Python Learning Day 9

1. Working with Modules

- **Writing Your Own Modules**: Create a `.py` file with reusable code.
- **Using Standard Library Modules**: Utilize Python's built-in libraries like `math`, `os`, etc.

2. Modules and Packages

- **Importing Modules**: Use `import` to bring in existing modules.
- **Creating and Using Packages**: Organize modules into directories with an `__init__.py` file.

3. Debugging and Testing

- **Debugging Techniques**: Use print statements, `pdb`, and error handling.
- **Unit Testing with `unittest` Module**: Structure and automate tests for code.

4. Secure Coding Practices

- **Handling User Authentication and Authorization**: Use Django's authentication system.
- **Cross-Site Scripting (XSS) and SQL Injection Prevention**: Use Django's template escaping and ORM.

5. Database Access

- **SQL Databases (SQLite, PostgreSQL)**: Use Django's database configurations for these SQL databases.
- **Using ORM (Object-Relational Mapping)**: Interact with databases using Django models.

6. Concurrency

- **Threading vs. Multiprocessing**: Use threading for I/O-bound tasks and multiprocessing for CPU-bound tasks.
- **Asyncio for Asynchronous Programming**: Manage I/O-bound tasks using coroutines and event loops.

1. Working with Modules

- A **module** in Python is simply a file that contains Python code (like functions, variables, or classes). Modules allow you to organize your code into separate files, making it easier to manage, reuse, and maintain.

Writing Your Own Modules

- To write your own module, you simply create a `.py` file. For example, you might have a file named `mymodule.py`with the following content:

```
python
Copy code
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

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

- You can then use this module in another Python file by importing it:

```
python
Copy code
import mymodule

print(mymodule.greet("Alice"))  # Output: Hello, Alice!
print(mymodule.add(5, 3))       # Output: 8
```

Using Standard Library Modules

- Python has a rich set of **standard library modules** that are included by default. Some commonly used modules include:
  - `math`: For mathematical functions.
  - `datetime`: For working with dates and times.
  - `os`: For interacting with the operating system.
  - `json`: For working with JSON data.
- Example of using a standard library module:

```
python
Copy code
import math

print(math.sqrt(16))  # Output: 4.0
```

2. Modules and Packages

- A **package** in Python is a way of organizing related modules into a directory hierarchy. A package is simply a directory with an `__init__.py` file and one or more module files.

Importing Modules

- You can import modules in various ways:

```
python
Copy code
import mymodule            # Import the whole module
from mymodule import greet # Import a specific function
import math as m           # Import with an alias
```

Creating and Using Packages

- A package is a directory that contains a special `__init__.py` file. This file can be empty or contain initialization code for the package.
- Example structure:

```
markdown
Copy code
mypackage/
    __init__.py
    module1.py
    module2.py
```

- You can then use the package like this:

```
python
Copy code
from mypackage import module1
import mypackage.module2 as mod2

result = module1.some_function()
another_result = mod2.another_function()
```

3. Debugging and Testing

- **Debugging** is the process of finding and fixing errors (bugs) in your code.
- **Testing** involves verifying that your code behaves as expected. Python provides built-in tools to aid both processes.

Debugging Techniques

- **Print Statements**: A quick way to debug by printing variable values:

```
python
Copy code
def add(a, b):
    print(f"Adding {a} and {b}")
    return a + b

result = add(2, 3)  # Adding 2 and 3
```

- **Using the `pdb` Module**: Python has a built-in debugger called `pdb`. You can add `pdb.set_trace()` in your code to start debugging.

```
python
Copy code
import pdb

def divide(a, b):
    pdb.set_trace()  # Start the debugger
    return a / b

result = divide(4, 2)
```

- **Error Handling**: Use `try...except` blocks to catch and handle exceptions:

```
python
Copy code
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

Unit Testing with `unittest` Module

- The `unittest` module is part of Python's standard library and is used to create test cases for your code.
- **Creating a Test Case**:

```
python
Copy code
import unittest

# Function to be tested
def add(a, b):
    return a + b

# Creating a Test Case
class TestMathFunctions(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(3, 4), 7)  # Test for equality
        self.assertNotEqual(add(3, 4), 8)  # Test for inequality

# Run the tests
if __name__ == "__main__":
    unittest.main()
```

- **Running the Tests**:
  - Save the test file and run it using the command line:

    ```
    bash
    Copy code
    python -m unittest test_file.py
    ```

Summary

- **Modules** help in organizing code, and you can write your own or use Python's built-in ones.
- **Packages** are directories of related modules, aiding larger codebases.
- **Debugging** includes techniques like using print statements, error handling, or `pdb`.
- **Testing** is handled by Python's `unittest`, allowing structured and automated verification of code.

