BROWSEJOBS

PYTHON QUESTIONNAIRE

1. **What is the difference between list and tuples in python?**

LISTS	TUPLES
Lists are mutable, which means their contents can be changed after they are created, while	Tuples are immutable, which means once they are created, their contents cannot be changed.
Lists are created using square brackets [],	tuples are created using parentheses ().
Lists are commonly used to store and manipulate data that can be changed, such as a list of names that may need to be sorted or appended to.	Tuples are commonly used to store data that cannot be changed, such as a pair of coordinates or a date and time.
Lists can be slower to access and process, especially when they are modified frequently.	tuples are immutable, they can be faster to access and process than lists, especially when dealing with large amounts of data.

2. **What is the difference between Sets and tuples in python?**

SETS	TUPLES
Sets are immutable, which means their contents cannot be changed after they are created	Tuples are immutable, which means once they are created, their contents cannot be changed.

Lists are created using Curly brackets {},	tuples are created using parentheses ().
Sets do not allow duplicate values	tuples can have duplicate values.
Sets are unordered, which means the order of elements in a set is not guaranteed	while tuples are ordered, which means the order of elements in a tuple is preserved.

3. **Lists or tuples which is better, and why?**

   Here are some factors to consider:

1. **Mutability:** If you need to modify the contents of the collection after it has been created, then a list is the better choice, as it is mutable. If the contents of the collection should not be changed after it has been created, then a tuple is the better choice, as it is immutable.
2. **Performance:** Tuples can be faster to access and process than lists, especially when dealing with large amounts of data. This is because tuples are stored in a more compact format than lists, and their immutability allows them to be optimized by the Python interpreter. However, if you need to modify the contents of the collection frequently, then a list may be the better choice, even if it is slightly slower to access and process.
3. **Use case:** Lists are commonly used to store and manipulate data that can be changed, such as a list of names that may need to be sorted or appended to. Tuples are commonly used to store data that cannot be changed, such as a pair of coordinates or a date and time.

   In general, if you need a collection that can be modified after it has been created, then a list is the better choice. If you need a collection that cannot be modified after it has been created, or if you need a collection that is faster to access and process, then a tuple is the better choice.

4. **What is memory management in python?**

   ANS: Memory management in Python refers to the way that Python handles the allocation and deallocation of memory used by objects in a Python program. In memory management memory is stored in 2 spaces that are Stack and Heap. Where stack stores all variables and heap stores all values.

   In addition to reference counting, Python also uses a mechanism called "garbage collection" to deallocate memory that is no longer being used by the program. Garbage collection is a process where Python identifies objects that are no longer reachable from the program's execution context, and deallocates their memory.

5. **What is memory allocation in the list?**

   ANS: In Python, a list is a dynamic data structure that can grow or shrink in size as needed. When a new list is created, Python allocates a block of memory to store the list's elements. The amount of memory allocated depends on the initial size of the list, which can be specified when the list is created, or it can be left unspecified.

   When the size of the list exceeds the currently allocated memory, Python allocates a new block of memory that is larger than the current block, and copies the list's elements to the new block. The old block of memory is then deallocated.

   The amount of memory allocated for a list can be controlled using the **sys.getsizeof()** function, which returns the size of an object in bytes. The actual amount of memory used by a list may be larger than the size returned by **getsizeof(),** due to the over-allocation technique used by Python.

6. **What are List Methods?**

append()

The append() method appends an element to the end of the list.

Syntax list.append(elmnt)

Example:

Add an element to the fruits list:

```
fruits = ['apple', 'banana', 'cherry']

fruits.append("orange")

fruits
```

clear()

The clear() method removes all the elements from a list.

Syntax list.clear()

Example:

```
fruits = ['apple', 'banana', 'cherry', 'orange']

fruits.clear()

fruits
```

remove()

The remove() method removes the first occurrence of the element with the specified value.

Syntax list.remove(elmnt)

Example:

Remove the "banana" element of the fruit list:

```
fruits = ['apple', 'banana', 'cherry']

fruits.remove("banana")

fruits
```

copy()

The copy() method returns a copy of the specified list.

Syntax list.copy()

Example:

Copy the fruits list:

```
fruits = ['apple', 'banana', 'cherry', 'orange']

x = fruits.copy()

x
```

count()

The count() method returns the number of elements with the specified value.

Syntax list.count(value)

Example:

Return the number of times the value "cherry" appears in the fruits list:

```
fruits = ['apple', 'banana', 'cherry']

x = fruits.count("cherry")

x
```

extend()

The extend() method adds the specified list elements (or any iterable) to the end of the current list.

Syntax list.extend(iterable)

Example:

Add the elements of cars to the fruits list

```
fruits = ['apple', 'banana', 'cherry']

cars = ['Ford', 'BMW', 'Volvo']

fruits.extend(cars)

fruits
```

index()

The index() method returns the position at the first occurrence of the specified value.

Syntax list.index(elmnt)

Example:

What is the position of the value 32:

```
fruits = [4, 55, 64, 32, 16, 52]

x = fruits.index(32)

x
```

insert()

The insert() method inserts the specified value at the specified position.

Syntax list.insert(pos, elmnt)

Example:

Insert the value "orange" as the second element of the fruit list:

```
fruits = ['apple', 'banana', 'cherry']

fruits.insert(1, "orange")

fruits
```

pop()

The pop() method removes the element at the specified position.

Syntax list.pop(pos)

Remove the second element of the fruit list:

```
fruits = ['apple', 'banana', 'cherry']

fruits.pop(1)

fruits
```

reverse()

Syntax list.reverse()

Example:

Reverse the order of the fruit list:

```
fruits = ['apple', 'banana', 'cherry']

fruits.reverse()

fruits
```

sort()

Syntax list.sort(reverse=True) for descending

Example:

Sort the list alphabetically:

```
cars = ['Ford', 'BMW', 'Volvo']

cars.sort()

cars
```

for sorting in descending order

Sort the list descending:

```
cars = ['Ford', 'BMW', 'Volvo']

cars.sort(reverse=True)

cars
```

7. **What are Dictionary Methods?**

clear()

The clear() method removes all the elements from a dictionary.

Syntax dictionary.clear()

Example:

car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

car.clear()

car

copy()

The copy() method returns a copy of the specified dictionary.

Syntax dictionary.copy()

Example:

car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.copy()

x

fromkeys()

The fromkeys() method returns a dictionary with the specified keys and the specified value.

Syntax dict.fromkeys(keys, value)

Example:

x = ('key1', 'key2', 'key3') y = 0

thisdict = dict.fromkeys(x, y)

thisdict

get()

The get() method returns the value of the item with the specified key.

Example:

car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.get("model")

x

items()

The items() method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list.

The view object will reflect any changes done to the dictionary..

Syntax dictionary.items()

Example:

car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.items()

x

keys()

The keys() method returns a view object. The view object contains the keys of the dictionary, as a list.

The view object will reflect any changes done to the dictionary

Syntax dictionary.keys()

Example:1 car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.keys()

x

Example:2

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.keys()

car["color"] = "white"

x
```

pop()

The pop() method removes the specified item from the dictionary.

Syntax dictionary.pop(keyname)

Example

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

car.pop("model")

car
```

setdefault()

The setdefault() method returns the value of the item with the specified key.

If the key does not exist, insert the key

Syntax dictionary.setdefault(keyname)

Example:1

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.setdefault("model")

print(x)
```

Example:2

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.setdefault("color", "white")

car
```

update()

The update() method inserts the specified items to the dictionary.

The specified items can be a dictionary

Syntax dictionary.update(iterable)

Example:

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

car.update({"color": "White"})

car
```

values()

The values() method returns a view object. The view object contains the values of the dictionary, as a list.

The view object will reflect any changes done to the dictionary

Syntax dictionary.values()

Example:1

```
car = { "brand": "Ford", "model": "Mustang", "year": 1964 }

x = car.values()

print(x)
```

8. **What are Set Methods?**

add()

The add() method adds an element to the set.

If the element already exists, the add() method does not add the element.

Syntax set.add(elmnt)

clear()

The clear() method removes all elements in a set.

Syntax set.clear()

copy()

The copy() method copies the set.

Syntax set.copy()

difference()

The difference() method returns a set that contains the difference between two sets. Meaning: The returned set contains items that exist only in the first set, and not in both sets.

Syntax set.difference(set)

difference_update()

The difference_update() method removes the items that exist in both sets.

The difference_update() method is different from the difference() method, because the difference() method returns a new set, without the unwanted

items, and the difference_update() method removes the unwanted items from the original set.

Syntax set.difference_update(set)

discard()

Syntax set.discard(value)

The discard() method removes the specified item from the set.

This method is different from the remove() method, because the remove() method will raise an error if the specified item does not exist, and the discard() method will not.

intersection()

The intersection() method returns a set that contains the similarity between two or more sets.

Meaning: The returned set contains only items that exist in both sets, or in all sets if the comparison is done with more than two sets.

Syntax set.intersection(set1, set2 ... etc)

intersection_update()

The intersection_update() method removes the items that are not present in both sets (or in all sets if the comparison is done between more than two sets).

The intersection_update() method is different from the intersection() method, because the intersection() method returns a new set, without the unwanted items, and the intersection_update() method removes the unwanted items from the original set.

Syntax set.intersection_update(set1, set2 ... etc)

isdisjoint()

The isdisjoint() method returns True if none of the items are present in both sets, otherwise it returns False.

Syntax set.isdisjoint(set)

issubset()

The issubset() method returns True if all items in the set exists in the specified set, otherwise it returns False.

Syntax set.issubset(set)

issuperset()

The issuperset() method returns True if all items in the specified set exists in the original set, otherwise it returns False.

Syntax set.issuperset(set)

pop()

The pop() method removes a random item from the set.

This method returns the removed item.

Syntax set.pop()

remove()

The remove() method removes the specified element from the set.

This method is different from the discard() method, because the remove() method will raise an error if the specified item does not exist, and the discard() method will not.

Syntax set.remove(item)

symmetric_difference()

The symmetric_difference() method returns a set that contains all items from both set, but not the items that are present in both sets.

Meaning: The returned set contains a mix of items that are not present in both sets.

Syntax set.symmetric_difference(set)

union()

The union() method returns a set that contains all items from the original set, and all items from the specified set(s). If an item is present in more than one set, the result will contain only one appearance of this item.

Syntax set.union(set1, set2...)

update()

The update() method updates the current set, by adding items from another set (or any other iterable).

If an item is present in both sets, only one appearance of this item will be present in the updated set.

Syntax set.update(set)

9. **What are Tuple Methods?**

len ()

Returns the length of a tuple

Example:

```
companies=('apple','google','bmw','skoda')
len(companies)
```

max()

it shows the max value in the tuple

Example:

z=(22,3444,554,5566,332)

max(z)

min()

it shows the max value in the tuple

Example:

z=(22,3444,554,5566,332)

min(z)

index()

it returns the index of the existing elements

Example:

z=(22,3444,554,5566,332)

z.index(554)

count()

Its shows how many times the occurrence has occurred in a particular tuple

Example:

x=(1,2,2,3,3,3,3,4,4,43,7)

x.count(2)

tuple()

its is used to create different type of values into tuple

Example:

u=[1,3,4,45,5,5]

```
f= tuple(u)
```

10. What is the difference between append and extend?

In Python, append() and extend() are methods that can be used to add elements to a list. Here are the main differences between the two:

- **append():** This method is used to add a single element to the end of a list. The argument passed to append() is added as a single element to the list.

  For example:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
```

Output: [1, 2, 3, 4]

- **extend():** This method is used to add multiple elements to the end of a list. The argument passed to extend() should be an iterable object (e.g. another list, tuple, or set) that contains the elements to be added to the list. The elements in the iterable object are added to the list as separate elements.

```
my_list = [1, 2, 3]
my_list.extend([4, 5])
print(my_list)
```

Output: [1, 2, 3, 4, 5]

11. What is the difference between remove and pop?

- **Pop:** removes and returns the last element of a list by default, or the element at the specified index if an index is provided. The pop method modifies the list in place.
  Here's an example:

```
# Remove the element at index 1
second_element = my_list.pop(1)
print(my_list)                    #output - [1, 3, 4]
```

- **Remove:** removes the first occurrence of a specified element from a list. The remove method modifies the list in place.

  Here's an example:

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

# Remove the element 3
my_list.remove(3)
print(my_list)                    #output - [1, 2, 4]
```

12. Why is string immutable?

In many programming languages, including Python, strings are immutable. This means that once a string is created, it cannot be modified.
In Python, when you create a string, the memory is allocated for that string and the contents of the string are stored in that memory. Any attempt to modify the contents of the string will result in an error.

13. What are List comprehensions and dictionary comprehensions? Why do we use them, give examples.

List comprehensions and dictionary comprehensions are two powerful features in Python that allow us to create new lists and dictionaries in a more concise and readable way.

List comprehensions allow us to generate a new list by applying a transformation to each element of an existing list or other iterable.

Here's an example:

```
# Create a list of the first 10 squares of natural numbers
squares = [x**2 for x in range(1, 11)]
print(squares)
```

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

Dictionary comprehension is a concise way to create a new dictionary by applying a transformation to each element of an existing iterable. Here's an example:

```
# Create a dictionary that maps each number to its square
squares_dict = {x: x**2 for x in range(1, 11)}
print(squares_dict)
```

Output - {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}

14. How to unpack nested dictionaries?

To unpack nested dictionaries in Python, you can use a combination of square brackets and keys to access the values stored in each dictionary. Here's an example of how to unpack a nested dictionary with two levels of nesting:

```
# Define a nested dictionary
person = {
    'name': 'John',
```

```
    'age': 30,
    'address': {
        'street': '123 Main St',
        'city': 'Anytown',
        'state': 'CA'
    }
}


# Unpack the nested dictionary
street = person['address']['street']
city = person['address']['city']
state = person['address']['state']

# Print the unpacked values
print(street)   # Output: 123 Main St
print(city)     # Output: Anytown
print(state)    # Output: CA
```

To unpack the nested dictionary, we use square brackets to access the values stored in each dictionary. We first access the 'address' key using person['address'], which returns the nested dictionary. We then use another set of square brackets to access the values of the keys within the nested dictionary, such as person['address']['street'] to access the 'street' value. You can continue this process to unpack nested dictionaries with more levels of nesting by using multiple sets of square brackets and keys to access the values stored in each dictionary.

15. What are the common built-in data types in Python?

Ans: The common built-in data types in python are

1. List– An ordered sequence of items is called a list. The elements of a list may belong to different data types. Eg. [5,'market',2.4]

2. Tuple– It is also an ordered sequence of elements. Unlike lists , tuples are immutable, which means they can't be changed. Eg. (3,'tool',1)

3. String– A sequence of characters is called a string. They are declared within single or double-quotes. Eg. "Sana", 'She is going to the market', etc.

4. Set– Sets are a collection of unique items that are not in order. Eg. {7,6,8}

5. Dictionary– A dictionary stores values in key and value pairs where each value can be accessed through its key. The order of items is not important. Eg. {1:'apple',2:'mango}

6. Boolean– There are 2 boolean values- True and False.

7. Numbers– They include integers, floating-point numbers, and complex numbers. eg. 1, 7.9,3+4i

16. What are docstrings in Python?

Docstrings are not actually comments, but, they are **documentation strings**

In Python, a docstring is a string literal that appears as the first statement in a module, function, class, or method definition. Its purpose is to document the object and provide useful information about its purpose, usage, and behavior. Docstrings can be written using single or triple quotes, and they should be written in a clear and concise manner, using proper grammar and punctuation.

