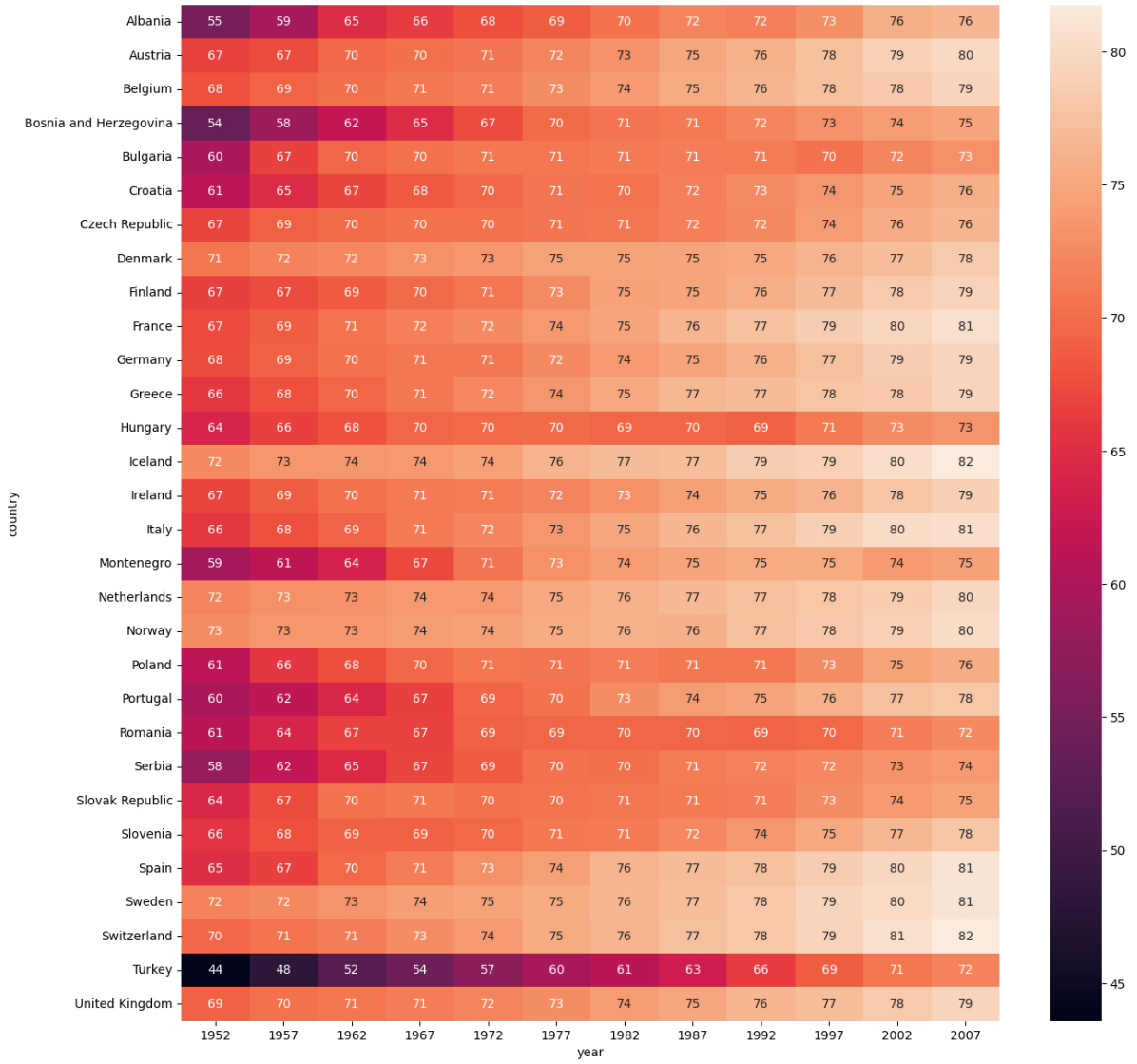
In [ ]:
```
# Heatmap

# Plot rectangular data as a color-encoded matrix
temp_df = gap.pivot(index='country',columns='year',values='lifeExp')

# axes level function
plt.figure(figsize=(15,10))
sns.heatmap(temp_df)
```

Out[ ]:
```
<Axes: xlabel='year', ylabel='country'>
```



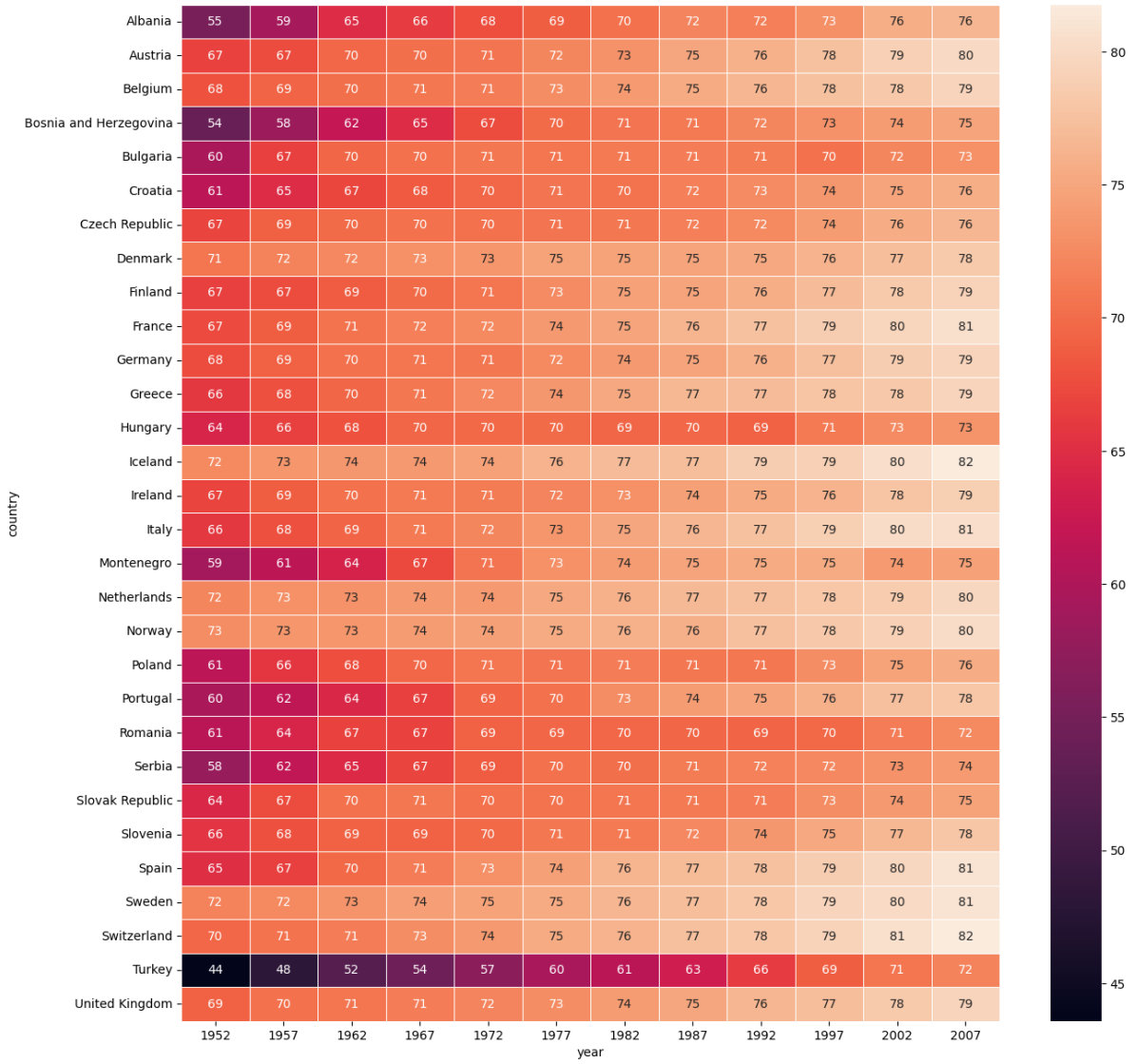
In [ ]:
```
# annot
temp_df = gap[gap['continent'] == 'Europe'].pivot(index='country',columns='year',va

plt.figure(figsize=(15,15))
sns.heatmap(temp_df,annot=True)
```

Out[ ]:
```
<Axes: xlabel='year', ylabel='country'>
```



In [ ]:
```
# linewidth
# annot
temp_df = gap[gap['continent'] == 'Europe'].pivot(index='country',columns='year',va

plt.figure(figsize=(15,15))
sns.heatmap(temp_df,annot=True,linewidth=0.5)
```

Out[ ]:
```
<Axes: xlabel='year', ylabel='country'>
```



In [ ]:
```
# cmap
# annot
temp_df = gap[gap['continent'] == 'Europe'].pivot(index='country',columns='year',va

plt.figure(figsize=(15,15))
sns.heatmap(temp_df,annot=True,linewidth=0.5, cmap='summer')
```

Out[ ]:
```
<Axes: xlabel='year', ylabel='country'>
```

In [ ]:
```
# Clustermap

# Plot a matrix dataset as a hierarchically-clustered heatmap.

# This function requires scipy to be available.

iris = px.data.iris()
iris
```

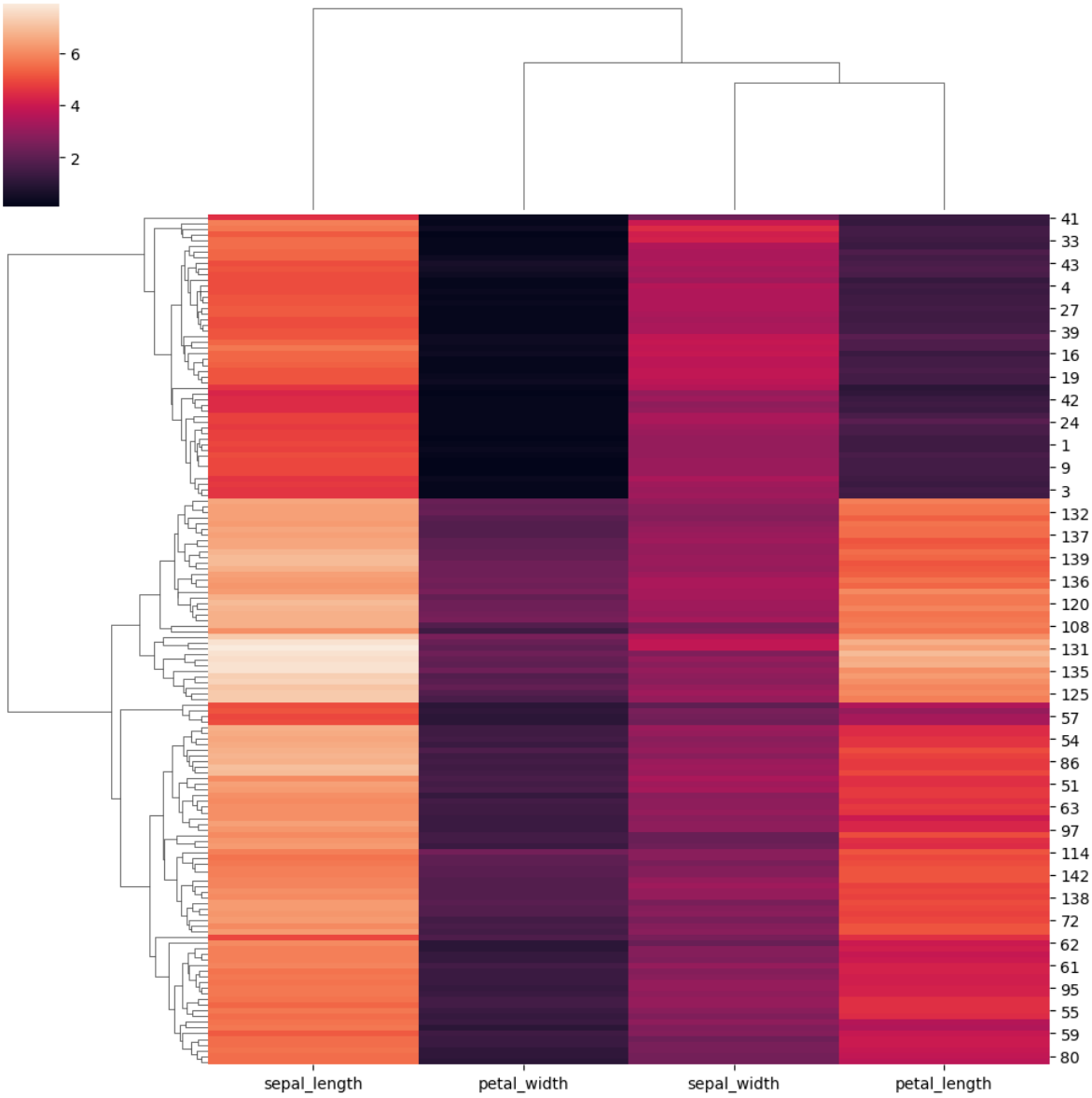
Out[ ]:

	sepal_length	sepal_width	petal_length	petal_width	species	species_id
0	5.1	3.5	1.4	0.2	setosa	1
1	4.9	3.0	1.4	0.2	setosa	1
2	4.7	3.2	1.3	0.2	setosa	1
3	4.6	3.1	1.5	0.2	setosa	1
4	5.0	3.6	1.4	0.2	setosa	1
...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica	3
146	6.3	2.5	5.0	1.9	virginica	3
147	6.5	3.0	5.2	2.0	virginica	3
148	6.2	3.4	5.4	2.3	virginica	3
149	5.9	3.0	5.1	1.8	virginica	3

150 rows × 6 columns

In [ ]:
```
sns.clustermap(iris.iloc[:,[0,1,2,3]])
```

Out[ ]:
```
<seaborn.matrix.ClusterGrid at 0x7a3bfb6797b0>
```


6
4
2
41
33
43
4
27
39
16
19
42
24
1
9
3
132
137
139
136
120
108
131
135
125
57
54
86
51
63
97
114
142
138
72
62
61
95
55
59
80
sepal_length
petal_width
sepal_width
petal_length

Basic 5 Interview Questions on Python

1. How do you reverse a string in Python?

In [ ]:
```
# if you have a string s
s = "Hello World"

def reverse_the_string(s):
    return s[::-1]

reverse_the_string(s)
```

Out[ ]:
```
'dlroW olleH'
```

2. What is the difference between a list and a Dictionary?

1. List

• A mutable data structure. • Can store heterogeneous data types. • Uses an index-based data structure. • Useful for storing multiple values of the same data type.

