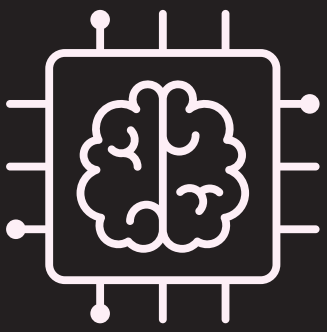
PYTHON

and

AI



Introduction

WELCOME fellow Python Comrade.

This E-Book goes over various **Beginner** & **Advanced** Python concepts that are a must known to any Python Developer.

The E-Book features in-depth Guide on many libraries & Frameworks, such as Pandas, Flask, NLP, and AI Libraries like NumPy & TensorFlow.

So, Let's Get Started.

Author: Rishabh Singh

Contents

Python Best Practices 5

Beginner

Data Structures and Algorithms 9

- Arrays, Linked Lists, Stacks, Queues,
- Search Algorithm, Trees, Hash Tables
- Sorting Algorithms, Dijkstra's algorithm

OOP in Python 19

- Classes, Inheritance,
- Encapsulation, Polymorphism
- OOP Interview Questions

Advanced

Python Decorators 27

- Function Decorators, Class Decorators,
- Chaining Decorators, Decorators Arguments

Python Metaclasses 32

- Metaclasses Interview Questions

Magic Functions in Python 35

Python Generators 38

- Generator Uses

Context Managers 43

Multi-Threading & Multi-Processing 48

Asynchronous Programming 53

- Coroutines, Asyncio, async - await,
- Synchronous vs Asynchronous

Expert

SQLite3 for Python **60**

- Installing SQLite
- Advanced SQLite
  - Transactions, Triggers,
  - Indexes, Views
- SQLite3 Interview Questions

Natural Language Processing (NLP) **70**

- Text Processing, Linguistic Secrets
- Sentiment Analysis with NLTK - **Project**

Pandas **79**

- Installation & Set-up
- Series, Dataframes,
- Indexing & Selecting Data,
- Handling Missing Data
- Pandas FAANG Interview Questions
- Pandas Project - Analyzing Sales Data

FLASK **95**

- Installation & Set-up
- Creating New Flask App
- Flask Dev Server, Adding Dynamic Content,
- Flask Extensions, Flask-WTF, Flask Forms,
- Flask-SQLalchemy, Flask Database,
- Flask Login, Flask RESTful
- Flask FAANG Interview Questions

Projects

Awesome Python Projects **121**

- Building a BlackJack Game
- Building Weather App using Python
- Python MARS Project 

E-Book Part - 2

Python and AI

NumPy 137

- Installation & Set-up
- NumPy Arrays
- Arrays Indexing and Slicing
- NumPy & Data Analysis
- NumPy and Image Processing
- NumPy FAANG Interview Questions

TensorFlow 146

- Installation & Set-up
- Building a Simple TensorFlow Program
- Working with Tensors
- Sessions and Variables
- Handling Data with Tensorflow
- TensorFlow FAANG Interview Questions

Python Best Practices to follow:

In many companies, Python is a key language used for developing cutting-edge AI systems and applications.

However, it's not enough to simply write code in Python - it's equally important to write high-quality, maintainable code that follows best practices.

In this chapter, we'll explore the importance of Python best practices in industry.

Importance of Python Best Practices::

Collaboration: As a professional developer, you are expected to work in collaboration with thousands of other developers around the Globe. So, your code should be easily understandable by anyone reading it.

Optimization for performance: You are also expected to write code that is optimized and executes fast. i.e. there should be no extra lines, no unused variables, no unused functions. Your code should be as clutter-free as possible.

Follow the existing code structure: You will be expected to follow a code structure that is already being used by the senior devs. This makes collaboration easier (which is the most important thing while working).

Modularity and reusability: Your code should be reusable so that any future developer can easily edit/modify/add new features to your code. Breaking down code into smaller, modular components promotes code reuse and makes it easier to understand, test, and maintain.

If you write a neat and clean code, you will thank yourself and others will thank you too.

In the Next Chapter, We'll go through a series of **Best Practices** that you must follow:

1. Variable Naming Convention

Setting intuitive variable names is very helpful for anyone to understand 'what' we are trying to do.
Often, we create variables using single characters such as i, j, k — these are not self-explanatory variables and should be avoided.

Note: Never use l, o, or I single-letter names as these can be mistaken for "1" and "0", depending on the typeface.

```
o = 2 # This may look like you're trying to reassign 2 to zero
```

2. Don't use random values in your code. Define them as constants.

So you dont have to update your whole code later if you need to change just 1 value.

```
# some code here
weight = 9.8 * mass
# some more code here

#The above can be Written as:

GRAVITATIONAL_ACCELERATION = 9.8

# some code here
weight = GRAVITATIONAL_ACCELERATION * mass
# some more code here
```

3. Comments in function

Add comments to your function to make it more understandable.