```
"""
Using docstring as a comment.
This code divides 2 numbers
"""
x=8
y=4
z=x/y
print(z)
```

Output: 2.0

17. What is pep 8?

PEP 8 (Python Enhancement Proposal 8) is a style guide for writing Python code. It is a set of guidelines that describes how to write Python code in a way that is easy to read and understand. PEP 8 is widely used and respected in the Python community, and many popular Python projects follow its guidelines.

PEP 8 covers a variety of topics related to coding style, including naming conventions, indentation, whitespace, comments, and more

18. Python is an interpreted language. Explain.

An interpreted language is any programming language which is not in machine level code before runtime. Therefore, Python is an interpreted language.

Python is an interpreted language, which means that code written in Python is executed directly by the Python interpreter, rather than being compiled into machine code and executed by the computer's processor.

19. What are the benefits of using Python?

Ans: The benefits of using python are

1. Easy to use– Python is a high-level programming language that is easy to use, read, write and learn.
2. Interpreted language– Since python is an interpreted language, it executes the code line by line and stops if an error occurs in any line.
3. Dynamically typed– the developer does not assign data types to variables at the time of coding. It automatically gets assigned during execution.
4. Free and open-source– Python is free to use and distribute. It is open source.
5. Extensive support for libraries– Python has vast libraries that contain almost any function needed. It also further provides the facility to import other packages using Python Package Manager(pip).
6. Portable– Python programs can run on any platform without requiring any change.
7. The data structures used in python are user friendly.
8. It provides more functionality with less coding

20. What are Python namespaces?

Ans: A namespace in python refers to the name which is assigned to each object in python. The objects are variables and functions. As each object is created, its name along with space(the address of the outer function in which the object is), gets created. The namespaces are maintained in python like a dictionary where the key is the namespace and value is the address of the object.

There are 4 types of namespace in python

1. Built-in namespace– These namespaces contain all the built-in objects in python and are available whenever python is running.
2. Global namespace– These are namespaces for all the objects created at the level of the main program.
3. Enclosing namespaces– These namespaces are at the higher level or outer function.
4. Local namespaces– These namespaces are at the local or inner function.

21. What are decorators?Give an example.

In Python, a decorator is a special type of function that can modify the behavior of another function without changing its source code.
Decorators are used to add functionality to functions or classes, or to modify their behavior in some way.

22. Is integer immutable or mutable and why

In Python, integers are immutable. This means that once an integer object is created, it cannot be changed. Any operation that appears to change an integer actually creates a new integer object.

23. What is the difference between .py and .pyc files?

In Python, .py files are the source code files that contain the Python code that you write. When you run a Python program, the interpreter reads the .py file and executes the code.
On the other hand, .pyc files are compiled bytecode files that are generated by the Python interpreter when it compiles your source code. The .pyc files contain the compiled version of your code, which can be executed more efficiently by the interpreter.

The main differences between .py and .pyc files are:

1. **File extension: .py** files have the extension .py, while **.pyc** files have the extension .pyc.
2. **Human-readable vs. machine-readable**: **.py** files are human-readable and can be edited using a text editor, while **.pyc** files are machine-readable and cannot be easily edited by humans.
3. **Compilation: .py** files are compiled at runtime by the Python interpreter, while **.pyc** files are pre-compiled by the interpreter and stored on disk for later use.
4. **Performance:** Since **.pyc** files are pre-compiled, they can be executed more quickly than **.py** files, especially for larger programs.

24. What are Keywords in Python?

Ans: Keywords in python are reserved words that have special meaning.They are generally used to define types of variables. Keywords cannot be used for variable or function names.

There are following 33 keywords in python-

- **And**
- **Else**
- **For**
- **While**
- **Break**
- **As**
- **Def**
- **Lambda**
- **Or**
- **Not**
- **If**
- **Elif**
- **Pass**
- **Return**
- **True**
- **False**
- **Try**
- **With**
- **Assert**
- **Class**

• **Continue**

25. What are python Modules and packages?

In Python, a module is a file containing Python definitions and statements that can be imported and used in other Python code. A package is a collection of related modules that are organized into a directory hierarchy.

Here are some more details about modules and packages in Python:
Modules:

- A module is a single file containing Python code that defines functions, classes, and variables that can be used in other Python programs.
- To use a module in your Python program, you can import it using the import statement.
- When you import a module, Python executes the code in the module and creates a module object that contains the module's definitions.
- You can access the definitions in a module using dot notation, such as module_name.function_name() or module_name.variable_name.

Packages:

- A package is a collection of related modules that are organized into a directory hierarchy. The directory that contains the package is called the package directory.
- A package directory must contain a special file called __init__.py that is executed when the package is imported.
- You can import a module from a package using dot notation, such as package_name.module_name.
- You can also use the from statement to import specific functions, classes, or variables from a module, such as from package_name.module_name import function_name.

26. What is the Latest version of python?

The latest version of Python was version 3.10.0, which was released on October 4, 2021.
Before version 3.10, the latest version of Python was version 3.9.7, which was released on August 30, 2021. Prior to that, the latest version was 3.9.6, which was released on June 28, 2021.

It's worth noting that the latest version of Python is generally recommended for new projects, as it includes the most up-to-date features, bug fixes, and security updates. However, if you are working on an existing project, you may want to check its compatibility with the latest version of Python before upgrading.

27. What are local variables and global variables in Python?

Global Variables: Variables declared outside a function or in global space are called global variables. These variables can be accessed by any function in the program.

Local Variables: Any variable declared inside a function is known as a local variable. This variable is present in the local space and not in the global space.

Example:

```
a=2
def add():
b=3
c=a+b
print(c)
add()
```

Output: 5

When you try to access the local variable outside the function add(), it will throw an error.

28. What is type conversion in Python?

Ans: Type conversion refers to the conversion of one data type into another.

- **int() –** converts any data type into integer type
- **float() –** converts any data type into float type
- **ord() –** converts characters into integer
- **hex() –** converts integers to hexadecimal
- **oct() –** converts integer to octal
- **tuple() –** This function is used to convert to a tuple.
- **set() –** This function returns the type after converting to set.
- **list() –** This function is used to convert any data type to a list type.
- **dict() –** This function is used to convert a tuple of order (key, value) into a dictionary.
- **str() –** Used to convert an integer into a string.
- **complex(real,imag) –** This function converts real numbers to complex(real,imag) numbers.

29. What is the difference between Python Arrays and lists?

Python arrays and lists are both used to store collections of data. However, there are some key differences between the two:

1. **Data Types:** Python **lists** can store elements of different data types, whereas **arrays** are designed to store elements of the same data type. This means that all elements in an array must be of the same type, such as integers or floating-point numbers.
2. **Memory Allocation**: **Lists** are implemented as dynamic arrays, which means that their size can be changed dynamically during runtime. **Arrays**,

on the other hand, are implemented as fixed-size arrays, which means that their size cannot be changed once they are created.

3. **Performance: Arrays** are generally faster than lists when it comes to indexing and accessing elements, as they use a contiguous block of memory for storage. **Lists**, on the other hand, may need to allocate additional memory when they grow in size, which can slow down operations that involve accessing or modifying elements.

30. What is __init__?

Ans: __init__ is a method or constructor in Python. This method is automatically called to allocate memory when a new object/ instance of a class is created. All classes have the __init__ method.

Here is an example of how to use it.

```
class Employee:
def __init__(self, name, age,salary):
self.name = name
self.age = age
self.salary = 20000
E1 = Employee("XYZ", 23, 20000)
# E1 is the instance of class Employee.
#__init__ allocates memory for E1.
print(E1.name)
print(E1.age)
print(E1.salary)
```

Output:

```
XYZ
23
20000
```

31. What are magic and dunder methods?

Magic methods and dunder methods are special methods in Python that allow objects to behave in certain ways when certain operators or built-in functions are used on them. "Magic" is short for "magic methods," and "dunder" is short for "double underscore."

These methods are typically defined with special names that start and end with double underscores, such as __init__, __str__, __len__, __eq__, and __add__.

Magic methods are used to define the behavior of an object when certain built-in functions or operators are used on it. **For example**, __len__ is a magic method that defines how the built-in len() function should behave when called on an object of that class.
Dunder methods are a specific type of magic method that use double underscores at the beginning and end of their name. These methods are used to define special behavior for certain operators or built-in functions. **For example,** __add__ is a dunder method that defines how the + operator should behave when used with an object of that class.

32. What is a lambda function?

In Python, a lambda function is a small, anonymous function that can take any number of arguments but can only have one expression. The expression is evaluated and returned as the result of the function.

Lambda functions are defined using the lambda keyword, followed by the arguments and the expression, separated by a colon.

Example:
a = lambda x,y : x+y

```
print(a(5, 6))
```

Output: 11

33. What is self in Python?

Ans: Self is an instance or an object of a class. In Python, this is explicitly included as the first parameter. However, this is not the case in Java where it's optional. It helps to differentiate between the methods and attributes of a class with local variables. The self variable in the init method refers to the newly created object while in other methods, it refers to the object whose method was called.

34. How does break, continue and pass work?

Break - Allows loop termination when some condition is met and the control is transferred to the next statement

Continue - Allows skipping some part of a loop when some specific condition is met and the control is transferred to the beginning of the loop

Pass - Used when you need some block of code syntactically, but you want to skip its execution. This is basically a null operation. Nothing happens when this is executed.

35. Reverse string without using indexing.

One way to reverse a string in Python without using indexing is to use slicing with a step of -1

Here's an example:

```
string = "Hello, world!"
reversed_string = string[::-1]
print(reversed_string)
```

Output

!dlrow ,olleH

36. What are iterators in python?

In Python, an iterator is an object that can be iterated (looped) upon, meaning that you can traverse through all the values that it contains one by one. An iterator is used to represent a stream of data, such as the contents of a list or the characters in a string.

Iterators are implemented using the iterator protocol, which requires two methods: **__iter__()** and **__next__()**. The **__iter__()** method returns the iterator object itself, while the **__next__()** method returns the next value in the stream

37. What are generators in python?

In Python, a generator is a type of iterator that allows you to generate a sequence of values on the fly, as opposed to storing them all in memory at once. Generators are implemented using a special type of function called a generator function, which is defined using the ***yield*** keyword instead of the ***return*** keyword.

38. What is encapsulation in python?

Encapsulation is a fundamental concept of object-oriented programming (OOP) that refers to the practice of hiding the internal details of an object from the outside world and instead exposing a public interface that can be used to interact with the object.

In Python, encapsulation is achieved through the use of access modifiers, which are keywords that determine the level of visibility of an object's attributes and methods. Python provides two types of access modifiers:

1. **Public:** Public attributes and methods are those that can be accessed from outside the object using the dot notation. In Python, all attributes and methods are public by default.
2. **Private**: Private attributes and methods are those that are only accessible from within the object itself. In Python, private attributes and methods are denoted by a double underscore prefix (__) before the attribute or method name. This tells Python to **"mangle"** the name of the attribute or method, making it harder to access from outside the object.

39. What is the difference between range & xrange?

Ans: For the most part, xrange and range are the exact same in terms of functionality. They both provide a way to generate a list of integers for you to use, however you please. The only difference is that range returns a Python list object and x range returns an xrange object.

This means that xrange doesn't actually generate a static list at run-time like range does. It creates the values as you need them with a special technique called yielding. This technique is used with a type of object known as generators. That means that if you have a really gigantic range you'd like to generate a list for, say one billion, xrange is the function to use.

This is especially true if you have a really memory sensitive system such as a cell phone that you are working with, as range will use as much memory as

it can to create your array of integers, which can result in a Memory Error and crash your program. It's a memory hungry beast.

40. What is pickling and unpickling?

Ans: Pickle module accepts any Python object and converts it into a string representation and dumps it into a file by using the dump function, this process is called pickling. While the process of retrieving original Python objects from the stored string representation is called unpickling.

41. What does this mean: *args, **kwargs? And why would we use it?

Ans: We use ***args** when we aren't sure how many arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function. ****kwargs** is used when we don't know how many keyword arguments will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments. The identifiers args and kwargs are a convention, you could also use *bob and **billy but that would not be wise.

42. How to convert python file into a package

Converting a Python file into a package involves a few steps:

1. **Create a directory:** Create a new directory with a name that will be the name of your package. This directory will contain your package files.
2. **Move the Python file into the directory:** Move the Python file that you want to convert into a package into the directory you created in the previous step.
3. **Create an empty init.py file:** This file tells Python that the directory is a package. It can be an empty file, or it can contain initialization code.

4. **Refactor the code:** Refactor your code so that it can be imported as a module. This means you'll need to define the functions, classes, and variables you want to export from the package in the Python file.
5. **Update import statements:** In the Python file, update any import statements to reflect the new package structure.
6. **Test the package:** Test the package by importing it in a new Python script and running the code.

43. What is a shallow and deep copy? Give an example.

A shallow copy creates a new object which is a copy of the original object. However, the new object only contains references to the same nested objects as the original object. In other words, changes made to the nested objects in the new object will also affect the original object.

Here's an example of a shallow copy:

```
original_list = [[1, 2], [3, 4]]
new_list = original_list.copy()

new_list[0][0] = 5

print(original_list)

# Output: [[5, 2], [3, 4]]
```

As you can see, changing the first element of the first nested list in the new list also changed the corresponding element in the original list. This is because the new list is a shallow copy of the original list.
Deep copy creates a new object with a new memory location for the original object and all of its nested objects. Changes made to the nested objects in the new object will not affect the original object.

Here's an example of a deep copy:

```
import copy

original_list = [[1, 2], [3, 4]]
new_list = copy.deepcopy(original_list)

new_list[0][0] = 5

print(original_list)

# Output:   [[1, 2], [3, 4]]
```

As you can see, changing the first element of the first nested list in the new list did not affect the corresponding element in the original list. This is because the new list is a deep copy of the original list.

44. What is Inheritance and mention different types of inheritance with examples.

Inheritance is a fundamental feature of object-oriented programming (OOP) that allows classes to inherit properties and behaviors from other classes. In OOP, inheritance is used to create new classes that are built upon existing classes.

The basic syntax for creating a class that inherits from another class in Python is as follows:

```
class ChildClass(ParentClass):
# Child class definition
```

There are several types of inheritance, including:

- **Single Inheritance:**
  Single inheritance is the most common type of inheritance. In single inheritance, a class inherits from only one parent class.

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

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

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        print(f"{self.name} barks")

dog = Dog("Rover")
dog.speak()                          # Output: "Rover barks"
```

- **Multiple Inheritance:**
  Multiple inheritance is when a class inherits from more than one parent class.

```
class Flyer:
    def fly(self):
        print("I'm flying!")

class Swimmer:
    def swim(self):
        print("I'm swimming!")
```

```
class Duck(Flyer, Swimmer):
    pass

duck = Duck()
duck.fly()                    # Output: "I'm flying!"
duck.swim()                   # Output: "I'm swimming!"
```

- **Multi-level Inheritance:**
  Multi-level inheritance is when a class inherits from a parent class, which in turn inherits from another parent class.