2. Dictionary

• A key-value data structure. • A mutable data structure. • Useful for storing student data, customer details, etc.. Here are some other data structures in Python: Arrays, Strings, Queues, Stacks, Trees, Linked lists, Graphs.

3. Write a function that checks if a given word is a palindrome.

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

4. How can you remove duplicates from a list?

In [ ]:
```
l = [1,2,2,4,4,5,6]

def remove_duplicates(l):
    return list(set(l))

remove_duplicates(l)
```

Out[ ]:
```
[1, 2, 4, 5, 6]
```

`5. What is a lambda function? Provide an example.`

In Python, a lambda function is a small, anonymous function defined using the `lambda` keyword. Lambda functions are also sometimes referred to as anonymous functions or lambda expressions. The primary purpose of lambda functions is to create small, one-time-use functions without formally defining a full function using the `def` keyword.

In [ ]:
```
add = lambda x, y: x + y
print(add(2, 3))
```

```
5
```

In this example, the lambda function takes two arguments (`x` and `y`) and returns their sum. It is equivalent to the following regular function:

In [ ]:
```
def add(x, y):
    return x + y
```

Lambda functions are often used for short, simple operations, especially in situations where you need to pass a function as an argument to another function

Interview Questions

`1.What does the else clause in a loop do?`

- The else clause in a loop is executed when the loop finishes execution (i.e., when the loop condition becomes False). It won't execute if the loop was exited using a break statement.

In [ ]:
```
for i in range(5):
    print(i)
else:
    print("Loop finished")
```

```
0
1
2
3
4
Loop finished
```

`2.What is the purpose of the pass statement in Python?`

- The pass statement is a no-op (does nothing). It's used as a placeholder where syntactically some code is required, but you don't want to execute any command or code.

`3.How do you retrieve all the keys, values, and items from a dictionary?`

In [ ]:
```
d = {'name':'John','age':24,}

print(d.keys())
print(d.values())
print(d.items())
```

```
dict_keys(['name', 'age'])
dict_values(['John', 24])
dict_items([('name', 'John'), ('age', 24)])
```

In [ ]:
```
# how can you extract only john from dict?
d['name']
```

Out[ ]:
```
'John'
```

`4.How can you achieve inheritance in Python?`

- Inheritance is achieved by defining a new class, derived from an existing class. The derived class inherits attributes and behaviors of the base class and can also have additional attributes or behaviors.

In [ ]:
```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
```

```
    def speak(self):
        return "Woof"
```

In [ ]:
```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks")

# Child class inheriting from Animal
class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows")

# Creating instances of the classes
animal = Animal("Generic Animal")
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Calling methods
animal.speak()  # Output: Generic Animal makes a sound
dog.speak()     # Output: Buddy barks
cat.speak()     # Output: Whiskers meows
```

```
Generic Animal makes a sound
Buddy barks
Whiskers meows
```

In this example:

- The Animal class is the parent class with a **init** method and a speak method.
- The Dog and Cat classes are child classes that inherit from the Animal class. They override the speak method to provide their own implementation.
- Instances of Dog and Cat can access both the attributes and methods of the Animal class, and they can also have their own additional attributes and methods.

`5.What is the __str__ method in a class and when is it used?`

- The **str** method is a special method that should return a string representation of the object. It's invoked by the built-in str() function and by the print() function when outputting the object.

In [ ]:
```
class Person:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"Person named {self.name}"

p = Person("Alice")
print(p)
```

```
Person named Alice
```

Sets in Python - Interview Questions

`1.Can you explain the concept of sets in Python?`

- In Python, a set is an **unordered** and **mutable** collection of unique elements.
- The key characteristics of sets are their ability to store distinct values and the absence of any defined order among the elements. Sets are defined using curly braces {}, and elements are separated by commas
- The uniqueness property of sets ensures that **duplicate elements are automatically eliminated**. Sets support a variety of operations such as union, intersection, difference, and symmetric difference, making them versatile for tasks involving the comparison and manipulation of collections.
- it's important to note that while sets themselves are **mutable** (elements can be added or removed), elements within a set must be of immutable data types (e.g., numbers, strings, or tuples). (note-If mutability is not required, Python offers an immutable version of sets called "frozensets," created using the frozenset() constructor.)

`2.How do you create an empty set in Python, and what is the key difference between an empty set and an empty dictionary?`

In [ ]:
```
# if we use curley braces for empty set then

s = {}

print(type(s))
```

```
<class 'dict'>
```

but it gives type is Dictionary so for the creation of empty set we use differnet Syntax

In [ ]:
```
# create empty set

s = set()

print(type(s))
```

```
<class 'set'>
```

Use Cases:

- Empty Set:

1. Suitable when there is a need for a collection with distinct and unordered elements.
2. Ideal for tasks involving set operations such as union, intersection, and difference.

- Empty Dictionary:

1. Used when data needs to be stored and retrieved based on specific keys.
2. Essential for scenarios where a mapping between keys and values is necessary.

In summary, an empty set is a collection of unique and unordered elements created using the set() constructor, while an empty dictionary is a data structure used for storing key-value

pairs, created using the {} notation or the dict() constructor. Understanding the intended purpose and structure of each is crucial in choosing the appropriate data structure for a given task.

`3.If you have an empty set, how would you add elements using a built-in function in Python?`

In [ ]:
```
# Create an empty set
my_set = set()

# Add elements to the set using the add() method
my_set.add(1)
my_set.add('hello')
my_set.add(3.14)

# Display the updated set
print(my_set)
```

```
{'hello', 1, 3.14}
```

`4.If you have a tuple and a set, which one is faster in terms of performance, and why?`

The performance difference between a tuple and a set in Python is influenced by their inherent characteristics and use cases.

1. **Access Time:**
   - **Tuple:**
     - Tuples are generally faster for element access since they are indexed and ordered.
     - Accessing elements in a tuple has a constant time complexity O(1).
   - **Set:**
     - Sets are unordered collections, and they do not support indexing.
     - Accessing elements in a set involves a hash lookup, resulting in an average time complexity of O(1) but with some variability.
2. **Uniqueness and Mutability:**
   - **Tuple:**
     - Tuples are immutable, meaning their elements cannot be changed after creation.
     - Ideal for scenarios where data should remain constant throughout the program.
   - **Set:**
     - Sets are mutable and designed for efficient membership tests and unique element storage.
     - Suitable for scenarios where the collection of distinct and unordered elements needs to be dynamically modified.
3. **Use Cases:**

- **Tuple:**
  - Prefer tuples when the order of elements matters, and the data should remain constant.
  - Useful in scenarios like representing coordinates (x, y, z) or holding a fixed set of values.
- **Set:**
  - Opt for sets when dealing with collections of unique elements and the order is not significant.
  - Useful for tasks involving set operations like union, intersection, and difference.

4. **Memory Overhead:**

- **Tuple:**
  - Tuples typically have lower memory overhead compared to sets, as they store only the elements.
- **Set:**
  - Sets use additional memory for hash tables and other internal structures, which can lead to higher memory consumption.

In conclusion, the choice between a tuple and a set depends on the specific requirements of the task. If fast element access, order preservation, and immutability are crucial, a tuple is preferred. On the other hand, if the focus is on efficient membership tests, unique element storage, and dynamic modification, a set is more suitable. Understanding the trade-offs and characteristics of each data structure is key to making an informed decision based on the specific needs of the application.

`5.How can you find the intersection of two sets in Python?`

In [ ]:
```
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

intersection_set = set1.intersection(set2)

print(intersection_set)
```

```
{3, 4, 5}
```

Sets in Python - Interview Questions

`1.Can you explain the concept of sets in Python?`

- In Python, a set is an **unordered** and **mutable** collection of unique elements.
- The key characteristics of sets are their ability to store distinct values and the absence of any defined order among the elements. Sets are defined using curly braces {}, and elements are separated by commas
- The uniqueness property of sets ensures that **duplicate elements are automatically eliminated**. Sets support a variety of operations such as union, intersection, difference, and symmetric difference, making them versatile for tasks involving the comparison and manipulation of collections.
- it's important to note that while sets themselves are **mutable** (elements can be added or removed), elements within a set must be of immutable data types (e.g., numbers, strings, or tuples). (note-If mutability is not required, Python offers an immutable version of sets called "frozensets," created using the frozenset() constructor.)

`2.How do you create an empty set in Python, and what is the key difference between an empty set and an empty dictionary?`

In [ ]:
```
# if we use curley braces for empty set then

s = {}

print(type(s))
```

```
<class 'dict'>
```

but it gives type is Dictionary so for the creation of empty set we use differnet Syntax

In [ ]:
```
# create empty set

s = set()

print(type(s))
```

```
<class 'set'>
```

Use Cases:

- Empty Set:

1. Suitable when there is a need for a collection with distinct and unordered elements.
2. Ideal for tasks involving set operations such as union, intersection, and difference.

- Empty Dictionary:

1. Used when data needs to be stored and retrieved based on specific keys.
2. Essential for scenarios where a mapping between keys and values is necessary.

In summary, an empty set is a collection of unique and unordered elements created using the set() constructor, while an empty dictionary is a data structure used for storing key-value

pairs, created using the {} notation or the dict() constructor. Understanding the intended purpose and structure of each is crucial in choosing the appropriate data structure for a given task.

`3.If you have an empty set, how would you add elements using a built-in function in Python?`

In [ ]:
```
# Create an empty set
my_set = set()

# Add elements to the set using the add() method
my_set.add(1)
my_set.add('hello')
my_set.add(3.14)

# Display the updated set
print(my_set)
```

```
{'hello', 1, 3.14}
```

`4.If you have a tuple and a set, which one is faster in terms of performance, and why?`

The performance difference between a tuple and a set in Python is influenced by their inherent characteristics and use cases.

1. **Access Time:**
   - **Tuple:**
     - Tuples are generally faster for element access since they are indexed and ordered.
     - Accessing elements in a tuple has a constant time complexity O(1).
   - **Set:**
     - Sets are unordered collections, and they do not support indexing.
     - Accessing elements in a set involves a hash lookup, resulting in an average time complexity of O(1) but with some variability.
2. **Uniqueness and Mutability:**
   - **Tuple:**
     - Tuples are immutable, meaning their elements cannot be changed after creation.
     - Ideal for scenarios where data should remain constant throughout the program.
   - **Set:**
     - Sets are mutable and designed for efficient membership tests and unique element storage.
     - Suitable for scenarios where the collection of distinct and unordered elements needs to be dynamically modified.
3. **Use Cases:**

- **Tuple:**
  - Prefer tuples when the order of elements matters, and the data should remain constant.
  - Useful in scenarios like representing coordinates (x, y, z) or holding a fixed set of values.
- **Set:**
  - Opt for sets when dealing with collections of unique elements and the order is not significant.
  - Useful for tasks involving set operations like union, intersection, and difference.

4. **Memory Overhead:**

- **Tuple:**
  - Tuples typically have lower memory overhead compared to sets, as they store only the elements.
- **Set:**
  - Sets use additional memory for hash tables and other internal structures, which can lead to higher memory consumption.

In conclusion, the choice between a tuple and a set depends on the specific requirements of the task. If fast element access, order preservation, and immutability are crucial, a tuple is preferred. On the other hand, if the focus is on efficient membership tests, unique element storage, and dynamic modification, a set is more suitable. Understanding the trade-offs and characteristics of each data structure is key to making an informed decision based on the specific needs of the application.

`5.How can you find the intersection of two sets in Python?`

In [ ]:
```
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

intersection_set = set1.intersection(set2)

print(intersection_set)
```

```
{3, 4, 5}
```

Tuples Interview Questions

`1.Explain the difference between a tuple and a list in Python. Provide an example of when you would use each.`

Answer: A tuple is an immutable sequence, and its elements cannot be changed after creation, while a list is mutable. Tuples are created using parentheses, and lists use square brackets. Use a tuple when you want to represent a fixed collection of items, and a list when you need a dynamic and mutable sequence.

`2.How can you reverse a tuple in Python?`

Answer: Tuples are immutable, so you cannot reverse them in place. However, you can create a new tuple with reversed elements using slicing

In [ ]:
```
original_tuple = (1, 2, 3, 4)
reversed_tuple = original_tuple[::-1]
reversed_tuple
```

Out[ ]:
```
(4, 3, 2, 1)
```

`3.How can you sort a list of tuples based on the second element of each tuple?`

In [ ]:
```
my_list = [(3, 7), (1, 4), (2, 8)]
sorted_list = sorted(my_list, key=lambda x: x[1])
sorted_list
```

Out[ ]:
```
[(1, 4), (3, 7), (2, 8)]
```

`4.Find the length of a tuple.`

In [ ]:
```
# Solution 1:
def tuple_length(tup):
    return len(tup)

# Solution 2:
tup = (1, 2, 3, 4, 5)
length = len(tup)
print(length)
```

```
5
```

`5.Find the first and last elements of a tuple.`

In [ ]:
```
# Solution:
def first_last_elements(tup):
    return tup[0], tup[-1]

tup = (10, 20, 30, 40, 50)
first, last = first_last_elements(tup)
print("First:", first)  # Output: First: 10
print("Last:", last)    # Output: Last: 50
```

```
First: 10
Last: 50
```

List Interview Questions

`1.How do you reverse a list in Python?`

In [ ]:
```
l = [1,2,3,4,5]

# solution 1
def reverse_list(l):
    return l[::-1]

reverse_list(l)

# solution 2
reverse_list = l[::-1]
print(reverse_list)
```

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

`2. Explain the difference between append() and extend() methods in Python lists.`

Answer:

- append() adds a single element to the end of the list.
- extend() adds elements of an iterable (list, tuple, etc.) to the end of the list.

