Q1: Explain the MVC pattern in the context of Django. How does it relate to the structure of a Django project?

The MVC pattern stands for Model View And Controller. Model: It is something that deals with databases and is responsible for representing the database and performing different data-related tasks also known as business logic.

View: It is something that deals with the user side (user-request) and is responsible for generating the user interface in which the user can interact and provide necessary data to the UI by getting the required data from the database.

Controller: It is something that stays between the Model and View. It is responsible for understanding the user request and calling the right View (class or method). It is responsible for handling different routes effectively.

In Django, we have something like (MVT), which stands for Model View Template.

Model: It is defined in the models.py file. We use an object-oriented approach to represent the entity. and use Object Relational Mapping (ORM) to manipulate the data in the database

for eg:

```
from django.db import models

class Author(models.Model):

    name = models.CharField(max_length=255)


    def __str__(self):

        return self.name


class Book(models.Model):

    title = models.CharField(max_length=255)

    # one author can write multiple books

    author = models.ForeignKey(Author,on_delete=models.CASCADE)

    publication_date = models.DateField()


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

Q2: Write a Python program to swap the values of two variables without using a temporary variable.

```
# initialization
a = 10
b = 20
print("Before Swapping")

print(f"a :{a}")
print(f"b :{b}")

# Swapping
b,a = a,b

print("After Swapping")
print(f"a :{a}")
print(f"b :{b}")
```

Q3: Explain the differences between lists and tuples in Python.

List: It is a mutable data structure in Python where we can modify, the data later as per our need for eg :

```
books = ["DSA", "Python", "Java"]
books.append("OS")
books.remove("DSA")
```

we use the list to store the dynamic data that might need to be updated in the future.

Tuples: It is an immutable data structure in Python where we store such info that is not supposed to be changed in the future, such as config data.

for eg : mytuple = (1,2,3,4)

```
STATUS_CHOICES = (
('success', 'success'),
('failure', 'failure'),
('process', 'process'),
)
```

we use this to store choices in models, in Django

Q4. What is encapsulation, and how is it implemented in Python?

Encapsulation is one of the 4 pillars of OOP. It is all about building the data and the methods on those data into a single unit called a class. By doing so the data is hidden and only accessed through the defined methods on that class. Although the encapsulation leads to data hiding but it is not data hiding.

```
class Author(models.Model):
   # public attribute
   name = models.CharField(max_length=255)

   # private attribute
   __contact = models.CharField(max_length=255)

   # protected attribute
   _address = models.CharField(max_length=255)
```

```
    # setter method for contact
    def set_contact(self, contact):
        self.__contact = contact

    # setter method for address
    def set_address(self, address):
        self._address = address

    # getter method for contact
    def get_contact(self):
        return self.__contact

    # getter method for address
    def get_address(self):
        return self._address

    def __str__(self):
        return self.name

# Creating author
python_author = Author()
python_author.name = "Elon Musk"

# setting author address and contact
python_author.set_address("USA")
python_author.set_contact("0000")
```

# getting author name address and contact

author_name = python_author.name

author_address = python_author.get_address()

author_contact = python_author.get_contact()

Q5: Explain the purpose of Django's ORM (Object-Relational Mapping).

Django ORM acts as the bridge between the Django code and the database. It allows us to interact with the database using Python objects rather than writing raw SQL queries. for eg:

Allows us to interact with databases using Python objects and methods

We can use the same ORM to query all types of databases. For example, to get all the books we can write: books = Book.objects.al(). This will return all the books no matter which databases we use. In short, we don't need to worry about the raw SQL statements to handle different databases

It also saves our application from SQL injection attacks

It helps to represent the entities as a Python object which makes it easier to store data and manipulate them

for example: we can use ORM to interact with the databse holding book record, such as:

all_books = Books.objects.all()

first_book = Books.objects.get(id=1)

change the name of the book first_book.name = "first book 2.0"

delete the book first_book.delete()

Q6: Create a function in Python that takes a list of numbers as input and returns the sum of all even numbers in the list.