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

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

class Mammal(Animal):
    def feed_young_with_milk(self):
        print(f"{self.name} is feeding its young with milk")

class Dog(Mammal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        print(f"{self.name} barks")

dog = Dog("Rover")
dog.feed_young_with_milk()

# Output: "Rover is feeding its young with milk"
```

- **Hierarchical Inheritance:**
  Hierarchical inheritance is when multiple classes inherit from the same parent class.

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

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

class Cat(Animal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        print(f"{self.name} meows")

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        print(f"{self.name} barks")

cat = Cat("Whiskers")
dog = Dog("Rover")
cat.speak()                    # Output: "Whiskers meows"
dog.speak()                    # Output: "Rover barks"
```

45. What is anagram?

An anagram is a word or phrase formed by rearranging the letters of a different word or phrase. For example, the word "listen" is an anagram of the word "silent". Anagrams can be formed by rearranging the letters of any word or phrase, as long as the resulting arrangement of letters is a valid word or phrase. Anagrams are often used in puzzles and word games, and they can also be a fun way to challenge your vocabulary and spelling skills.

46. What are palindromes?

A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward. In other words, if you reverse the order of the letters or characters in a palindrome, you get the same word, phrase, or number.

For example, "level" is a palindrome because if you read it backward, it is still "level". Similarly, the phrase "Madam, I'm Adam" is a palindrome because if you read it backward, it is still "Madam, I'm Adam". Palindromes can be formed from any combination of letters, numbers, or characters, as long as the resulting sequence reads the same forward and backward. They are often used in word games, puzzles, and other forms of entertainment, and they can also be a fun way to challenge your cognitive skills.

47. What is the Fibonacci series?

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding numbers, starting from 0 and 1.

In mathematical terms, the Fibonacci sequence is defined recursively by the following formula:
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2), for n > 1

The first ten numbers in the Fibonacci sequence are:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
The Fibonacci sequence has many interesting mathematical properties and appears in many different areas of mathematics, science, and nature. For example, it can be used to model the growth of populations, the spiral patterns in seashells and galaxies, and the relationships between musical notes.

48. What is Try,except and exception handling?

In Python, try-except blocks are used for exception handling. An exception is a runtime error that occurs during the execution of a program, and exception handling allows the program to gracefully handle these errors and continue running without crashing.
The basic syntax of a try-except block in Python is as follows:

```
try:
    # some code that might raise an exception
except ExceptionType:
    # code to handle the exception
```

In this code, the try block contains the code that might raise an exception, and the except block contains the code to handle the exception if it occurs. The ExceptionType is the type of exception that you want to catch and handle. You can replace ExceptionType with the specific type of exception that you want to catch, or you can use a more general exception type like Exception to catch any type of exception.

Here's an example of a try-except block in Python:

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
    print(y)
except ValueError:
    print("Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

In this code, the try block asks the user to enter a number, then tries to divide 10 by that number and print the result. If the user enters an invalid input (like a string instead of a number), a ValueError exception is raised and the program prints an error message. If the user enters the number 0, a ZeroDivisionError exception is raised and the program prints a different error message.

Using try-except blocks for exception handling can make your programs more robust and prevent them from crashing due to unexpected errors.

49. What is Map,filter and reduce?

In Python, map(), filter(), and reduce() are built-in functions that are used to manipulate iterables like lists, tuples, and sets. These functions can be used to perform common data transformations on sequences of data in a concise and readable way.

1.map(): The map() function applies a function to each item in an iterable and returns a new iterable with the results.

The syntax of map() function is:

```
map(function, iterable)
```

Here, function is the function that you want to apply to each item in the iterable, and iterable is the iterable (e.g. list, tuple, or set) that you want to apply the function to.

Here's an example of using map() to convert a list of strings to uppercase:

```
fruits = ['apple', 'banana', 'cherry']
upper_fruits = list(map(str.upper, fruits))
print(upper_fruits)
# Output: ['APPLE', 'BANANA', 'CHERRY']
```

2.filter(): The filter() function creates a new iterable that contains only the elements from the original iterable for which a given function returns True.

The syntax of filter() function is:

```
filter(function, iterable)
```

Here, function is the function that you want to use to filter the iterable, and iterable is the iterable that you want to filter.

Here's an example of using filter() to filter out even numbers from a list:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)

# Output: [1, 3, 5, 7, 9]
```

3.reduce(): The reduce() function applies a function of two arguments cumulatively to the items of an iterable from left to right to reduce the iterable to a single value.

The syntax of reduce() function is:

```
reduce(function, iterable)
```

Here, function is the function that you want to apply to the iterable to reduce it to a single value, and iterable is the iterable that you want to reduce.

Here's an example of using reduce() to find the product of a list of numbers:

```
from functools import reduce
```

```
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)
# Output: 120
```

In this example, the reduce() function applies the lambda function lambda x, y: x * y to the first two elements of the list (1 and 2), then applies the same function to the result and the next element (3), and so on, until the entire list has been reduced to a single value (120).

50. What is monkey patching in Python?

Ans: In Python, the term monkey patch only refers to dynamic modifications of a class or module at run-time.

Consider the below example:

```
# m.py class
MyClass:
def f(self):
print "f()"
```

We can then run the monkey-patch testing like this:

```
import m
def monkey_f(self):
print "monkey_f()"

m.MyClass.f = monkey_f
obj = m.MyClass()
obj.f()
```

The output:

monkey_f()

As we can see, we did make some changes in the behavior of f() in MyClass using the function we defined, monkey_f(), outside of the module m.

51.Does python support multiple inheritance?

Ans: Multiple inheritance means that a class can be derived from more than one parent class. Python does support multiple inheritance, unlike Java.

52. Write a program in Python to check if a number is prime.

```
 a=int(input("enter number"))
if a=1:
   for x in range(2,a):
          if(a%x)==0:
            print("not prime")
    break
    else:
         print("Prime")
else:
      print("not prime")
```

Output: enter number 3 Prime

53. Write a program in Python to check if a sequence is a Palindrome.

```
a=input("enter sequence")
b=a[::-1]
if a==b:
```

```
    print("palindrome")
else:
    print("Not a Palindrome")
```

Output: enter sequence 323 palindrome

54. Write a one-liner that will count the number of capital letters in a file.Your code should work even if the file is too big to fit in memory.

Ans: Let us first write a multiple line solution and then convert it to one-liner code.

```
with open(SOME_LARGE_FILE) as fh:
count = 0
text = fh.read()
for character in text:
      if character.isupper():
count += 1
```

We will now try to transform this into a single line.

```
count sum(1 for line in fh for character in line if character.isupper())
```

SQL

BROWSEJOBS

1.How to Extract values from Table

The SELECT statement is used to query the database and retrieve data from one or more tables.
Syntax:-SELECT column1, column2, ... FROM table_name;

In the above syntax:

- SELECT is the keyword used to select data from a table.
- column1, column2, ... are the names of the columns you want to retrieve data from. If you want to retrieve data from all columns in the table, you can use the * wildcard character.
- FROM is the keyword used to specify the table from which you want to retrieve data.
- Table_name is the name of the table from which you want to retrieve data.

2.What are Having , where and group by clause?

The WHERE clause is used to filter records based on a specified condition. It is used with the SELECT, UPDATE, and DELETE statements. The syntax for the WHERE clause is:
SELECT column1, column2, ... FROM table_name WHERE condition;

In the above syntax, condition is the condition that each record must satisfy to be included in the result set.
For example, if you have a table called "customers" with columns "customer_id", "customer_name", and "customer_country", and you want to retrieve all customers who are from the United States, you can use the following SQL query:

SELECT customer_id, customer_name FROM customers WHERE customer_country = 'United States';

The HAVING clause, on the other hand, is used to filter records that are returned by a GROUP BY clause based on a specified condition. It is used with the SELECT statement.

The syntax for the HAVING clause is:
SELECT column1, column2, ... FROM table_name GROUP BY column_name HAVING condition;

In the above syntax, column_name is the name of the column on which you want to group the data, and condition is the condition that each group must satisfy to be included in the result set.
For example, if you have a table called "orders" with columns "order_id", "customer_id", and "order_total", and you want to retrieve the total orders of customers who have placed orders more than $1000, you can use the following SQL query:
SELECT customer_id, SUM(order_total) as total_orders FROM orders GROUP BY customer_id HAVING SUM(order_total) > 1000;

This query will group the orders by customer_id and then retrieve the total orders of each customer. The HAVING clause will filter the result set to include only those customers whose total orders are greater than $1000.
The GROUP BY clause is used in SQL to group rows based on the values in one or more columns. It is used with the SELECT statement and is typically used with an aggregate function (such as SUM, AVG, COUNT, MAX, or MIN) to perform calculations on each group of rows.

SELECT column1, column2, ..., aggregate_function(column_name) FROM table_name GROUP BY column1, column2, ...;

In the above syntax, column1, column2, ... are the columns that you want to group the data by, and aggregate_function(column_name) is the aggregate function that you want to apply to the data in the specified column.
For example, if you have a table called "orders" with columns "order_id", "customer_id", "order_date", and "order_total", and you want to retrieve the total orders for each customer, you can use the following SQL query:
SELECT customer_id, SUM(order_total) as total_orders FROM orders GROUP BY customer_id;

This query will group the orders by customer_id and then retrieve the total orders for each customer. The result set will have one row for each customer, with the customer_id and the total_orders calculated using the SUM function.
Note that any columns that are included in the SELECT statement that are not in the GROUP BY clause must be aggregated using an aggregate function. This is because the GROUP BY clause groups the rows based on the values in the specified columns, so any non-aggregated columns would have multiple values per group, which is not allowed.

3.How to find the Ascending and Descending order in SQL

To sort data in ascending or descending order in SQL, you can use the ORDER BY clause in the SELECT statement.

The basic syntax of the ORDER BY clause is as follows:
SELECT column1, column2, ... FROM table_name ORDER BY column1 ASC/DESC;

In the above syntax, column1, column2, ... are the columns that you want to retrieve data from, and ASC or DESC is used to specify the sort order. ASC means ascending order (from lowest to highest), while DESC means descending order (from highest to lowest).

For example, if you have a table called "products" with columns "product_id", "product_name", and "product_price", and you want to retrieve the data sorted in ascending order by product_price, you can use the following SQL query:

```
SELECT product_id, product_name, product_price FROM products
ORDER BY product_price ASC;
```

This will retrieve the product_id, product_name, and product_price columns from the "products" table and sort the data in ascending order based on the product_price column.

If you want to sort the data in descending order, you can use the following SQL query:

```
SELECT product_id, product_name, product_price FROM products
ORDER BY product_price DESC;
```

This will retrieve the same columns from the "products" table but sort the data in descending order based on the product_price column.

4.SQl Query to find Second Highest salary from 3 tables.

Assuming you have three tables named employee, manager, and ceo with columns name and salary, you can get the second-highest salary from all three tables using the following SQL query:

```
SELECT MAX(salary) as second_highest_salary FROM (
        SELECT salary FROM employee
        UNION ALL
        SELECT salary FROM manager
        UNION ALL
        SELECT salary FROM ceo
) AS all_salaries
WHERE salary < (
        SELECT MAX(salary) FROM (
        SELECT salary FROM employee
        UNION ALL
        SELECT salary FROM manager
```

```
        UNION ALL
        SELECT salary FROM ceo
        ) AS all_salaries_inner
)
```

This query uses a subquery to first retrieve all salaries from the three tables using the UNION ALL operator. It then filters out the maximum salary from the combined list of salaries by using a WHERE clause with a subquery to retrieve the maximum salary from the same list. Finally, the MAX function is used to retrieve the second highest salary from the filtered list of salaries.

5.Mention Types of joins with Example

1. **INNER JOIN:** The INNER JOIN returns only the rows that have matching values in both tables.

```
SELECT *
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

For example, if you have a table of customers and a table of orders, and you want to retrieve all orders along with the customer information for each order, you can use the following INNER JOIN query:

```
SELECT *
FROM orders
INNER JOIN customers
ON orders.customer_id = customers.customer_id;
```

2. **LEFT JOIN:** The LEFT JOIN returns all rows from the left table and the matching rows from the right table. If there is no match in the right table, NULL values are returned.

```
SELECT *
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

For example, if you have a table of employees and a table of departments, and you want to retrieve all employees along with their department information, you can use the following LEFT JOIN query:

```
SELECT *
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

3. **RIGHT JOIN:** The RIGHT JOIN returns all rows from the right table and the matching rows from the left table. If there is no match in the left table, NULL values are returned.

```
SELECT *
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

For example, if you have a table of departments and a table of employees, and you want to retrieve all departments along with their employee information, you can use the following RIGHT JOIN query:

```
SELECT *
FROM departments
RIGHT JOIN employees
ON departments.department_id = employees.department_id;
```

4. **FULL OUTER JOIN:** The FULL OUTER JOIN returns all rows from both tables, including the unmatched rows. If there is no match in one of the tables, NULL values are returned for the missing values.