In [ ]:
```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

list1.append(4)
print(list1)
list1.extend(list2)
print(list1)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4, 4, 5, 6]
```

`4. Explain the pop() method in Python lists.`

Answer: The pop() method removes and returns the item at the specified index (default is the last item).

In [ ]:
```
my_list = [1, 2, 3, 4, 5]
popped_item = my_list.pop(2)  # Removes and returns the item at index 2 (3)
my_list
```

Out[ ]:
```
[1, 2, 4, 5]
```

`5. Explain list comprehension in Python.`

Answer: List comprehension is a concise way to create lists. It consists of an expression followed by at least one for clause and zero or more if clauses.

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

```
[1, 4, 9, 16, 25]
```

Python Usecases Interview Questions

1.Given two lists, write a function that returns the elements that are common to both lists.

In [ ]:
```
# given lists
l1 = [1, 2, 3, 4, 5]
l2 = [3, 4, 5, 6, 7]

# function for the commmon elements
def find_common_element(list1,list2):
    common_element = list(set(list1) & set(list2))
    return common_element

find_common_element(l1,l2)
```

Out[ ]:
```
[3, 4, 5]
```

2. Write a Python function to merge two dictionaries. If both dictionaries have the same key, prefer the second dictionary's value.

In [ ]:
```
# function
def merge_two_dictionaries(dict1,dict2):
    merged = dict1.copy()
    merged.update(dict2)
    return merged

dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
print(merge_two_dictionaries(dict1, dict2))
```

```
{'a': 1, 'b': 3, 'c': 4}
```

3.Given a list of numbers, write a function to compute the mean, median, and mode.

In [ ]:
```
# list
numbers = [1, 2, 3, 4, 4, 5, 5, 5, 6]

from statistics import mean, median, mode
def central_measures(numbers):
    return {
        'mean': mean(numbers),
        'median': median(numbers),
        'mode': mode(numbers)
    }

print(central_measures(numbers))
```

```
{'mean': 3.888888888888889, 'median': 4, 'mode': 5}
```

4.Write a function to check if a given string is a palindrome.

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

```
string = "radar"
print(is_palindrome(string))
```

```
True
```

5.Write a function to compute the factorial of a number using iteration.

In [ ]:
```
def factorial_iterative(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result

number = 5
print(factorial_iterative(number))
```

```
120
```

OOPs Interview Questions

1. What is meant by the term OOPs? and What is the need for OOPs?

OOPs refers to Object-Oriented Programming. It is the programming paradigm that is defined using objects. Objects can be considered as real-world instances of entities like class, that have some characteristics and behaviors.

There are many reasons why OOPs is mostly preferred, but the most important among them are:

- OOPs helps users to understand the software easily, although they don't know the actual implementation.
- With OOPs, the readability, understandability, and maintainability of the code increase multifold.
- Even very big software can be easily written and managed easily using OOPs.

2. What is a class?

A class can be understood as a template or a blueprint, which contains some values, known as member data or member, and some set of rules, known as behaviors or functions. So when an object is created, it automatically takes the data and functions that are defined in the class. Therefore the class is basically a template or blueprint for objects. Also one can create as many objects as they want based on a class.

For example, first, a car's template is created. Then multiple units of car are created based on that template.

Here's a basic example of a class in Python:

In [ ]:
```python
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def bark(self):
        print(f"{self.name} says Woof!")

# Creating instances of the Dog class
dog1 = Dog(name="Buddy", age=2)
dog2 = Dog(name="Molly", age=4)

# Accessing attributes and calling methods
print(f"{dog1.name} is {dog1.age} years old.")
```

```
print(f"{dog2.name} is {dog2.age} years old.")
dog1.bark()
dog2.bark()
```

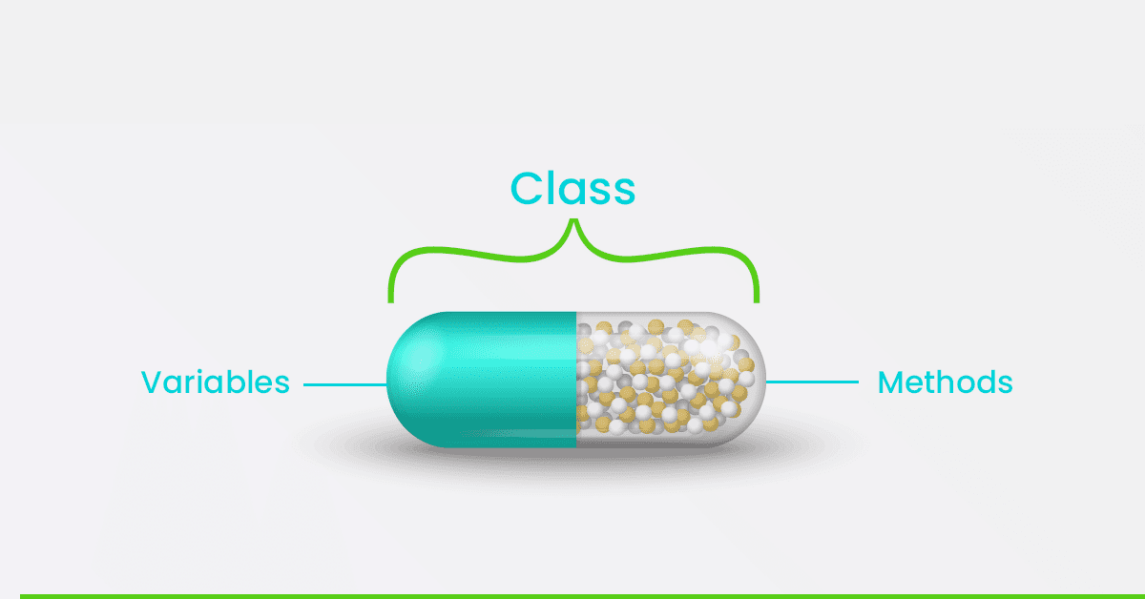
```
Buddy is 2 years old.
Molly is 4 years old.
Buddy says Woof!
Molly says Woof!
```

3. What is an object?

An object refers to the instance of the class, which contains the instance of the members and behaviors defined in the class template. In the real world, an object is an actual entity to which a user interacts, whereas class is just the blueprint for that object. So the objects consume space and have some characteristic behavior.

For example, a specific car

4. What is encapsulation?



One can visualize Encapsulation as the method of putting everything that is required to do the job, inside a capsule and presenting that capsule to the user. What it means is that by Encapsulation, all the necessary data and methods are bind together and all the unnecessary details are hidden to the normal user. So Encapsulation is the process of binding data members and methods of a program together to do a specific job, without revealing unnecessary details.

5.what is inheritance

Inheritance is a key concept in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit attributes and behaviors from an existing class (base class or superclass). The new class can extend or override the functionalities of the existing class, promoting code reuse and the creation of a hierarchy of classes.

In [ ]:
```python
# Base class (superclass)
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

# Derived class (subclass)
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

# Another derived class (subclass)
class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# Creating instances of the classes
dog = Dog("Buddy")
cat = Cat("Mittens")

# Using inherited methods
print(dog.speak())
print(cat.speak())
```

```
Buddy says Woof!
Mittens says Meow!
```

In this example, the `Animal` class is the superclass, and the `Dog` and `Cat` classes are subclasses. Both `Dog` and `Cat` inherit the `__init__` method (constructor) and the `speak` method from the `Animal` class. However, each subclass provides its own implementation of the `speak` method, allowing them to exhibit different behaviors.

Inheritance is a powerful mechanism in OOP that helps in organizing and structuring code in a hierarchical manner, making it easier to manage and extend software systems.

Pandas Interview Questions

1. What is Pandas, and why is it popular in data analyst?

Pandas is a popular open-source data manipulation and analysis library for the Python programming language. It provides data structures for efficiently storing and manipulating large datasets and tools for reading and writing data in various formats. The two primary data structures in Pandas are:

1. **Series:** A one-dimensional labeled array capable of holding any data type.
2. **DataFrame:** A two-dimensional labeled data structure with columns that can be of different types.

Pandas is widely used in the field of data analysis for several reasons:

1. **Ease of Use:** Pandas provides a simple and intuitive syntax for data manipulation. Its data structures are designed to be easy to use and interact with.

2. **Data Cleaning and Transformation:** Pandas makes it easy to clean and transform data. It provides functions for handling missing data, reshaping data, merging and joining datasets, and performing various data transformations.

3. **Data Exploration:** Pandas allows data analysts to explore and understand their datasets quickly. Descriptive statistics, data summarization, and various methods for slicing and dicing data are readily available.

4. **Data Input/Output:** Pandas supports reading and writing data in various formats, including CSV, Excel, SQL databases, and more. This makes it easy to work with data from different sources.

5. **Integration with Other Libraries:** Pandas integrates well with other popular data science and machine learning libraries in Python, such as NumPy, Matplotlib, and Scikit-learn. This allows for a seamless workflow when performing more complex analyses.

6. **Time Series Analysis:** Pandas provides excellent support for time series data, including tools for date range generation, frequency conversion, and resampling.

7. **Community and Documentation:** Pandas has a large and active community, which means there is extensive documentation and a wealth of online resources, tutorials, and forums available for users to seek help and guidance.

8. **Open Source:** Being an open-source project, Pandas allows users to contribute to its development and improvement. This collaborative nature has helped Pandas evolve and stay relevant in the rapidly changing landscape of data analysis and data science.

In summary, Pandas is popular in data analysis because it simplifies the process of working with structured data, provides powerful tools for data manipulation, and has become a

standard tool in the Python ecosystem for data analysis tasks.

2. What is DataFrame in Pandas?

In Pandas, a DataFrame is a two-dimensional, tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or SQL table, where data can be stored in rows and columns. The key features of a DataFrame include:

1. **Tabular Structure:** A DataFrame is a two-dimensional table with rows and columns. Each column can have a different data type, such as integer, float, string, or even custom types.
2. **Labeled Axes:** Both rows and columns of a DataFrame are labeled. This means that each row and each column has a unique label or index associated with it, allowing for easy access and manipulation of data.
3. **Flexible Size:** DataFrames can grow and shrink in size. You can add or remove rows and columns as needed.
4. **Heterogeneous Data Types:** Different columns in a DataFrame can have different data types. For example, one column might contain integers, while another column contains strings.
5. **Data Alignment:** When performing operations on DataFrames, Pandas automatically aligns the data based on labels, making it easy to work with data even if it is not perfectly clean or aligned.
6. **Missing Data Handling:** DataFrames can handle missing data gracefully. Pandas provides methods for detecting, removing, or filling missing values.
7. **Powerful Operations:** DataFrames support a wide range of operations, including arithmetic operations, aggregation, filtering, merging, and reshaping. This makes it a powerful tool for data analysis and manipulation.

In [ ]:
```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {'Name': ['John', 'Jane', 'Bob'],
        'Age': [28, 24, 22],
        'City': ['New York', 'San Francisco', 'Los Angeles']}

df = pd.DataFrame(data)

# Displaying the DataFrame
print(df)
```

```
   Name  Age           City
0  John   28       New York
1  Jane   24  San Francisco
2   Bob   22    Los Angeles
```

In this example, each column represents a different attribute (Name, Age, City), and each row represents a different individual. The DataFrame provides a convenient way to work with this tabular data in a structured and labeled format.

3. What is diffrence between loc and iloc in pandas?

In Pandas, `loc` and `iloc` are two different methods used for indexing and selecting data from a DataFrame. They are primarily used for label-based and integer-location-based indexing, respectively. Here's the key difference between `loc` and `iloc`:

1. **`loc` (Label-based Indexing):**
   - The `loc` method is used for selection by label.
   - It allows you to access a group of rows and columns by labels or a boolean array.
   - The syntax is `df.loc[row_label, column_label]` or `df.loc[row_label]` for selecting entire rows.
   - The labels used with `loc` are the actual labels of the index or column names, not the integer position.
   - Inclusive slicing is supported with `loc`, meaning both the start and stop index are included in the selection.

In [ ]:
```
import pandas as pd

   # Assuming 'df' is our DataFrame
selected_data = df.loc[2:4, 'Name':'City']
selected_data
```

Out[ ]:

	Name	Age	City
2	Bob	22	Los Angeles

1. **`iloc` (Integer-location based Indexing):**
   - The `iloc` method is used for selection by position.
   - It allows you to access a group of rows and columns by integer positions.
   - The syntax is `df.iloc[row_index, column_index]` or `df.iloc[row_index]` for selecting entire rows.
   - The indices used with `iloc` are integer-based, meaning you specify the position of the rows and columns based on their numerical order (0-based indexing).
   - Exclusive slicing is used with `iloc`, meaning the stop index is not included in the selection.

In [ ]:
```
import pandas as pd

   # Assuming 'df' is our DataFrame
selected_data = df.iloc[2:5, 0:3]
selected_data
```

Out[ ]:

	Name	Age	City
2	Bob	22	Los Angeles

In summary, if you want to select data based on the labels of rows and columns, you use `loc`. If you prefer to select data based on the integer positions of rows and columns, you

use `iloc`. The choice between them depends on whether you are working with labeled or integer-based indexing.

4. How do you filter rows in a dataframe based on condition?

To filter rows in a DataFrame based on a condition, you can use boolean indexing. Boolean indexing involves creating a boolean Series that represents the condition you want to apply and then using that boolean Series to filter the rows of the DataFrame. Here's a step-by-step guide:

Assuming you have a DataFrame named df, and you want to filter rows based on a condition, let's say a condition on the 'Age' column:

In [ ]:
```
import pandas as pd

# Assuming 'df' is your DataFrame
data = {'Name': ['John', 'Jane', 'Bob', 'Alice'],
        'Age': [28, 24, 22, 30],
        'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']}
df = pd.DataFrame(data)

# Condition for filtering (e.g., selecting rows where Age is greater than 25)
condition = df['Age'] > 25

# Applying the condition to filter rows
filtered_df = df[condition]

# Displaying the filtered DataFrame
print(filtered_df)
```

```
    Name  Age      City
0   John   28  New York
3  Alice   30   Chicago
```

5. How do you handle missing values in data with the help of pandas?

Handling missing values is a crucial step in the data cleaning process. Pandas provides several methods for working with missing data in a DataFrame. Here are some common techniques:

1. **Detecting Missing Values:**
   - The `isnull()` method can be used to detect missing values in a DataFrame. It returns a DataFrame of the same shape, where each element is a boolean indicating whether the corresponding element in the original DataFrame is missing.
   - The `notnull()` method is the opposite of `isnull()` and returns `True` for non-missing values.

