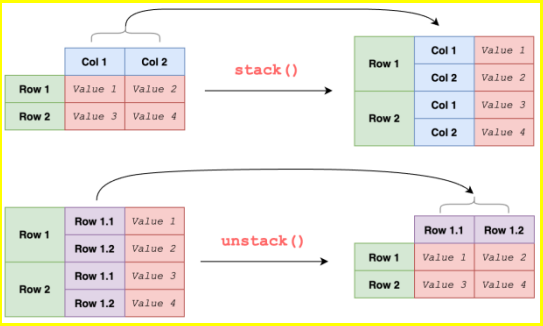
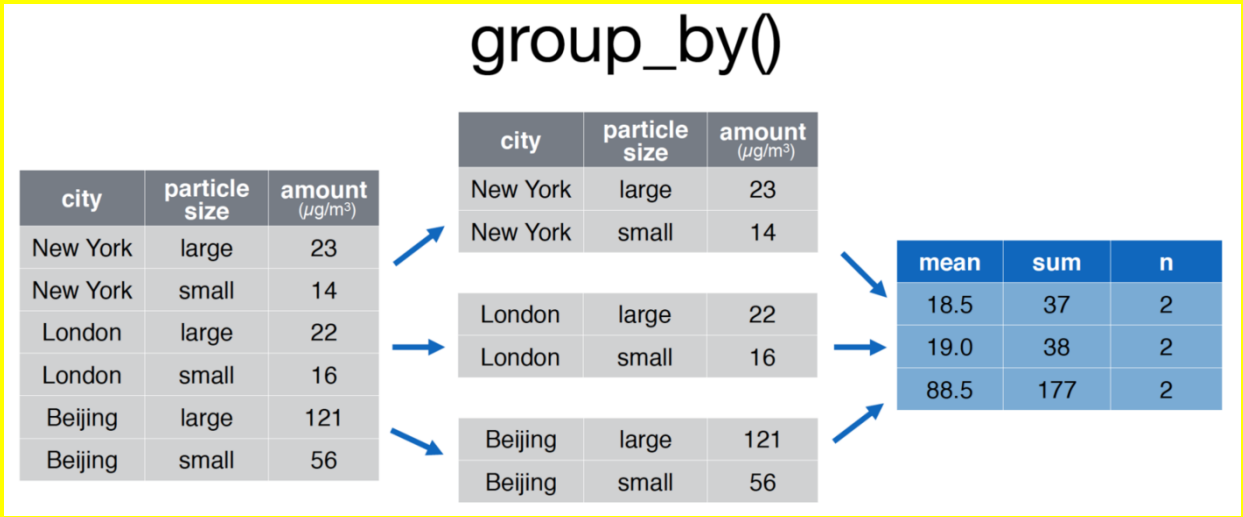
Machine Learning Part 05

01. List comprehension Questions
02. Python Programming Questions
03. Tuple and Sets Python Coding Questio
04. Programming Questions
05. Pandas Questions
06. Python Interview Question
07. Revision of Encapsulation in OOP in Python
08. Revision of Aggregation OOP
09. Revision of Inheritance in OOP
10. Revision Of Polymorphism and Abstraction
11. Revision of
12. Revision of GroupBy object in pandas Part 1
13. Revision Of GroupBy object in Pandas Part 2
14. Revision of Vectorized String operation
15. Python Use cases Interview Questions
16. Python Mastery Unleashed!
17. Python Interview Use-cases Questions
18. OOP (Python) Interview Questions
19. List comprehension Questions
20. Dictionary comprehension Interview Questions





List Comprehension Questions

1.Write a Python code to create a new list that contains only the even numbers from the given list.

In [ ]:
```
# Sample Input
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Solution using list comprehension
even_numbers = [num for num in numbers if num % 2 == 0]

# Output
print(even_numbers)
```

```
[2, 4, 6, 8, 10]
```

2. Write a Python code to create a new list that contains squares of even numbers and cubes of odd numbers from the given list.

In [ ]:
```
# Sample Input
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Solution using list comprehension
result = [num ** 2 if num % 2 == 0 else num ** 3 for num in numbers]

# Output
print(result)
```

```
[1, 4, 27, 16, 125, 36, 343, 64, 729, 100]
```

3.Write a Python code to create a new list that contains the squares of numbers from 1 to 10.

In [ ]:
```
squred_numbers = [num ** 2 for num in range(1,11)]

# output
print(squred_numbers)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

4.Given two lists, create a new list that contains the common elements using list comprehension.

In [ ]:
```
list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 5, 6, 7]

common_elements = [x for x in list1 if x in list2]

print(common_elements)
```

```
[3, 4, 5]
```

5.Given a list of numbers, create a new list that contains the square of each number if it is positive, and zero otherwise.

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

squares = [x**2 if x > 0 else 0 for x in numbers]

print(squares)
```

```
[1, 0, 9, 0, 25, 0]
```

6.Write a list comprehension to extract all the vowels from a given string.

In [ ]:
```
input_string = "hello world"

vowels = [char for char in input_string if char.lower() in 'aeiou']

print(vowels)
```

```
['e', 'o', 'o']
```

Python Programming Questions

Problem 1: Running Sum on list, Write a program to print a list after performing running sum on it

i.e:

Input:

list1 = [1,2,3,4,5,6]

Output:

[1,3,6,10,15,21]

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

# Perform running sum
for i in range(1, len(list1)):
    list1[i] += list1[i - 1]

print(list1)
```

```
[1, 3, 6, 10, 15, 21]
```

Problem 2: Find list of common unique items from two list. and show in increasing order

Input

num1 = [23,45,67,78,89,34] num2 = [34,89,55,56,39,67]

Output:

[34, 67, 89]

In [ ]:
```
num1 = [23, 45, 67, 78, 89, 34]
num2 = [34, 89, 55, 56, 39, 67]

# Find common unique items
common_items = list(set(num1) & set(num2))

# Sort the common items in increasing order
common_items.sort()

print(common_items)
```

```
[34, 67, 89]
```

Problem 3: Write a program that can perform union operation on 2 lists

Example:

Input:

```
[1,2,3,4,5,1]
[2,3,5,7,8]
```

Output:

```
[1,2,3,4,5,7,8]
```

In [ ]:
```
list1 = [1, 2, 3, 4, 5, 1]
list2 = [2, 3, 5, 7, 8]

# Perform union operation
union_result = list(set(list1) | set(list2))

print(union_result)
```

```
[1, 2, 3, 4, 5, 7, 8]
```

Problem 4: Write a program that can find the max number of each row of a matrix

Example:

Input:

```
[[1,2,3],[4,5,6],[7,8,9]]
```

Output:

```
[3,6,9]
```

In [ ]:
```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Find the max number of each row
max_numbers = [max(row) for row in matrix]

print(max_numbers)
```

```
[3, 6, 9]
```

Tuple and Sets Python Coding Questions

Questions 1: Check is tuples are same or not?

Two tuples would be same if both tuples have same element at same index

```
t1 = (1,2,3,0)
t2 = (0,1,2,3)

t1 and t2 are not same
```

In [ ]:
```python
t1 = (1,2,3,0)
t2 = (0,1,2,3)

# define function for same tuple

def are_tuple_same(tuple1,tuple2):
    # check if the length of the tuple is same or not
    if len(tuple1)!=len(tuple2):
        return False

    # check if elements at corresponding indices are the same
    for i in range(len(tuple1)):
        if tuple1[i] != tuple2[i]:
            return False

    # tuples are same if all elemets are equal
    return True

if are_tuple_same(t1,t2):
    print('t1 and t2 are the same')

else:
    print('t1 and t2 are not same')
```

```
t1 and t2 are not same
```

In [ ]:
```python
# EXAMPLE 2
t1 = (1,2,3,4,5)
t2 = (1,2,3,4,5)

if are_tuple_same(t1,t2):
    print('t1 and t2 are the same')

else:
    print('t1 and t2 are not same')
```

```
t1 and t2 are the same
```

Question 2: Count no of tuples, list and set from a list

`list1 = [{'hi', 'bye'},{'Geeks', 'forGeeks'},('a', 'b'),['hi', 'bye'], ['a', 'b']]`

Output:

```
List-2
Set-2
Tuples-1
```

In [ ]:
```python
list1 = [{'hi', 'bye'}, {'Geeks', 'forGeeks'}, ('a', 'b'), ['hi', 'bye'], ['a', 'b'

# Initialize counters
tuple_count = 0
list_count = 0
set_count = 0

# Iterate through the elements in the list
for elem in list1:
    if isinstance(elem, tuple):
        tuple_count += 1
    elif isinstance(elem, list):
        list_count += 1
    elif isinstance(elem, set):
        set_count += 1

# Print the counts
print(f"List-{list_count}")
print(f"Set-{set_count}")
print(f"Tuples-{tuple_count}")
```

```
List-2
Set-2
Tuples-1
```

Question 3: Write a program to find set of common elements in three lists using sets.

```
Input : ar1 = [1, 5, 10, 20, 40, 80]
        ar2 = [6, 7, 20, 80, 100]
        ar3 = [3, 4, 15, 20, 30, 70, 80, 120]

Output : [80, 20]
```

In [ ]:
```python
arr1 = [1,5,10,20,40,80]
arr2 = [6,7,20,80,100]
arr3 = [3,4,15,20,30,70,80,120]

# convert list into sets
set1 = set(arr1)
set2 = set(arr2)
set3 = set(arr3)

# Find the common elements using set intersection
common_elements = set1.intersection(set2, set3)

# Convert the result back to a list
result_list = list(common_elements)

# Output
print("Output:", result_list)
```

```
Output: [80, 20]
```

Question 4: find union of n arrays.

Example 1:

Input:

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

Output:

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

In [ ]:

```python
def find_union(arrays):
    # Initialize an empty set for the union
    union_set = set()

    # Iterate through each array and update the union set
    for array in arrays:
        union_set.update(array)

    # Convert the set to a list (if needed)
    union_list = list(union_set)

    return union_list

# Example usage
input_arrays = [
    [1, 2, 2, 4, 3, 6],
    [5, 1, 3, 4],
    [9, 5, 7, 1],
    [2, 4, 1, 3]
]

output_union = find_union(input_arrays)
print("Output:", output_union)
```

```
Output: [1, 2, 3, 4, 5, 6, 7, 9]
```

Python Programming Questions

Problem-1: Write a Python function that takes a list and returns a new list with unique elements of the first list.

Input:

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

Output:

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

In [ ]:
```
def unique_elements(input_list):
    # Convert the list to a set to remove duplicates
    unique_set = set(input_list)

    # Convert the set back to a list
    result_list = list(unique_set)

    return result_list

# Example usage
input_list = [1, 2, 3, 3, 3, 3, 4, 5]


output_list = unique_elements(input_list)
print(output_list)
```

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

Problem-2: Write a Python function that accepts a hyphen-separated sequence of words as parameter and returns the words in a hyphen-separated sequence after sorting them alphabetically.

Input:

```
green-red-yellow-black-white
```

Output:

```
black-green-red-white-yellow
```

In [ ]:
```
def sort_words(input_sequence):
    # Split the hyphen-separated sequence into a list of words
    words_list = input_sequence.split('-')

    # Sort the list of words alphabetically
    sorted_words = sorted(words_list)

    # Join the sorted words into a hyphen-separated sequence
    result_sequence = '-'.join(sorted_words)
```

```
    return result_sequence

# Example
input_sequence = "green-red-yellow-black-white"
output_sequence = sort_words(input_sequence)
print(output_sequence)
```

```
black-green-red-white-yellow
```

Problem 3: Write a Python program to print the even numbers from a given list.