```
SELECT *
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

For example, if you have a table of customers and a table of orders, and you want to retrieve all customers and all orders, including those with no matching customer or order, you can use the following FULL OUTER JOIN query:

```
SELECT *
FROM customers
```

```
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

5.Can Self Join be used to join with second table

No, a self-join cannot be used to join with a second table.
A self-join is used when you need to join a table to itself, typically to compare data within the same table. In a self-join, you treat the table as if it were two separate tables by giving each instance of the table a different alias.
For example, if you have a table of employees with columns "employee_id", "first_name", "last_name", and "manager_id" where the manager_id refers to another employee in the same table, you can use a self-join to retrieve the manager name for each employee:

```
SELECT e1.employee_id, e1.first_name, e1.last_name, e2.first_name AS manager_first_name, e2.last_name AS manager_last_name
FROM employees e1
INNER JOIN employees e2
ON e1.manager_id = e2.employee_id;
```

In the above example, the "employees" table is joined to itself using the "manager_id" column to match each employee with their corresponding manager. The alias "e1" is used for the first instance of the "employees" table, while the alias "e2" is used for the second instance of the table.
However, if you need to join a table to another table, you cannot use a self-join. In that case, you would need to use a regular join or one of the other types of joins available in SQL.

7.what is primary key and Foreign Key

A primary key is a column or set of columns in a database table that uniquely identifies each row in that table. It is used to ensure that each record in the table can be identified and accessed without duplication, and it

is always unique and not null. A primary key can be composed of a single column or multiple columns.
A foreign key, on the other hand, is a column or set of columns in one table that refers to the primary key of another table. It is used to establish a link or relationship between two tables, and it enforces referential integrity between them. The foreign key column in one table is used to identify the corresponding row(s) in the other table.
For example, consider two tables: "Orders" and "Customers". The "Orders" table has a foreign key column "customer_id" that refers to the primary key column "customer_id" in the "Customers" table. This ensures that each order in the "Orders" table is associated with a valid customer in the "Customers" table. If a customer is deleted from the "Customers" table, the related orders in the "Orders" table can either be deleted or set to a default value, depending on the actions specified in the foreign key constraint.

8.What are SQL Views?

SQL views are virtual tables that are created based on an SQL query. They are not physical tables, but rather a virtual representation of a subset of data from one or more tables in a database. A view is defined by a SELECT statement that retrieves data from one or more tables and defines the columns and rows that make up the view.
Views can be used to simplify complex queries, provide a simplified interface to a complex database schema, or provide a simplified interface to data for users who don't have direct access to the underlying tables. Views can also be used to restrict access to sensitive data by only exposing a subset of the data to users.
Once a view is created, it can be treated like a table in subsequent queries, with the added advantage of being able to hide the underlying complexity of the database schema. Views can be queried, updated, and deleted in the same way as tables, and they can also be joined with other tables and views to produce more complex queries.
Views are created using the CREATE VIEW statement, which defines the SQL query that will be used to create the view. Views can also be modified

or dropped using the ALTER VIEW or DROP VIEW statements, respectively.

9.Types of Normalization?

Normalization is the process of organizing data in a database in such a way as to reduce redundancy and ensure data integrity. There are several normal forms that a database can be designed to meet, with each subsequent normal form building on the previous one. The most common normal forms are:

1. **First Normal Form (1NF):** This requires that all tables have a primary key and that each column in the table contains atomic (indivisible) values. In other words, each field in the table should contain a single value, and not a list of values.
2. **Second Normal Form (2NF):** This requires that all non-key columns in a table be functionally dependent on the primary key. In other words, each non-key column in the table should be directly related to the primary key, and not to any other non-key column.
3. **Third Normal Form (3NF):** This requires that all non-key columns in a table be independent of each other. In other words, no non-key column should be dependent on any other non-key column.
4. **Fourth Normal Form (4NF):** This requires that all multi-valued dependencies be removed from the database. This means that no table should have multiple independent many-to-many relationships.
5. **Fifth Normal Form (5NF):** This requires that all join dependencies be removed from the database. This means that all relationships between tables should be based on primary key/foreign key relationships.

It's worth noting that not all databases need to be designed to meet all of these normal forms. The level of normalization needed will depend on the specific requirements of the database and the applications that will be using it.

10.what is Indexing,why do we go for Indexing?

Indexing is a technique used in databases to speed up data retrieval operations, particularly when querying large data sets. An index is essentially a data structure that allows for quick lookups of data based on specific columns in a table. When a table is indexed, the database creates a separate structure that stores the indexed column(s) along with a pointer to the corresponding data rows.

Indexes are used to speed up search operations and queries by allowing the database to quickly locate the relevant data. Without an index, the database would have to search through every row in a table to find the desired data, which can be very slow and resource-intensive. With an index, the database can quickly narrow down the search to only the relevant rows.

Indexes can be created on one or more columns in a table, and can be created as unique or non-unique. Unique indexes ensure that no two rows in a table have the same values for the indexed column(s), while non-unique indexes allow for duplicates.

We go for indexing for the following reasons:

1. Faster data retrieval: Indexing speeds up data retrieval operations by allowing the database to quickly locate the relevant data without searching through every row in a table.
2. Improved query performance: Indexing can significantly improve query performance, particularly for complex queries or queries that involve large data sets.
3. Reduced disk I/O: Indexing can reduce the amount of disk I/O required for data retrieval operations, which can help improve overall database performance.
4. Better scalability: Indexing can help improve the scalability of a database by allowing it to handle larger data sets more efficiently.
5. Enforced data integrity: Unique indexes can be used to enforce data integrity by ensuring that no two rows in a table have the same values for the indexed column(s).

11.Can we do Indexing for Multiple Columns?

Multicolumn indexes can: be created on up to 32 columns. be used for partial indexing.

12.Advantage and Disadvantages of using Multiple Indexing

Advantages:

1. **Improved Query Performance:** By using multiple indexing, you can significantly improve the performance of your queries. When you have multiple indexes, the database can quickly narrow down the search to a smaller subset of data, reducing the time it takes to find the required information.
2. **Increased Flexibility:** With multiple indexing, you can retrieve data using a combination of different criteria. This makes it easier to find the exact information you need, even if it's spread out across multiple columns.
3. **Reduced Data Duplication:** By using multiple indexing, you can avoid duplicating data in your database. Instead of storing the same data in multiple tables, you can store it in one table and create multiple indexes to access it more efficiently.

Disadvantages:

1. **Increased Storage Requirements:** Multiple indexing requires additional storage space, which can be a significant disadvantage if you are working with a large dataset.
2. **Increased Complexity:** As you add more indexes to your database, the complexity of the system increases. This can make it more difficult to maintain and troubleshoot, especially if you have a large number of indexes.
3. **Slower Write Performance:** Multiple indexing can also slow down the performance of writes, as the database needs to update multiple indexes every time a change is made to the data.
4.

Overall, the decision to use multiple indexing will depend on your specific requirements and the size of your dataset. While it can provide significant performance benefits, it can also add complexity and storage requirements to your system.

13.What is Transaction in SQL

In SQL, a transaction is a group of one or more database operations that are performed as a single, atomic unit of work. Transactions are used to ensure the integrity of the database and to guarantee that data is consistent and accurate. Transactions are typically used when multiple database operations need to be performed together, and if any of the operations fail, the entire transaction should be rolled back or canceled.
A transaction in SQL consists of the following four properties, which are collectively known as the ACID properties:

1. **Atomicity:** This property ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction succeed, or they are all rolled back to their original state.
2. **Consistency:** This property ensures that the database remains in a consistent state before and after the transaction. If a transaction violates any constraints, the transaction is rolled back to its original state.
3. **Isolation:** This property ensures that each transaction is isolated from other transactions, so that transactions do not interfere with each other. Each transaction should be executed independently, without affecting the outcome of other transactions.
4. **Durability:** This property ensures that once a transaction has been committed, it will be permanently stored in the database, even in the event of a power failure, system crash, or other unexpected event.

14.what is Pass by Value and Pass by Reference

Pass by value and pass by reference are two different ways that programming languages use to pass arguments to a function or method.
*Pass by value means that when an argument is passed to a function, a copy of the argument is created and passed to the function. Any changes made to the argument within the function are made to the copy, not the original value. This means that the original value remains unchanged.

*Pass by reference means that when an argument is passed to a function, a reference to the original value is passed instead of a copy. Any changes made to the argument within the function are reflected in the original value. In pass by reference, a reference to the original value is passed to the function, and any changes made to the argument are reflected in the original value.
In summary, pass by value creates a copy of the argument that is passed to the function, while pass by reference passes a reference to the original value.

Django

1.What is MVT Architecture?

MVT (Model-View-Template) is a design pattern used in the Django web framework to structure web applications. MVT is similar to the more well-known Model-View-Controller (MVC) pattern, but with some differences in implementation. In MVT, the three main components are:

1. **Model:** This component defines the data structure of the application and provides an interface for interacting with the database. Models are typically defined as Python classes that inherit from Django's built-in Model class.
2. **View:** This component handles the business logic of the application and determines how data is presented to the user. Views are typically defined as Python functions or classes that take input from a user and return a response.
3. **Template:** This component defines how the data is presented to the user. Templates are typically written in HTML and use Django's templating language to insert dynamic content.

In the MVT architecture, the flow of data is as follows: the user interacts with the interface, which sends a request to the view. The view retrieves data from the model and performs any necessary business logic. The view then sends the data to the template, which renders it as HTML and sends it back to the user.
Django provides a built-in set of tools and conventions for implementing the MVT pattern. For example, Django's ORM (Object-Relational Mapping) maps database tables to model classes, allowing developers to interact with the database using Python code. Django's view functions and classes provide a way to process incoming requests and return responses. And Django's templating language provides a way to render dynamic content in HTML. Overall, the MVT architecture in Django provides a clear and organized way to structure web applications, separating concerns into distinct components and allowing developers to focus on specific areas of functionality.

2.What is Middleware

Middleware is a software component that acts as a bridge between different systems or applications. In the context of web development, middleware is often used to handle common tasks that are required for most web applications, such as authentication, logging, and request/response handling. In the context of web development, middleware is software that sits between the web server and the web application, intercepting requests and responses and performing some action or modification before passing them on to the next stage of processing. Middleware can be used to implement a wide range of functionality, such as:

- **Authentication and authorization:** Middleware can check whether a user is authorized to access a particular resource or perform a particular action.
- **Logging and debugging:** Middleware can log requests and responses for debugging and monitoring purposes.
- **Compression and caching:** Middleware can compress responses and cache data to improve performance.

- **Error handling:** Middleware can handle errors and exceptions in a consistent way across the application.

Middleware is often used in frameworks and libraries, such as Django and Express.js, to provide a set of common functionality that can be reused across different applications. Middleware can be stacked together to create a pipeline of processing that is executed in a specific order. This pipeline is executed for each incoming request, allowing developers to implement complex processing logic in a modular and reusable way.

3.Explain Request life cycle.

While setting up the django application on the server we need a webserver and a wsgi server. Webserver helps us in serving static files/content. If we did not use the webserver to serve static files then it has to be served by WSGI server which results in more requests to the server. So, gradually it slows down the application performance. Web server balances the requests load on the server.

We can divide request resources in two types.

1. static resource
2. dynamic resource(It has to process the data based on request to provide resource)

If browser request for the static resources like images/css/javascript/etc, then NGINX serves the request without sending it to the uWSGI server.If browser requests for a dynamic request then NGINX passes it to the uWSGI to process the request.At this stage NGINX acts like a reverse proxy server. A reverse proxy server is a type of proxy server that typically sits behind the firewall in a private network and directs browser/client requests to the appropriate back-end server(uWSGI).Advantages of reverse proxy server are Load balancing, Web acceleration, Security and anonymity.

WSGI is a tool created to solve a basic problem: connecting a web server to a web framework. WSGI has two sides: the ‘server’ side and the ‘application’ side. To handle a WSGI response, the server executes the application and provides a callback function to the application side. The application processes the request and returns the response to the server using the provided callback. Essentially, the WSGI handler acts as the gatekeeper between your web server (Apache, NGINX, etc) and your Django project. While deploying the django application on the server “Nginx, gunicorn” combination widely used.When a client sends a request to the server it first passed to the web server(NGINX), It contains the configuration rules to dispatch the request to the WSGI(gunicorn) server or served by itself(serving static files). WSGI server passes the request to the Django application. Django has the following layers in dealing with request-response lifecycle of a Django application.

Layers of Django Application

1. Request Middlewares
2. URL Router(URL Dispatcher)
3. Views
4. Context Processors
5. Template Renderers
6. Response Middlewares

4.Different types of middlewares,Can we create a custom middleware?

There are various types of middleware that can be used in different contexts, including:

1. **Web middleware:** This type of middleware is used in web applications and is responsible for handling HTTP requests and responses. Examples of web middleware include authentication middleware, authorization middleware, session middleware, and error handling middleware.
2. **Network middleware:** This type of middleware is used in networked applications and is responsible for managing network connections, routing, and message passing. Examples of network middleware include message queues, load balancers, and firewalls.
3. **Database middleware:** This type of middleware is used to abstract and simplify interactions with databases. Examples of database middleware include ORMs (object-relational mappers) and database connection pools.
4. **Application middleware:** This type of middleware is used to provide additional functionality to an application, such as logging, caching, and performance monitoring.

Yes, we can create custom middleware as per the requirements of the application. Custom middleware can be developed using various programming languages and frameworks. The process of creating custom middleware typically involves writing code that intercepts requests and responses and performs the desired processing or modification. Custom middleware can be used to add additional functionality to an application or to customize the behavior of existing middleware

5.How do we build a simple API?

Django REST Framework is a wrapper over default Django Framework, basically used to create APIs of various kinds. There are three stages before creating an API through REST framework, Converting a Model's data to JSON/XML format (Serialization), Rendering this data to the view, Creating a URL for mapping to the viewset.

Steps

- Add rest_framework to INSTALLED_APPS
- Create a app and model
- Serialization
- Creating a viewset
- Define URLs of API
- Run server and check API

Add rest_framework to INSTALLED_APPS

To initialize REST Framework in your project, go to settings.py, and in INSTALLED_APPS add ‘rest_framework’ at the bottom.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
]
```

Create a app and model

Now, let's create a app using command, python manage.py startapp apis

A folder with name apis would have been registered by now. let's add this app to INSTALLED_APPS and urls.py also.