There are 2 types of comments:

#1 Inline Comment: Comments which are in the same line as our code. '#' is used to set the inline comments.

```
a = 10 # Single line comment
```

#2 Block Comment: Multi-line comments are called block comments.

```
""" Example of Multi-line comment
    Line 2 of a multiline comment
"""
```

4. Consistent Indentation

Use consistent indentation (usually four spaces) to improve code readability.

Python relies on indentation to define code blocks, so make sure to be consistent.

5. Modularize Your Code

Break down your code into reusable modules, classes, and functions.

This promotes code reusability and helps maintain a clean and organized codebase.

6. Avoid Magic Numbers

Instead of using arbitrary numbers in your code, assign them to named constants. This improves code readability and makes it easier to modify those values later.

7. Virtual Environments

Use virtual environments like virtualenv or conda to isolate project dependencies.

This ensures that your project's dependencies are separate from the system-level Python installation.

8. Use Built-in Data Structures and Functions

Python provides a rich set of built-in data structures (lists, dictionaries, sets, etc.) and functions (sorting, filtering, mapping, etc.).

Utilize these features to write efficient and concise code.

9. Testing

Write unit tests for your code to ensure it functions as expected.

Use frameworks like pytest or unittest to create and run tests, allowing you to catch bugs and regressions early on.

10. Version Control

Utilize a version control system like Git to track changes in your codebase.

This helps you keep a history of your project and collaborate with others effectively.

Learning Phase

In this phase, you'll Learn Skills that will improve your chances of landing a job at one of the big tech giants.

Lets start with the most important ones **Data Structures & Algorithms**

Introduction to Data Structures and Algorithms

Data structures and algorithms are fundamental concepts in computer science, and they play a crucial role in software development.

Simply put, a data structure is a way of organizing and storing data, while an algorithm is a set of instructions that can be used to solve a specific problem.

Data structures can be thought of as containers for data.

They can be used to store and organize information in a way that allows for efficient access and manipulation. Some common data structures include arrays, linked lists, stacks, queues, and trees.

Algorithms, on the other hand, are sets of instructions that can be used to solve specific problems.

They can be used to manipulate and analyze data stored in data structures. Some common algorithms include searching, sorting, and graph traversal algorithms.

Let's take a closer look at some common data structures and algorithms:

Arrays: An array is a data structure that stores a collection of elements, each identified by an index or a key.

Arrays can be used for a variety of tasks, such as storing lists of items or performing mathematical computations.

Linked Lists: A linked list is a data structure that consists of a sequence of nodes, each containing data and a reference to the next node in the sequence.

Linked lists can be used for tasks such as implementing stacks or queues.

Searching Algorithms: Searching algorithms are used to find a specific element in a data structure. Some common searching algorithms include linear search and binary search.

Sorting Algorithms: Sorting algorithms are used to sort elements in a data structure in a specific order. Some common sorting algorithms include bubble sort, selection sort, and quicksort.

Graph Traversal Algorithms: Graph traversal algorithms are used to traverse a graph data structure in a specific order. Some common graph traversal algorithms include breadth-first search and depth-first search.

Implementing Data Structures in Python

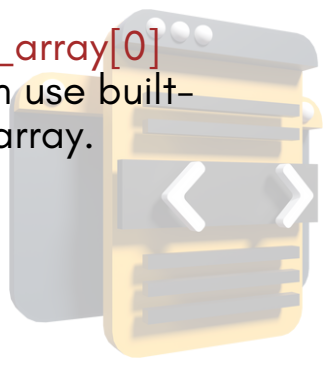
Python is a popular programming language that offers many built-in data structures, such as lists, tuples, and dictionaries.

However, sometimes you may need to implement a custom data structure to suit your specific needs.

In this chapter, we'll cover how to implement some common data structures in Python.

Arrays: In Python, arrays can be implemented using lists. For example, to create an array of integers, you can simply use a list of integers: `my_array = [1, 2, 3, 4, 5]`.

To access an element in the array, you can use indexing: `my_array[0]` will return the first element of the array. Additionally, you can use built-in methods like `append()` to add elements to the end of the array.



Linked Lists: Linked lists can be implemented in Python using classes. Each node in the linked list can be represented by an instance of a node class, which contains a data attribute and a next attribute that points to the next node in the list.

For example, a simple node class could be defined as follows:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

To create a linked list, you can create instances of the Node class and connect them using the next attribute.

Stacks: Stacks can also be implemented using lists in Python. To create a stack, you can simply use a list and perform operations like append() to push elements onto the stack, and pop() to remove elements from the top of the stack.

For example:

```
my_stack = []
my_stack.append(1)  # push element onto stack
my_stack.append(2)
my_stack.pop()  # remove top element from stack
```

Queues: Queues can also be implemented using lists in Python. However, to efficiently remove elements from the front of the queue, you can use the deque class from the collections module.