```
Sample List : [1, 2, 3, 4, 5, 6, 7, 8, 9]
Expected Result : [2, 4, 6, 8]
```

In [ ]:
```
def print_even_numbers(input_list):
    # Use a list comprehension to filter even numbers
    even_numbers = [i for i in input_list if i % 2 == 0]

    # Print the result or return the list
    print(even_numbers)


sample_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print_even_numbers(sample_list)
```

```
[2, 4, 6, 8]
```

Problem-4: Write a Python function to concatenate any no of dictionaries to create a new one.

```
Sample Dictionary :
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
Expected Result : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
```

In [ ]:
```
def concatenate_dictionaries(*dict):
    # Initialize an empty dictionary to store the result
    result_dict = {}

    # Iterate through each dictionary and update the result_dict
    for d in dict:
        result_dict.update(d)

    return result_dict

# Example usage
dic1 = {1: 10, 2: 20}
dic2 = {3: 30, 4: 40}
dic3 = {5: 50, 6: 60}

result_dictionary = concatenate_dictionaries(dic1, dic2, dic3)
print(result_dictionary)
```

```
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
```

Pandas Questions

Que-1: Write a program to create an empty series.

In [ ]:
```
import pandas as pd

# Creating an empty series
empty_series = pd.Series()


print(empty_series)
```

```
Series([], dtype: object)
```

Que-2: Write a Pandas program to add, subtract, multiple and divide two Pandas Series.

In [ ]:
```
# Creating two Pandas Series
series1 = pd.Series([1, 2, 3, 4])
series2 = pd.Series([5, 6, 7, 8])

# Addition
addition_result = series1 + series2

# Subtraction
subtraction_result = series1 - series2

# Multiplication
multiplication_result = series1 * series2

# Division
division_result = series1 / series2



print("\nAddition Result:\n", addition_result)
print("Subtraction Result:\n", subtraction_result)
print("Multiplication Result:\n", multiplication_result)
print("Division Result:\n", division_result)
```

```
Addition Result:
 0     6
1     8
2    10
3    12
dtype: int64
Subtraction Result:
 0   -4
1   -4
2   -4
3   -4
dtype: int64
Multiplication Result:
 0     5
1    12
2    21
3    32
dtype: int64
Division Result:
 0    0.200000
1    0.333333
2    0.428571
3    0.500000
dtype: float64
```

Que-3: 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]

In [ ]:

```
# Creating two Pandas Series
series1 = pd.Series([2, 4, 6, 8, 10])
series2 = pd.Series([1, 3, 5, 7, 10])

# Comparing elements
comparison_result_greater = series1 > series2
comparison_result_less = series1 < series2
comparison_result_equal = series1 == series2



print("\nElements in Series 1 greater than Series 2:\n", comparison_result_greater)
print("Elements in Series 1 less than Series 2:\n", comparison_result_less)
print("Elements in Series 1 equal to Series 2:\n", comparison_result_equal)
```

```
Elements in Series 1 greater than Series 2:
 0     True
1     True
2     True
3     True
4    False
dtype: bool
Elements in Series 1 less than Series 2:
 0    False
1    False
2    False
3    False
4    False
dtype: bool
Elements in Series 1 equal to Series 2:
 0    False
1    False
2    False
3    False
4     True
dtype: bool
```

Que-4.Write a function to change the data type of given a column or a Series. Function takes series and data type as input, returns the converted series.

```
series = pd.Series([1,2,'Python', 2.0, True, 100])
change to float type data
```

In [ ]:

```python
def convert_to_dtype(series, target_dtype):
    # Convert the series to numeric values
    numeric_series = pd.to_numeric(series, errors='coerce')

    # Convert the numeric series to the target data type
    converted_series = numeric_series.astype(target_dtype)

    return converted_series


input_series = pd.Series([1, 2, 'Python', 2.0, True, 100])
target_data_type = float

converted_series = convert_to_dtype(input_series, target_data_type)

# Displaying the results
print("Original Series:\n", input_series)
print("\nConverted Series to", target_data_type, ":\n", converted_series)
```

```
Original Series:
 0         1
1         2
2    Python
3       2.0
4      True
5       100
dtype: object

Converted Series to <class 'float'> :
 0      1.0
1      2.0
2      NaN
3      2.0
4      1.0
5    100.0
dtype: float64
```

Python Interview Question

`1. What is the difference between range and xrange?`

In Python 2, there were two functions for creating sequences of numbers: `range()` and `xrange()`. However, in Python 3, the `xrange()` function was removed, and `range()` was optimized to behave like `xrange()` from Python 2.

In Python 2:

1. **`range()`**:
    - It returns a list containing a sequence of numbers.
    - It generates the entire sequence and stores it in memory.
    - This can be inefficient for large ranges because it consumes a significant amount of memory.
2. **`xrange()`**:
    - It returns an xrange object, which is an iterator that generates numbers on-the-fly.
    - It is more memory-efficient for large ranges because it generates numbers as needed and doesn't store the entire sequence in memory.

In Python 3:

- The `xrange()` function is no longer available. The `range()` function in Python 3 was modified to behave like `xrange()` from Python 2, meaning it is an iterator and generates values on-the-fly.

So, in Python 3, you can use `range()` for creating sequences, and it will be memory-efficient like `xrange()` in Python 2. The use of `xrange()` is no longer necessary in Python 3.

2. What is pickling and unpickling in Python?

Pickling and unpickling are processes in Python used to serialize and deserialize objects, respectively. Serialization is the process of converting a Python object into a byte stream, and deserialization is the process of reconstructing the object from that byte stream. This is particularly useful for storing or transmitting complex data structures.

Pickling:

- **Definition:** Pickling is the process of converting a Python object into a byte stream.
- **Module:** The `pickle` module in Python is used for pickling.
- **Function:** The primary function for pickling is `pickle.dump()` or `pickle.dumps()`. The former writes the pickled representation to a file-like object, and the latter returns a string containing the pickled representation.

In [ ]:
```python
import pickle

data = {'name': 'John', 'age': 30, 'city': 'New York'}

with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)
```

Unpickling:

- **Definition:** Unpickling is the process of converting a byte stream back into a Python object.
- **Module:** The `pickle` module in Python is also used for unpickling.
- **Function:** The primary function for unpickling is `pickle.load()` or `pickle.loads()`. The former reads the pickled representation from a file-like object, and the latter reads from a string containing the pickled representation.

In [ ]:
```python
import pickle

with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)

print(loaded_data)
```

```
{'name': 'John', 'age': 30, 'city': 'New York'}
```

It's important to note that while pickling is a convenient way to serialize Python objects, caution should be exercised when unpickling data from untrusted or unauthenticated sources, as it can lead to security vulnerabilities (e.g., arbitrary code execution). In such cases, alternative serialization formats like JSON may be more appropriate.

3.What is _init_ in Python?

In Python, `__init__` is a special method (also known as a dunder method or magic method) that is used for initializing objects of a class. It is called automatically when a new instance of the class is created. The full name of the method is `__init__` (with double underscores before and after "init").

Here's a brief explanation of how `__init__` works:

- **Initialization:** The primary purpose of `__init__` is to initialize the attributes of the object. When you create a new instance of a class, `__init__` is called automatically, and you can use it to set up the initial state of the object.
- **Parameters:** `__init__` can take parameters, which are passed when you create an instance of the class. These parameters are used to initialize the attributes of the object.

In [ ]:
```python
class MyClass:
      def __init__(self, name, age):
          self.name = name
          self.age = age

  # Creating an instance of MyClass
```

```
obj = MyClass("John", 30)

  # Accessing the attributes
print(obj.name)
print(obj.age)
```

```
John
30
```

In the example above, the `__init__` method takes `self` (which represents the instance being created) along with two additional parameters (`name` and `age`). Inside the method, it initializes the attributes `name` and `age` with the values passed during the object creation.

Remember that `__init__` is just one of several special methods in Python classes. Other dunder methods include `__str__` for string representation, `__repr__` for a detailed representation, and so on. These methods allow you to define how objects of your class behave in various contexts.

Revision Of Encapsulation - OOP in Python

Encapsulation is one of the fundamental principles of object-oriented programming (OOP) and is supported in Python, just like in many other object-oriented languages. It refers to the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class. Additionally, it restricts direct access to some of the object's components, providing a controlled and well-defined interface to interact with the object.

In Python, encapsulation is implemented using access modifiers and naming conventions:

1. **Public:** Attributes and methods that are accessible from outside the class without any restrictions. By default, all attributes and methods in Python are public.
2. **Protected:** Attributes and methods that are intended to be used only within the class and its subclasses. In Python, you can denote a protected attribute or method by prefixing it with a single underscore (e.g., `_my_attribute`).
3. **Private:** Attributes and methods that are intended to be used only within the class. In Python, you can denote a private attribute or method by prefixing it with a double underscore (e.g., `__my_attribute`).

In [ ]:
```python
# if we have a one class

class Myclass:
    def __init__(self):
        self.public_attribute = 55
        self._protected_attribute = 'Hello'
        self.__private_attribute = 'World'

    def public_method(self):
        return "This is public method"

    def _protected_method(self):
        return "This is Protected Method"

    def __private_method(self):
        return "this is private Method"
```

In [ ]:
```python
obj = Myclass()
```

In [ ]:
```python
# we have to access the public attribaute dirctly and don't give any tyoe of error

print(obj.public_attribute)
print(obj.public_method())
```

```
55
This is public method
```

In [ ]:
```python
# so now we try for Protected attribute
print(obj._protected_attribute)
print(obj._protected_method())
```

```
Hello
This is Protected Method
```

In [ ]:
```
# Accessing private attributes and methods directly will result in an error:
print(obj.__private_attribute)   # Raises an AttributeError
print(obj.__private_method())    # Raises an AttributeError
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
c:\Users\disha\Downloads\Python 100 Days\100_Days_OF_Python\Day17 - Encapsulation in OOP in Python.ipynb Cell 7 line 2
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Pythoon%20100%20Days/100_Days_OF_Python/Day17%20-%20Encapsulation%20in%20OOP%20in%20Python.ipynb#W6sZmlsZQ%3D%3D?line=0'>1</a> # Accessing private attributes and methods directly will result in an error:
----> <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Pythoon%20100%20Days/100_Days_OF_Python/Day17%20-%20Encapsulation%20in%20OOP%20in%20Python.ipynb#W6sZmlsZQ%3D%3D?line=1'>2</a> print(obj.__private_attribute)   # Raises an AttributeError
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Pythoon%20100%20Days/100_Days_OF_Python/Day17%20-%20Encapsulation%20in%20OOP%20in%20Python.ipynb#W6sZmlsZQ%3D%3D?line=2'>3</a> print(obj.__private_method())