1. Secure Coding Practices

Secure coding practices are methods and techniques to ensure that the code you write is secure and resilient to attacks. In Python and Django, secure coding practices are essential to safeguard web applications from common vulnerabilities.

Key Secure Coding Practices in Python and Django:

1. **Input Validation and Sanitization**:
   - Always validate and sanitize user inputs to prevent malicious data from causing harm. In Django, you can use form validation to check if input data meets the expected format.
   - Example in Django using forms:

     ```
     python
     Copy code
     from django import forms

     class UserForm(forms.Form):
         username = forms.CharField(max_length=50)
         age = forms.IntegerField(min_value=18, max_value=100)  #
     Validating age
     ```

2. **Use Django's Built-in Features**:
   - Django provides many security features out-of-the-box, such as CSRF protection, SQL injection prevention, and secure password hashing. Rely on Django's built-in tools instead of creating custom implementations.
   - Ensure that `DEBUG` mode is set to `False` in production:

     ```
     python
     Copy code
     DEBUG = False  # In production
     ```

3. **Secure Data Storage**:
   - Use Django's encrypted fields to store sensitive information. For example, storing passwords should always be done using Django's built-in `User` model which uses strong password hashing.

- Example:

```
python
Copy code
from django.contrib.auth.models import User

user = User.objects.create_user(username='john',
password='securepassword123')
user.save()
```

4. **Error Handling and Logging**:
   - Avoid displaying detailed error messages to end-users, as it may expose sensitive information about the server or the application. Use custom error pages (like `404.html` or `500.html`).
   - Log errors using Django's logging system:

```
python
Copy code
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'ERROR',
            'class': 'logging.FileHandler',
            'filename': '/path/to/django/error.log',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file'],
            'level': 'ERROR',
            'propagate': True,
        },
    },
}
```

5. **Secure API Endpoints**:
   - Use secure practices when building APIs. Django provides the `django-rest-framework` (DRF) for building APIs with built-in tools for secure authentication.
   - Use HTTPS instead of HTTP to encrypt data between the client and server.

2. Handling User Authentication and Authorization

Authentication is the process of verifying a user's identity, while authorization is about checking what permissions an authenticated user has. Django offers robust tools for handling both authentication and authorization.

User Authentication in Django:

1. **Built-in Authentication System**:

- Django comes with a built-in authentication system that provides login, logout, password management, and user registration.
- Example of using Django’s built-in authentication:

```
python
Copy code
from django.contrib.auth import authenticate, login, logout
from django.shortcuts import render, redirect

def login_view(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, 
password=password)

        if user is not None:
            login(request, user)
            return redirect('dashboard')
        else:
            return render(request, 'login.html', {'error': 
'Invalid credentials'})

    return render(request, 'login.html')
```

2. **Password Security**:
   - Use Django's `make_password` and `check_password` functions to handle passwords securely:

```
python
Copy code
from django.contrib.auth.hashers import make_password, 
check_password

hashed_password = make_password('mypassword123')
is_valid = check_password('mypassword123', hashed_password)
```

3. **Django's Authentication Middleware**:
   - Django includes middleware for authentication which handles session cookies and authorization.
   - Use `@login_required` decorator to protect views that require user authentication.

```
python
Copy code
from django.contrib.auth.decorators import login_required

@login_required
def dashboard(request):
    return render(request, 'dashboard.html')
```

User Authorization in Django:

1. **Using Groups and Permissions**:

- Django allows grouping users with specific permissions. You can manage these groups and permissions via the Django Admin interface or programmatically.

  ```
  python
  Copy code
  from django.contrib.auth.models import Group, Permission

  # Creating a new group
  editors = Group.objects.create(name='Editors')

  # Adding permissions to the group
  permission = Permission.objects.get(codename='can_edit_article')
  editors.permissions.add(permission)
  ```

2. **Custom Permissions**:
   - You can create custom permissions for models by adding them to the `Meta` class:

     ```
     python
     Copy code
     class Article(models.Model):
         title = models.CharField(max_length=100)
         content = models.TextField()

         class Meta:
             permissions = [
                 ("can_edit_article", "Can edit article"),
             ]
     ```

3. **Access Control with `has_perm`**:
   - Use `request.user.has_perm('app_label.permission_codename')` to check if a user has specific permissions.

     ```
     python
     Copy code
     if request.user.has_perm('myapp.can_edit_article'):
         # Allow user to edit the article
     ```

3. Cross-Site Scripting (XSS) and SQL Injection Prevention

XSS and SQL Injection are common web vulnerabilities. Here's how to handle them in Django:

Preventing Cross-Site Scripting (XSS):

- XSS occurs when an attacker injects malicious scripts into web pages viewed by other users.
- Django escapes user inputs by default when rendering templates, preventing most XSS attacks.

1. **Template Auto-Escaping**:

- Django templates automatically escape user inputs, so if you use Django templates, user inputs will be safe:

  ```
  html
  Copy code
  <!-- Safe: Output will be auto-escaped -->
  <p>{{ user_input }}</p>
  ```

- If you must allow HTML, use `|safe` carefully:

  ```
  html
  Copy code
  <!-- Use with caution -->
  <p>{{ user_input|safe }}</p>
  ```

2. **Form Handling**:
   - Use Django's forms to handle user input, as it automatically escapes form data.

Preventing SQL Injection:

- SQL Injection is a technique where an attacker can execute malicious SQL code. Django's ORM (Object-Relational Mapper) is designed to prevent SQL Injection.

1. **Using Django's ORM**:
   - Avoid raw SQL queries. Use Django's ORM to interact with the database, which automatically escapes user input.

     ```
     python
     Copy code
     # Safe queries using Django ORM
     articles = Article.objects.filter(title__icontains='Django')
     ```

2. **Parameterized Queries**:
   - If raw SQL is necessary, use parameterized queries to avoid injection:

     ```
     python
     Copy code
     from django.db import connection

     def get_article_by_title(title):
         with connection.cursor() as cursor:
             cursor.execute("SELECT * FROM myapp_article WHERE title =
     %s", [title])
             result = cursor.fetchone()
         return result
     ```

3. **Avoid Direct String Concatenation in SQL**:
   - Never concatenate user inputs directly into SQL queries.

1. Database Access

In Django, interacting with databases is streamlined and simplified using built-in tools. While SQL Databases such as SQLite and PostgreSQL are commonly used, Django provides a high-level API called Object-Relational Mapping (ORM) to abstract away most SQL commands, making database interactions easy and safe.

2. SQL Databases (SQLite, PostgreSQL)

SQLite

- **SQLite** is a lightweight, file-based database that doesn't require a separate server to operate. It's an excellent choice for small-scale projects or development environments.
- **Advantages**:
  - Zero configuration: No installation or configuration required.
  - Portable: Database is stored in a single file.
  - Suitable for small projects or development.
- **Using SQLite in Django**:
  - SQLite is the default database when you create a new Django project.
  - Example of `settings.py` configuration for SQLite:

    ```
    python
    Copy code
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.sqlite3',
            'NAME': BASE_DIR / 'db.sqlite3',  # Database file
        }
    }
    ```

- **Running SQLite Commands**:
  - Use the built-in `sqlite3` command-line tool to interact with the SQLite database:

    ```
    bash
    Copy code
    sqlite3 db.sqlite3
    ```

PostgreSQL

- **PostgreSQL** is a powerful, open-source object-relational database system. It is known for reliability, robustness, and advanced features, making it a preferred choice for production environments.
- **Advantages**:
  - ACID-compliant: Ensures reliable transactions.
  - Scalable: Suitable for large applications.
  - Advanced features: Full-text search, JSON support, complex queries.
- **Installing PostgreSQL**:
  - On Ubuntu:

```
bash
Copy code
sudo apt-get update
sudo apt-get install postgresql postgresql-contrib
```