In, settings.py,

```
INSTALLED_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest_framework',

'apis',

]
```

```
from django.contrib import admin

# include necessary libraries

from django.urls import path, include
```

```
urlpatterns = [
        path('admin/', admin.site.urls),
        # add apis urls
        path('', include("apis.urls"))
]
```

Create a model

To demonstrate creating and using an API, let's create a model named "Model". In apis/models.py

```
from django.db import models


class Model(models.Model):
        title = models.CharField(max_length = 200)
        description = models.TextField()


        def __str__(self):
        return self.title
```

Serialization

Serializers allow complex data such as querysets and model instances to be converted to native Python data types that can then be easily rendered into JSON, XML or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data. Let's start creating a serializer, in file apis/serializers.py,

```
# import serializer from rest_framework

from rest_framework import serializers


# import model from models.py
from .models import Model


# Create a model serializer
class Serializer(serializers.HyperlinkedModelSerializer):
        # specify model and fields
        class Meta:
        model = Model
        fields = ('title', 'description')
```

Creating a viewset

To render data into the frontend, and handle requests from users, we need to create a view. In Django REST Framework, we call these as viewsets, so let's create a view in apis/views.py,

```
# import viewsets

from rest_framework import viewsets


# import local data

from .serializers import Serializer

from .models import Model


# create a viewset

class ViewSet(viewsets.ModelViewSet):

    # define queryset

    queryset = Model.objects.all()


    # specify serializer to be used

    serializer_class = Serializer
```

Define URLs of API

Specify the url path of APIs to be accessed, In apis/urls.py,

```
# basic URL Configurations
```

```
from django.urls import include, path

# import routers

from rest_framework import routers


# import everything from views

from .views import *


# define the router

router = routers.DefaultRouter()


# define the router path and viewset to be used

router.register(r'browsejobs', ViewSet)

# specify URL Path for rest_framework

urlpatterns = [

        path('', include(router.urls)),

        path('api-auth/', include('rest_framework.urls'))

]
```

After everything is successfully ready, let's run some commands to activate the server.

Run server and check API

Run following commands to create the database, and run server,

python manage.py makemigrations

python manage.py migrate

python manage.py runserver

6.Difference between Flask and Django

Django	Flask
Django could be a Python-based free, open source system which takes after the MVT(model view Template) approach of structural design	**Flask could be a Python-based smaller scale system without any set of specific instruments or outside libraries. It too doesn't have a database layer or arrangements for shape approval and makes utilize of expansions.**
Urls.py is utilized to set the association properties and demands are handled by the	**URI is most regularly than not set by the see decorator and centralized setup is additionally conceivable. Sometimes the recent designs are**

primary coordinating see of regex list	**coordinated with the URIs, the last mentioned is sorted in a default arrange**
Doesn’t exclude setting flexibility	**It is accepted that all the conceivable stages to organize a Flask code rises to the applications number show in flask as of now**
Extend Layout is Conventional extend structure	**Extend Layout is Arbitrary structure**
Django gives an all-inclusive encounter: you get an admin board, database interfacing, an ORM, and registry structure for your apps and ventures out of the box.	**Flask gives straightforwardness, adaptability and fine-grained control. It is un-opinionated**
It is suitable for multi page applications.	**It is suitable for single page applications only.**

Its framework structure is more conventional.	**Random web framework structure.**
It doesn't support any virtual debugging.	**It has a built-in debugger that provides virtual debugging.**
Its working style is Monolithic.	**Its working style is diversified style.**

7.Modules in Django?

1. **Models:** This module provides an Object-Relational Mapping (ORM) system that makes it easy to work with databases. It allows developers to define database models as Python classes, and provides a set of tools for querying and manipulating data.
2. **Views:** This module provides the logic for handling HTTP requests and generating HTTP responses. Views are Python functions that take a request object as input and return a response object.
3. **Templates:** This module provides a system for generating HTML templates that can be used to render the output of views. Django's

template engine supports a wide range of features, including template inheritance, template tags, and filters.

4. **Forms:** This module provides a set of tools for working with HTML forms, including form validation, form rendering, and form processing.
5. **Admin:** This module provides a built-in administration interface that allows developers to manage the content of their website using a web-based interface.
6. **Middleware:** This module provides a way to intercept and modify HTTP requests and responses at various stages of the request/response cycle. Middleware can be used for a variety of purposes, such as authentication, caching, and logging.
7. **URL routing:** This module provides a way to map URLs to views using a flexible and powerful routing system. URLs can be matched based on regular expressions, and can include named parameters.

These are some of the core modules of Django, but the framework also provides many additional modules that can be used to add functionality to a web application, such as authentication, caching, and internationalization.

8.Templates in Django?

Templates are the third and most important part of Django's MVT Structure. A template in Django is basically written in HTML, CSS, and Javascript in a .html file. Django framework efficiently handles and generates dynamically HTML web pages that are visible to the end-user. Django mainly functions with a backend so, in order to provide a frontend and provide a layout to our website, we use templates. There are two methods of adding the template to our website depending on our needs.

We can use a single template directory which will be spread over the entire project.

For each app of our project, we can create a different template directory.

For our current project, we will create a single template directory that will be spread over the entire project for simplicity. App-level templates are generally used in big projects or in case we want to provide a different layout to each component of our webpage.

Configuration

Django Templates can be configured in app_name/settings.py,

```
TEMPLATES = [
    {
    # Template backend to be used, For example Jinja
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    # Directories for templates
    'DIRS': [],
    'APP_DIRS': True,

    # options to configure
    'OPTIONS': {
    'context_processors': [
```

```
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Using Django Templates

Illustration of How to use templates in Django using an Example Project. Templates not only show static data but also the data from different databases connected to the application through a context dictionary. Consider a project named Browsejobs having an app named Browse.

To render a template one needs a view and a URL mapped to that view. Let's begin by creating a view in Browse/views.py,

```
# import Http Response from django
from django.shortcuts import render


# create a function
def browse_view(request):
    # create a dictionary to pass
```

```
        # data to the template
        context ={
        "data":"Browse is the best",
        "list":[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        }
        # return response with template and context
        return render(request, "browse.html", context)
from django.urls import path

# importing views from views..py
from .views import browse_view

urlpatterns = [
        path('', browse_view),
]

<!DOCTYPE html>

<html lang="en">
<head>
```

```
        <meta charset="UTF-8">

        <meta name="viewport" content="width=device-width,
initial-scale=1.0">

        <meta http-equiv="X-UA-Compatible" content="ie=edge">

        <title>Homepage</title>


</head>

<body>

        <h1>Welcome to Browsejobs.</h1>

<p> Data  is {{  data }}</p>

        <h4>List is </h4>

        <ul>

        {% for i in list %}

        <li>{{ i }}</li>

        <li>{{ i }}</li>

</body>

</html>
```

The Django template language

This is one of the most important facilities provided by Django Templates. A Django template is a text document or a Python string marked-up using the Django template language. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags. As we used for the loop in the above example, we used it as a tag. similarly, we can use various other conditions such as if, else, if-else, empty, etc. The main characteristics of Django Template language are Variables, Tags, Filters, and Comments.

Variables

Variables output a value from the context, which is a dict-like object mapping keys to values. The context object we sent from the view can be accessed in the template using variables of Django Template

Syntax

{{ variable_name }}

Example

Variables are surrounded by {{ and }} like this:

My first name is {{ first_name }}. My last name is {{ last_name }}.

With a context of {‘first_name’: ‘Naveen’, ‘last_name’: ‘Arora’}, this template renders to:

My first name is Naveen. My last name is Arora.

Tags

Tags provide arbitrary logic in the rendering process. For example, a tag can output content, serve as a control structure e.g. an “if” statement or a “for” loop, grab content from a database, or even enable access to other template tags.

Syntax

{% tag_name %}

Example

Tags are surrounded by {% and %} like this:

{% csrf_token %}

Most tags accept arguments, for example :

{% cycle 'odd' 'even' %}

Filters

Django Template Engine provides filters that are used to transform the values of variables and tag arguments. We have already discussed major Django Template Tags. Tags can't modify the value of a variable whereas filters can be used for incrementing the value of a variable or modifying it to one's own need.

Syntax

{{ variable_name | filter_name }}

Filters can be "chained." The output of one filter is applied to the next. {{ text|escape|linebreaks }} is a common idiom for escaping text contents, then converting line breaks to <p> tags.

Example

{{ value | length }}

If value is ['a', 'b', 'c', 'd'], the output will be 4.

Comments

Template ignores everything between {% comment %} and {% end comment %}. An optional note may be inserted in the first tag. For example, this is useful when commenting out code for documenting why the code was disabled.

Syntax

{% comment 'comment_name' %}

{% endcomment %}

Example :

{% comment "Optional note" %}

Commented out text with {{ create_date|date:"c" }}

{% endcomment %}

Template Inheritance

The most powerful and thus the most complex part of Django's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines blocks that child templates can override. extends tag is used for

the inheritance of templates in Django. One needs to repeat the same code again and again. Using extends we can inherit templates as well as variables.

Syntax

{% extends 'template_name.html' %}

Example :

assume the following directory structure:

dir1/

template.html

base2.html

my/

base3.html

base1.html

In template.html, the following paths would be valid:

{% extends "./base2.html" %}

{% extends "../base1.html" %}

{% extends "./my/base3.html" %}

9.Django Admin use

Django's admin interface provides a web-based graphical user interface (GUI) for managing your application's data. It is a powerful tool that allows you to perform CRUD (Create, Read, Update, Delete) operations on your models without writing any code.

To use Django's admin interface, you need to follow these steps:

1. **Create a** Django project and an app: If you haven't already, create a new Django project using the django-admin startproject command. Then, create a new app using the python manage.py startapp command.
2. Define models: Define your models in the models.py file of your app.
3. Register models: Register your models with the admin interface by creating an admin.py file in your app and importing your models.
4. Create a superuser: Create a superuser account using the python manage.py createsuperuser command. This will allow you to access the admin interface.
5. Access the admin interface: To access the admin interface, go to http://localhost:8000/admin in your web browser and enter your superuser credentials.
6. Use the admin interface: Once you are logged in, you can use the admin interface to manage your data. You can add new objects, edit existing ones, delete them, and view their details. You can also perform bulk operations on multiple objects at once.

Overall, the Django admin interface is a powerful tool that can save you a lot of time and effort when managing your application's data.

10.Authentication and authorization in Django

Authentication is the process of verifying the identity of a user or system. It is typically done by requiring users to enter a username and password, which are then checked against a database of known user credentials. Once the user's identity is confirmed, they are granted access to the application and its resources.

Authorization, on the other hand, is the process of determining what actions a user is allowed to perform once they have been authenticated. This is typically done by assigning roles or permissions to users based on their

identity and the level of access they need. For example, an administrator might have full access to all resources, while a regular user might only be able to view certain pages or perform certain actions.

In Django, authentication and authorization are implemented through the built-in auth app. The auth app provides a user model for managing user accounts, and a set of views and templates for handling user authentication and authorization. To use these features, you will need to configure your application to use the appropriate authentication and authorization backends, and then use the @login_required decorator to protect views that require authentication.

Overall, authentication and authorization are important concepts in web application security that should be carefully implemented to ensure that only authorized users can access sensitive resources and perform certain actions.

11.what are Query sets?

A QuerySet is a collection of data from a database.

A QuerySet is built up as a list of objects.

QuerySets makes it easier to get the data you actually need, by allowing you to filter and order the data at an early stage.

Member:

id	firstname	lastname	phone	joined_date

BROWSEJOBS

1	Emil	Refsnes	5551234	2022-01-05
2	Tobias	Refsnes	5557777	2022-04-01
3	Linus	Refsnes	5554321	2021-12-24
4	Lene	Refsnes	5551234	2021-05-01
5	Stalikken	Refsnes	5559876	2022-09-29

Querying Data

In views.py, we have a view for testing called testing where we will test different queries.

In the example below we use the .all() method to get all the records and fields of the Member model:

View

views.py:

```
from django.http import HttpResponse

from django.template import loader

from .models import Member:

def testing(request):

  mydata = Member.objects.all()

  template = loader.get_template('template.html')

  context = {

      'mymembers': mydata,
```

```
}

return HttpResponse(template.render(context, request))
```

```
<QuerySet [
 <Member: Member object (1)>,
 <Member: Member object (2)>,
 <Member: Member object (3)>,
 <Member: Member object (4)>,
 <Member: Member object (5)>
]>
```

Template

```
<table border='1'>

 <tr>

      <th>ID</th>

      <th>Firstname</th>

      <th>Lastname</th>

 </tr>

 {% for x in mymembers %}

      <tr>

      <td>{{ x.id }}</td>
```

```
        <td>{{ x.firstname }}</td>
        <td>{{ x.lastname }}</td>
        </tr>
    {% endfor %}
</table>
```

Get Data

There are different methods to get data from a model into a QuerySet.

The values() Method

The values() method allows you to return each object as a Python dictionary, with the names and values as key/value pairs:

views.py:

```
from django.http import HttpResponse
from django.template import loader
from .models import Member
def testing(request):
```

```
    mydata = Member.objects.all().values()

    template = loader.get_template('template.html')

    context = {

        'mymembers': mydata,

    }

  return HttpResponse(template.render(context, request)
```

Return Specific Columns

The values_list() method allows you to return only the columns that you specify.

views.py:

```
    from django.http import HttpResponse

    from django.template import loader

    from .models import Member

    def testing(request):

      mydata = Member.objects.values_list('firstname')

      template = loader.get_template('template.html')
```

```
  context = {
      'mymembers': mydata,
  }
  return HttpResponse(template.render(context, request))
```

Return Specific Rows

You can filter the search to only return specific rows/records, by using the filter() method.

views.py:

```
from django.http import HttpResponse
from django.template import loader
from .models import Member

def testing(request):
  mydata = Member.objects.filter(firstname='Emil').values()
  template = loader.get_template('template.html')
  context = {
```

```
    'mymembers': mydata,
}
return HttpResponse(template.render(context, request))
```

QuerySet Filter

The filter() method is used to filter your search, and allows you to return only the rows that match the search term.

As we learned in the previous chapter, we can filter on field names like this:

Example

Return only the records where the firstname is 'Emil':

```
mydata = Member.objects.filter(firstname='Emil').values()
```

In SQL, the above statement would be written like this:

```
SELECT * FROM members WHERE firstname = 'Emil';
```

AND

The filter() method takes the arguments as **kwargs (keyword arguments), so you can filter on more than one field by separating them by a comma.

Return records where lastname is "Refsnes" and id is 2:

```
mydata = Member.objects.filter(lastname='Refsnes',
id=2).values()
```

OR

To return records where firstname is Emil or firstname is Tobias (meaning: returning records that matches either query, not necessarily both) is not as easy as the AND example above.

We can use multiple filter() methods, separated by a pipe | character. The results will merge into one model.

Example
Return records where firstname is either "Emil" or Tobias":

```
mydata = Member.objects.filter(firstname='Emil').values() |
Member.objects.filter(firstname='Tobias').values
```

Return records where firstname is either "Emil" or Tobias":