AttributeError: 'Myclass' object has no attribute '__private_attribute'
```

You can undestand with Encapsulation with this short story:

In a bustling city, there was a high-tech security system protecting a valuable treasure. The system had a keypad (public interface) that allowed authorized personnel to enter a secret code. Behind the keypad was a locked room (protected layer) with advanced security features, and inside that room was a safe (private layer) holding the treasure. Only a select few knew the secret code, ensuring the treasure's safety while providing access to those who needed it. This is encapsulation in action, safeguarding valuable data while allowing controlled access.

Top five benefits of encapsulation:

1. **Controlled Access:** Encapsulation restricts direct access to internal data, ensuring that it is only modified through well-defined methods, which helps prevent unintended errors and maintain data integrity.

2. **Modularity:** Encapsulation promotes modular code by isolating the implementation details of a class. This makes it easier to update or replace components without affecting the rest of the system.

3. **Information Hiding:** It hides complex internal details, reducing the complexity for users of a class and allowing developers to change the implementation without impacting external code.

4. **Security:** By controlling access to sensitive data and methods, encapsulation enhances security, reducing the risk of unauthorized or malicious manipulation.

5. **Reusability:** Encapsulation facilitates the creation of reusable and self-contained classes, which can be easily integrated into different parts of a program or shared across projects, saving development time and effort.

Common uses of encapsulation in programming:

1. **Data Protection:** Encapsulation restricts direct access to data, ensuring that it can only be modified through controlled methods. This protects data integrity and prevents unintended modifications or errors.

2. **Abstraction:** It allows you to present a simplified and high-level view of an object, hiding the complex implementation details. This abstraction makes it easier for users of the class to understand and work with it.

3. **Code Organization:** Encapsulation helps organize code by bundling related data and methods within a class. This modular approach enhances code readability and maintainability, making it easier to manage large codebases.

4. **Security:** By encapsulating sensitive data and operations, you can control who has access to them. This enhances security by preventing unauthorized access or manipulation of critical information.

5. **Reusability:** Encapsulated classes can be reused in different parts of a program or in different projects. This reusability saves development time and promotes the creation of robust, well-tested components.

Revision of Class Relationships in object-oriented programming (OOP) with Python:

1. **Aggregation (Has-a Relationship):**
    - **Definition:** Aggregation is a class relationship where one class (the container or composite class) contains or references another class (the contained or component class) as part of its structure. It models a "has-a" relationship, where the container class has one or more instances of the contained class.
    - **Usage:** Aggregation is often used to model objects that are composed of or contain other objects. It can be implemented using attributes or references in the container class.
    - **Example:** In a university system, the `University` class may contain `Department` objects, and each `Department` contains `Professor` objects. This represents an aggregation relationship because a university has departments, and each department has professors.
2. **Inheritance (Is-a Relationship):**
    - **Definition:** Inheritance is a type of class relationship where a subclass (derived class) inherits properties and behaviors from a superclass (base class). It models an "is-a" relationship, where the subclass is a more specific or specialized version of the superclass.
    - **Usage:** In Python, inheritance is established using the `class SubClass(BaseClass)` syntax. The subclass inherits attributes and methods from the superclass and can also add its own or override the inherited ones.
    - **Example:** If you have a base class `Vehicle`, you can create subclasses like `Car` and `Bicycle` that inherit attributes and methods from `Vehicle` while adding their own specific properties.

These two class relationships, inheritance and aggregation, are fundamental concepts in OOP and are used extensively in software design to model different types of relationships between classes and objects.

1. Aggregation(Has a Realtionship)

In object-oriented programming (OOP) with Python, aggregation is a type of association between classes where one class, known as the "container" or "composite" class, contains or references another class, known as the "contained" or "component" class. Aggregation represents a "has-a" relationship, where the container class has one or more instances of the contained class as part of its own structure.

Aggregation is often used to model relationships between objects when one object is composed of or contains other objects. It is a way to create more complex objects by combining simpler objects. A classic example of aggregation is a university system where a

University class can contain **Department** objects, and each Department can contain `**Professor** objects.

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

class Department:
    def __init__(self, name):
        self.name = name
        self.professors = []  # Aggregation: Department contains Professor objects

class University:
    def __init__(self, name):
        self.name = name
        self.departments = []  # Aggregation: University contains Department object

# Creating objects
professor1 = Professor("John Doe")
professor2 = Professor("Jane Smith")

department1 = Department("Computer Science")
department1.professors.append(professor1)
department1.professors.append(professor2)

university = University("ABC University")
university.departments.append(department1)
```

In this example:

- 'University' has an aggregation relationship with 'Department' because it contains a list of 'Department' objects.
- 'Department' has an aggregation relationship with 'Professor' because it contains a list of Professor objects.

Aggregation is a way to represent the idea that one class is composed of or contains instances of another class, and it is often used to model real-world relationships between objects in a system.

Top five reasons why we use aggregation in object-oriented programming:

1. **Modularity and Code Reusability:** Aggregation allows for the creation of modular, reusable components, making it easier to build and maintain complex systems.

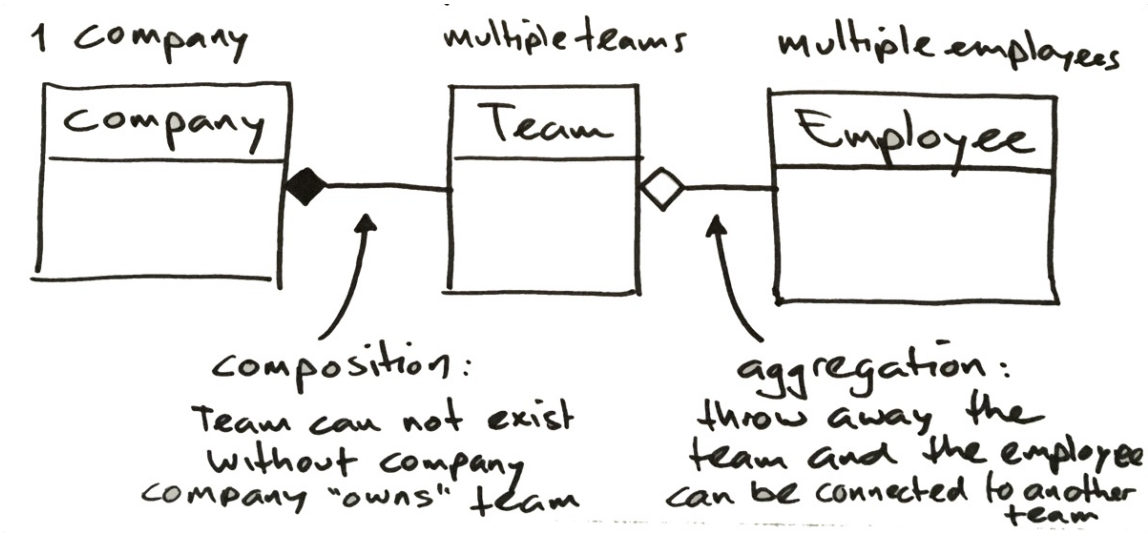
2. **Clearer Object Relationships:** It helps model real-world relationships accurately, enhancing the code's clarity and alignment with the problem domain.

3. **Flexible System Design:** Aggregation permits changes and extensions to be made to the codebase without affecting the entire system, ensuring adaptability to evolving requirements.

4. **Enhanced Encapsulation:** It supports improved data and behavior encapsulation by encapsulating the interaction details between components within the container class.

5. **Efficient Resource Utilization:** Aggregation can lead to efficient memory usage by sharing component instances among multiple container objects, making it particularly useful in resource-constrained environments.

One example of class diagram of Aggreagation



- The solid diamond indicates Composition. Notice how teams belong to the single company object. If the company object is deleted, the teams will have no reason to exist anymore.
- The open diamond indicates Aggregation. When a team is deleted, the employees that were in the team, still exist. Employees might also belong to multiple teams. A team object does not "own" an employee object.

Revision of Inheritance in OOP

What is Inheritance(is a relationship)

Inheritance is one of the fundamental concepts in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit properties and behaviors (attributes and methods) from an existing class (base class or superclass). This allows you to create a new class that is a modified or specialized version of an existing class, promoting code reuse and establishing a hierarchical relationship between classes.

Inheritance is typically used to model an "is-a" relationship between classes, where the derived class is a more specific or specialized version of the base class. The derived class inherits the attributes and methods of the base class and can also have its own additional attributes and methods or override the inherited ones.

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

    def speak(self):
        pass  # Base class method, to be overridden by subclasses

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

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

# Creating objects
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Calling the speak method on objects
print(dog.speak())
print(cat.speak())
```

```
Buddy says Woof!
Whiskers says Meow!
```

In this example:

- 'Animal' is the base class, and 'Dog' and 'Cat' are subclasses of 'Animal'.
- Both 'Dog' and 'Cat' inherit the 'name' attribute and the 'speak' method from the 'Animal' class.
- However, each subclass overrides the 'speak' method to provide its own implementation, representing the specific behavior of each animal. Inheritance is a powerful mechanism in OOP because it allows you to create a hierarchy of classes, with each level of the hierarchy adding or modifying functionality as needed. This promotes code reuse, encapsulation, and abstraction, making it easier to manage and extend your codebase.

Inheritance is a fundamental concept in object-oriented programming (OOP) that offers several benefits for software development. Here are five key advantages of inheritance:

1. **Code Reusability:** Inheritance allows you to reuse code from existing classes (base or parent classes) in new classes (derived or child classes). This promotes code reusability, reduces redundancy, and saves development time. Developers can leverage well-tested and established code when creating new classes.

2. **Hierarchical Structure:** Inheritance enables the creation of a hierarchical structure of classes, with each derived class inheriting attributes and methods from its parent class. This hierarchical organization makes it easier to understand and manage the relationships between different classes in a complex system.

3. **Promotes Polymorphism:** Inheritance is closely linked to the concept of polymorphism, which allows objects of different classes to be treated as objects of a common base class. This promotes flexibility and extensibility in your code, making it easier to work with diverse types of objects in a unified manner.

4. **Supports Code Extensibility:** With inheritance, you can extend existing classes to add or modify functionality. Derived classes can override inherited methods to provide specialized behavior while still benefiting from the base class's shared attributes and methods. This makes it straightforward to adapt and extend your code to accommodate changing requirements.

5. **Enhances Maintenance:** Inheritance improves code maintenance because changes made to the base class are automatically reflected in all its derived classes. This reduces the risk of introducing bugs during updates and ensures that modifications are consistently applied throughout the codebase. It also helps maintain a consistent interface for objects derived from the same base class.

In summary, inheritance is a powerful mechanism in OOP that promotes code reusability, structure, flexibility, extensibility, and ease of maintenance. It is a cornerstone of building complex software systems by organizing and leveraging existing code effectively.

Inheritance in summary

- A class can inherit from another class.

- Inheritance improves code reuse
- Constructor, attributes, methods get inherited to the child class
- The parent has no access to the child class
- Private properties of parent are not accessible directly in child class
- Child class can override the attributes or methods. This is called method overriding
- super() is an inbuilt function which is used to invoke the parent class methods and constructor

Types of Inheritance

1. **Single Inheritance:**
    - In single inheritance, a class inherits properties and behaviors from a single parent class. This is the simplest form of inheritance, where each class has only one immediate base class.
2. **Multilevel Inheritance:**
    - In multilevel inheritance, a class derives from a class, which in turn derives from another class. This creates a chain of inheritance where each class extends the previous one. It forms a hierarchy of classes.
3. **Hierarchical Inheritance:**
    - In hierarchical inheritance, multiple derived classes inherit from a single base or parent class. This results in a structure where several classes share a common ancestor.
4. **Multiple Inheritance (Diamond Problem):**
    - Multiple inheritance occurs when a class inherits properties and behaviors from more than one parent class. This can lead to a complication known as the "diamond problem," where ambiguity arises when a method or attribute is called that exists in multiple parent classes. Some programming languages, like Python, provide mechanisms to resolve this ambiguity.
5. **Hybrid Inheritance:**
    - Hybrid inheritance is a combination of two or more types of inheritance mentioned above. It is used to model complex relationships in a system where multiple inheritance hierarchies may intersect.

These different types of inheritance allow developers to model various relationships and structures in object-oriented programming. Choosing the appropriate type of inheritance depends on the specific requirements and design of the software being developed.

In [ ]:
```
# single inheritance
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
```