```
# list initialization
mynums = []


user_input = input("Enter numbers and -1 when done : ")


while user_input != "-1":
    try:
        mynums.append(int(user_input))
    except ValueError:
        print("Please enter int number")


    user_input = input()


print("My list is :")
print(mynums)



# function that takes a list and check if it's even and sum all the even number in the list
and returns the result
def calc_sum(myList:list):
    sum = 0
    for i in myList:
        if i % 2 == 0:
            sum +=i


    return sum
```

```
result = calc_sum(mynums)
print(f"The sum of all even numbers in the list is {result}")
```

Q7: Write a Python function that takes user input for a number and handles the exception if the input is not a valid integer.

```
def get_int_number():
    try:
        num = int(input("Enter a number : "))
        print(f"You entered {num}")

    except ValueError:
        print("Please enter a number")
```

Q8: Create a Django model representing a Book with attributes like title, author, and publication date.

```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self):
        return self.name



class Book(models.Model):
    title = models.CharField(max_length=255)
    # one author can write multiple books
    author = models.ForeignKey(Author,on_delete=models.CASCADE)
    publication_date = models.DateField()


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

Q9: Explain the concept of decorators in Python and provide an example of a custom decorator.

Decorators are a way of adding additional functionality to a function. They are primarily used for validating authentication, setting permissions, logging, etc in Django. for example in Django, if we have some protected views which require user to be logged in, we can use a decorator called @login_required

```
@login_required
def protected_view(request):
    # protected code
```

Example of custom decorators :

```
def log_sum(func):
    def wrapper(*args, **kwargs):
        print(f"Starting sum...")
        print(f"Calling function : {func.__name__}")
        print(f"Arguments are : {args}")

        result = func(*args, **kwargs)

        print(f"The result of the function is : {result}")
        return result

    return wrapper

@log_sum
def sum(a,b):
    return a + b



result = sum(3,2)
print(f"Result : {result}")
```

Q10: What is a generator in Python? Provide an example of a generator function.

Generator is a type of function that generates a sequence of values lazily. It returns an iterator that does not store all the data on the memory all at once, thus it makes it possible to deal with large datasets. We use the yield keyword which allows to pause and resume the operation. They are helpful while dealing with large datasets. I have used the generator to define the database session and pass that connection to each function which were responsible for crud on the database. That is, when we call the generator function, it does not perform the operation and returns the result, rather provides us with the iterator which we can use to generate the value

```
def count_to_n(n):
    value = 0

    while value < n:
        yield value
        value += 1
```

```
for num in count_to_n(10):
    print(num)
```

Q11: Explain the Global Interpreter Lock (GIL) in Python and how it affects multithreading.

It is a mutex, Mutual Exclusion Lock, It locks the critical part of the code to be executed by multiple threads at once to ensure data consistency. One of the major problems in multithreading is that multiple threads can access the data, or some critical part of the code. which might affect the CIA of the data. Thus Global Interpreter Lock helps to lock the data or the critical part of the code until the one thread that is using the common data or executing the critical part of the program releases the lock once the previous thread is done with the common and critical section.

It affects multithreading in the following ways.

1. As I mentioned above, it locks some part of the code until the thread is done with that part of the code, it affects the performance.
2. It affects the concurrency and parallel processing

Q12: Discuss the key features of Django REST Framework (DRF) and how it enhances API development in Django.

Django REST framework helps us to build REST API using the Django framework. It allows us to create a secure, robust, and scalable REST API quickly.

Some of the key features of the Django REST framework are as follows:

1. It has serialization which helps to serialize our Django models easily. It helps us to control the response and also validate the request from the user
2. It helps to define permissions and it also supports the authentication

3. It offers us a bunch of viewsets and routers which helps us to perform different operations on viewsets and with the help of routers, it makes it easy to route to different urls with a minimum set of code
4. It offers us built-in pagination and filtering, it saves us a lot of time and effort
5. One of the best things i like is, the browsable API, with this we don’t need to spend time on Postman

Q13: Explain what Django Channels is and provide a use case where it is beneficial.