```
from django.http import HttpResponse

from django.template import loader

from .models import Member

from django.db.models import Q

def testing(request):
```

```
    mydata = Member.objects.filter(Q(firstname='Emil') |
(firstname='Tobias')).values()

    template = loader.get_template('template.html')

    context = {

        'mymembers': mydata,

    }

  return HttpResponse(template.render(context, request))
```

In SQL, the above statement would be written like this:

```
SELECT * FROM members WHERE firstname = 'Emil' OR firstname
= 'Tobias';
```

Field Lookups

Django has its own way of specifying SQL statements and WHERE clauses.

To make specific where clauses in Django, use "Field lookups".

Field lookups are keywords that represents specific SQL keyword:

Example:

Use the __startswith keyword:

```
.filter(firstname__startswith='L');
```

Is the same as the SQL statement:

WHERE firstname LIKE 'L%'

Field Lookups Syntax

All Field lookup keywords must be specified with the fieldname, followed by two(!) underscore characters, and the keyword.

In our Member model, the statement would be written like this:

Example

Return the records where firstname starts with the letter 'L':

```
mydata = Member.objects.filter(firstname__startswith='L').values()
```

Order By

To sort QuerySets, Django uses the order_by() method:

Example

Order the result alphabetically by firstname:

```
mydata = Member.objects.all().order_by('firstname').values()
```

In SQL, the above statement would be written like this:

```
SELECT * FROM members ORDER BY firstname;
```

Descending Order

By default, the result is sorted ascending (the lowest value first), to change the direction to descending (the highest value first), use the minus sign (NOT), - in front of the field name:

Example

Order the result firstname descending:

Multiple Order Bys

To order by more than one field, separate the fieldnames with a comma in the order_by() method:

Example

Order the result first by lastname ascending, then descending on id:

```
mydata = Member.objects.all().order_by('lastname', '-id').values()
```

In SQL, the above statement would be written like this:

```
SELECT * FROM members ORDER BY lastname ASC, id DESC;
```

12.How to extract all names starting with "a" in Django model

To extract all names starting with "a" from a Django model, you can use the startswith lookup with the filter method. Here's an example assuming you have a Person model with a name field:

```
from myapp.models import Person
names_starting_with_a = Person.objects.filter(name__startswith='a')
```

This will return a QuerySet containing all the Person objects whose name starts with "a". You can then use this QuerySet to perform further operations, such as retrieving specific fields or performing aggregations.

13.How to extract name "krish Bhargav",only krish but not ending with Bhargav with filters

To extract the name "krish Bhargav" but not ending with "Bhargav" in Django model, you can use the startswith and endswith lookups together with the exclude method. Here's an example assuming you have a Person model with a name field:

```
from myapp.models import Person
krish_without_bhargav =
Person.objects.filter(name__startswith='krish').exclude(name__endswith='B
hargav')
```

This will return a QuerySet containing all the Person objects whose name starts with "krish" but does not end with "Bhargav". You can then use this QuerySet to perform further operations, such as retrieving specific fields or performing aggregations.

14.What is Queue in Django?

Django doesn't provide a built-in Queue implementation, but you can use Python's built-in queue module to implement a queue in Django.
Here's an example of how to use the queue module to implement a simple queue in Django:

```
import queue

# Initialize a queue

my_queue = queue.Queue()

# Add items to the queue

my_queue.put("item1")

my_queue.put("item2")

my_queue.put("item3")

# Get an item from the queue

item = my_queue.get()

# Check if the queue is empty

if my_queue.empty():

        print("The queue is empty")
```

You can use this same implementation in your Django project. For example, you might create a model to represent the items in your queue, and then use the queue module to manage the queue operations within your Django views or background tasks. Alternatively, you might use a third-party package like Celery to manage your task queues in Django**.**

15.What are Static files?

In Django, static files are files that are served directly to the client's web browser, such as images, CSS stylesheets, JavaScript files, and other assets that do not change dynamically.
Static files are separated from the dynamic parts of the web application and are typically located in a dedicated directory in the project's file structure. By default, Django looks for static files in a directory called static in each installed app, as well as in any directories specified in the STATICFILES_DIRS setting.
When a static file is requested by the client, Django will attempt to serve the file directly from the file system, without going through the Python code. To make this work, Django provides a view called django.contrib.staticfiles.views.serve that serves static files.
When a static file is requested by the client, Django will attempt to serve the file directly from the file system, without going through the Python code. To make this work, Django provides a view called django.contrib.staticfiles.views.serve that serves static files.

16.Where do we deploy django project?

Django is a versatile web framework and can be deployed on a variety of hosting platforms. Here are some options for deploying a Django project:

1. **Virtual Private Server (VPS):** A VPS is a virtual machine that runs its own operating system and allows you to install and run software as if you were using a physical server. You can choose a VPS provider and install the necessary software to run a Django project yourself.
2. **Platform as a Service (PaaS):** PaaS providers, such as Heroku, Google App Engine, or PythonAnywhere, provide a platform for deploying web applications without requiring you to manage the underlying infrastructure. They often provide built-in features such as automatic scaling, deployment pipelines, and database integration.
3. **Dedicated Server:** A dedicated server is a physical server that is rented or owned and managed entirely by the user. This option gives

you complete control over the server, but also requires more technical expertise to set up and maintain.

4. **Containerization:** Containerization platforms like Docker allow you to package your application and its dependencies into a container image, which can be run on any platform that supports Docker. This makes it easy to deploy your application to any platform that supports Docker.

Ultimately, the choice of where to deploy your Django project will depend on your technical expertise, budget, and specific needs for your project. It's important to choose a platform that can support your application's scalability, security, and performance requirements.

17.How do we migrate changes in Django

To migrate changes in Django, follow these steps:

1. **Make changes to your models:** If you want to change the structure of your database, you will need to modify your models in the models.py file.
2. **Create a new migration:** After making changes to your models, create a new migration file using the makemigrations management command. This command will create a new migration file that includes the changes you made to your models.

python manage.py makemigrations

3. **Apply the migration:** After creating a new migration file, apply the migration to your database using the migrate management command. This command will execute the changes defined in the migration file on your database.

python manage.py migrate

18.How do we build relations in modules of multiple classes?

In Django, building relations between modules of multiple classes can be achieved through the use of Django's built-in ORM (Object-Relational

Mapping) system. The ORM allows you to define relationships between different classes by creating fields on one class that reference another class. There are several types of relationships that can be defined in Django, including:

1. **One-to-one relationship:** This relationship is used when each instance of one class is related to only one instance of another class. To create a one-to-one relationship, you can define a OneToOneField on one of the classes, referencing the other class.
2. **One-to-many relationship:** This relationship is used when each instance of one class is related to multiple instances of another class. To create a one-to-many relationship, you can define a ForeignKey field on the "many" class, referencing the "one" class.
3. **Many-to-many relationship:** This relationship is used when each instance of one class can be related to multiple instances of another class, and vice versa. To create a many-to-many relationship, you can define a ManyToManyField on one of the classes, referencing the other class.

Here's an example of how to define a one-to-many relationship between two classes in Django:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
```

In this example, the Book class has a foreign key field called author that references the Author class. This creates a one-to-many relationship between the two classes, where each author can have multiple books associated with them.

To access the related objects, you can use the _set notation, like this:

```
# Create an author
author = Author.objects.create(name='Jane Austen')
# Create a book associated with the author
book = Book.objects.create(title='Pride and Prejudice', author=author)
# Access the author of the book
book.author # returns the Author object
# Access all books associated with an author
author.book_set.all() # returns a QuerySet of all Book objects with
author=author
```

19.What is Jwt Authentication?Explain

JWT (JSON Web Token) is a way of securely transmitting information between parties as a JSON object. It can be used for authentication purposes in web applications, where a user is required to log in and access protected resources.

The process of using JWT for authentication typically involves the following steps:

1. Users logs in with their credentials (e.g., email and password).
2. The server verifies the credentials and generates a JWT containing a payload with user-specific information (e.g., user ID, username, and expiration time).
3. The JWT is returned to the client, usually in the form of an HTTP response header.
4. The client stores the JWT in a cookie, local storage, or session storage.
5. The client sends the JWT with each subsequent request to the server in the Authorization header.
6. The server verifies the JWT signature, decodes the payload, and checks if the user is authorized to access the requested resource.

To implement JWT authentication in a Django web application, you can use third-party libraries like django-rest-framework-simple jwt, which provides ready-to-use views and serializers for handling JWT authentication.

Here's an example of how to use django-rest-framework-simplejwt:

1. Install the package:

```
pip install djangorestframework-simplejwt
```

2. Add the following settings to your Django project's settings.py file:

```
REST_FRAMEWORK = {
   'DEFAULT_AUTHENTICATION_CLASSES': [
      'rest_framework_simplejwt.authentication.JWTAuthentication',
        ],
}

SIMPLE_JWT = {
   'ACCESS_TOKEN_LIFETIME': timedelta(minutes=30),
   'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
}
```

3. Create a view that generates a JWT when a user logs in:

```
from rest_framework_simplejwt.views import TokenObtainPairView
class MyTokenObtainPairView(TokenObtainPairView):
        serializer_class = MyTokenObtainPairSerializer
```

4.Create a serializer that defines the fields required for generating a JWT:

```
from rest_framework_simplejwt.serializers import
TokenObtainPairSerializer
class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
        @classmethod
        def get_token(cls, user):
        token = super().get_token(user)
        # Add custom claims to the token
       token['username'] = user.username
        return token
```

5. Include the view in your Django project's urls.py file:

```
from django.urls import path
```

```
from .views import MyTokenObtainPairView
urlpatterns = [
   path('api/token/', MyTokenObtainPairView.as_view(),
name='token_obtain_pair'),
]
```

6. Use the @api_view decorator to protect views that require authentication:

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response

@api_view(['GET'])
@permission_classes([IsAuthenticated])
def protected_view(request):
        content = {'message': 'This is a protected view.'}
        return Response(content)
```

7. Make a POST request to the /api/token/ URL to obtain a JWT:

```
curl -X POST \
  http://localhost:8000/api/token/ \
  -H 'Content-Type: application/json' \
  -d '{
    "email": "john.doe@example.com",
    "password": "secret"
}'
```

8. Make a GET request to the /protected_view/ URL with the JWT in the Authorization header:

```
curl -X GET \
  http://localhost:8000/protected_view/ \
  -H 'Authorization: Bearer <your-access-token
```

20.What is Basic Authentication and Token Authentication

Basic Authentication and Token Authentication are two different ways of authenticating users in web applications.

1. **Basic Authentication:**

Basic Authentication is a simple authentication scheme that is used to send the user's credentials (username and password) in an encoded format with each request. The credentials are usually encoded in Base64 format and sent in the Authorization header of the HTTP request.
Here's an example of how to use Basic Authentication in a Django web application:

```
from django.contrib.auth.decorators import login_required
from django.http import HttpResponse

@login_required
def protected_view(request):
    return HttpResponse('This is a protected view.')
```

In this example, the @login_required decorator ensures that the user is authenticated using Basic Authentication before accessing the protected view. If the user is not authenticated, they will be redirected to the login page.

2. **Token Authentication:**

Token Authentication is a more advanced authentication scheme that involves the use of a token (typically a JWT) to authenticate the user. The token is usually generated by the server and sent to the client after the user has successfully logged in. The client stores the token and sends it with each subsequent request in the Authorization header of the HTTP request.

Here's an example of how to use Token Authentication in a Django web application using the django-rest-framework-simplejwt package:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated
from rest_framework_simplejwt.authentication import JWTAuthentication

class ProtectedView(APIView):
    permission_classes = [IsAuthenticated]
```

```
    authentication_classes = [JWTAuthentication]

    def get(self, request):
    content = {'message': 'This is a protected view.'}
    return Response(content)
```

In this example, the IsAuthenticated permission class ensures that the user is authenticated using Token Authentication before accessing the protected view. The JWTAuthentication authentication class is used to authenticate the user using a JWT token. If the user is not authenticated, they will receive a 401 Unauthorized response.

21.Model Inheritance in Django

Model inheritance is a powerful feature in Django that allows you to define a new model that inherits fields and methods from an existing model. There are three types of model inheritance in Django:

1. **Abstract Base Classes:**

An abstract base class is a model that is not intended to be instantiated. It provides a base set of fields and methods that can be inherited by other models. You can define an abstract base class by setting the abstract attribute to True in the Meta class of the model.
Here's an example:

```
from django.db import models

class BaseModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
    abstract = True

class MyModel(BaseModel):
    name = models.CharField(max_length=100)
    description = models.TextField()
```

In this example, BaseModel is an abstract base class that defines two fields (created_at and updated_at) and is not intended to be instantiated. MyModel inherits from BaseModel and defines two additional fields (name and description).

2. **Multi-table Inheritance:**

Multi-table inheritance is a type of model inheritance where each model in the inheritance hierarchy is stored in a separate database table. Each model inherits all the fields and methods of its parent model, and can define additional fields and methods of its own.
Here's an example:

```
from django.db import models

class Animal(models.Model):
        name = models.CharField(max_length=100)
        age = models.IntegerField()

class Dog(Animal):
        breed = models.CharField(max_length=100)

class Cat(Animal):
        color = models.CharField(max_length=100)
```

In this example, Animal is the base model, and Dog and Cat inherit from Animal. Each model is stored in a separate database table, with a foreign key linking it to its parent model.

3. **Proxy Models:**

A proxy model is a model that is based on an existing model, but has a different name and can have additional methods. The proxy model uses the same database table as the original model, but provides a different way of accessing the data.
Here's an example:

```
from django.db import models
class MyModel(models.Model):
   name = models.CharField(max_length=100)
```

```
class MyModelProxy(MyModel):
    class Meta:
        proxy = True
    def custom_method(self):
        return "This is a custom method"
```

In this example, MyModelProxy is a proxy model based on the MyModel model. The proxy model does not create a new database table, but instead provides a different way of accessing the data. The custom_method() method is an example of an additional method that can be defined on a proxy model.

22.Can we create custom permission in django from using python? give detail explanation

Ans - Yes, we can create custom permissions in Django programmatically using Python. Here is a detailed explanation of how to do it:

A) Import necessary modules

First, import the necessary modules:

***Python code ***

```
from django.contrib.auth.models import Permission

from django.contrib.contenttypes.models import ContentType
```

B) Create a content type

A content type is a way to organize models in Django. Before you can create a permission, you need to create a content type for your model.