```
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
```

```
Inside phone constructor
Buying a phone
```

In [ ]:
```
# multilevel
class Product:
    def review(self):
        print ("Product customer review")

class Phone(Product):
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass
```

In [ ]:
```
# Hierarchical
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass
```

In [ ]:
```
# Multiple
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class Product:
    def review(self):
        print ("Customer review")

class SmartPhone(Phone, Product):
    pass
```

Revision of Polymorphism and Abstraction - Object-Oriented Programming (OOP)

Polymorphism:

Polymorphism is one of the four fundamental principles of Object-Oriented Programming (OOP), along with encapsulation, inheritance, and abstraction. It allows objects of different classes to be treated as objects of a common superclass. In Python, polymorphism is implemented through method overriding and method overloading, which are two related concepts.

1. Method Overriding

2. Method Overloading

1. Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows the subclass to provide its own behavior for a method while still adhering to the method signature defined in the superclass.

Here's an example of method overriding in Python:

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

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

dog = Dog()
cat = Cat()

print(dog.speak())
print(cat.speak())
```

```
Woof!
Meow!
```

2.Method Overloading:

Method overloading allows a class to define multiple methods with the same name but different parameters. Python does not support traditional method overloading with different parameter types like some other languages (e.g., Java or C++). Instead, Python achieves a form of method overloading using default arguments and variable-length argument lists.

Here's an example:

In [ ]:
```
class Calculator:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

calc = Calculator()

#result1 = calc.add(1, 2)        # Error: The second add method with three paramete
result2 = calc.add(1, 2, 3)     # This works fine.
```

In Python, only the latest defined method with a particular name is accessible. Traditional method overloading with different parameter types isn't supported.

In [ ]:
```
class Shape:

  def area(self,a,b=0):
    if b == 0:
      return 3.14*a*a
    else:
      return a*b

s = Shape()

print(s.area(2))
print(s.area(3,4))
```

```
12.56
12
```

Polymorphism allows you to write more flexible and reusable code by treating different objects in a consistent way, regardless of their specific class. This promotes code flexibility and makes it easier to extend and maintain your programs as they grow in complexity.

Abstraction

Abstraction is one of the four fundamental principles of Object-Oriented Programming (OOP), along with encapsulation, inheritance, and polymorphism. Abstraction is the process of simplifying complex reality by modeling classes based on the essential properties and behaviors of objects, while ignoring or hiding the non-essential details.

In software development, abstraction allows you to create a simplified representation of an object or system that focuses on what's relevant to your application and hides the unnecessary complexity. Here are some key aspects of abstraction:

1. **Modeling:** Abstraction involves defining classes and objects that capture the essential characteristics and behaviors of real-world entities or concepts. You create classes to

represent these abstractions, defining their attributes (data) and methods (behavior) to interact with them.

2. **Hiding Complexity:** Abstraction allows you to hide the internal details and complexities of an object's implementation from the outside world. This is achieved through encapsulation, where you define private attributes and provide public methods to interact with the object.

3. **Generalization:** Abstraction often involves creating abstract classes or interfaces that define a common set of attributes and behaviors shared by multiple related classes. This promotes code reusability and flexibility through inheritance.

4. **Focus on What, Not How:** When you work with abstractions, you can focus on using objects based on what they do (their methods) rather than how they do it (their implementation details). This separation of concerns simplifies the design and maintenance of complex systems.

Here's a simple example of abstraction in Python:

In [ ]:
```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start(self):
        pass

    def stop(self):
        pass

class Car(Vehicle):
    def start(self):
        print(f"{self.make} {self.model} is starting")

    def stop(self):
        print(f"{self.make} {self.model} is stopping")

class Motorcycle(Vehicle):
    def start(self):
        print(f"{self.make} {self.model} is revving up")

    def stop(self):
        print(f"{self.make} {self.model} is slowing down")
```

In this example, the `Vehicle` class represents an abstraction of a general vehicle with common attributes (make and model) and methods (start and stop). Concrete subclasses like `Car` and `Motorcycle` inherit from `Vehicle` and provide specific implementations for the start and stop methods. Users of these classes can interact with them without needing to know the exact implementation details of each vehicle type.

Abstraction helps in managing complexity, improving code organization, and making code more maintainable and understandable by focusing on high-level concepts and behaviors.

How to Use Abstraction:

In [ ]:
```
from abc import ABC,abstractmethod
class BankApp(ABC):

  def database(self):
    print('connected to database')

  @abstractmethod    # This is absreactmethod
  def security(self):
    pass

  @abstractmethod
  def display(self):
    pass
```

In [ ]:
```
class MobileApp(BankApp):

  def mobile_login(self):
    print('login into mobile')

  def security(self):         # if we use some function in abstractmethod then we h
    print('mobile security')

  def display(self):
    print('display')
```

In [ ]:
```
mob = MobileApp()
```

In [ ]:
```
mob.security()
```

```
mobile security
```

In [ ]:
```
obj = BankApp()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
c:\Users\disha\Downloads\Pythoon 100 Days\100_Days_OF_Python\Polymorphism and Abst
raction - Object-Oriented Programming (OOP).ipynb Cell 19 line 1
----> <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Pythoon%20100%20Da
ys/100_Days_OF_Python/Polymorphism%20and%20Abstraction%20-%20Object-Oriented%20Pro
gramming%20%28OOP%29.ipynb#X26sZmlsZQ%3D%3D?line=0'>1</a> obj = BankApp()

TypeError: Can't instantiate abstract class BankApp with abstract methods display,
security
```

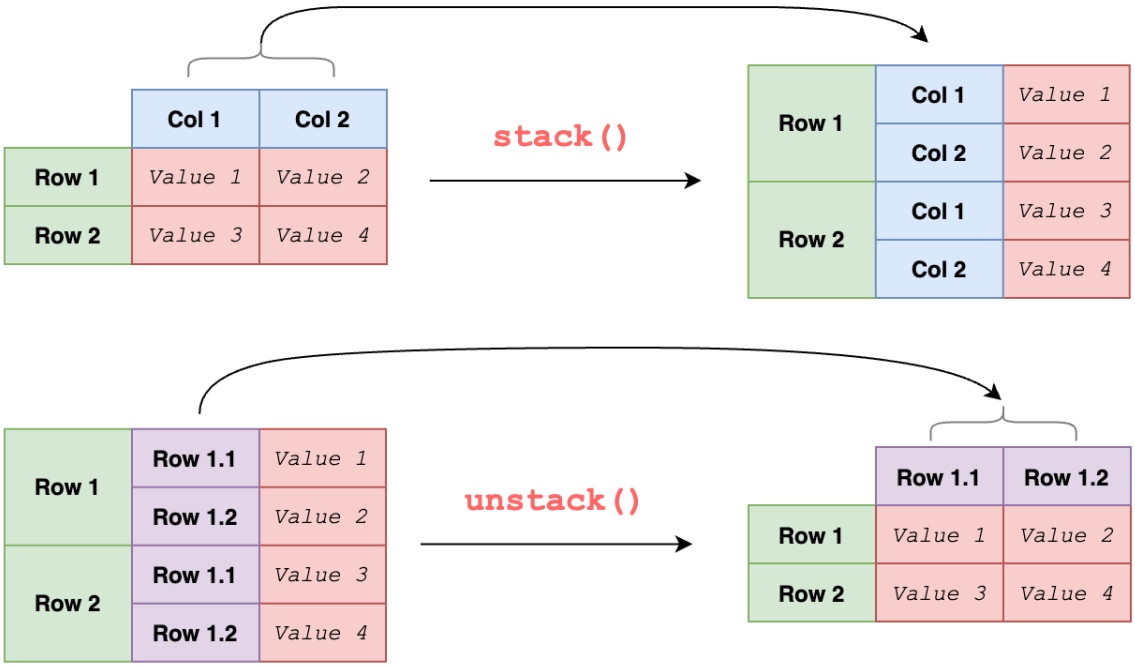
The error message indicates that you are trying to instantiate an abstract class called BankApp, but this class contains abstract methods display and security that have not been implemented in the BankApp class or its subclasses.

In Python, an abstract class is a class that cannot be instantiated directly, and it often serves as a blueprint for other classes. Abstract classes can contain abstract methods, which are methods declared without an implementation in the abstract class. Subclasses of the abstract class are required to provide implementations for these abstract methods.

Revision of Stacking and Unstacking in MultiIndex Object in Pandas

In pandas, a multi-index object is a way to represent hierarchical data structures within a DataFrame or Series. Multi-indexing allows you to have multiple levels of row or column indices, providing a way to organize and work with complex, structured data.

"Stacking" and "unstacking" are operations that you can perform on multi-indexed DataFrames to change the arrangement of the data, essentially reshaping the data between a wide and a long format (or vice versa).



1. Stacking:

- Stacking is the process of "melting" or pivoting the innermost level of column labels to become the innermost level of row labels.
- This operation is typically used when you want to convert a wide DataFrame with multi-level columns into a long format.
- You can use the `.stack()` method to perform stacking. By default, it will stack the innermost level of columns.