- **Using PostgreSQL in Django**:
  - You need the `psycopg2` package to connect Django to PostgreSQL:

    ```
    bash
    Copy code
    pip install psycopg2-binary
    ```

  - Example of `settings.py` configuration for PostgreSQL:

    ```
    python
    Copy code
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql',
            'NAME': 'your_database_name',
            'USER': 'your_username',
            'PASSWORD': 'your_password',
            'HOST': 'localhost',  # Use '127.0.0.1' for localhost
            'PORT': '5432',
        }
    }
    ```

- **Creating a PostgreSQL Database**:
  - Use PostgreSQL commands to create a user and database:

    ```
    bash
    Copy code
    sudo -u postgres psql
    CREATE DATABASE mydb;
    CREATE USER myuser WITH PASSWORD 'mypassword';
    GRANT ALL PRIVILEGES ON DATABASE mydb TO myuser;
    ```

3. Using ORM (Object-Relational Mapping)

What is ORM?

- **Object-Relational Mapping (ORM)** is a technique that allows you to interact with the database using high-level programming constructs instead of raw SQL queries.
- Django's ORM provides a powerful and intuitive way to work with relational databases, making it easy to define data models, execute complex queries, and manage database relationships without writing SQL.

Defining Models in Django ORM

- In Django, a **model** is a Python class that represents a database table. Each attribute of the class corresponds to a database field.
- Example of a simple `models.py` file:

```
python
Copy code
from django.db import models

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    birth_date = models.DateField()

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

- **Data Fields**:
  - `CharField`, `IntegerField`, `DateField`, `BooleanField`, etc.
  - `ForeignKey` is used to define a relationship between tables.

Creating Database Tables

- Once you define your models, Django needs to create the corresponding tables in the database:

```
bash
Copy code
python manage.py makemigrations
python manage.py migrate
```

Performing Database Operations with ORM

1. **Creating and Inserting Records**:
   - Use Django's ORM to create and save objects in the database:

```
python
Copy code
# Creating an Author
author = Author(first_name='John', last_name='Doe',
birth_date='1980-01-01')
author.save()

# Creating a Book associated with an Author
book = Book(title='My First Book', author=author,
published_date='2023-01-01', pages=200)
book.save()
```

- You can also use the `create` method to save data directly:

```
python
Copy code
author = Author.objects.create(first_name='Jane',
last_name='Smith', birth_date='1990-05-15')
```

2. **Reading and Querying Records**:
   - **Retrieving All Records**:

```
python
Copy code
authors = Author.objects.all()
```

   - **Filtering Records**:

```
python
Copy code
# Get authors born after 1980
authors = Author.objects.filter(birth_date__gt='1980-01-01')

# Get a specific author by first name
author = Author.objects.get(first_name='John')
```

   - **Field Lookups**:
     - Use lookups like `__gt` (greater than), `__lt` (less than), `__icontains` (case-insensitive contains).
   - **Chaining Queries**:

```
python
Copy code
# Filter and order results
books =
Book.objects.filter(published_date__year=2023).order_by('title')
```

3. **Updating Records**:
   - Update records using the `update` method or by modifying and saving objects:

```
python
Copy code
# Modify an object and save
author = Author.objects.get(id=1)
author.first_name = 'Jonathan'
author.save()

# Bulk update
Author.objects.filter(last_name='Doe').update(last_name='Smith')
```

4. **Deleting Records**:
   - Delete records using the `delete` method:

```
python
Copy code
# Delete a specific author
author = Author.objects.get(id=1)
author.delete()

# Bulk delete
Book.objects.filter(pages__lt=100).delete()
```

Relationships in Django ORM

- **One-to-Many (ForeignKey)**: Example is the `author` field in the `Book` model.
- **Many-to-Many**: Use `ManyToManyField` for relationships where multiple records relate to multiple others.