***Python code ***

```
content_type = ContentType.objects.get_for_model(MyModel)
```

C) Create a permission

Next, create a permission for your model. You can use the **Permission.objects.create()** method to create a new permission:

***Python code ***

```
permission = Permission.objects.create(
    codename='can_view_report',
    name='Can view report',
    content_type=content_type,
)
```

The **codename** parameter is a unique identifier for the permission. The **name** parameter is a human-readable name for the permission.

D) Add the permission to a group

To assign the permission to a group, you need to first create the group:

***Python code ***

```
from django.contrib.auth.models import Group
```

group = Group.objects.create(name='Reporters')

Then, add the permission to the group:

***Python code ***

group.permissions.add(permission)

E) Assign the group to a user

Finally, you can assign the group to a user:

***Python code ***

user.groups.add(group)

Now, the user has the custom permission that you created.

That's it! You have now created a custom permission in Django programmatically using Python.

23. What are the different inbuilt permission in Django

Django provides three built-in permissions:

A)add: Allows a user to add new objects of a model.

B)change: Allows a user to change existing objects of a model.

C)delete: Allows a user to delete existing objects of a model.

These permissions are available for every model in your Django application. When you define a model, Django automatically creates these permissions for it.

You can assign these permissions to users or groups using the Django admin interface or programmatically using Python.

In addition to the built-in permissions, Django provides a fourth permission view, which is not created automatically when you define a model. You can add the **view** permission to a model by including it in the **permissions** attribute of the model's **Meta** class.

For example:

*** Python Code***

```
class MyModel(models.Model):

    ...

    class Meta:

    permissions = [('view_mymodel', 'Can view MyModel')]
```

With the view permission, you can control who has access to view instances of a model.

24. How do you optimize rest api in Django

Optimizing a REST API in Django involves several techniques that you can apply to improve its performance and scalability. Here are some ways to optimize a REST API in Django:

Caching: Use caching to avoid processing requests that can be cached. Django provides a caching framework that supports various cache backends, such as Memcached and Redis

Pagination: Use pagination to limit the number of resources returned in a response. This can reduce the amount of data transferred between the server and client, improving the performance of the API

Throttling: Implement rate limiting or throttling to prevent clients from overwhelming the API with requests. Django provides a throttling framework that allows you to control the rate at which clients can access the API.

Serialization: Use efficient serialization techniques to reduce the size of the JSON response. Django supports various serialization formats, including JSON, XML, and YAML.

Database optimization: Optimize database queries and use database indexes to improve query performance. Django provides an ORM that supports various databases, including PostgreSQL, MySQL, and SQLite.

Caching database queries: Use a caching framework to cache database queries that are frequently used. Django provides a caching framework that supports caching of database queries.

Use async views: Use asynchronous views to handle requests more efficiently. Async views can improve the performance of the API by allowing the server to handle multiple requests concurrently.

Use a CDN: Use a Content Delivery Network (CDN) to cache static assets such as images, stylesheets, and JavaScript files. This can improve the performance of the API by reducing the amount of data transferred between the server and client.

By applying these techniques, you can optimize your REST API in Django and improve its performance and scalability.

25.Different type of http method?

HTTP (Hypertext Transfer Protocol) defines several methods (also known as verbs) that indicate the intended action to be performed on a resource. Here are the most commonly used HTTP methods:

GET: Retrieves a resource from the server. This method is safe and idempotent, meaning it should not modify the resource on the server and multiple identical requests should produce the same result.

POST: Submits an entity (such as form data) to the server for processing. This method is not idempotent, meaning multiple identical requests may have different results, such as creating multiple resources on the server.

PUT: Updates an existing resource on the server with a new representation provided in the request. This method is idempotent, meaning multiple identical requests should produce the same result.

DELETE: Deletes a resource on the server. This method is idempotent, meaning multiple identical requests should produce the same result.

PATCH: Applies a partial update to an existing resource on the server. This method is not idempotent, meaning multiple identical requests may have different results.

HEAD: Retrieves the headers of a resource, without the body. This method is similar to GET, but only retrieves metadata about the resource, not the full representation.

OPTIONS: Retrieves information about the communication options available for a resource. This method is typically used to retrieve information about the supported HTTP methods and other capabilities of the server.

These HTTP methods provide a standard way for clients and servers to communicate and interact with resources over the web.

26. What are generics and mixins

In Django, generics and mixins are used to simplify the development of reusable and extensible views.

Generics in Django refer to a set of reusable view classes that provide common functionality for handling common data models. For example, Django includes generic views for handling list views, detail views, and form views. These generic views can be subclassed and customized to handle specific data models, without having to write the view logic from scratch.

Mixins in Django refer to a set of reusable classes that can be used to extend the functionality of views. For example, Django includes mixins for adding login and authentication functionality, pagination, and filtering to views. These mixins can be added to any view that requires their functionality, without having to modify the original view.

Django provides a wide range of generic views and mixins that can be used to build web applications quickly and efficiently. By using these generics and mixins, developers can avoid writing repetitive code and focus on building the core functionality of their applications. Additionally, Django's generic views and mixins follow best practices for security and performance, ensuring that applications built with them are secure and scalable.

27.What is class based views and function based views

In Django, there are two ways to create views that handle HTTP requests and generate HTTP responses: class-based views and function-based views.

Function-based views: Function-based views are the simplest and most common way to create views in Django. A function-based view is a Python function that takes an HTTP request as its first argument and returns an HTTP response. Function-based views are easy to write and understand, and are suitable for simple use cases.

Class-based views: Class-based views are a more powerful and flexible way to create views in Django. A class-based view is a Python class that defines methods for handling different HTTP methods (such as GET, POST, PUT, DELETE, etc.). Class-based views allow you to encapsulate common functionality in base classes and inherit from them, and provide more advanced features such as mixins and inheritance.

Both function-based views and class-based views have their own advantages and disadvantages, and the choice between them depends on the specific requirements of the application. Function-based views are easy to write and understand, but can become difficult to maintain and reuse as the application grows. Class-based views are more powerful and flexible, but can be more complex and require more advanced knowledge of Python and Django.

28.What are different status codes?

HTTP (Hypertext Transfer Protocol) defines several status codes that servers can use to provide information about the response to a client's request. Here are some of the most commonly used status codes:

1xx (Informational): Indicates that the server has received the request and is continuing to process it.

2xx (Successful): Indicates that the request was successfully received, understood, and accepted by the server. The most common status code in this category is 200 OK, which indicates a successful response.

3xx (Redirection): Indicates that the client must take additional action to complete the request. For example, a 302 Found status code indicates that the requested resource has been temporarily moved to a new URL.

4xx (Client Error): Indicates that the client's request was invalid or cannot be completed by the server. The most common status code in this category is 404 Not Found, which indicates that the requested resource does not exist on the server.

5xx (Server Error): Indicates that the server was unable to complete the request due to an error on the server side. The most common status code in this category is 500 Internal Server Error, which indicates a generic error on the server.

There are many other HTTP status codes defined in the HTTP specification, each with a specific meaning and use case. Understanding HTTP status codes is important for developing and debugging web applications, as they can provide valuable information about the success or failure of requests and responses.

29.What is oauth 2.0 explain? how does it work

OAuth 2.0 is an authorization framework that allows third-party applications to access a user's resources without having to share the user's credentials (such as username and password). It is widely used by many popular websites and mobile applications for authentication and authorization purposes.

OAuth 2.0 works by separating the roles of the resource owner (user), client (application), and authorization server. The client requests access to the user's resources through the authorization server, which authenticates the user and grants the client access to the requested resources. The client then uses an access token to access the user's resources without needing the user's credentials.

Here is a basic flow of how OAuth 2.0 works:

The client (application) requests access to the user's resources (such as photos, contacts, or documents).

The authorization server prompts the user to authenticate and authorize the client's access to the requested resources.

If the user grants access, the authorization server issues an access token to the client.

The client uses the access token to access the user's resources from the resource server

The resource server validates the access token and grants or denies access to the requested resources.

OAuth 2.0 uses different grant types (such as authorization code, implicit, client credentials, and resource owner password credentials) depending on the specific use case and security requirements. OAuth 2.0 is widely adopted because it is flexible, extensible, and allows users to grant granular permissions to third-party applications without sharing their credentials.

30.What are viewsets?

In Django REST Framework, ViewSets are classes that provide a set of methods for handling CRUD (Create, Read, Update, Delete) operations for a particular type of data. ViewSets can be used to group related API views into a single class, which can simplify the code and make it more maintainable.

ViewSets are based on the concept of RESTful resources, where each resource is identified by a unique URL and can be accessed using HTTP methods. A ViewSet typically defines a set of standard methods, such as **list**, **retrieve**, **create**, **update**, and **delete**, that correspond to the standard HTTP methods **GET**, **POST**, **PUT**, **PATCH**, and **DELETE**.

There are two main types of ViewSets in Django REST Framework:

ModelViewSet: This ViewSet provides default implementations for all CRUD operations for a Django model. It includes standard methods for listing, retrieving, creating, updating, and deleting objects from the database.

GenericViewSet: This ViewSet provides basic implementations for common CRUD operations, but requires explicit method implementations for some actions, such as listing or retrieving objects.

ViewSets can also be customized to include additional methods, permissions, authentication, and other features as needed. They are a powerful tool for building RESTful APIs in Django, as they allow developers to define a set of related views in a single class, which can simplify the code and improve maintainability.

31.What are serializers and their different types?

In Django REST Framework, serializers are a key component that allow you to convert complex data types (such as Django models) into Python data types that can be easily rendered into JSON, XML, or other content types that are commonly used in web applications.

There are two main types of serializers in Django REST Framework:

Model Serializer: This is a serializer that automatically generates serializer fields based on a Django model. It is a shortcut for creating a serializer that corresponds to a model and can save a lot of time when building APIs that expose Django models. The ModelSerializer class automatically includes fields for all model fields and can automatically handle relationships between models.

Serializer: This is a more flexible serializer that allows you to define the fields and data conversion logic explicitly. You can use Serializer when you need to customize the serialization or deserialization process, or when you are working with non-model data.

In addition to these main types, there are also several other serializer classes available in Django REST Framework:

Hyperlinked Model Serializer: This is a subclass of Model Serializer that adds support for hyperlinking between related resources.

JSON Serializer: This is a serializer that converts Python data to JSON format.

XML Serializer: This is a serializer that converts Python data to XML format.

Form Serializer: This is a serializer that converts Django form data to JSON format.

Nested Serializer: This is a serializer that allows you to include nested serializers for related objects.

File Field Serializer: This is a serializer that allows you to handle file uploads in your API.

Overall, serializers are a powerful tool for converting complex data types into simple, easily-renderable formats, making it easier to build and maintain RESTful APIs in Django.

32.What are routers explain?

In Django REST Framework, routers are a convenient way to automatically generate a set of URL patterns for a ViewSet. Routers can be used to map ViewSets to URL patterns, allowing you to create a clean and consistent API URL structure without having to write URL patterns by hand.

A router typically consists of two components: a router instance and a set of URL patterns. The router instance is responsible for mapping ViewSets to URL patterns, while the URL patterns are used to define the URLs that will be used to access the API views.

There are two main types of routers in Django REST Framework :

Simple Router: This is a basic router that provides a set of default URL patterns for a View Set, including URLs for list and detail views.

Default Router: This is a more advanced router that provides additional URL patterns for views that support actions other than the standard CRUD methods. The Default Router includes the URL patterns for list, detail, and custom actions that are defined on the View Set.

Using routers can greatly simplify the process of defining URL patterns for a Django REST Framework API. By mapping View Sets to URL patterns automatically, you can ensure that your API has a consistent and predictable structure, making it easier for clients to work with. Additionally, routers can help reduce the amount of boilerplate code needed to define URL patterns, which can save time and reduce the chance of errors.

33.Explain in detail multithreading and multiprocessing with example in python?

Multi-threading and multi-processing are two techniques used to achieve concurrent execution of code in Python. In this answer, I will explain both concepts in detail and provide examples of how to use them in Python.

Multi-threading:

Multi-threading is a technique used to achieve concurrent execution of code by running multiple threads within a single process. Each thread can run independently and perform a separate task simultaneously. Threads can communicate with each other and share data within the same process.

In Python, multi-threading is achieved using the threading module. Here is an example of how to create and start a new thread in Python:

Python Copy code

*** Python code***

```
import threading

def print_numbers():
    for i in range(10):
    print(i)

# Create a new thread
t = threading.Thread(target=print_numbers)

# Start the thread
t.start()
```

In this example, we define a function **print_numbers()** that prints numbers from 0 to 9 . We then create a new thread using the **Thread** class and pass

the **print_numbers()** function as the target. Finally, we start the thread using the **start()** method.

Multi-processing:

Multi-processing is a technique used to achieve concurrent execution of code by running multiple processes in parallel. Each process can run independently and perform a separate task simultaneously. Processes cannot communicate with each other directly, but they can share data using inter-process communication mechanisms such as pipes, queues, and shared memory.

In Python, multi-processing is achieved using the multiprocessing module. Here is an example of how to create and start a new process in Python:

*** Python code ***

```
import multiprocessing

def print_numbers():
    for i in range(10):
    print(i)

# Create a new process
```

```
p = multiprocessing.Process(target=print_numbers)


# Start the process

p.start()
```

In this example, we define a function **print_numbers()** that prints numbers from 0 to 9. We then create a new process using the **Process** class and pass the **print_numbers()** function as the target. Finally, we start the process using the **start()** method.

The main difference between multithreading and multiprocessing is that multi-threading runs multiple threads within a single process, while multi-processing runs multiple processes in parallel. Multi-threading is generally more lightweight and efficient than multi-processing, but it is also more prone to synchronization issues and race conditions.

In summary, multi-threading and multi-processing are two techniques used to achieve concurrent execution of code in Python. Multi-threading is used to run multiple threads within a single process, while multi-processing is used to run multiple processes in parallel. Both techniques have their own advantages and disadvantages, and the choice between them depends on the specific requirements of the application.

34.What are docker commands? explain

Docker is a popular platform for creating and running containers that allow developers to easily package, deploy, and run their applications in any environment. Here are some of the most commonly used Docker commands:

docker run: This command is used to create and start a new container based on an image. For example, **docker run -it ubuntu** will start a new container based on the Ubuntu image and open an interactive terminal inside the container.

docker build: This command is used to build a new Docker image based on a Dockerfile. For example, **docker build -t myimage .** will build a new image with the tag **myimage** based on the Dockerfile in the current directory.

docker pull: This command is used to download a Docker image from a registry. For example, **docker pull nginx** will download the latest version of the Nginx image from the Docker Hub registry.

docker push: This command is used to upload a Docker image to a registry. For example, **docker push myusername/myimage** will upload the **myimage** image to the Docker Hub registry under the **myusername** account.

docker ps: This command is used to list all running containers. For example, **docker ps** will show a list of all containers that are currently running.

docker stop: This command is used to stop a running container. For example, **docker stop mycontainer** will stop the container with the name **mycontainer**.

docker rm: This command is used to remove a container. For example, **docker rm mycontainer** will remove the container with the name **mycontainer**.

docker rmi: This command is used to remove an image. For example, **docker rmi myimage** will remove the image with the tag **myimage**.

docker exec: This command is used to run a command inside a running container. For example, **docker exec mycontainer ls -l** will run the **ls -l** command inside the container with the name **mycontainer**.