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

In [ ]:
```
# Create a DataFrame with multi-level columns
df = pd.DataFrame(np.random.rand(4, 4), columns=[['A', 'A', 'B', 'B'], ['X', 'Y', '
print(df)
```

```
          A                   B
          X         Y         X         Y
0  0.960684  0.118538  0.900984  0.485585
1  0.946716  0.049913  0.444658  0.991469
2  0.656110  0.158270  0.759727  0.203801
3  0.360581  0.965035  0.797212  0.102426
```

In [ ]:
```
# Stack the innermost level of columns
stacked_df = df.stack()
print(stacked_df)
```

```
            A         B
0 X  0.960684  0.900984
  Y  0.118538  0.485585
1 X  0.946716  0.444658
  Y  0.049913  0.991469
2 X  0.656110  0.759727
  Y  0.158270  0.203801
3 X  0.360581  0.797212
  Y  0.965035  0.102426
```

2. **Unstacking**:

- Unstacking is the reverse operation of stacking. It involves pivoting the innermost level of row labels to become the innermost level of column labels.
- You can use the `.unstack()` method to perform unstacking. By default, it will unstack the innermost level of row labels.

Example:

In [ ]:
```
# Unstack the innermost level of row labels
unstacked_df = stacked_df.unstack()
print(unstacked_df)
```

```
          A                   B
          X         Y         X         Y
0  0.960684  0.118538  0.900984  0.485585
1  0.946716  0.049913  0.444658  0.991469
2  0.656110  0.158270  0.759727  0.203801
3  0.360581  0.965035  0.797212  0.102426
```

You can specify the level you want to stack or unstack by passing the `level` parameter to the `stack()` or `unstack()` methods. For example:

In [ ]:
```
# Stack the second level of columns
stacked_df = df.stack(level=1)
stacked_df
```

Out[ ]:

		A	B
0	**X**	0.960684	0.900984
	Y	0.118538	0.485585
1	**X**	0.946716	0.444658
	Y	0.049913	0.991469
2	**X**	0.656110	0.759727
	Y	0.158270	0.203801
3	**X**	0.360581	0.797212
	Y	0.965035	0.102426

In [ ]:
```
# Unstack the first level of row labels
unstacked_df = stacked_df.unstack(level=0)
unstacked_df
```

Out[ ]:

	A				B			
	0	**1**	**2**	**3**	**0**	**1**	**2**	**3**
X	0.960684	0.946716	0.65611	0.360581	0.900984	0.444658	0.759727	0.797212
Y	0.118538	0.049913	0.15827	0.965035	0.485585	0.991469	0.203801	0.102426

In [ ]:
```
index_val = [('cse',2019),('cse',2020),('cse',2021),('cse',2022),('ece',2019),('ece
multiindex = pd.MultiIndex.from_tuples(index_val)
multiindex.levels[1]
```

Out[ ]:
```
Index([2019, 2020, 2021, 2022], dtype='int64')
```

In [ ]:
```
branch_df1 = pd.DataFrame(
    [
        [1,2],
        [3,4],
        [5,6],
        [7,8],
        [9,10],
        [11,12],
        [13,14],
        [15,16],
    ],
    index = multiindex,
    columns = ['avg_package','students']
)

branch_df1
```

Out[ ]:

		avg_package	students
cse	2019	1	2
	2020	3	4
	2021	5	6
	2022	7	8
ece	2019	9	10
	2020	11	12
	2021	13	14
	2022	15	16

In [ ]:
```
# multiindex df from columns perspective
branch_df2 = pd.DataFrame(
    [
        [1,2,0,0],
        [3,4,0,0],
        [5,6,0,0],
        [7,8,0,0],
    ],
    index = [2019,2020,2021,2022],
    columns = pd.MultiIndex.from_product([['delhi','mumbai'],['avg_package','studer
)

branch_df2
```

Out[ ]:

	delhi		mumbai	
	avg_package	students	avg_package	students
2019	1	2	0	0
2020	3	4	0	0
2021	5	6	0	0
2022	7	8	0	0

In [ ]:
```
branch_df1
```

Out[ ]:

		avg_package	students
cse	2019	1	2
	2020	3	4
	2021	5	6
	2022	7	8
ece	2019	9	10
	2020	11	12
	2021	13	14
	2022	15	16

In [ ]:
```
branch_df1.unstack().unstack()
```

Out[ ]:
```
avg_package  2019  cse     1
                    ece     9
             2020  cse     3
                    ece    11
             2021  cse     5
                    ece    13
             2022  cse     7
                    ece    15
students     2019  cse     2
                    ece    10
             2020  cse     4
                    ece    12
             2021  cse     6
                    ece    14
             2022  cse     8
                    ece    16
dtype: int64
```

In [ ]:
```
branch_df1.unstack().stack()
```

Out[ ]:

		avg_package	students
cse	**2019**	1	2
	2020	3	4
	2021	5	6
	2022	7	8
ece	**2019**	9	10
	2020	11	12
	2021	13	14
	2022	15	16

In [ ]:
```
branch_df2
```

Out[ ]:

	delhi		mumbai	
	avg_package	**students**	**avg_package**	**students**
2019	1	2	0	0
2020	3	4	0	0
2021	5	6	0	0
2022	7	8	0	0

In [ ]:
```
branch_df2.stack()
```

Out[ ]:

		delhi	mumbai
2019	**avg_package**	1	0
	students	2	0
2020	**avg_package**	3	0
	students	4	0
2021	**avg_package**	5	0
	students	6	0
2022	**avg_package**	7	0
	students	8	0

In [ ]:
```
branch_df2.stack().stack()
```

Out[ ]:
```
2019  avg_package  delhi     1
                   mumbai    0
      students     delhi     2
                   mumbai    0
2020  avg_package  delhi     3
                   mumbai    0
      students     delhi     4
                   mumbai    0
2021  avg_package  delhi     5
                   mumbai    0
      students     delhi     6
                   mumbai    0
2022  avg_package  delhi     7
                   mumbai    0
      students     delhi     8
                   mumbai    0
dtype: int64
```

Stacking and unstacking can be very useful when you need to reshape your data to make it more suitable for different types of analysis or visualization. They are common operations in data manipulation when working with multi-indexed DataFrames in pandas.

Revision Of GroupBy Object in Pandas: GroupBy Foundation

In pandas, a `GroupBy` object is a crucial part of the data manipulation process, specifically for data aggregation and transformation. It is a result of the `groupby()` method applied to a pandas DataFrame, which allows you to group the data in the DataFrame based on one or more columns.

When you apply `groupby()` to a DataFrame, it creates a `GroupBy` object, which acts as a kind of intermediate step before applying aggregation functions or other operations to the grouped data. This intermediate step helps you perform operations on subsets of data based on the grouping criteria. Some common aggregation functions you can apply to a `GroupBy` object include `sum()`, `mean()`, `count()`, `max()`, `min()`, and more.

Here's a basic example of how you can create a `GroupBy` object and perform aggregation with it:

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

# Create a sample DataFrame
data = {
    'Category': ['A', 'B', 'A', 'B', 'A'],
    'Value': [10, 20, 15, 25, 30]
}

df = pd.DataFrame(data)

# Group the data by the 'Category' column
grouped = df.groupby('Category')
```

In [ ]:
```
# Calculate the sum of 'Value' for each group
sum_values = grouped['Value'].sum()
```

In [ ]:
```
# Display the result
print(sum_values)
```

```
Category
A    55
B    45
Name: Value, dtype: int64
```

In this example, we group the DataFrame `df` by the 'Category' column, creating a `GroupBy` object. Then, we calculate the sum of 'Value' for each group using the `sum()` method on the `GroupBy` object, resulting in a new DataFrame or Series with the aggregated values.

Practical Use

In [ ]:
```
movies = pd.read_csv('Data\Day35\imdb-top-1000.csv')
```

In [ ]:
```
movies.head(3)
```

Out[ ]:

	Series_Title	Released_Year	Runtime	Genre	IMDB_Rating	Director	Star1	No_of_Votes
0	The Shawshank Redemption	1994	142	Drama	9.3	Frank Darabont	Tim Robbins	2343110
1	The Godfather	1972	175	Crime	9.2	Francis Ford Coppola	Marlon Brando	1620367
2	The Dark Knight	2008	152	Action	9.0	Christopher Nolan	Christian Bale	2303232

Applying builtin aggregation fuctions on groupby objects

In [ ]:
```
genres = movies.groupby('Genre')
```

In [ ]:
```
genres.sum(3)
```

Out[ ]:

Genre	Runtime	IMDB_Rating	No_of_Votes	Gross	Metascore
Action	22196	1367.3	72282412	3.263226e+10	10499.0
Adventure	9656	571.5	22576163	9.496922e+09	5020.0
Animation	8166	650.3	21978630	1.463147e+10	6082.0
Biography	11970	698.6	24006844	8.276358e+09	6023.0
Comedy	17380	1224.7	27620327	1.566387e+10	9840.0
Crime	13524	857.8	33533615	8.452632e+09	6706.0
Drama	36049	2299.7	61367304	3.540997e+10	19208.0
Family	215	15.6	551221	4.391106e+08	158.0
Fantasy	170	16.0	146222	7.827267e+08	0.0
Film-Noir	312	23.9	367215	1.259105e+08	287.0
Horror	1123	87.0	3742556	1.034649e+09	880.0
Mystery	1429	95.7	4203004	1.256417e+09	633.0
Thriller	108	7.8	27733	1.755074e+07	81.0
Western	593	33.4	1289665	5.822151e+07	313.0

In [ ]:
```
genres.mean(3)
```

Out[ ]:

Genre	Runtime	IMDB_Rating	No_of_Votes	Gross	Metascore
Action	129.046512	7.949419	420246.581395	1.897224e+08	73.419580
Adventure	134.111111	7.937500	313557.819444	1.319017e+08	78.437500
Animation	99.585366	7.930488	268032.073171	1.784326e+08	81.093333
Biography	136.022727	7.938636	272805.045455	9.404952e+07	76.240506
Comedy	112.129032	7.901290	178195.658065	1.010572e+08	78.720000
Crime	126.392523	8.016822	313398.271028	7.899656e+07	77.080460
Drama	124.737024	7.957439	212343.612457	1.225259e+08	79.701245
Family	107.500000	7.800000	275610.500000	2.195553e+08	79.000000
Fantasy	85.000000	8.000000	73111.000000	3.913633e+08	NaN
Film-Noir	104.000000	7.966667	122405.000000	4.197018e+07	95.666667
Horror	102.090909	7.909091	340232.363636	9.405902e+07	80.000000
Mystery	119.083333	7.975000	350250.333333	1.047014e+08	79.125000
Thriller	108.000000	7.800000	27733.000000	1.755074e+07	81.000000
Western	148.250000	8.350000	322416.250000	1.455538e+07	78.250000

find the top 3 genres by total earning

In [ ]:
```
movies.groupby('Genre')['Gross'].sum().sort_values(ascending=False).head(3)
```

Out[ ]:
```
Genre
Drama     3.540997e+10
Action    3.263226e+10
Comedy    1.566387e+10
Name: Gross, dtype: float64
```

In [ ]:
```
movies.groupby('Genre').sum()['Gross'].sort_values(ascending=False).head(3)
```

Out[ ]:
```
Genre
Drama     3.540997e+10
Action    3.263226e+10
Comedy    1.566387e+10
Name: Gross, dtype: float64
```

find the genre with highest avg IMDB rating

In [ ]:
```
movies.groupby('Genre')['IMDB_Rating'].mean().sort_values(ascending=False).head(1)
```

Out[ ]:
```
Genre
Western    8.35
Name: IMDB_Rating, dtype: float64
```

find director with most popularity

In [ ]:
```
movies.groupby('Director')['No_of_Votes'].sum().sort_values(ascending=False).head(1
```

Out[ ]:
```
Director
Christopher Nolan    11578345
Name: No_of_Votes, dtype: int64
```

find number of movies done by each actor

In [ ]:
```
movies.groupby('Star1')['Series_Title'].count().sort_values(ascending=False)
```

Out[ ]:
```
Star1
Tom Hanks             12
Robert De Niro        11
Clint Eastwood        10
Al Pacino             10
Leonardo DiCaprio      9
                      ..
Glen Hansard           1
Giuseppe Battiston     1
Giulietta Masina       1
Gerardo Taracena       1
Ömer Faruk Sorak       1
Name: Series_Title, Length: 660, dtype: int64
```

A GroupBy object is a powerful tool for performing group-wise operations on data. It enables data analysts and scientists to gain insights into their data by aggregating, filtering, and transforming information based on specific grouping criteria. These operations are essential for understanding data patterns and making informed decisions.

Revision of GroupBy Attributes and Methods in Pandas

- len - find total number of groups
- size - find items in each group
- nth item - first()/nth/last items
- get_group - vs filtering
- groups
- describe
- sample
- nunique
- agg method
- apply -> builtin function

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

movies = pd.read_csv('Data\Day35\imdb-top-1000.csv')
```