In [ ]:
```
import pandas as pd

# Assuming 'df' is your DataFrame
missing_values = df.isnull()
```

1. **Dropping Missing Values:**
   - The `dropna()` method can be used to remove rows or columns containing missing values.
   - The `thresh` parameter can be used to specify a threshold for the number of non-null values required to keep a row or column.

In [ ]:
```
# Drop rows with any missing values
df_no_missing_rows = df.dropna()

# Drop columns with any missing values
df_no_missing_cols = df.dropna(axis=1)

# Drop rows with at least 3 non-null values
df_thresh = df.dropna(thresh=3)
```

1. **Filling Missing Values:**
   - The `fillna()` method can be used to fill missing values with a specified constant or using various filling methods like forward fill or backward fill.
   - Commonly, mean or median values are used to fill missing values in numerical columns.

In [ ]:
```
# Fill missing values with a constant
df_fill_constant = df.fillna(0)

# Fill missing values with the mean of the column
df_fill_mean = df.fillna(df.mean())

# Forward fill missing values (use the previous value)
df_ffill = df.fillna(method='ffill')

# Backward fill missing values (use the next value)
df_bfill = df.fillna(method='bfill')
```

Pandas Interview Questions(Part2)

1. Explain the process of merging two dataframe in pandas

Merging two DataFrames in Pandas involves combining their rows based on a common column (or index) called the key. This operation is similar to SQL joins. Pandas provides the merge() function to perform various types of joins between two or more DataFrames.

Step 1: Import Pandas

In [ ]:
```
import pandas as pd
```

Step 2: Create Two DataFrames Assuming you have two DataFrames, df1 and df2, with a common key column:

In [ ]:
```
# Example DataFrames
data1 = {'Key': ['A', 'B', 'C'], 'Value1': [1, 2, 3]}
data2 = {'Key': ['A', 'B', 'D'], 'Value2': ['X', 'Y', 'Z']}

df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)
```

Step 3: Choose the Type of Join Decide on the type of join you want to perform. The common types are:

- Inner Join (how='inner'): Keeps only the rows with keys present in both DataFrames.
- Outer Join (how='outer'): Keeps all rows from both DataFrames and fills in missing values with NaN where there is no match.
- Left Join (how='left'): Keeps all rows from the left DataFrame and fills in missing values with NaN where there is no match in the right DataFrame.
- Right Join (how='right'): Keeps all rows from the right DataFrame and fills in missing values with NaN where there is no match in the left DataFrame.

Step 4: Merge the DataFrames

In [ ]:
```
# Performing the merge (for example, an inner join)
merged_df = pd.merge(df1, df2, on='Key', how='inner')
merged_df
```

Out[ ]:

	Key	Value1	Value2
0	A	1	X
1	B	2	Y

In this example, we're performing an inner join based on the 'Key' column. The resulting DataFrame (merged_df) will have columns from both original DataFrames, and rows where the 'Key' values match in both DataFrames.

`2.What is purpose of the groupby function in pandas?`

The `groupby` function in Pandas is used for grouping data based on some criteria, and it is a powerful and flexible tool for data analysis and manipulation. The primary purpose of `groupby` is to split the data into groups based on some criteria, apply a function to each group independently, and then combine the results back into a DataFrame.

Here's an overview of the purpose and usage of the `groupby` function:

Purpose of `groupby`:

1. **Data Splitting:**
   - `groupby` is used to split the data into groups based on one or more criteria, such as a column's values or a combination of columns.
2. **Operations on Groups:**
   - After splitting the data into groups, you can perform operations on each group independently. This might include aggregations, transformations, filtering, or other custom operations.
3. **Aggregation:**
   - One of the most common use cases for `groupby` is to perform aggregation operations on each group, such as calculating the mean, sum, count, minimum, maximum, etc.
4. **Data Transformation:**
   - `groupby` allows you to apply transformations to the groups and create new features or modify existing ones.
5. **Filtering:**
   - You can use `groupby` in combination with filtering operations to select specific groups based on certain conditions.

`3.Expalin the difference between Series and DataFrame in Pandas`

In Pandas, both Series and DataFrame are fundamental data structures, but they serve different purposes and have distinct characteristics.

Series:

1. **1-Dimensional Data Structure:**
   - A Series is essentially a one-dimensional labeled array that can hold any data type, such as integers, floats, strings, or even Python objects.
2. **Homogeneous Data:**

- All elements in a Series must be of the same data type. It is a homogeneous data structure.

3. **Labeled Index:**
   - Each element in a Series has a label (index), which can be customized or can be the default integer index. This index facilitates easy and efficient data retrieval.
4. **Similar to a Column in a DataFrame:**
   - A Series is similar to a single column in a DataFrame. In fact, you can think of a DataFrame as a collection of Series with the same index.
5. **Creation:**
   - You can create a Series from a list, NumPy array, or dictionary.

In [ ]:
```
import pandas as pd

# Creating a Series from a list
s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
s
```

Out[ ]:
```
a    1
b    2
c    3
d    4
dtype: int64
```

DataFrame:

1. **2-Dimensional Data Structure:**
   - A DataFrame is a two-dimensional tabular data structure where you can store data of different data types. It consists of rows and columns.
2. **Heterogeneous Data:**
   - Different columns in a DataFrame can have different data types, making it a heterogeneous data structure.
3. **Labeled Index and Columns:**
   - Similar to a Series, a DataFrame also has a labeled index for rows, and additionally, it has labeled columns. The column names can be customized.
4. **Collection of Series:**
   - A DataFrame can be thought of as a collection of Series. Each column is a Series, and the columns share the same index.
5. **Creation:**
   - You can create a DataFrame from a dictionary, a list of dictionaries, a NumPy array, or another DataFrame.

In [ ]:
```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {'Name': ['John', 'Jane', 'Bob'],
        'Age': [28, 24, 22],
```

```
        'City': ['New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data)
df
```

Out[ ]:

	Name	Age	City
0	John	28	New York
1	Jane	24	San Francisco
2	Bob	22	Los Angeles

In summary, while both Series and DataFrames have labeled indices and support various operations, a Series is a one-dimensional array, and a DataFrame is a two-dimensional table. Series are often used to represent a single column, while DataFrames are used to represent a collection of columns with potentially different data types.

`4.what is purpose ofthe apply unction in pandas, and how is it used?`

The `apply` function in Pandas is a powerful tool used for applying a function along the axis of a DataFrame or a Series. It is particularly useful for performing element-wise operations, transformations, or custom functions on the data. The primary purpose of `apply` is to allow users to apply a function to each element in a DataFrame or Series, row-wise or column-wise.

Purpose of `apply`:

1. **Element-wise Operations:**
   - Apply allows you to perform element-wise operations on each element in a Series or DataFrame.
2. **Custom Functions:**
   - You can use `apply` to apply custom functions that you define to each element or row/column of your data.
3. **Aggregation:**
   - When used with DataFrames, `apply` can be used for column-wise or row-wise aggregation.
4. **Function Composition:**
   - `apply` is often used in combination with lambda functions or other callable objects, allowing for flexible and concise code.

Examples:

In [ ]:

```
#### Example 1: Element-wise Operation on Series
import pandas as pd

# Creating a Series
s = pd.Series([1, 2, 3, 4])
```

```
# Applying a square function element-wise
result_series = s.apply(lambda x: x**2)
result_series
```

Out[ ]:
```
0     1
1     4
2     9
3    16
dtype: int64
```

In [ ]:
```
#### Example 2: Applying a Function to Each Column in a DataFrame


import pandas as pd

# Creating a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Applying a sum function column-wise
result_dataframe = df.apply(sum, axis=0)
result_dataframe
```

Out[ ]:
```
A     6
B    15
dtype: int64
```

In [ ]:
```
#### Example 3: Applying a Function to Each Row in a DataFrame

# Creating a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Applying a sum function row-wise
result_dataframe = df.apply(sum, axis=1)
result_dataframe
```

Out[ ]:
```
0    5
1    7
2    9
dtype: int64
```

In these examples, the `apply` function is used to perform operations on each element in a Series or each row/column in a DataFrame. The flexibility of `apply` makes it a versatile tool for various data manipulation tasks in Pandas.

5.Explain the concept of Pivot Tables in pandas

In Pandas, a pivot table is a data summarization tool that allows you to reshape and analyze your data by aggregating and restructuring it. Pivot tables provide a way to rearrange, reshape, and aggregate data in a DataFrame to gain insights and perform complex analyses more efficiently. The concept is inspired by Excel's pivot table functionality.

Key Concepts of Pivot Tables:

1. **Rows and Columns:**
   - In a pivot table, you specify which columns of the original DataFrame should become the new index (rows) and which columns should become the new columns.
2. **Values:**

- You specify which columns of the original DataFrame should be used as values in the new table. These values are then aggregated based on the specified rows and columns.

3. **Aggregation Function:**

   - You can specify an aggregation function to apply to the values. Common aggregation functions include sum, mean, count, max, min, etc.

How to Create a Pivot Table:

- `df` : The original DataFrame.
- `values` : The column to aggregate (can be a list if you want to aggregate multiple columns).
- `index` : The column(s) to become the new index.
- `columns` : The column(s) to become the new columns.
- `aggfunc` : The aggregation function to apply to the values.

Example:

In [ ]:
```
import pandas as pd

# Creating a sample DataFrame
data = {'Date': ['2022-01-01', '2022-01-01', '2022-01-02', '2022-01-02'],
        'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles'],
        'Temperature': [32, 75, 30, 78],
        'Humidity': [80, 10, 85, 12]}

df = pd.DataFrame(data)
print(df)

print('-----------------------------------------------------------------------------

# Creating a pivot table to show average temperature for each city on each date
pivot_table = pd.pivot_table(df, values='Temperature', index='Date', columns='City'

print(pivot_table)
```

```
         Date         City  Temperature  Humidity
0  2022-01-01     New York           32        80
1  2022-01-01  Los Angeles           75        10
2  2022-01-02     New York           30        85
3  2022-01-02  Los Angeles           78        12
-------------------------------------------------------------------------------
City        Los Angeles  New York
Date
2022-01-01           75        32
2022-01-02           78        30
```

In this example, the pivot table calculates the average temperature for each city on each date. The resulting table will have dates as rows, cities as columns, and the average temperature as values.

Pivot tables are particularly useful for exploring and summarizing data with multiple dimensions, making complex data analysis tasks more accessible and efficient. They allow you to quickly gain insights into your data's patterns and relationships.

NumPy Interview Questions

1. What is NumPy, and how is it different from Python List?

NumPy is a powerful numerical computing library in Python. It provides support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays. NumPy is a fundamental package for scientific computing in Python, and many other libraries, such as pandas and scikit-learn, are built on top of it.

Here are some key differences between NumPy arrays and Python lists:

1. **Homogeneity:**
   - NumPy arrays are homogeneous, meaning that all elements of an array must be of the same data type. This allows for more efficient storage and computation.
   - Python lists can contain elements of different data types.
2. **Size:**
   - NumPy arrays are more compact in memory compared to Python lists. This is because NumPy arrays are implemented in C and allow for more efficient storage of data.
   - Python lists have more overhead and are generally less memory-efficient.
3. **Performance:**
   - NumPy operations are implemented in C, which makes them much faster than equivalent operations on Python lists, especially for large datasets.
   - NumPy provides a wide range of efficient functions for array operations, such as element-wise operations, linear algebra, and statistical operations.
4. **Functionality:**
   - NumPy provides a variety of functions for performing operations on arrays, such as element-wise operations, linear algebra, statistical operations, and more.
   - While Python lists have some built-in functions, NumPy provides a more extensive set of tools specifically designed for numerical operations.
5. **Syntax:**
   - NumPy arrays support vectorized operations, which means that operations can be performed on entire arrays without the need for explicit loops. This leads to more concise and readable code.
   - In Python lists, you often need to use explicit loops for element-wise operations.

Here's a simple example to illustrate the differences:

In [ ]:
```
# Using Python lists
python_list1 = [1, 2, 3]
python_list2 = [4, 5, 6]
sum_python_lists = python_list1 + python_list2  # Concatenation, not element-wise c
```

```
print('Sum_Python_list -',sum_python_lists)

# Using NumPy arrays
import numpy as np

numpy_array1 = np.array([1, 2, 3])
numpy_array2 = np.array([4, 5, 6])
sum_numpy_arrays = numpy_array1 + numpy_array2  # Element-wise addition
print('Numpy_list_sum -',sum_numpy_arrays)
```

```
Sum_Python_list - [1, 2, 3, 4, 5, 6]
Numpy_list_sum - [5 7 9]
```

In the NumPy example, the addition is performed element-wise, which is a fundamental concept in numerical computing. In contrast, the Python list example concatenates the two lists without performing element-wise addition.

2.Explain the concept of Broadcasting in NumPy?

Broadcasting is a powerful feature in NumPy that allows you to perform operations on arrays of different shapes and sizes. The term "broadcasting" refers to how NumPy treats arrays with different shapes during arithmetic operations. The smaller array is "broadcast" across the larger array so that they have compatible shapes.

The broadcasting rule in NumPy is as follows:

1. If the arrays do not have the same rank (number of dimensions), pad the smaller shape with ones on its left side.
2. Compare the sizes of the corresponding dimensions:
   - If the sizes of the dimensions are different, but one of them is 1, then the arrays are compatible for broadcasting.
   - If the sizes in a dimension are neither equal nor one, broadcasting is not possible, and a `ValueError` is raised.
3. After the broadcasting, each array behaves as if its shape was the element-wise maximum of both shapes.