  ```
  python
  Copy code
  class Publisher(models.Model):
      name = models.CharField(max_length=100)

  class Book(models.Model):
      title = models.CharField(max_length=100)
      publishers = models.ManyToManyField(Publisher)
  ```

- **One-to-One**: Use `OneToOneField` when a record in one table is linked to exactly one record in another.

  ```
  python
  Copy code
  class Profile(models.Model):
      user = models.OneToOneField(User, on_delete=models.CASCADE)
      bio = models.TextField()
  ```

Advanced Querying with Django ORM

- **Aggregation**:

  ```
  python
  Copy code
  from django.db.models import Count, Avg

  # Count the number of books per author
  author_book_count = Author.objects.annotate(num_books=Count('book'))

  # Average number of pages
  avg_pages = Book.objects.all().aggregate(Avg('pages'))
  ```

- **Prefetching Related Data**:
  - Use `select_related` for foreign key relationships (One-to-Many) and `prefetch_related` for Many-to-Many.

    ```
    python
    Copy code
    # Efficiently fetch related data
    ```

```
books = Book.objects.select_related('author').all()
```

Summary

- **SQLite** is great for development or small projects; it's a file-based database.
- **PostgreSQL** is robust, scalable, and suitable for production with advanced features.
- **Django's ORM** allows you to interact with SQL databases using Python classes and methods, abstracting away raw SQL. You define models, and Django handles the SQL queries.
- The ORM supports creating, reading, updating, and deleting records easily, while also managing relationships like one-to-many and many-to-many.

By leveraging Django's ORM, you gain both simplicity and power in managing databases, with a layer of security that reduces common vulnerabilities like SQL Injection.

1. Concurrency Overview

Concurrency in Python refers to the ability of a program to manage multiple tasks at the same time. It does not necessarily mean executing tasks simultaneously (that would be parallelism), but rather managing multiple tasks in a way that makes the program more efficient.

Why Use Concurrency?

- To make programs faster or more efficient, especially when waiting for I/O operations (e.g., reading files, network requests).
- To utilize multiple cores or CPUs for heavy computational tasks.
- To improve responsiveness in applications that involve multiple tasks.

2. Threading vs. Multiprocessing

Threading

- **Threading** is a way to run multiple threads (lightweight sub-processes) concurrently within the same process.
- Threads share the same memory space, making communication between them fast and easy. However, due to Python's **Global Interpreter Lock (GIL)**, only one thread can execute Python code at a time. This makes threading suitable for I/O-bound tasks but not for CPU-bound tasks.

Key Characteristics of Threading:

- Shared memory space.
- Lightweight and low memory usage.

- Limited by the GIL in CPU-bound tasks.
- Ideal for I/O-bound operations (e.g., file I/O, network requests).

Using Threading in Python:

- Example using the `threading` module:

```
python
Copy code
import threading
import time

def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")
        time.sleep(1)

def print_letters():
    for letter in 'ABCDE':
        print(f"Letter: {letter}")
        time.sleep(1)

# Create threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start threads
thread1.start()
thread2.start()

# Wait for threads to complete
thread1.join()
thread2.join()

print("Done!")
```

Advantages of Threading:

- Great for I/O-bound operations (e.g., downloading files, reading data from disk, network operations).
- Easier to implement for simple concurrent tasks.

Disadvantages of Threading:

- GIL limits performance for CPU-bound tasks.
- Requires careful handling of shared resources to avoid race conditions.

Multiprocessing

- **Multiprocessing** involves running multiple processes concurrently, each with its own memory space. This bypasses the GIL, making it suitable for CPU-bound tasks.

- Each process runs in its own memory space, so they do not share memory, which avoids the limitations of the GIL but requires more memory.

Key Characteristics of Multiprocessing:

- Separate memory space for each process.
- No GIL limitation, making it suitable for CPU-bound tasks.
- Higher memory usage compared to threading.
- Ideal for computationally intensive operations.

Using Multiprocessing in Python:

- Example using the `multiprocessing` module:

```
python
Copy code
from multiprocessing import Process
import os

def worker(number):
    print(f"Process {number} running (PID: {os.getpid()})")

# Create multiple processes
processes = []
for i in range(5):
    process = Process(target=worker, args=(i,))
    processes.append(process)
    process.start()