In [ ]:
```
genres = movies.groupby('Genre')
```

1. len

In [ ]:
```
len(movies.groupby('Genre'))
```

Out[ ]:
```
14
```

2. nunique

In [ ]:
```
movies['Genre'].nunique()
```

Out[ ]:
```
14
```

3. size

In [ ]:
```
movies.groupby('Genre').size()
```

Out[ ]:
```
Genre
Action       172
Adventure     72
Animation     82
Biography     88
Comedy       155
Crime        107
Drama        289
Family         2
Fantasy        2
Film-Noir      3
Horror        11
Mystery       12
Thriller       1
Western        4
dtype: int64
```

4. nth

In [ ]:
```
genres = movies.groupby('Genre')
# genres.first()
# genres.last()
genres.nth(6)
```

Out[ ]:

	Series_Title	Released_Year	Runtime	Genre	IMDB_Rating	Director	Star1	No_of_V
16	Star Wars: Episode V - The Empire Strikes Back	1980	124	Action	8.7	Irvin Kershner	Mark Hamill	1159
27	Se7en	1995	127	Crime	8.6	David Fincher	Morgan Freeman	1445
32	It's a Wonderful Life	1946	130	Drama	8.6	Frank Capra	James Stewart	405
66	WALL·E	2008	98	Animation	8.4	Andrew Stanton	Ben Burtt	999
83	The Great Dictator	1940	125	Comedy	8.4	Charles Chaplin	Charles Chaplin	203
102	Braveheart	1995	178	Biography	8.3	Mel Gibson	Mel Gibson	959
118	North by Northwest	1959	136	Adventure	8.3	Alfred Hitchcock	Cary Grant	299
420	Sleuth	1972	138	Mystery	8.0	Joseph L. Mankiewicz	Laurence Olivier	44
724	Get Out	2017	104	Horror	7.7	Jordan Peele	Daniel Kaluuya	492

5. get_group

In [ ]:
```
genres.get_group('Fantasy')
```

Out[ ]:

	Series_Title	Released_Year	Runtime	Genre	IMDB_Rating	Director	Star1	No_of_Votes	
321	Das Cabinet des Dr. Caligari	1920	76	Fantasy	8.1	Robert Wiene	Werner Krauss	57428	3
568	Nosferatu	1922	94	Fantasy	7.9	F.W. Murnau	Max Schreck	88794	4

In [ ]:
```
# with simple but its slow
movies[movies['Genre'] == 'Fantasy']
```

Out[ ]:

	Series_Title	Released_Year	Runtime	Genre	IMDB_Rating	Director	Star1	No_of_Votes	
321	Das Cabinet des Dr. Caligari	1920	76	Fantasy	8.1	Robert Wiene	Werner Krauss	57428	3
568	Nosferatu	1922	94	Fantasy	7.9	F.W. Murnau	Max Schreck	88794	4

6. describe

In [ ]:
```
genres.describe()
```

Out[ ]:

	Runtime								IMDB_Rating		...
	count	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**	**count**	**mean**	**...**
Genre											
Action	172.0	129.046512	28.500706	45.0	110.75	127.5	143.25	321.0	172.0	7.949419	...
Adventure	72.0	134.111111	33.317320	88.0	109.00	127.0	149.00	228.0	72.0	7.937500	...
Animation	82.0	99.585366	14.530471	71.0	90.00	99.5	106.75	137.0	82.0	7.930488	...
Biography	88.0	136.022727	25.514466	93.0	120.00	129.0	146.25	209.0	88.0	7.938636	...
Comedy	155.0	112.129032	22.946213	68.0	96.00	106.0	124.50	188.0	155.0	7.901290	...
Crime	107.0	126.392523	27.689231	80.0	106.50	122.0	141.50	229.0	107.0	8.016822	...
Drama	289.0	124.737024	27.740490	64.0	105.00	121.0	137.00	242.0	289.0	7.957439	...
Family	2.0	107.500000	10.606602	100.0	103.75	107.5	111.25	115.0	2.0	7.800000	...
Fantasy	2.0	85.000000	12.727922	76.0	80.50	85.0	89.50	94.0	2.0	8.000000	...
Film-Noir	3.0	104.000000	4.000000	100.0	102.00	104.0	106.00	108.0	3.0	7.966667	...
Horror	11.0	102.090909	13.604812	71.0	98.00	103.0	109.00	122.0	11.0	7.909091	...
Mystery	12.0	119.083333	14.475423	96.0	110.75	117.5	130.25	138.0	12.0	7.975000	...
Thriller	1.0	108.000000	NaN	108.0	108.00	108.0	108.00	108.0	1.0	7.800000	...
Western	4.0	148.250000	17.153717	132.0	134.25	148.0	162.00	165.0	4.0	8.350000	...

14 rows × 40 columns

7. sample

In [ ]:
```
genres.sample(2,replace=True)
```

Out[ ]:

	Series_Title	Released_Year	Runtime	Genre	IMDB_Rating	Director	Star1	No_
648	The Boondock Saints	1999	108	Action	7.8	Troy Duffy	Willem Dafoe	
908	Kick-Ass	2010	117	Action	7.6	Matthew Vaughn	Aaron Taylor-Johnson	
193	The Gold Rush	1925	95	Adventure	8.2	Charles Chaplin	Charles Chaplin	
361	Blood Diamond	2006	143	Adventure	8.0	Edward Zwick	Leonardo DiCaprio	
61	Coco	2017	105	Animation	8.4	Lee Unkrich	Adrian Molina	
758	Papurika	2006	90	Animation	7.7	Satoshi Kon	Megumi Hayashibara	
328	Lion	2016	118	Biography	8.0	Garth Davis	Dev Patel	
159	A Beautiful Mind	2001	135	Biography	8.2	Ron Howard	Russell Crowe	
256	Underground	1995	170	Comedy	8.1	Emir Kusturica	Predrag 'Miki' Manojlovic	
667	Night on Earth	1991	129	Comedy	7.8	Jim Jarmusch	Winona Ryder	
441	The Killing	1956	84	Crime	8.0	Stanley Kubrick	Sterling Hayden	
288	Cool Hand Luke	1967	127	Crime	8.1	Stuart Rosenberg	Paul Newman	
773	Brokeback Mountain	2005	134	Drama	7.7	Ang Lee	Jake Gyllenhaal	
515	Mulholland Dr.	2001	147	Drama	7.9	David Lynch	Naomi Watts	
698	Willy Wonka & the Chocolate Factory	1971	100	Family	7.8	Mel Stuart	Gene Wilder	
688	E.T. the Extra-Terrestrial	1982	115	Family	7.8	Steven Spielberg	Henry Thomas	
568	Nosferatu	1922	94	Fantasy	7.9	F.W. Murnau	Max Schreck	
321	Das Cabinet des Dr. Caligari	1920	76	Fantasy	8.1	Robert Wiene	Werner Krauss	
456	The Maltese Falcon	1941	100	Film-Noir	8.0	John Huston	Humphrey Bogart	
456	The Maltese Falcon	1941	100	Film-Noir	8.0	John Huston	Humphrey Bogart	
932	Saw	2004	103	Horror	7.6	James Wan	Cary Elwes	

	Series_Title	Released_Year	Runtime	Genre	IMDB_Rating	Director	Star1	No_
876	The Invisible Man	1933	71	Horror	7.7	James Whale	Claude Rains	
420	Sleuth	1972	138	Mystery	8.0	Joseph L. Mankiewicz	Laurence Olivier	
119	Vertigo	1958	128	Mystery	8.3	Alfred Hitchcock	James Stewart	
700	Wait Until Dark	1967	108	Thriller	7.8	Terence Young	Audrey Hepburn	
700	Wait Until Dark	1967	108	Thriller	7.8	Terence Young	Audrey Hepburn	
12	Il buono, il brutto, il cattivo	1966	161	Western	8.8	Sergio Leone	Clint Eastwood	
691	The Outlaw Josey Wales	1976	135	Western	7.8	Clint Eastwood	Clint Eastwood	

8. nunique()

In [ ]:
```
genres.nunique()
```

Out[ ]:

Genre	Series_Title	Released_Year	Runtime	IMDB_Rating	Director	Star1	No_of_Votes	Gross
Action	172	61	78	15	123	121	172	172
Adventure	72	49	58	10	59	59	72	72
Animation	82	35	41	11	51	77	82	82
Biography	88	44	56	13	76	72	88	88
Comedy	155	72	70	11	113	133	155	155
Crime	106	56	65	14	86	85	107	107
Drama	289	83	95	14	211	250	288	287
Family	2	2	2	1	2	2	2	2
Fantasy	2	2	2	2	2	2	2	2
Film-Noir	3	3	3	3	3	3	3	3
Horror	11	11	10	8	10	11	11	11
Mystery	12	11	10	8	10	11	12	12
Thriller	1	1	1	1	1	1	1	1
Western	4	4	4	4	2	2	4	4

9. agg method

In [ ]:
```
# passing dict
genres.agg(
    {
```

```
        'Runtime':'mean',
        'IMDB_Rating':'mean',
        'No_of_Votes':'sum',
        'Gross':'sum',
        'Metascore':'min'
    }
)
```

Out[ ]:

Genre	Runtime	IMDB_Rating	No_of_Votes	Gross	Metascore
Action	129.046512	7.949419	72282412	3.263226e+10	33.0
Adventure	134.111111	7.937500	22576163	9.496922e+09	41.0
Animation	99.585366	7.930488	21978630	1.463147e+10	61.0
Biography	136.022727	7.938636	24006844	8.276358e+09	48.0
Comedy	112.129032	7.901290	27620327	1.566387e+10	45.0
Crime	126.392523	8.016822	33533615	8.452632e+09	47.0
Drama	124.737024	7.957439	61367304	3.540997e+10	28.0
Family	107.500000	7.800000	551221	4.391106e+08	67.0
Fantasy	85.000000	8.000000	146222	7.827267e+08	NaN
Film-Noir	104.000000	7.966667	367215	1.259105e+08	94.0
Horror	102.090909	7.909091	3742556	1.034649e+09	46.0
Mystery	119.083333	7.975000	4203004	1.256417e+09	52.0
Thriller	108.000000	7.800000	27733	1.755074e+07	81.0
Western	148.250000	8.350000	1289665	5.822151e+07	69.0

In [ ]:

```
genres.agg(
    {
        'Runtime':['min','mean'],
        'IMDB_Rating':'mean',
        'No_of_Votes':['sum','max'],
        'Gross':'sum',
        'Metascore':'min'
    }
)
```

Out[ ]:

	Runtime		IMDB_Rating	No_of_Votes		Gross	Metascore
	min	mean	mean	sum	max	sum	min
Genre							
Action	45	129.046512	7.949419	72282412	2303232	3.263226e+10	33.0
Adventure	88	134.111111	7.937500	22576163	1512360	9.496922e+09	41.0
Animation	71	99.585366	7.930488	21978630	999790	1.463147e+10	61.0
Biography	93	136.022727	7.938636	24006844	1213505	8.276358e+09	48.0
Comedy	68	112.129032	7.901290	27620327	939631	1.566387e+10	45.0
Crime	80	126.392523	8.016822	33533615	1826188	8.452632e+09	47.0
Drama	64	124.737024	7.957439	61367304	2343110	3.540997e+10	28.0
Family	100	107.500000	7.800000	551221	372490	4.391106e+08	67.0
Fantasy	76	85.000000	8.000000	146222	88794	7.827267e+08	NaN
Film-Noir	100	104.000000	7.966667	367215	158731	1.259105e+08	94.0
Horror	71	102.090909	7.909091	3742556	787806	1.034649e+09	46.0
Mystery	96	119.083333	7.975000	4203004	1129894	1.256417e+09	52.0
Thriller	108	108.000000	7.800000	27733	27733	1.755074e+07	81.0
Western	132	148.250000	8.350000	1289665	688390	5.822151e+07	69.0

10. apply -> builtin function

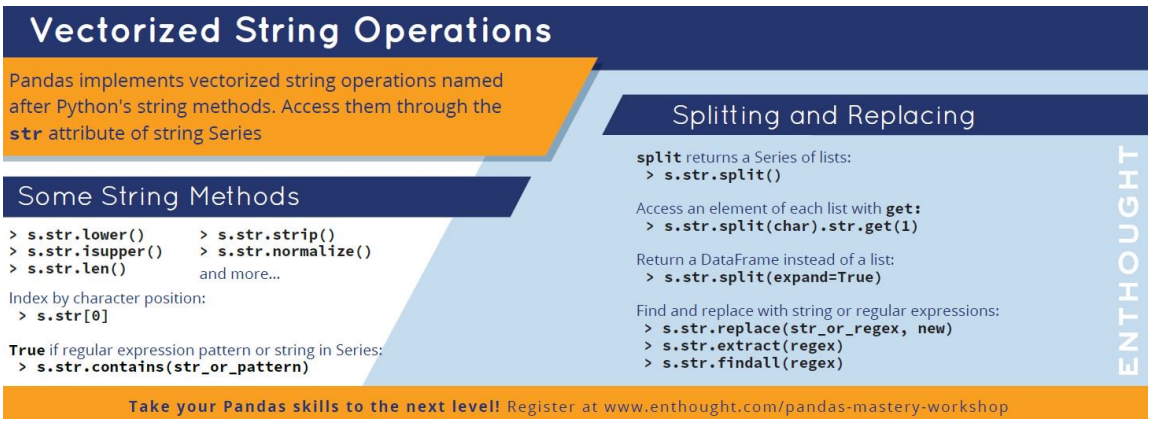
In [ ]:
```
genres.apply(min)
```

Out[ ]:

Genre	Series_Title	Released_Year	Runtime	Genre	IMDB_Rating	Director	Star1	N
Action	300	1924	45	Action	7.6	Abhishek Chaubey	Aamir Khan	
Adventure	2001: A Space Odyssey	1925	88	Adventure	7.6	Akira Kurosawa	Aamir Khan	
Animation	Akira	1940	71	Animation	7.6	Adam Elliot	Adrian Molina	
Biography	12 Years a Slave	1928	93	Biography	7.6	Adam McKay	Adrien Brody	
Comedy	(500) Days of Summer	1921	68	Comedy	7.6	Alejandro G. Iñárritu	Aamir Khan	
Crime	12 Angry Men	1931	80	Crime	7.6	Akira Kurosawa	Ajay Devgn	
Drama	1917	1925	64	Drama	7.6	Aamir Khan	Abhay Deol	
Family	E.T. the Extra-Terrestrial	1971	100	Family	7.8	Mel Stuart	Gene Wilder	
Fantasy	Das Cabinet des Dr. Caligari	1920	76	Fantasy	7.9	F.W. Murnau	Max Schreck	
Film-Noir	Shadow of a Doubt	1941	100	Film-Noir	7.8	Alfred Hitchcock	Humphrey Bogart	
Horror	Alien	1933	71	Horror	7.6	Alejandro Amenábar	Anthony Perkins	
Mystery	Dark City	1938	96	Mystery	7.6	Alex Proyas	Bernard-Pierre Donnadieu	
Thriller	Wait Until Dark	1967	108	Thriller	7.8	Terence Young	Audrey Hepburn	
Western	Il buono, il brutto, il cattivo	1965	132	Western	7.8	Clint Eastwood	Clint Eastwood	

Revision of Vectorized String Opearations in Pandas

- Vectorized string operations in Pandas refer to the ability to apply string operations to a series or dataframe of strings in a single operation, rather than looping over each string individually. This can be achieved using the str attribute of a Series or DataFrame object, which provides a number of vectorized string methods.
- Vectorized string operations in Pandas refer to the ability to apply string functions and operations to entire arrays of strings (columns or Series containing strings) without the need for explicit loops or iteration. This is made possible by Pandas' integration with the NumPy library, which allows for efficient element-wise operations.
- When you have a Pandas DataFrame or Series containing string data, you can use various string methods that are applied to every element in the column simultaneously. This can significantly improve the efficiency and readability of your code. Some of the commonly used vectorized string operations in Pandas include methods like `.str.lower()`, `.str.upper()`, `.str.strip()`, `.str.replace()`, and many more.
- Vectorized string operations not only make your code more concise and readable but also often lead to improved performance compared to explicit for-loops, especially when dealing with large datasets.



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

In [ ]:
```
s = pd.Series(['cat','mat',None,'rat'])
```

In [ ]:
```
# str ->  string accessor
s.str.startswith('c')
```

Out[ ]:
```
0     True
1    False
2     None
3    False
dtype: object
```

Real-world Dataset - Titanic Dataset

In [ ]:
```
df = pd.read_csv('Data\Day44\Titanic.csv')
```

In [ ]:
```
df.head()
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN

In [ ]:
```
df['Name']
```

Out[ ]:
```
0                              Braund, Mr. Owen Harris
1    Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss. Laina
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
                           ...
886                              Montvila, Rev. Juozas
887                       Graham, Miss. Margaret Edith
888           Johnston, Miss. Catherine Helen "Carrie"
889                              Behr, Mr. Karl Howell
890                                Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

Common Functions

In [ ]:
```
# lower/upper/capitalize/title
df['Name'].str.upper()
df['Name'].str.capitalize()
df['Name'].str.title()
```

Out[ ]:
```
0                                Braund, Mr. Owen Harris
1      Cumings, Mrs. John Bradley (Florence Briggs Th...
2                                 Heikkinen, Miss. Laina
3           Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                               Allen, Mr. William Henry
                             ...
886                                Montvila, Rev. Juozas
887                         Graham, Miss. Margaret Edith
888             Johnston, Miss. Catherine Helen "Carrie"
889                                Behr, Mr. Karl Howell
890                                  Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

In [ ]:
```
# len

df['Name'][df['Name'].str.len()]
```

Out[ ]:
```
23                       Sloper, Mr. William Thompson
51                       Nosworthy, Mr. Richard Cater
22                        McGowan, Miss. Anna "Annie"
44                      Devaney, Miss. Margaret Delia
24                      Palsson, Miss. Torborg Danira
                            ...
21                           Beesley, Mr. Lawrence
28                      O'Dwyer, Miss. Ellen "Nellie"
40    Ahlin, Mrs. Johan (Johanna Persdotter Larsson)
21                               Beesley, Mr. Lawrence
19                            Masselmani, Mrs. Fatima
Name: Name, Length: 891, dtype: object
```

In [ ]:
```
df['Name'][df['Name'].str.len() == 82].values[0]
```

Out[ ]:
```
'Penasco y Castellana, Mrs. Victor de Satode (Maria Josefa Perez de Soto y Vallej
o)'
```

In [ ]:
```
# strip
df['Name'].str.strip()
```

Out[ ]:
```
0                                Braund, Mr. Owen Harris
1      Cumings, Mrs. John Bradley (Florence Briggs Th...
2                                 Heikkinen, Miss. Laina
3           Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                               Allen, Mr. William Henry
                             ...
886                                Montvila, Rev. Juozas
887                         Graham, Miss. Margaret Edith
888             Johnston, Miss. Catherine Helen "Carrie"
889                                Behr, Mr. Karl Howell
890                                  Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

In [ ]:
```
# split -> get
df['lastname'] = df['Name'].str.split(',').str.get(0)
df.head()
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN

In [ ]:
```
df[['title','firstname']] = df['Name'].str.split(',').str.get(1).str.strip().str.sp
df.head()
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN

In [ ]:
```
df['title'].value_counts()
```

Out[ ]:
```
title
Mr.          517
Miss.        182
Mrs.         125
Master.       40
Dr.            7
Rev.           6
Mlle.          2
Major.         2
Col.           2
the            1
Capt.          1
Ms.            1
Sir.           1
Lady.          1
Mme.           1
Don.           1
Jonkheer.      1
Name: count, dtype: int64
```

In [ ]:
```
# replace
df['title'] = df['title'].str.replace('Ms.','Miss.')
df['title'] = df['title'].str.replace('Mlle.','Miss.')
```