Let's go through a simple example to illustrate broadcasting:

In [ ]:
```
import numpy as np

# Example 1: Broadcasting a scalar to an array
scalar_value = 5
array = np.array([1, 2, 3])
result = scalar_value + array
# The scalar is broadcasted to the shape of the array, resulting in [6, 7, 8]
result
```

Out[ ]:
```
array([6, 7, 8])
```

In [ ]:
```
# Example 2: Broadcasting a 1D array to a 2D array
array_1d = np.array([1, 2, 3])
array_2d = np.array([[4, 5, 6], [7, 8, 9]])
result = array_1d + array_2d
# The 1D array is broadcasted along the second dimension of the 2D array,
```

```
# resulting in [[5, 7, 9], [8, 10, 12]]
result
```

Out[ ]:
```
array([[ 5,  7,  9],
       [ 8, 10, 12]])
```

In [ ]:
```
# Example 3: Broadcasting both arrays
array_a = np.array([[1], [2], [3]])
array_b = np.array([4, 5, 6])
result = array_a + array_b
# Both arrays are broadcasted, resulting in [[5, 6, 7], [6, 7, 8], [7, 8, 9]]
result
```

Out[ ]:
```
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

In these examples, the smaller array or scalar is broadcasted to match the shape of the larger array, and the element-wise operation is then performed. This broadcasting mechanism allows for more concise and readable code when working with arrays of different shapes, making NumPy operations more flexible.

3.How do you perform Element wise addition of two NumPy arrays?

Performing element-wise addition of two NumPy arrays is straightforward. You can use the + operator, and NumPy will handle the element-wise addition automatically. Here's an example:

In [ ]:
```
import numpy as np

# Create two NumPy arrays
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

# Perform element-wise addition
result_array = array1 + array2

# Display the result
print(result_array)
```

```
[5 7 9]
```

In this example, array1 + array2 performs element-wise addition, resulting in a new NumPy array [5, 7, 9]. NumPy takes care of broadcasting if the arrays have different shapes but are still compatible according to the broadcasting rules.

You can perform element-wise addition, subtraction, multiplication, and division using the standard arithmetic operators (+, -, *, /). NumPy will apply these operations element-wise to the corresponding elements of the arrays.

In [ ]:
```
import numpy as np

# Create two NumPy arrays
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

# Element-wise operations
```

```
result_addition = array1 + array2  # Element-wise addition
result_subtraction = array1 - array2  # Element-wise subtraction
result_multiplication = array1 * array2  # Element-wise multiplication
result_division = array1 / array2  # Element-wise division

# Display the results
print("Addition:", result_addition)
print("Subtraction:", result_subtraction)
print("Multiplication:", result_multiplication)
print("Division:", result_division)
```

```
Addition: [5 7 9]
Subtraction: [-3 -3 -3]
Multiplication: [ 4 10 18]
Division: [0.25 0.4  0.5 ]
```

4.What is the purpose of the np.mean() function in NumPy?

The np.mean() function in NumPy is used to compute the arithmetic mean, or average, of elements along a specified axis or of the entire array. The mean is calculated by summing up all the elements and dividing the sum by the total number of elements.

In [ ]:
```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Calculate the mean of the array
mean_value = np.mean(arr)

print("Mean:", mean_value)
```

```
Mean: 3.0
```

In this example, np.mean(arr) will calculate the mean of the elements [1, 2, 3, 4, 5], which is (1 + 2 + 3 + 4 + 5) / 5 = 3.0.

5. How do you reshape a NumPy Arrays?

You can reshape a NumPy array using the reshape() function. Reshaping means changing the shape (dimensions) of the array while keeping the total number of elements the same. The reshape() function takes the desired shape as an argument.

In [ ]:
```
import numpy as np

# Create a NumPy array
original_array = np.array([1, 2, 3, 4, 5, 6])

# Reshape the array to a 2x3 matrix
reshaped_array = original_array.reshape(2, 3)

print("Original array:")
print(original_array)

print("\nReshaped array:")
print(reshaped_array)
```

```
Original array:
[1 2 3 4 5 6]

Reshaped array:
[[1 2 3]
 [4 5 6]]
```

In this example, original_array is reshaped into a 2x3 matrix using reshape(2, 3).

NumPy Interview Questions(part-2)

`1. Explain the use of NumPy's random module for generating random Numbers`

NumPy's `random` module provides a suite of functions for generating random numbers, which is useful for various applications such as simulations, statistical modeling, and machine learning. The `random` module is a sub-module of NumPy, and it offers a wide range of functions to generate random numbers with different distributions.

Here are some key functions and concepts related to NumPy's `random` module:

1. Simple Random Data:

- NumPy provides functions to generate arrays of random data with a uniform distribution.

In [ ]:
```
import numpy as np

# Generate random floats in the half-open interval [0.0, 1.0)
random_numbers = np.random.random(size=5)
random_numbers
```

Out[ ]:
```
array([0.11442764, 0.70706516, 0.17581463, 0.12075097, 0.19547172])
```

2. Random Integers:

- You can generate random integers within a specified range.

In [ ]:
```
# Generate random integers in the half-open interval [low, high)
random_integers = np.random.randint(low=1, high=10, size=5)
random_integers
```

Out[ ]:
```
array([1, 5, 6, 8, 7])
```

3. Random Sampling:

- Functions like `choice` allow you to sample from a given array.

In [ ]:
```
# Randomly sample from a given array
choices = np.random.choice(['a', 'b', 'c'], size=5)
choices
```

Out[ ]:
```
array(['a', 'b', 'a', 'c', 'c'], dtype='<U1')
```

4. Permutations:

- NumPy can be used to generate random permutations.

In [ ]:
```
# Randomly permute a sequence
permutation = np.random.permutation([1, 2, 3, 4, 5])
permutation
```

Out[ ]:
```
array([3, 4, 1, 5, 2])
```

5. **Distributions:**

- NumPy's `random` module supports various probability distributions such as normal, binomial, exponential, etc.

In [ ]:
```
# Generate random numbers from a normal distribution
normal_distribution = np.random.normal(loc=0, scale=1, size=5)
normal_distribution
```

Out[ ]:
```
array([ 0.96949722,  0.27283578, -0.27559331, -0.5644573 , -0.36100325])
```

These functions provide flexibility in generating random data according to different needs, and they are crucial tools for various statistical and numerical simulations. The ability to set a seed is especially useful when reproducibility is important, allowing you to recreate the same set of random numbers for debugging or sharing code with others.

2. How can you find the maximum and minimum values in a NumPy Array?

In NumPy, you can find the maximum and minimum values in an array using the `np.max()` and `np.min()` functions, respectively. These functions offer flexibility in finding the maximum and minimum values across different axes or the entire array.

Here are the basic usages:

Finding the Maximum and Minimum in the Entire Array:

In [ ]:
```
import numpy as np

# Create a NumPy array
arr = np.array([1, 5, 3, 8, 2])

# Find the maximum and minimum values
max_value = np.max(arr)
min_value = np.min(arr)

print("Maximum Value:", max_value)
print("Minimum Value:", min_value)
```

```
Maximum Value: 8
Minimum Value: 1
```

3.How do you convert Pandas DataFrame to a NumPy array?

The to_numpy() method of the NumPy package can be used to convert Pandas DataFrame, Index and Series objects.

Consider we have a DataFrame df, we can either convert the whole Pandas DataFrame df to NumPy array or even select a subset of Pandas DataFrame to NumPy array by using the to_numpy() method as shown in the example below:

In [ ]:
```
import pandas as pd
import numpy as np
# Pandas DataFrame
df = pd.DataFrame(data={'A': [3, 2, 1], 'B': [6,5,4], 'C': [9, 8, 7]},
                  index=['i', 'j', 'k'])
print("Pandas DataFrame: ")
print(df)

# Convert Pandas DataFrame to NumPy Array
np_arr = df.to_numpy()
print("Pandas DataFrame to NumPy array: ")
print(np_arr)


# Convert specific columns of Pandas DataFrame to NumPy array
arr = df[['B', 'C']].to_numpy()
print("Convert B and C columns of Pandas DataFrame to NumPy Array: ")
print (arr)
```

```
Pandas DataFrame:
   A  B  C
i  3  6  9
j  2  5  8
k  1  4  7
Pandas DataFrame to NumPy array:
[[3 6 9]
 [2 5 8]
 [1 4 7]]
Convert B and C columns of Pandas DataFrame to NumPy Array:
[[6 9]
 [5 8]
 [4 7]]
```

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

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.

In [ ]:
```
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)
```

```
After vstack:
 [[1 2 3]
 [4 5 6]]
After hstack:
 [1 2 3 4 5 6]
```

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

In [ ]:
```
import numpy as np
arr = np.arange(10, 60)
print(arr)
```

```
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
 58 59]
```

Matplotlib Interview Questions

1.What is Matplotlib, and how is it used for data visualization in Python?

Matplotlib is a comprehensive data visualization library in Python used for creating static, animated, and interactive visualizations in a wide range of formats. It was originally developed by John D. Hunter in 2003 and has since become a go-to library for researchers, analysts, and scientists for creating high-quality plots and charts.

Key Features of Matplotlib:

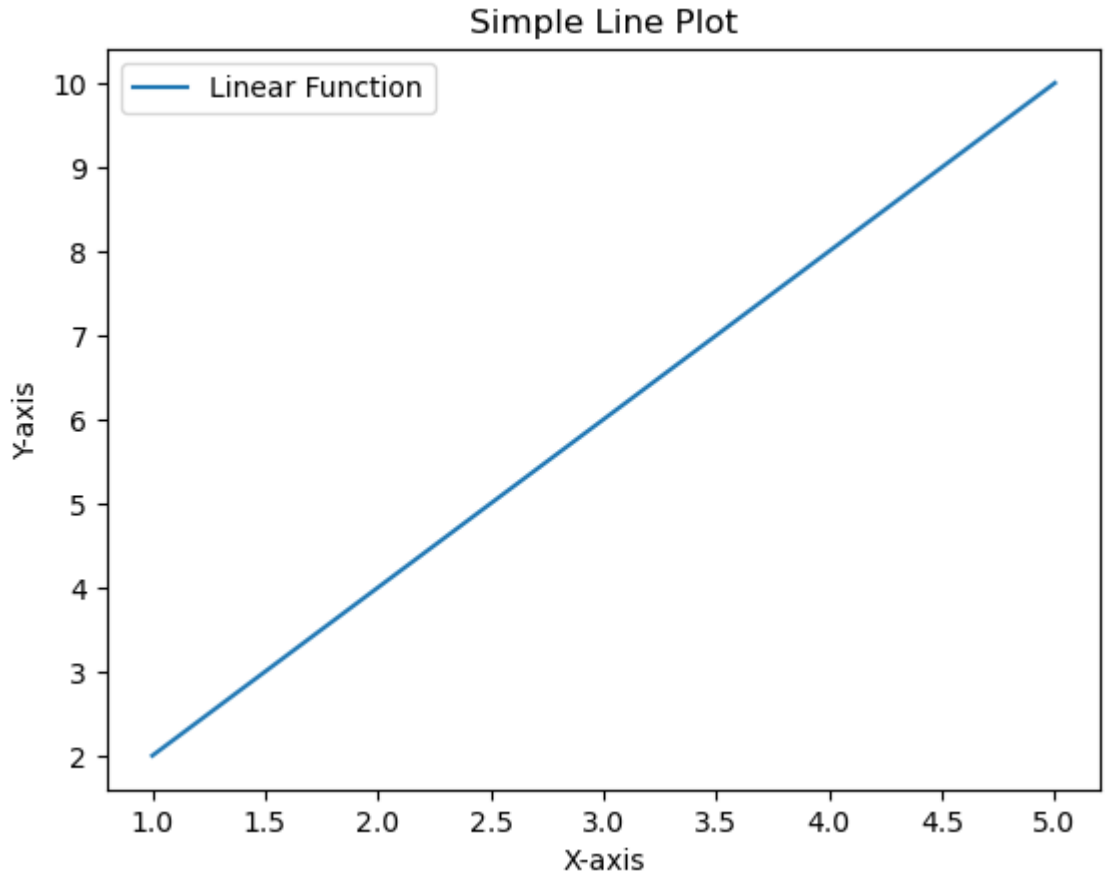
1. **Wide Range of Plot Types:**
   - Matplotlib supports various types of plots, including line plots, scatter plots, bar plots, histograms, pie charts, 3D plots, and more.
2. **Customization and Styling:**
   - Matplotlib allows extensive customization of plots, including control over colors, line styles, markers, fonts, and other stylistic elements.
3. **Publication-Quality Graphics:**
   - Matplotlib is designed to create publication-quality graphics with precise control over every aspect of the plot. This makes it suitable for creating figures for academic papers and presentations.
4. **Multiple Backends:**
   - Matplotlib supports multiple backends for rendering graphics, including interactive backends for use in GUI applications and non-interactive backends for generating static images.
5. **Integration with Jupyter Notebooks:**
   - Matplotlib seamlessly integrates with Jupyter Notebooks, providing an interactive environment for data exploration and visualization.

Basic Usage:

To use Matplotlib for data visualization, you need to import the `matplotlib.pyplot` module. Here's a simple example:

In [ ]:
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Create a line plot
plt.plot(x, y, label='Linear Function')
```

```
# Add labels and a legend
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
plt.legend()

# Display the plot
plt.show()
```