Django channels are used to extend the features of Django applications with a bunch of extra features like asynchronous operations. It allows us to build real-time applications for example chat apps, notification systems, and live data updating applications. It is like adding an extra worker which does the task of the application without blocking the original Django application. It supports web sockets which helps to establish a persistent connection between the client and the Django application. The main feature of the Django channel is to create a scalable backend, without affecting the performance of the Django application

Q14: What are metaclasses in Python, and how are they different from regular classes?

As we all know class is the blueprint for creating objects, and the metaclasses in Python are the blueprint for creating Python classes. It is useful to customize the behavior of the Python classes, and also for the creation of the classes. They define how the classes are created and behave

They are different from than class in the following ways

As classes are blueprints for the object, they are blueprints for creating and controlling the behavior of the classes

Q15: Explain the concept of asynchronous programming in Python using the asyncio module. Provide an example.

Asynchronous programming is all about running multiple tasks without blocking each other. It allows us to build applications that can perform other operations without waiting for the previous task to be complete. This we can perform multiple tasks which helps to enhance the performance and scalability of the application. In Python, the asyncio helps us to write asynchronous code, it helps us to use the async and await code just like in Javascript.

for example:

```
import asyncio


async def fetch_data(url):
    print(f"Started fetching data... for {url}")
    #simulate delay
    await asyncio.sleep(4)
    print(f"Completed : {url}")
    return url


async def main():
    #creating task
    task1 =  asyncio.create_task(fetch_data("bhandarikapil.com.np"))
    task2 = asyncio.create_task(fetch_data("bhandarikapi.com.np/projects"))


    result1 = await task1
    result2 = await task2


    print(f"Task1 result : {result1}")
    print(f"Task2 result : {result2}")
    print("Complete main function")



asyncio.run(main())
```

Q16: What are descriptors in Python? Provide an example of how they can be used.

Descriptors define how we can manage the object's attributes. Such as setting the value on attributes, getting the value of that attribute, etc. We basically use 3 methods , they are __getters__(),__setters__(), and __delete__()

It is most useful in such a situation when we want to perform certain checks before saving, or before returning to the value. for example:

```
class DescriptorDemo:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        print("Value ...")
        return self._value

    @value.getter
    def value(self):
        print("Value getter")
        return self._value

    @value.setter
    def value(self,value):
        print("Value setter")
        self._value = value

    @value.deleter
    def value(self):
        print("Value deleter")
        del self._value


v1 = DescriptorDemo(4)
print(v1.value)
v1.value = 10
print(v1.value)
del v1.value
```

Q17: Discuss the importance of migrations in Django. How can you handle schema migrations for a database?

Migrations in Django help us to create database schema automatically without having to write SQL scripts ourselves. It helps to sync our Django model with the database schema. If there was no migration, we would have to write SQL scripts like, create table, update table, etc each time we update our database model. But with the help of migration, we can achieve this automatically.

For example, we already created the database 2 models, (book and author) in the previous question.

To create a database schema of that model class. we can follow the following steps.

1. create migration file python3 manage.py make migrations This will create a migration file, in our Django project we got “0001_initial.py”, which contains the information about database schema for the author and book model
2. we can review the file if we want to
3. After that, we can apply the migrations python3 manage.py migrate This will actually, update the database, we have to understand that, the previous command is just to create the migration file, and this command will actually reflect the change to the database schema.
4. We can also verify if the migration was applied python3 manage.py showmigrations This command helps us to verify the migrations applied to our application
5. Suppose we decided to add another field, to store the number of books written by an author, to do that, we just need to add that attribute on the author model and create a migration file(in our case we get file 00002_author_no_of_books_written.py) and apply migration

6. let’s suppose we decided not to add the no of books written field in the database. That is we wanted the previous database schema. we can easily do this by.
   python3 manage.py migrate core 0001