In [ ]:
```
df['title'].value_counts()
```

Out[ ]:
```
title
Mr.          517
Miss.        185
Mrs.         125
Master.       40
Dr.            7
Rev.           6
Major.         2
Col.           2
Don.           1
Mme.           1
Lady.          1
Sir.           1
Capt.          1
the            1
Jonkheer.      1
Name: count, dtype: int64
```

filtering

In [ ]:
```
# startswith/endswith
df[df['firstname'].str.endswith('A')]
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
64	65	0	1	Stewart, Mr. Albert A	male	NaN	0	0	PC 17605	27.7208	NaN
303	304	1	2	Keane, Miss. Nora A	female	NaN	0	0	226593	12.3500	E101

In [ ]:
```
# isdigit/isalpha...
df[df['firstname'].str.isdigit()]
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked

slicing

In [ ]:
```
df['Name'].str[::-1]
```

Out[ ]:
```
0                                sirraH newO .rM ,dnuarB
1      )reyahT sggirB ecnerolF( yeldarB nhoJ .srM ,sg...
2                                 aniaL .ssiM ,nenikkieH
3           )leeP yaM yliL( htaeH seuqcaJ .srM ,ellertuF
4                               yrneH mailliW .rM ,nellA
                              ...
886                                sazouJ .veR ,alivtnoM
887                           htidE teragraM .ssiM ,maharG
888              "eirraC" neleH enirehtaC .ssiM ,notsnhoJ
889                                llewoH lraK .rM ,rheB
890                                  kcirtaP .rM ,yelooD
Name: Name, Length: 891, dtype: object
```

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
```

Python Usecases Interview Questions

Write a Python function to merge two sorted lists into a new sorted list.

In [1]:
```
def merge_sorted_lists(list1, list2):
    merged_list = sorted(list1 + list2)
    return merged_list

# Example usage:
list1 = [1, 3, 5]
list2 = [2, 4, 6]
result = merge_sorted_lists(list1, list2)
print(result)
```

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

Find the Missing Number

In [2]:
```
def find_missing_number(nums):
    n = len(nums) + 1
    expected_sum = n * (n + 1) // 2
    actual_sum = sum(nums)
    return expected_sum - actual_sum

# Example usage:
numbers = [0, 1, 3, 4, 5]
result = find_missing_number(numbers)
print(result)
```

```
8
```

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

In [4]:
```
# 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[4]:
```
[3, 4, 5]
```

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

In [5]:
```
# 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}
```

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

In [7]:
```
# 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}
```

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

In [8]:
```
def is_palindrome(s):
    return s == s[::-1]

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

```
True
```

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

In [9]:
```
def factorial_iterative(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result

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

```
120
```

Write a Python function to check if two strings are anagrams.

In [1]:
```
def are_anagrams(str1, str2):
    return sorted(str1) == sorted(str2)

# Example usage:
word1 = "listen"
word2 = "silent"
result = are_anagrams(word1, word2)
print(result)
```

```
True
```

Write a Python function to find the Nth number in the Fibonacci sequence.

In [2]:
```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# Example usage:
n = 6
result = fibonacci(n)
print(result)
```

```
8
```

Write a Python function to remove duplicates from a list while preserving the order.

In [3]:
```
def remove_duplicates(nums):
    unique_nums = []
    for num in nums:
        if num not in unique_nums:
            unique_nums.append(num)
    return unique_nums

# Example usage:
numbers = [1, 2, 2, 3, 4, 4, 5]
result = remove_duplicates(numbers)
print(result)
```

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

Implement a simple queue in Python with enqueue and dequeue operations.

In [4]:
```
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)

    def is_empty(self):
        return len(self.items) == 0
```

```
# Example usage:
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.dequeue())
```

```
1
```

Write a Python function to check if a given number is prime.

In [5]:
```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

# Example usage:
number = 11
result = is_prime(number)
print(result)
```

```
True
```

Write a Python function to remove duplicates from a list.

In [1]:
```
def remove_duplicates(lst):
    return list(set(lst))

# Example usage:
my_list = [1, 2, 2, 3, 4, 4, 5]
result = remove_duplicates(my_list)
print(result)
```

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

Find the Maximum Subarray Sum

In [2]:
```
def max_subarray_sum(nums):
    max_sum = current_sum = nums[0]
    for num in nums[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum

# Example usage:
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
result = max_subarray_sum(arr)
print(result)
```

```
6
```

Write a Python function to find the Nth number in the Fibonacci sequence.

In [4]:
```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# Example usage:
N = 8
result = fibonacci(N)
print(result)
```

```
21
```

Write a Python function to find the intersection of two lists.

In [6]:
```
def find_intersection(list1, list2):
    return list(set(list1) & set(list2))

# Example usage:
list_x = [1, 2, 3, 4, 5]
list_y = [3, 4, 5, 6, 7]
result = find_intersection(list_x, list_y)
print(result)
```

```
[3, 4, 5]
```

Write a Python function to calculate the power of a number using recursion.

In [8]:
```
def power(base, exponent):
    if exponent == 0:
```

```
        return 1
    else:
        return base * power(base, exponent - 1)

# Example usage:
result = power(2, 3)
print(result)
```

```
8
```

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 [1]:
```
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()
```

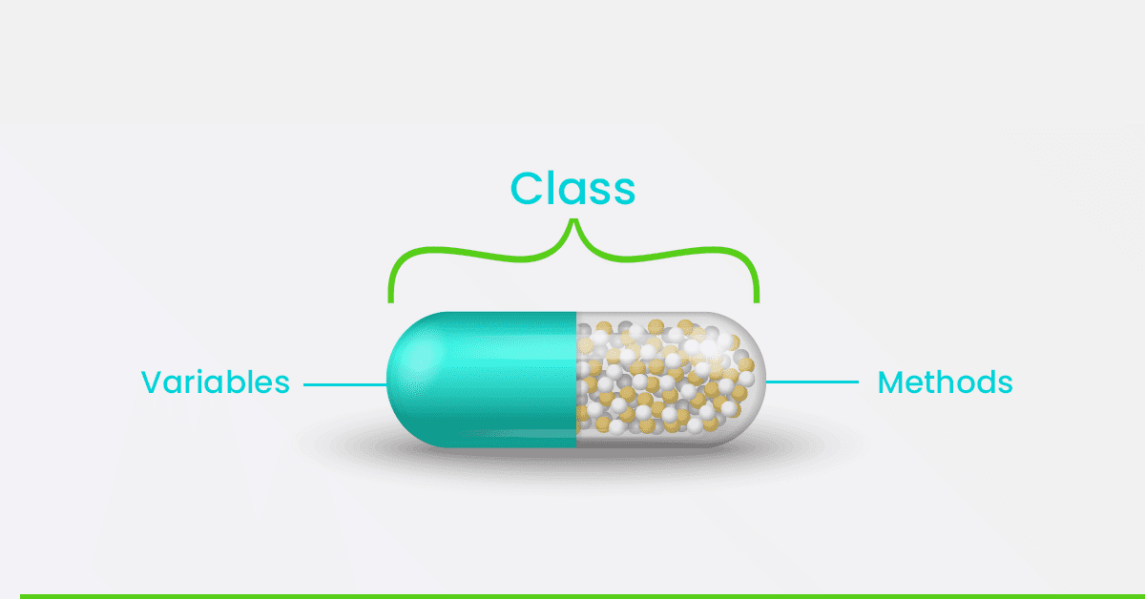
```
Buddy is 2 years old.
Molly is 4 years old.
Buddy says Woof!
Molly says Woof!
```

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 [3]:
```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.

List Comprehension Questions

Given a list of integers, numbers, write a Python program to generate a new list, squared_numbers, containing the squares of each number using list comprehension.

In [1]:
```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [num**2 for num in numbers]
print(squared_numbers)
```

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

Write a Python program that takes a list of integers, numbers, and generates a new list, even_numbers, containing only the even numbers using list comprehension.

In [2]:
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [num for num in numbers if num % 2 == 0]
print(even_numbers)
```

```
[2, 4, 6, 8, 10]
```

Write a Python code to create a new list that contains squares of even numbers and cubes of odd numbers from the given list.

In [2]:
```
# Sample Input
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Solution using list comprehension
result = [num ** 2 if num % 2 == 0 else num ** 3 for num in numbers]

# Output
print(result)
```

```
[1, 4, 27, 16, 125, 36, 343, 64, 729, 100]
```

Write a Python code to create a new list that contains the squares of numbers from 1 to 10.

In [3]:
```
squred_numbers = [num ** 2 for num in range(1,11)]

# output
print(squred_numbers)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Given two lists, create a new list that contains the common elements using list comprehension.

In [4]:
```
list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 5, 6, 7]

common_elements = [x for x in list1 if x in list2]

print(common_elements)
```

```
[3, 4, 5]
```

Given a list of numbers, create a new list that contains the square of each number if it is positive, and zero otherwise.

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

squares = [x**2 if x > 0 else 0 for x in numbers]

print(squares)
```

```
[1, 0, 9, 0, 25, 0]
```

Write a list comprehension to extract all the vowels from a given string.

In [6]:
```
input_string = "hello world"

vowels = [char for char in input_string if char.lower() in 'aeiou']

print(vowels)
```

```
['e', 'o', 'o']
```

Create a Python function that takes a list of strings, words, and returns a new list, word_lengths, containing the lengths of each string using list comprehension.

In [3]:
```
words = ["apple", "banana", "kiwi", "orange"]
word_lengths = [len(word) for word in words]
print(word_lengths)
```

```
[5, 6, 4, 6]
```

Dictionary comprehension Interview Questions

Question 1: Explain what a dictionary comprehension is in Python.

Answer: A dictionary comprehension is a concise way to create dictionaries in Python. It is similar to a list comprehension but produces key-value pairs in the form of a dictionary. The general syntax is

"{ `key_expression` : `value_expression` **for** `item` **in** `iterable` }""

Question 2: How can you use conditions in a dictionary comprehension? Provide an example.

Answer: Conditions can be incorporated into a dictionary comprehension using an if statement.

For instance, consider the following example that creates a dictionary of squares for even numbers from 0 to 8:

In [2]:
```
even_squares_dict = {x: x**2 for x in range(9) if x % 2 == 0}

print(even_squares_dict)
```

```
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

Question 3: Can you explain the role of expressions key_expression and value_expression in a dictionary comprehension?

Answer: key_expression and value_expression are the expressions that define the key and value for each key-value pair in the resulting dictionary. They are evaluated for each element in the iterable specified in the comprehension.

For example, in the expression

`{x: x**2 for x in range(5)}` , x is the key_expression, and x**2 is the value_expression.

Question 4: How would you use a dictionary comprehension to filter and transform data from an existing dictionary?

Answer: You can use a dictionary comprehension to filter and transform data by applying conditions and expressions to the existing dictionary.

For instance, consider the following example that filters out odd squares from a dictionary:

In [3]:
```
original_dict = {x: x**2 for x in range(5)}
filtered_dict = {key: value for key, value in original_dict.items() if value % 2 ==

print(original_dict)

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

print(filtered_dict)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
--------------
{0: 0, 2: 4, 4: 16}
```

Question 5: Are dictionary comprehensions more ef`icient than traditional methods of creating dictionaries in Python?

Answer:

- In terms of efficiency, dictionary comprehensions are generally more concise and can be as efficient as traditional methods.
- They provide a more readable and compact way to create dictionaries. However, for very large datasets or complex conditions, the difference in performance may be negligible.
- It's always essential to consider readability and maintainability alongside performance when choosing between different approaches.