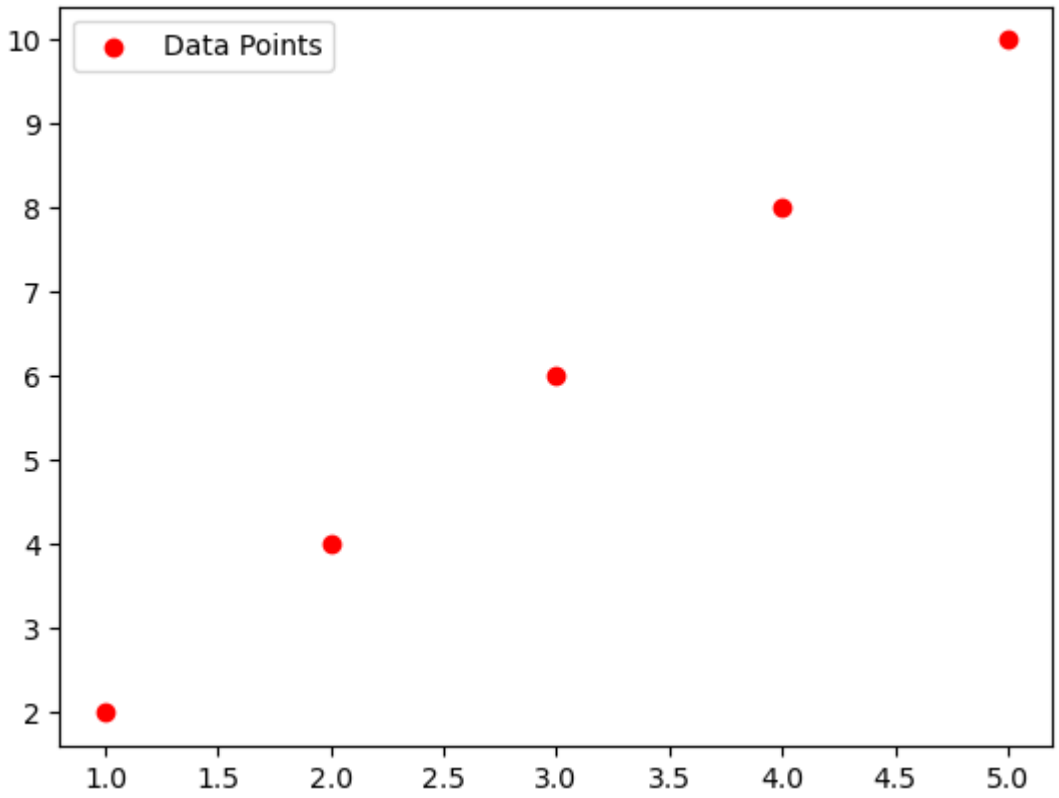


This example creates a basic line plot using Matplotlib. You can customize various aspects of the plot, such as colors, markers, and line styles, to suit your preferences.

Example Plot Types:

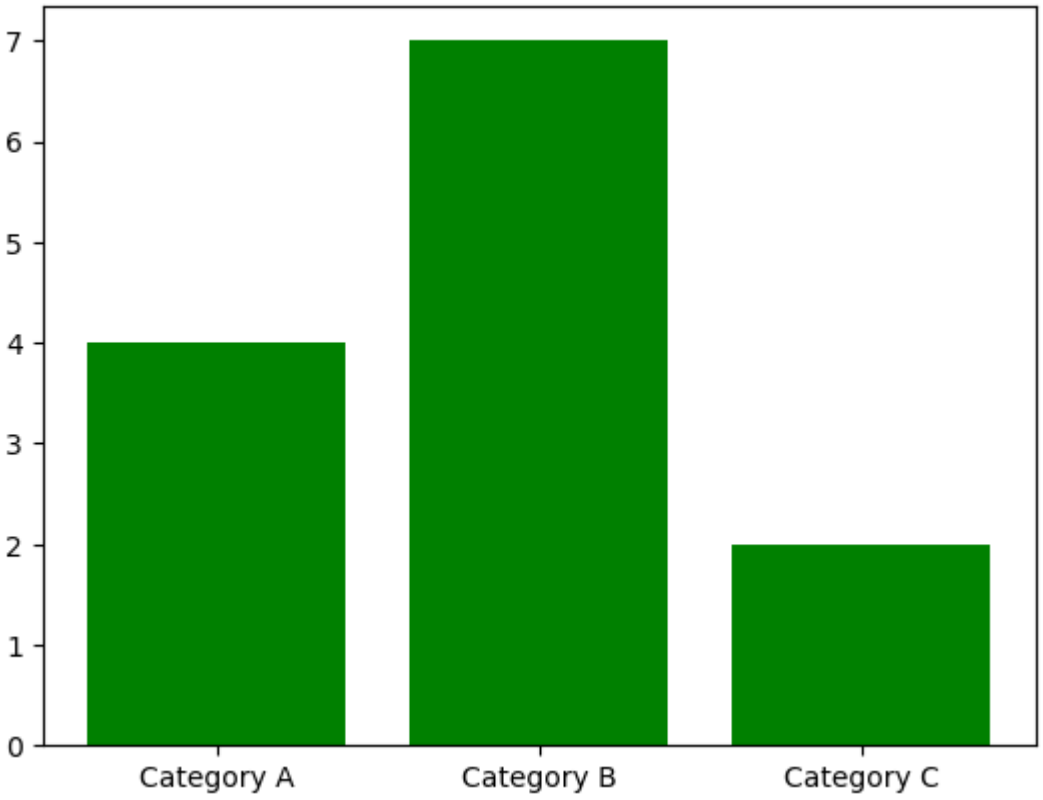
1. Scatter Plot:

In [ ]:
```
# Scatter plot
plt.scatter(x, y, label='Data Points', color='red', marker='o')
plt.legend()
plt.show()
```



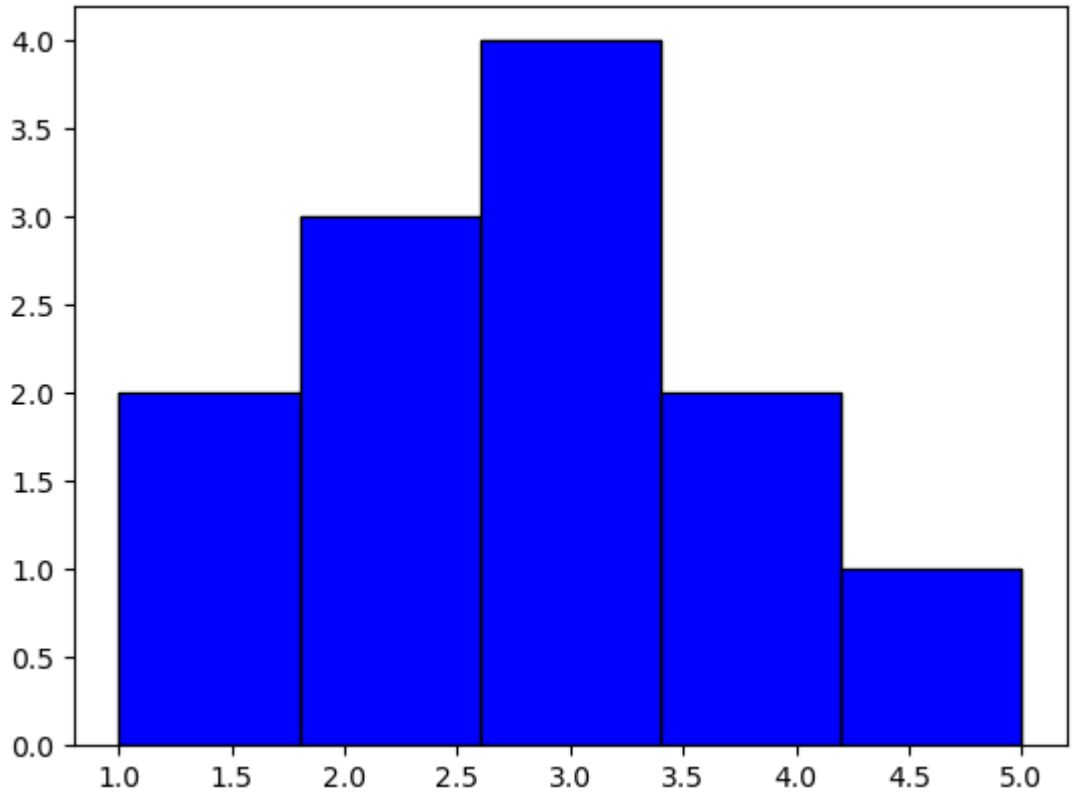
2. Bar Plot:

In [ ]:
```
# Bar plot
categories = ['Category A', 'Category B', 'Category C']
values = [4, 7, 2]
plt.bar(categories, values, color='green')
plt.show()
```



3. Histogram:

In [ ]:
```
# Histogram
data = [1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5]
plt.hist(data, bins=5, color='blue', edgecolor='black')
plt.show()
```



These are just a few examples, and Matplotlib provides a wide range of options and customization possibilities. Whether you need simple visualizations or complex, publication-quality graphics, Matplotlib is a powerful and flexible tool for data visualization in Python.

2. How do you create a basic line plot using Matplotlib?

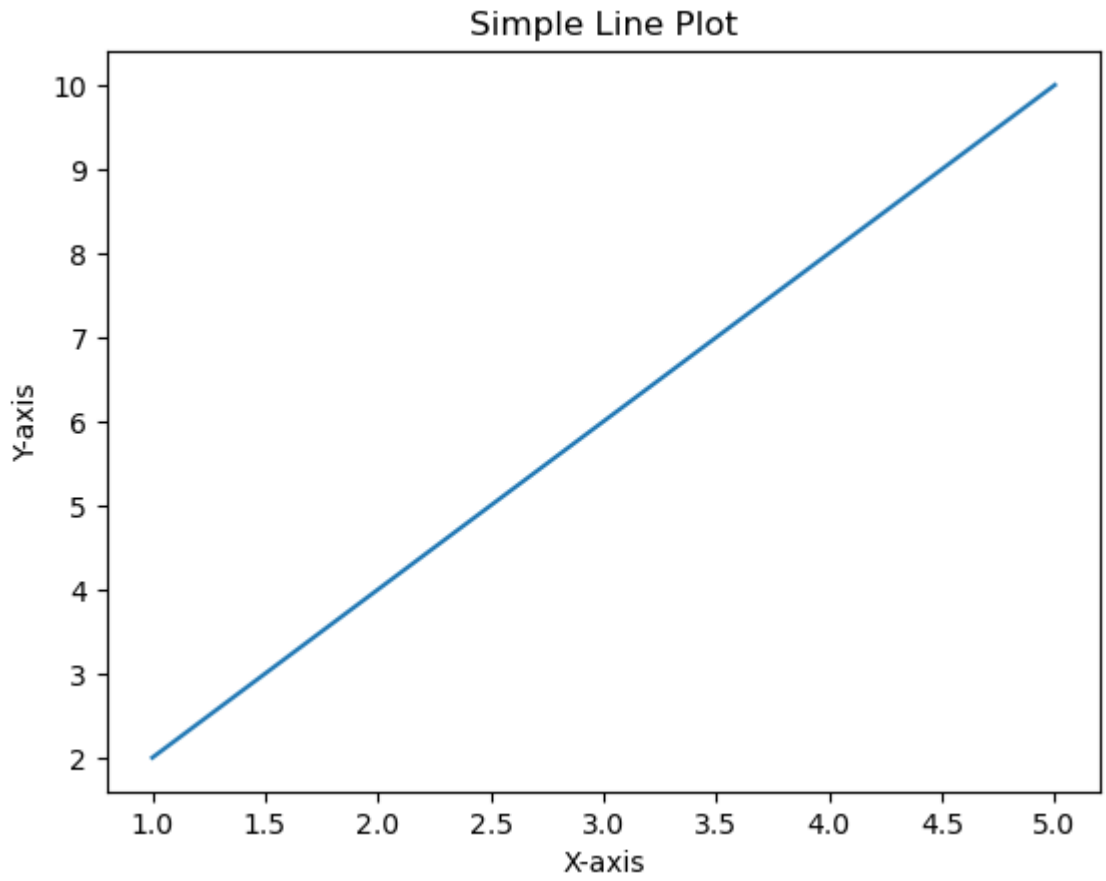
Creating a basic line plot using Matplotlib involves several steps:

- importing the matplotlib.pyplot module,
- providing data, and
- using the plot function to create the plot.