# Wait for all processes to complete
for process in processes:
    process.join()

print("All processes are done!")
```

Advantages of Multiprocessing:

- Bypasses the GIL, allowing true parallelism on multiple cores/CPUs.
- Great for CPU-bound operations (e.g., calculations, simulations).

Disadvantages of Multiprocessing:

- Higher memory consumption.
- Inter-process communication is more complex than thread communication.

3. Choosing Between Threading and Multiprocessing

- **Use Threading** for I/O-bound tasks where the program is mostly waiting (e.g., web scraping, handling network requests, file I/O).

- **Use Multiprocessing** for CPU-bound tasks where heavy computations are involved (e.g., data processing, machine learning computations, complex mathematical calculations).

4. Asyncio for Asynchronous Programming

- **`asyncio`** is a Python module that supports asynchronous programming by using **coroutines**. It allows you to handle concurrent I/O-bound operations without creating multiple threads or processes.
- Instead of blocking a thread while waiting for I/O, `asyncio` uses an event loop to manage and schedule tasks, making it highly efficient for I/O-bound tasks.

Key Concepts in `asyncio`:

- **Event Loop**: The core of the `asyncio` module, which schedules and runs asynchronous tasks.
- **Coroutines**: Functions defined with `async def` that can be paused and resumed.
- **Tasks**: Coroutines that have been wrapped for execution in the event loop.
- **Awaiting**: Using `await` to pause the execution of a coroutine until the awaited task is complete.

Basic Example of `asyncio`:

- A simple asynchronous program:

```
python
Copy code
import asyncio

async def say_hello():
    print("Hello!")
    await asyncio.sleep(1)  # Pause for 1 second
    print("World!")

# Run the coroutine
asyncio.run(say_hello())
```

Creating and Running Multiple Coroutines:

- Example using `asyncio.gather` to run multiple coroutines concurrently:

```
python
Copy code
import asyncio

async def task1():
    print("Task 1 starting")
    await asyncio.sleep(2)
    print("Task 1 done")

async def task2():
    print("Task 2 starting")
    await asyncio.sleep(1)
    print("Task 2 done")

async def main():
    await asyncio.gather(task1(), task2())

# Run the main coroutine
asyncio.run(main())
```

Concurrency with `asyncio`:

- Unlike threading or multiprocessing, `asyncio` achieves concurrency by quickly switching between tasks while they are waiting for I/O operations to complete.
- This makes `asyncio` suitable for tasks like web scraping, database access, and network operations where the code spends a lot of time waiting.

Using `asyncio` with I/O-bound Operations:

- Example of performing asynchronous HTTP requests:

```
python
Copy code
import asyncio
import aiohttp  # Install aiohttp for asynchronous HTTP requests

async def fetch_url(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    urls = ['http://example.com', 'http://example.org',
'http://example.net']

    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        responses = await asyncio.gather(*tasks)
        for response in responses:
            print(response)

# Run the main coroutine
asyncio.run(main())
```

Comparison: Threading, Multiprocessing, and Asyncio

Feature	Threading	Multiprocessing	Asyncio
Best For	I/O-bound tasks	CPU-bound tasks	I/O-bound tasks
Memory Usage	Low	High	Very Low

Feature	Threading	Multiprocessing	Asyncio
Parallelism	Limited due to GIL (pseudo-concurrent)	True parallelism (multi-core)	Cooperative multitasking (non-blocking)
Complexity	Medium (shared memory management)	High (inter-process communication)	Low (event-driven)
Speed	Good for I/O	Best for heavy computations	Excellent for I/O, fast switching
Example Libraries	`threading`, `concurrent.futures`	`multiprocessing`	`asyncio`, `aiohttp`

Summary

- **Threading** is best for I/O-bound tasks, and it works well if you don't need heavy computations. Python's GIL limits its performance for CPU-bound tasks.
- **Multiprocessing** is the go-to choice for CPU-heavy operations, as it allows parallel processing without GIL constraints, but it requires more memory.
- **Asyncio** is ideal for asynchronous I/O operations where you need to handle thousands of tasks without blocking, making it efficient for web scraping, network communication, and database I/O.