These are just a few examples of the many Docker commands available. Docker is a powerful tool that can help developers easily manage and deploy their applications, and learning how to use Docker effectively is an important skill for modern software development.

35.How and where do we use docker commands?

Docker commands are used in the terminal or command prompt of your local machine or remote server. The commands are used to manage Docker images and containers, which can be used to run applications or services.

Docker is often used in software development and deployment to create a consistent and isolated environment for running applications. Developers can use Docker to package their application and its dependencies into a Docker image, which can then be run on any system that has Docker installed.

Here are some examples of where and how Docker commands can be used:

Development environment: Developers can use Docker to create a consistent development environment for their applications, which can be easily shared with other developers. They can use Docker to create a container with the necessary dependencies, such as a specific version of Python or a particular database, and then use that container to develop their application.

Testing and staging environments: Docker can be used to create isolated environments for testing and staging applications. Developers can use Docker to create a container with the application and its dependencies, and then deploy that container to a testing or staging environment to ensure that the application works as expected.

Production environment: Docker can be used to deploy applications to production environments, where they can be run in a consistent and isolated manner. Developers can use Docker to create a container with the

application and its dependencies, and then deploy that container to a production server.

Overall, Docker commands are used to manage Docker images and containers, which can be used to create consistent and isolated environments for running applications. Docker is a powerful tool that can help developers easily manage and deploy their applications, and learning how to use Docker effectively is an important skill for modern software development.

36.What are Git commands and Git rebase explain

Git is a popular version control system used in software development to manage code changes and collaborate with others. Here are some of the most commonly used Git commands:

git init: This command is used to create a new Git repository in a directory.

git clone: This command is used to create a local copy of a remote Git repository.

git add: This command is used to add changes to the staging area

git commit: This command is used to commit changes to the local repository with a commit message.

git push: This command is used to push changes from the local repository to a remote repository.

git pull: This command is used to pull changes from a remote repository to the local repository.

git branch: This command is used to create, list, and delete branches.

git merge: This command is used to merge changes from one branch into another.

git rebase: This command is used to rebase a branch onto another branch.

Git rebase is a command used to rewrite the commit history of a branch. It works by taking the changes in one branch and reapplying them onto another branch. The result is a linear history of commits, without any merge commits.

37.Here's an example of how to use git rebase:

Create a new branch: **git checkout -b feature-branch**

Make changes and commit them: **git add .** and **git commit -m "Commit message"**

Switch to the main branch: **git checkout main**

Pull changes from the remote repository: **git pull origin main**

Rebase the feature branch onto the main branch: **git rebase main**

Resolve any conflicts that arise.

Push the changes to the remote repository: **git push origin feature-branch -f**

Overall, Git is a powerful tool that can help developers manage code changes and collaborate with others. Git rebase is a useful command for creating a linear history of commits, without merge commits, and can help make the commit history more readable and easier to understand.

38.What is the recent version tool of git?

As of March 2023, the most recent version of Git is version 2.36.1, which was released on February 7, 2022. This version includes several bug fixes, improvements, and new features. Some of the notable changes in version 2.36.1 include:

Improved performance when cloning large repositories.

Added support for sparse-checkout patterns, which allow users to specify which files or directories to include in a working directory.

Improved handling of merge conflicts.

Added a new **--allow-read-only** option to **git-branch** to allow creating a branch that can only be used for reading.

It's recommended to keep Git up-to-date to take advantage of the latest features and improvements. You can download the latest version of Git from the official Git website: https://git-scm.com/downloads.

39. Explain the most commonly used git command ?

Here are some of the most commonly used Git commands:

git init: This command is used to create a new Git repository in a directory.

Git clone: This command is used to create a local copy of a remote Git repository.

git add: This command is used to add changes to the staging area.

git commit: This command is used to commit changes to the local repository with a commit message.

git push: This command is used to push changes from the local repository to a remote repository.

git pull: This command is used to pull changes from a remote repository to the local repository.

git branch: This command is used to create, list, and delete branches.

git merge: This command is used to merge changes from one branch into another.

git status: This command is used to display the current status of the repository, including which files have been modified, which files are staged for commit, and which files are untracked.

git log: This command is used to display the commit history of the repository.

git diff: This command is used to display the differences between files in the working directory and the staged changes.

git stash: This command is used to temporarily save changes that are not ready to be committed, so that they can be applied later.

These are just some of the most commonly used Git commands, and there are many more commands and options available. Understanding and becoming comfortable with these commands is essential for using Git effectively in software development.

40.commonly used docker command explain

Here are some commonly used Docker commands:

docker run: This command is used to create and start a new container from a specified image.

docker pull: This command is used to download an image from a remote repository.

docker build: This command is used to build a Docker image from a Docker file.

docker ps: This command is used to list all running containers.

docker images: This command is used to list all available Docker images.

docker stop: This command is used to stop a running container.

docker rm: This command is used to remove a container.

docker rmi: This command is used to remove an image.

docker logs: This command is used to view the logs of a container.

docker exec: This command is used to execute a command inside a running container.

docker-compose up: This command is used to start all services defined in a Docker Compose file.

docker-compose down: This command is used to stop and remove all containers created by **docker-compose up**.

These are just a few of the commonly used Docker commands. Docker has a rich set of commands and options available that allow developers to manage containers and images effectively. Understanding and becoming comfortable with these commands is essential for working with Docker efficiently.

41.what are signals in django explain

In Django, signals are a way of allowing decoupled applications to get notified when certain actions occur elsewhere in the application. Signals allow certain senders to notify a set of receivers that some action has taken place.

In simple terms, a signal is a way of sending a message or notification from one part of your application to another. Signals can be used to perform a wide variety of tasks, such as updating a cache when a model is saved or sending an email when a new user is registered.

Here's a brief overview of how signals work in Django:

A signal is created using the **Signal** class provided by Django.

The signal is then connected to one or more receiver functions using the **receiver** decorator.

When the signal is sent by the sender, all of the connected receiver functions are called.

The sender does not need to know anything about the receiver functions or what they do, making it a decoupled way to perform actions.

Here's an example of how signals might be used in Django:

Python code

```
from django.db.models.signals import post_save

from django.dispatch import receiver

from myapp.models import MyModel


@receiver(post_save, sender=MyModel)

def my_handler(sender, **kwargs):

    # Do something here when MyModel is saved

    pass
```

In this example, a signal is created using the **post_save** signal that is sent whenever an instance of **MyModel** is saved. The **my_handler** function is then connected to the signal using the **receiver** decorator, which will be called whenever the signal is sent.

Overall, signals provide a powerful way to decouple different parts of your application and allow them to communicate with each other in a flexible and maintainable way.

42.What is the use admin.py and manage.py

admin.py and **manage.py** are both important files in a Django project.

admin.py is a file in your Django project where you can register your models with the Django admin site. By registering a model with the admin site, you can easily create, read, update, and delete instances of that model through a web interface, without having to write custom views or templates. In **admin.py**, you can also customize the behavior and appearance of the admin site by modifying the admin classes associated with your models.

manage.py is a command-line utility that comes with Django and is used to perform a variety of tasks related to managing your Django project. Some common tasks include running the development server, creating database tables, running database migrations, and running tests. When you run **manage.py**, you can pass various arguments to specify which command you want to run and any additional options or arguments that command requires.

Here are a few examples of how **manage.py** can be used:

python manage.py run server: Starts the Django development server.

python manage.py migrate: Runs any pending database migrations.

python manage.py create superuser: Creates a new superuser account.

python manage.py test: Runs your project's test suite.

Overall, **admin.py** and **manage.py** are both important files in a Django project that allow you to easily manage and customize various aspects of your application.

43.What are the features of the django framework?

Django is a popular web framework for building web applications using Python. Some of the key features of Django include:

Object-Relational Mapping (ORM): Django provides an ORM that allows you to define your database schema using Python code. This makes it easy to work with databases without having to write SQL code directly.

Built-in Admin Interface: Django comes with a built-in admin interface that makes it easy to manage your application's data through a web interface. You can use this interface to add, edit, and delete records, as well as perform other administrative tasks.

URL Routing: Django provides a URL routing system that maps URLs to view functions in your application. This makes it easy to build complex web applications with multiple pages and views.

Template System: Django provides a template system that allows you to separate the design and presentation of your web pages from the underlying Python code. This makes it easy to create and maintain consistent and reusable templates for your application.

Middleware: Django provides a middleware system that allows you to add functionality to your application at various stages of the request/response cycle. This includes things like authentication, caching, and compression.

Security Features: Django provides a number of security features, including protection against cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks. It also provides tools for handling user authentication and authorization.

Testing Framework: Django comes with a built-in testing framework that makes it easy to write and run tests for your application. This allows you to catch and fix bugs early in the development process.

Overall, Django is a powerful and flexible framework that provides a wide range of features for building web applications. Its ease of use, robustness, and scalability have made it a popular choice for developers around the world.

44.How is mvt different from mvc architecture

MVT (Model-View-Template) and MVC (Model-View-Controller) are both design patterns used for developing web applications, but they differ in some key ways.

In the MVC architecture, the Controller is responsible for handling user input and deciding which view to display. The Model represents the data and business logic of the application, and the View is responsible for presenting the data to the user.

In the MVT architecture, the Model represents the data and business logic of the application, just like in MVC. The Template is responsible for presenting the data to the user, and the View is responsible for handling user input and processing requests. The Template in MVT is similar to the View in MVC, but the key difference is that the Template only deals with the presentation logic and not with the business logic, which is the responsibility of the View.

One advantage of the MVT architecture is that it separates the presentation logic from the business logic more clearly, making it easier to maintain and modify the code. Another advantage is that it allows for more flexibility in the presentation layer, since the Template can be easily modified without affecting the business logic.

In summary, MVT and MVC are both widely used design patterns for web development, but MVT separates the presentation logic more clearly and allows for more flexibility in the presentation layer.

45.Explain jwt token and how it useful

JWT stands for JSON Web Token, which is an open standard for securely transmitting information between parties as a JSON object. JWT is a popular method for authentication and authorization in web applications.

A JWT token consists of three parts: a header, a payload, and a signature. The header and payload are base64-encoded JSON objects that contain information about the token and the user, such as the user ID, username, and expiration time. The signature is used to verify the authenticity of the token, and is calculated using a secret key known only to the server.

When a user logs in to a web application, the server generates a JWT token and sends it to the client, which can then store it in a cookie or local storage. On subsequent requests, the client sends the JWT token to the server, which can then verify the token's authenticity and use the information in the payload to identify the user and determine their permissions.

JWT tokens are useful because they are stateless, meaning that the server does not need to keep track of the user's session information. This makes them ideal for use in distributed systems and microservices, where multiple servers may need to authenticate and authorize requests. JWT tokens are also widely supported by libraries and frameworks, making them easy to use in many different programming languages.

In summary, JWT tokens are a secure and efficient method for authentication and authorization in web applications. They provide a standardized way of transmitting user information between parties, and are useful for building distributed systems and microservices.

46.What are authorization methods used?

Authorization methods are used to control access to resources in a web application. Some common authorization methods used in web development include:

Role-based access control (RBAC): This method assigns roles to users and defines what actions users with each role are allowed to perform. For example, an application might have roles like "admin," "manager," and "user," each with different permissions.

Attribute-based access control (ABAC): This method uses attributes like user location, department, or job title to determine access to resources. For example, an application might restrict access to certain resources based on the user's location or job title.

Rule-based access control: This method uses a set of rules to determine access to resources. For example, an application might have rules that allow access to certain resources only during certain times of day or only to users with certain privileges.

Discretionary access control (DAC): This method allows users to grant or deny access to resources they own. For example, a user might create a document and then grant access to it to specific users or groups.

Mandatory access control (MAC): This method uses a system of labels and rules to control access to resources. For example, a government agency might use MAC to control access to classified information.

These authorization methods can be used in combination or separately, depending on the needs of the application. It is important to choose the appropriate authorization method based on the security requirements and the level of access control needed for the application.

47.What is the uses of middleware in Django

In Django, middleware is a component that sits between the web server and the view, allowing you to modify requests and responses before they reach the view. The main uses of middleware in Django are:

Authentication and Authorization: Middleware can be used to enforce authentication and authorization rules for certain views or to redirect users to login pages.

Request/response modification: Middleware can modify request headers, response headers, or even the response content itself. For example, it can add security headers to the response or compress response content to reduce the size of the response.

Error handling: Middleware can handle errors and exceptions that occur during request processing. For example, it can catch exceptions and return a custom error page to the user.

Caching: Middleware can be used to cache responses for certain views, which can improve the performance of the application.

Request processing logging: Middleware can be used to log information about each request that comes into the application, such as the time it took to process the request or the IP address of the user.

Overall, middleware is a powerful tool in Django that can be used to modify the behavior of the application in a variety of ways. It allows developers to write reusable code that can be applied to multiple views, making it easier to maintain and extend the application over time.

48.what is return keyword

In Python, the **return** keyword is used to exit a function and return a value. When a function is called, it executes a block of code and may or may not

return a value. The **return** statement is used to specify what value the function should return, if any.

For example, consider the following function that adds two numbers and returns the result:

*** Python code ***

```
def is_even(number):
    if number % 2 == 0:
    return True
    else:
    return False
```

In this example, if the **number** argument is even, the function will return **True** and exit early. If the **number** argument is odd, the function will continue executing and eventually return **False**.

49.Write keyword to convert python function to python generator

To convert a Python function to a Python generator, you need to use the **yield** keyword instead of the **return** keyword. When a function uses **yield**, it becomes a generator function that can be iterated over using a **for** loop or the **next()** function.

Here's an example of a function that returns a list of even numbers:

*** Python code ***

```
def get_even_numbers(n):
    for i in range(n):
    if i % 2 == 0:
    yield i
```

In this example, the **get_even_numbers()** function returns a generator object that can be iterated over to obtain the even numbers. Each time the **yield** statement is encountered, the current value of **i** is returned, and the function is paused until the next value is requested.


BROWSEJOBS