Here's a simple example:

In [ ]:
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Create a line plot
plt.plot(x, y)

# Add labels and a title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
```

```
plt.title('Simple Line Plot')

# Display the plot
plt.show()
```



3. Explain the difference between plt.figure() and plt.subplot() in matplotlib

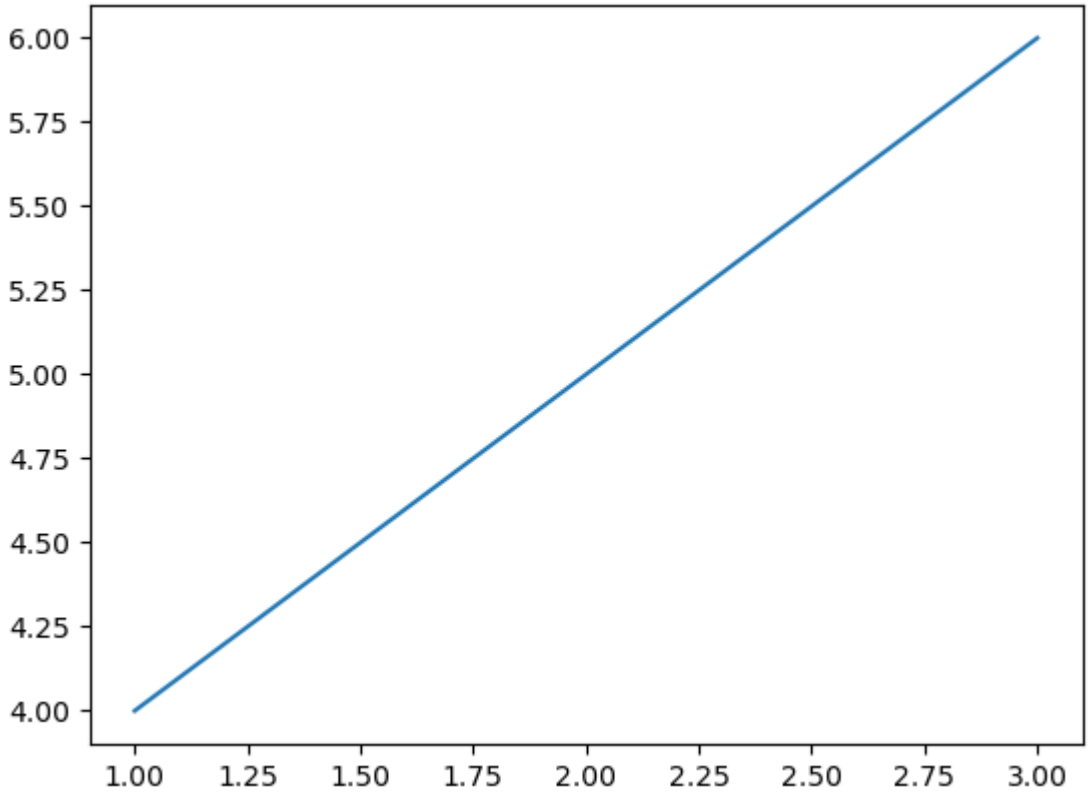
In Matplotlib, `plt.figure()` and `plt.subplot()` are two functions that are used for creating different components of a plot, and they serve distinct purposes.

`plt.figure()`:

The `plt.figure()` function is used to create a new figure or a new plotting window. A figure is the top-level container that holds all elements of a plot, including one or more axes (subplots), titles, legends, etc. When you create a figure using `plt.figure()`, you can customize the overall properties of the entire plot.

In [ ]:
```
import matplotlib.pyplot as plt

# Create a new figure
fig = plt.figure()

# Add an axes to the figure
ax = fig.add_subplot(111)  # 111 indicates a single subplot (1 row, 1 column, first

# Plot on the axes
ax.plot([1, 2, 3], [4, 5, 6])
```

```
# Display the plot
plt.show()
```

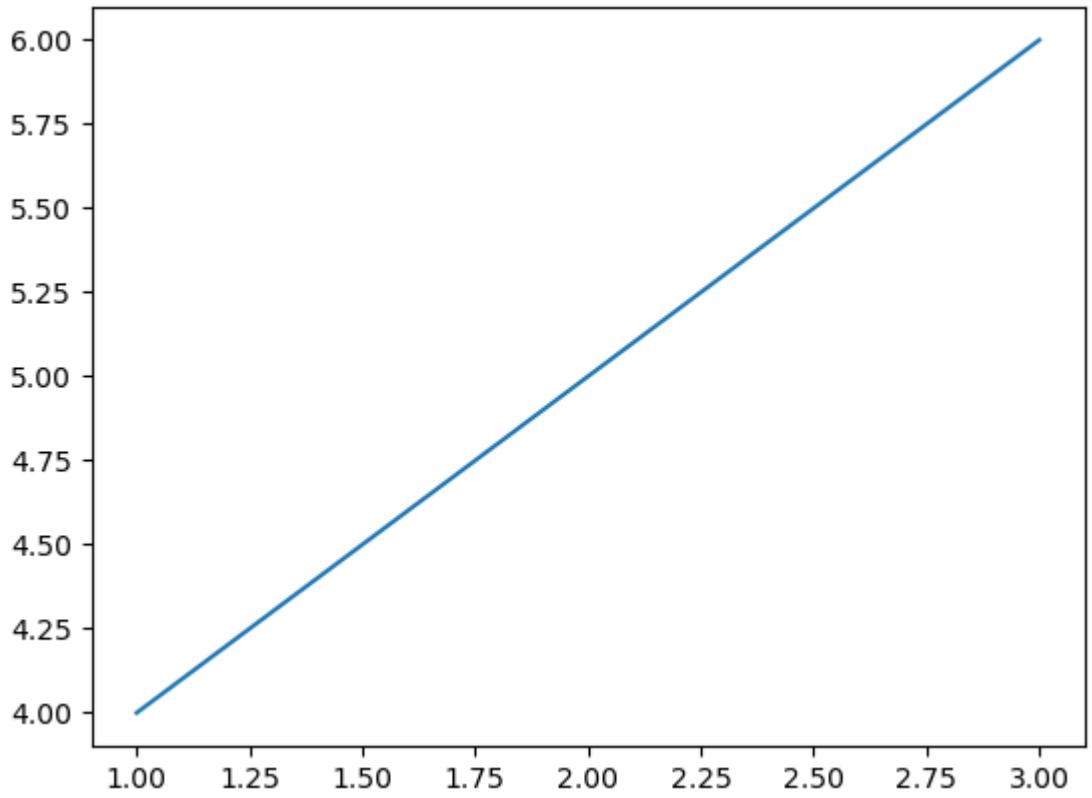


In this example, `plt.figure()` creates a new figure, and `fig.add_subplot(111)` adds a single subplot to the figure. The `111` argument indicates that the figure should have a 1x1 grid of subplots, and we are currently working with the first subplot.

`plt.subplot()`:

The `plt.subplot()` function is used to create and return a single subplot within a grid of subplots. It is a convenience function for creating a figure and adding a subplot to it in a single call. The function takes three arguments: the number of rows, the number of columns, and the index of the subplot you want to create.

In [ ]:
```
import matplotlib.pyplot as plt

# Create a single subplot within a 1x1 grid
plt.subplot(111)

# Plot on the subplot
plt.plot([1, 2, 3], [4, 5, 6])

# Display the plot
plt.show()
```



In this example, `plt.subplot(111)` creates a single subplot within a 1x1 grid. The `111` argument indicates that the subplot should be created in the first (and only) position in the grid.

Key Differences:

- **`plt.figure()` creates the top-level container:**
  - It is used to create a new figure, and you can then add one or more subplots (axes) to this figure.
  - Useful when you want to customize properties of the entire plot, such as size, title, or background color.
- **`plt.subplot()` creates a single subplot:**
  - It creates and returns a single subplot within a grid of subplots.
  - It is convenient when you need only one subplot or when you want to create a specific arrangement of subplots.

In summary, `plt.figure()` is used to create the overall figure, while `plt.subplot()` is used to create and return a single subplot within that figure. The choice between them depends on your specific needs and how you want to organize and customize your plots.

`4.What is purpose of the plt.legend() function in matplotlib?`

The `plt.legend()` function in Matplotlib is used to add a legend to the current axes or subplot. A legend is a key that labels the elements of a plot and helps the viewer identify

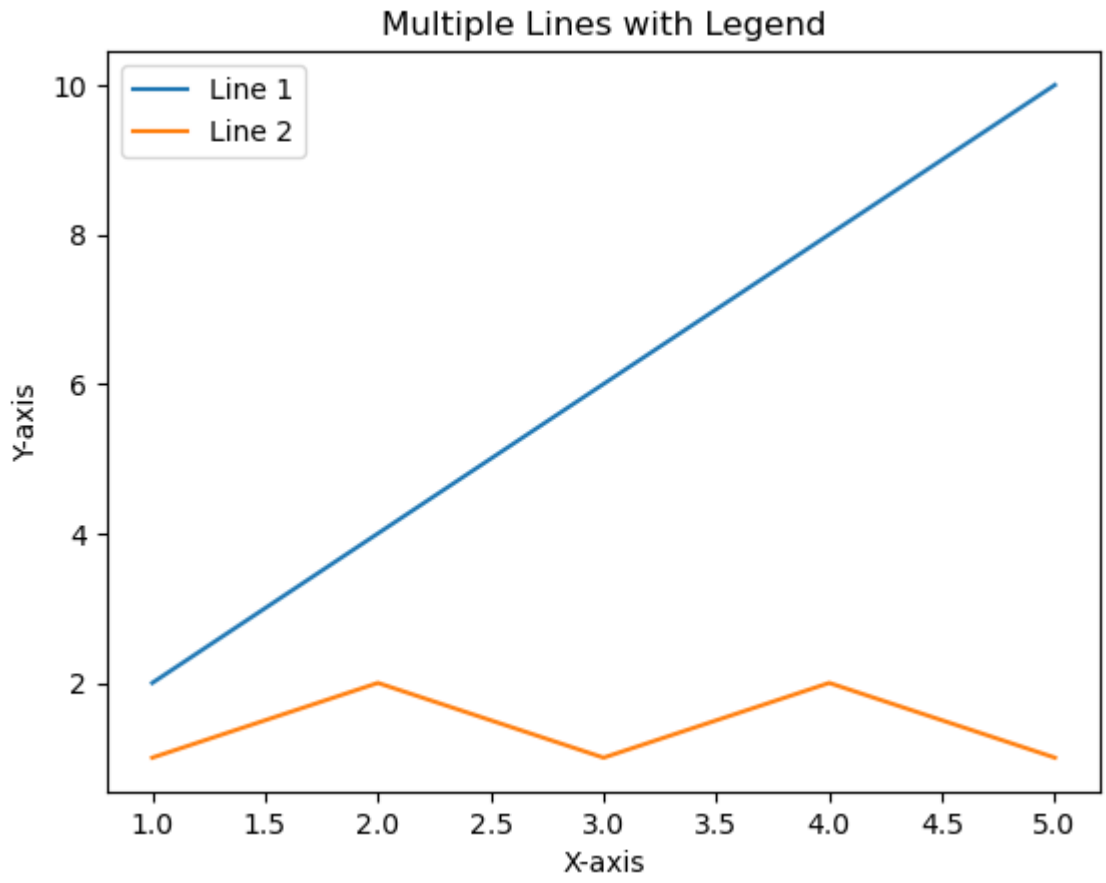
which data corresponds to which part of the plot. This is particularly useful when multiple datasets or multiple plot elements are present on the same axes.

Purpose of `plt.legend()`:

1. **Identifying Plot Elements:**
   - The primary purpose of the legend is to label different plot elements, such as lines, markers, or other graphical elements, with descriptive text. This helps the viewer understand the meaning of each element in the plot.
2. **Adding Context to the Plot:**
   - The legend provides additional context to the plot by associating labels with specific data series or plot components. This is crucial when presenting complex visualizations.
3. **Improving Readability:**
   - In cases where multiple datasets or plot elements overlap, a legend can improve the readability of the plot by clearly indicating which element belongs to which dataset.

Here's a simple example of using `plt.legend()`:

In [ ]:
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [1, 2, 1, 2, 1]

# Plotting two lines
plt.plot(x, y1, label='Line 1')
plt.plot(x, y2, label='Line 2')

# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Multiple Lines with Legend')

# Adding legend
plt.legend()

# Display the plot
plt.show()
```



In this example, `plt.legend()` is used to add a legend to the plot. The `label` argument in the `plot` function is used to assign labels to each line, and these labels are then displayed in the legend.

5. Explain the difference between a scatter plot and line plot in Matplotlib

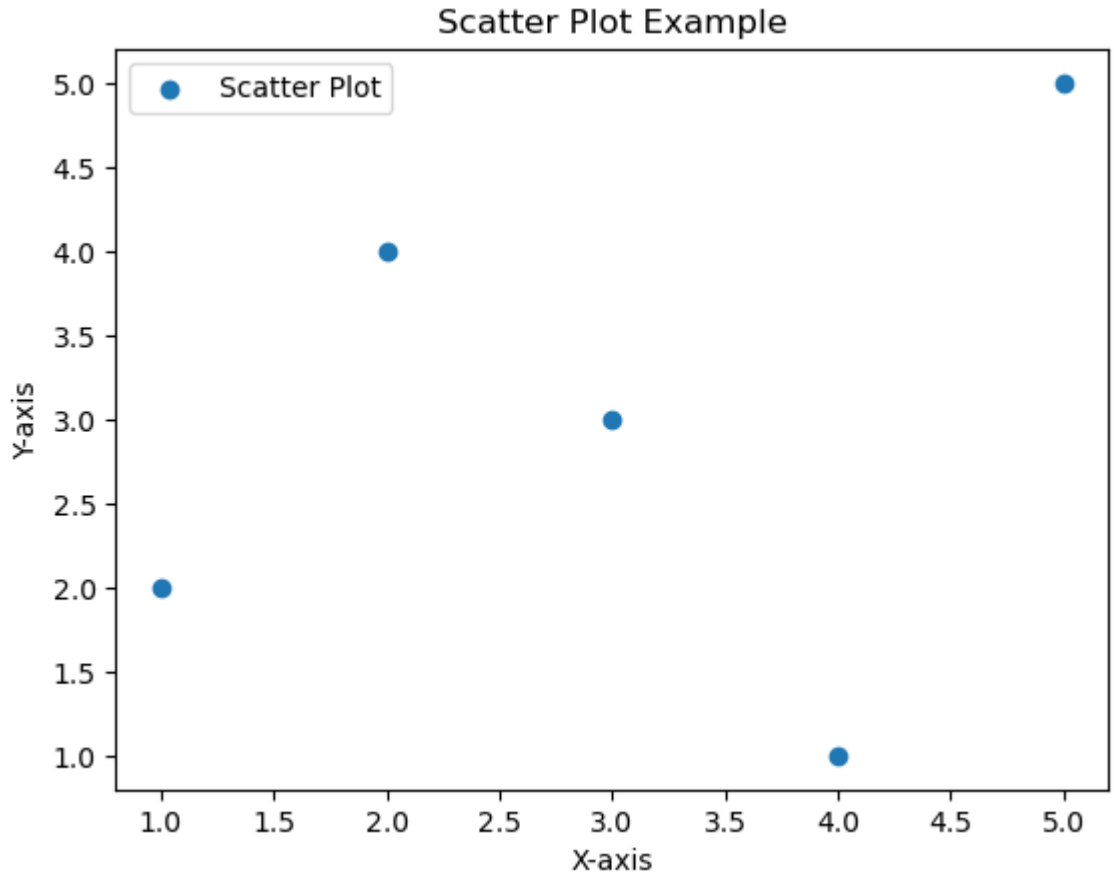
Both scatter plots and line plots are types of 2D plots used in Matplotlib to visualize relationships and patterns in data, but they represent data in different ways.

Scatter Plot:

- **Representation:**
  - A scatter plot represents individual data points as markers (dots) on the plot. Each point corresponds to a single data entry.
- **Use Case:**
  - Scatter plots are useful when you want to visualize the distribution of individual data points, examine relationships between two variables, or identify patterns such as clusters or outliers.

In [ ]:
```
import matplotlib.pyplot as plt

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

plt.scatter(x, y, label='Scatter Plot')
```

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot Example')
plt.legend()
plt.show()
```



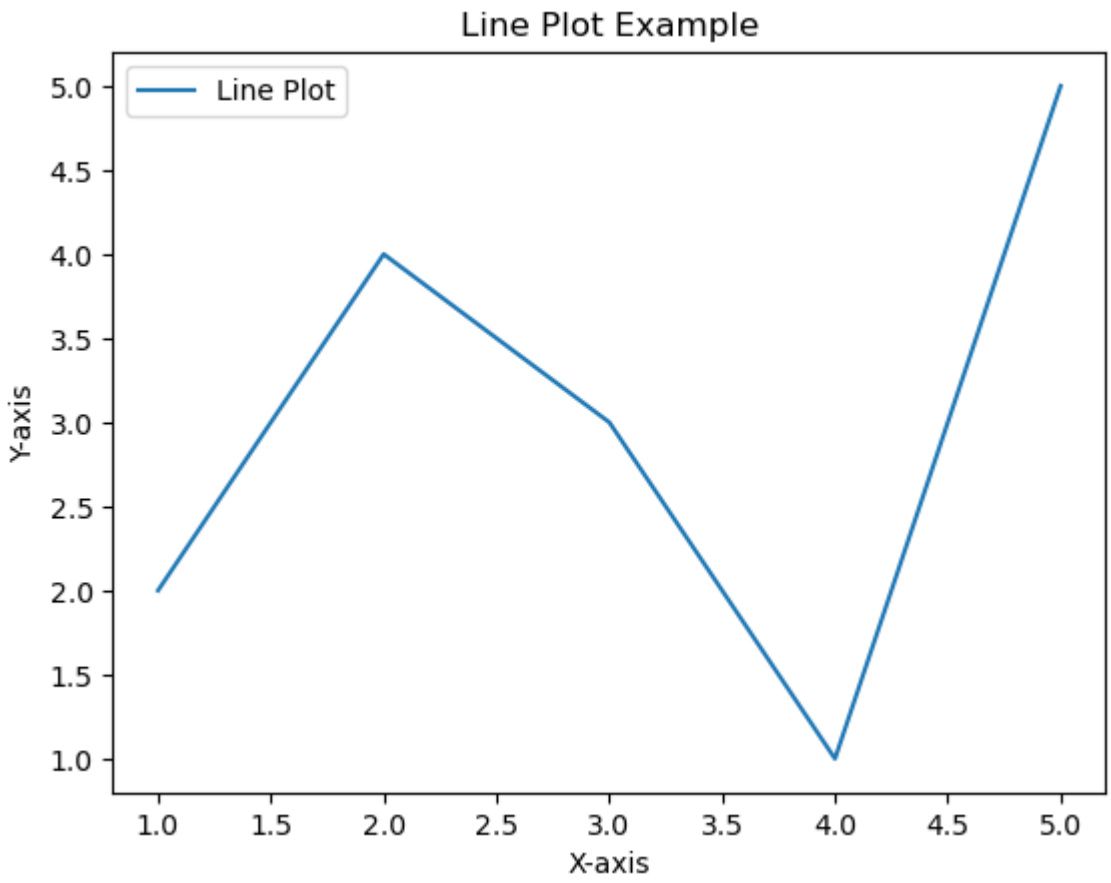
- **Customization:**
  - Scatter plots offer customization options for marker styles, sizes, and colors to enhance the visualization.

Line Plot:

- **Representation:**
  - A line plot represents data points by connecting them with straight lines. It is used to visualize trends, patterns, or the overall shape of the data.
- **Use Case:**
  - Line plots are commonly used to show the relationship between two continuous variables, particularly when there's a sequential order or a sense of continuity in the data.

In [ ]:
```
import matplotlib.pyplot as plt

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

plt.plot(x, y, label='Line Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
```

```
plt.title('Line Plot Example')
plt.legend()
plt.show()
```



- **Customization:**
  - Line plots provide customization options for line styles, colors, and markers, allowing you to emphasize specific aspects of the data.

Key Differences:

1. **Representation:**
   - Scatter plots show individual data points, while line plots connect data points with lines.
2. **Use Case:**
   - Scatter plots are suitable for examining distributions and relationships between two variables. Line plots are commonly used to show trends and patterns in sequential or continuous data.
3. **Data Type:**
   - Scatter plots work well with discrete or continuous data. Line plots are most effective when the data is sequential or continuous.
4. **Visual Emphasis:**
   - Scatter plots emphasize the distribution of individual data points. Line plots emphasize the overall trend or pattern in the data.

Python Coding Questions

1. Write a program that will convert celsius value to fahrenheit.

In [ ]:
```
def celsius_to_fahrengeit(celsius):
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit

# get the celsius input from the user
celsius_input = float(input('Enter the temperature in celsius: '))

print('celsius input:',celsius_input)

# convert celsius to far
result = celsius_to_fahrengeit(celsius_input)

# display the result
print(f"{celsius_input} degrees Celsius is equal to {result} degrees Fahrenheit.")
```

```
celsius input: 43.0
43.0 degrees Celsius is equal to 109.4 degrees Fahrenheit.
```

2.Take 2 numbers as input from the user.Write a program to swap the numbers without using any special python syntax.

In [ ]:
```
# get two numbers from the user
num1 = float(input('first number'))
num2 = float(input('secound number'))

# display the number which is provided by user
print(f"Before swapping: num1 = {num1}, num2 = {num2}")

# swap the number using temporial vaariable
temp = num1
num1 = num2
num2 = temp

# Display the numbers after swapping
print(f"After swapping: num1 = {num1}, num2 = {num2}")
```

```
Before swapping: num1 = 12.0, num2 = 34.0
After swapping: num1 = 34.0, num2 = 12.0
```

3. Write a program to find the euclidean distance between two coordinates.Take both the coordinates from the user as input.

In [ ]:
```
import math

# Get input from the user for the first coordinate
x1 = float(input("Enter the x-coordinate of the first point: "))
```

```
y1 = float(input("Enter the y-coordinate of the first point: "))

# Get input from the user for the second coordinate
x2 = float(input("Enter the x-coordinate of the second point: "))
y2 = float(input("Enter the y-coordinate of the second point: "))

# Calculate the Euclidean distance
distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# Display the result
print(f"The Euclidean distance between ({x1}, {y1}) and ({x2}, {y2}) is {distance:.
```

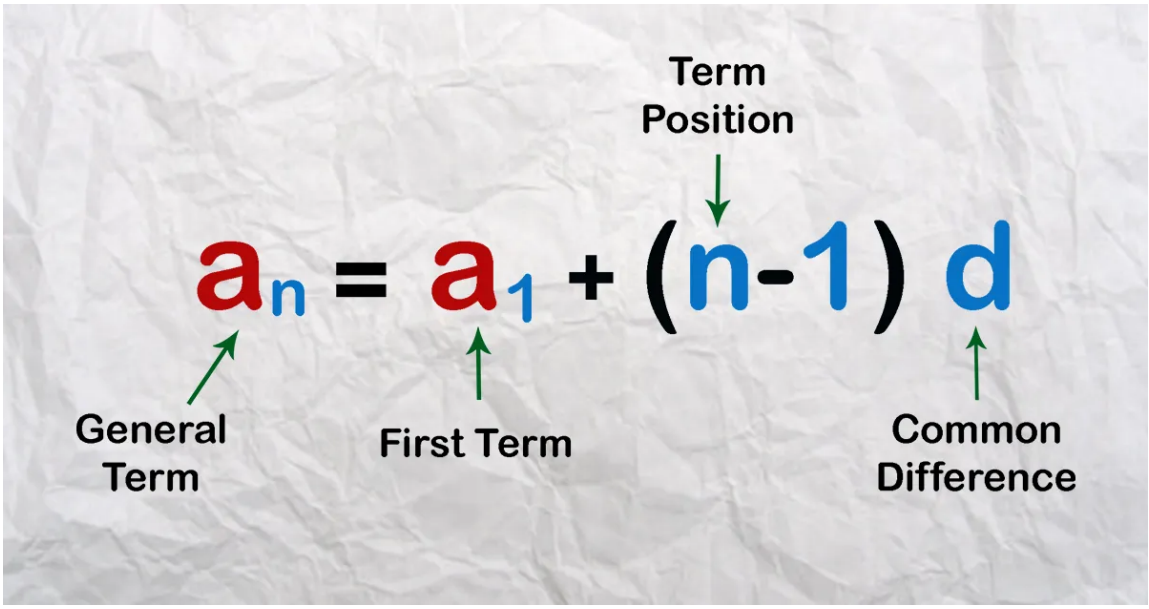
```
The Euclidean distance between (4.0, 3.0) and (3.0, 2.0) is 1.41
```

4. Write a program to find the sum of squares of first n natural numbers where n will be provided by the user.

In [ ]:
```
# # get the n input fron the user
n = int(input('enter the postive integer (n): '))

# calculate the sum of the square
sum_of_squares = sum(i**2 for i in range(1,n+1))

# Display the result
print(f"The sum of squares of the first {n} natural numbers is: {sum_of_squares}")
```

```
The sum of squares of the first 54 natural numbers is: 53955
```

5. Given the first 2 terms of an Arithmetic Series.Find the Nth term of the series. Assume all inputs are provided by the user.



In [ ]:
```
# Get input from the user for the first term, second term, and term number (n)
first_term = float(input("Enter the first term of the arithmetic series: "))
second_term = float(input("Enter the second term of the arithmetic series: "))
n = int(input("Enter the term number (n) you want to find: "))

# Calculate the common difference
```

```
common_difference = second_term - first_term

# Calculate the nth term of the arithmetic series
nth_term = first_term + (n - 1) * common_difference

# Display the result
print(f"The {n}th term of the arithmetic series is: {nth_term}")
```

```
The 5th term of the arithmetic series is: -1.0
```

Python Pattern Programing Questions

Problem 1. Print the following pattern. Write a program to use for loop to print the following reverse number pattern.

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

In [ ]:
```
rows = 5

for i in range(rows, 0, -1):
    for j in range(i, 0, -1):
        print(j, end=' ')
    print()
```

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

Problem 2: Print the following pattern.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

In [ ]:
```
rows = 5

for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()
```

```
*
* *
* * *
* * * *
* * * * *
```

In [ ]:
```
for i in range(rows - 1, 0, -1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()
```

```
* * * *
* * *
* *
*
```

In [ ]:
```
rows = 5

for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()

for i in range(rows - 1, 0, -1):
    for j in range(1, i + 1):
        print("*", end=" ")
    print()
```

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

Problem 3:Write a program to pring the following pattern

```
    *
  * * *
* * * * *
```

In [ ]:
```
rows = 5

for i in range(1, rows + 1):
    # Print leading spaces
    for j in range(rows - i):
        print(" ", end=" ")

    # Print asterisks
    for k in range(2 * i - 1):
        print("*", end=" ")

    print()
```

```
    *
  * * *
* * * * *
  * * * * * * *
* * * * * * * * *
```

Problem 4: Write a program to print the following pattern

1

2 1

3 2 1

4 3 2 1

5 4 3 2 1

In [ ]:
```
rows = 5

for i in range(1, rows + 1):
    for j in range(i, 0, -1):
        print(j, end=" ")
    print()
```

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
```

5. Inverted Half Pyramid Number Pattern

Pattern:

0 1 2 3 4 5

0 1 2 3 4

0 1 2 3

0 1 2

0 1

0

In [ ]:
```
rows = 6

for i in range(rows, 0, -1):
    for j in range(0, i):
        print(j, end=" ")
    print()
```

```
0 1 2 3 4 5
0 1 2 3 4
0 1 2 3
0 1 2
0 1
0
```

Python Programming Questions

Problem 1 - Find the sum of the series upto n terms.

Write a program to calculate the sum of series up to n term. For example, if n =5 the series will become 2 + 22 + 222 + 2222 + 22222 = 24690. Take the user input and then calculate. And the output style should match which is given in the example.

Example 1:

Input:

5

Output:

2+22+222+2222+22222
Sum of above series is: 24690

In [ ]:
```
def calculate_series_sum(n):
    series_sum = 0
    series = ""

    for i in range(1, n + 1):
        term = int("2" * i)
        series_sum += term
        series += str(term)

        if i < n:
            series += "+"

    return series, series_sum

# Taking user input
n = int(input("Enter the value of n: "))

# Calculating series sum
result_series, result_sum = calculate_series_sum(n)

# Displaying the series and its sum
print(f"Series: {result_series}")
print(f"Sum of above series is: {result_sum}")
```

```
Series: 2+22+222+2222+22222
Sum of above series is: 24690
```

For visualizing -> Python Tutor - https://pythontutor.com/render.html#mode=edit

Problem 2: Create Short Form from initial character

Given a string create short form ofthe string from Initial character. Short form should be capitalised.

Example:

`Input:`

```
Oh My God
```

`Output:`

```
OMG
```

In [ ]:
```python
def short_form_of_string(string):
    words = string.split()
    short_form = "".join(word[0].upper() for word in words)
    return short_form

string = input('Enter the string: ')

# result
result = short_form_of_string(string)

# Displaying the short form
print(string)
print(f"Short form: {result}")
```

```
oh my god
Short form: OMG
```

Problem 3 : Reverse words in a given String

Statement: We are given a string and we need to reverse words of a given string.

Example 1:

`Input:`

```
my name is Dishant
```

`Output:`

```
Dishant is name my
```

In [ ]:
```python
def reverse_words(string):
    words = string.split()
    reverse_words = ' '.join(reversed(words))
    return reverse_words

# for input
string = input('enter the string: ')

# reversimg string
result = reverse_words(string)

# print result
# Displaying the reversed words

print(string)
print(f"Reversed words: {result}")
```

```
my name is Dishant
Reversed words: Dishant is name my
```