For example:

```
from collections import deque

my_queue = deque()
my_queue.append(1)  # add element to end of queue
my_queue.append(2)
my_queue.popleft()  # remove element from front of queue
```

Trees: Trees can be implemented in Python using classes. Each node in the tree can be represented by an instance of a node class, which contains data, as well as references to its left and right children.

For example:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

Hash Tables: A hash table is a data structure that maps keys to values.

Hash tables use a hash function to compute an index into an array of buckets or slots, from which the corresponding value can be found.

Hash tables are commonly used for fast lookups and insertions, and are often used in database systems.

Commonly asked questions in FAANG interviews related to Data Structures and Algorithms

1. Given a sorted array, implement a binary search algorithm to find a specific element.

```python
def binary_search(arr, x):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

2. Implement a function to check if a given string is a palindrome.

```python
def is_palinfrome(s):
    return s == s[s::-1]
```

3. Implement a function to reverse a linked list.

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def reverse_linked_list(head):
    prev = None
    curr = head
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node
    return prev
```

4. Given a string, implement a function to find the longest substring that contains only unique characters.

```python
def longest_substring_with_unique_chars(s):
    char_set = set()
    max_len = 0
    start = 0
    for end in range(len(s)):
        while s[end] in char_set:
            char_set.remove(s[start])
            start += 1
        char_set.add(s[end])
        max_len = max(max_len, end - start + 1)
    return max_len
```

5. Given a binary tree, implement a function to find the maximum depth of the tree.

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def max_depth(root):
    if not root:
        return 0
    left_depth = max_depth(root.left)
    right_depth = max_depth(root.right)
    return max(left_depth, right_depth) + 1
```

6. Write a Python program to find whether a queue is empty or not.

```python
import queue
p = queue.Queue()
q = queue.Queue()
for x in range(4):
    q.put(x)
print(p.empty())
print(q.empty())
```

Commonly used Algorithms and their implementation in Python

Bubble Sort

Bubble sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order.

The algorithm passes through the list repeatedly until no more swaps are needed, indicating that the list is sorted.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Selection Sort

Selection sort is another simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the list and putting it at the beginning.

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Insertion Sort

Insertion sort is a simple sorting algorithm that works by building the final sorted array one item at a time. It works by iterating through the array and inserting each element into its proper place in the sorted part of the array.

```python
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr
```

Quick Sort

Quick sort is a divide-and-conquer sorting algorithm that works by selecting a pivot element and partitioning the array into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Breadth-First Search

Breadth-first search (BFS) is an algorithm that traverses a graph or tree by exploring all the nodes at the current depth before moving on to the next level. This algorithm is often used for finding the shortest path between two nodes in an unweighted graph.

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited
```

Depth-First Search

Depth-first search (DFS) is an algorithm that traverses a graph or tree by exploring as far as possible along each branch before backtracking. This algorithm is often used for finding connected components in a graph.

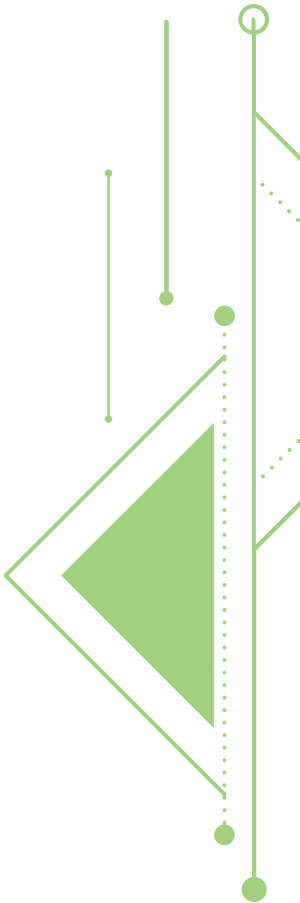
```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for neighbor in graph[start] - visited:
        dfs(graph, neighbor, visited)
    return visited
```

Dijkstra's Algorithm

Dijkstra's algorithm is a shortest-path algorithm that finds the shortest path between two nodes in a graph with non-negative edge weights.

The algorithm maintains a set of visited nodes and a set of tentative distances from the start node to each node in the graph. It repeatedly selects the node with the smallest tentative distance and updates the distances of its neighbors.

```python
import heapq
import math

def dijkstra(graph, start):
    distances = {node: math.inf for node in graph}
    distances[start] = 0
    queue = [(0, start)]
    while queue:
        current_distance, current_node = heapq.heappop(queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))
    return distances
```



The code starts by importing two modules: heapq and math. The heapq module is used to implement priority queues, which are needed for implementing Dijkstra's algorithm.

The math module is used to set initial distances to infinity, since we don't know the actual distances until we calculate them.

```
import heapq
import math

def dijkstra(graph, start):
```

The dijkstra function takes two arguments: graph, which is a dictionary that represents the graph we want to find the shortest path in, and start, which is the starting node for the algorithm.

```
distances = {node: math.inf for node in graph}
distances[start] = 0
queue = [(0, start)]
```

This code initializes the distances dictionary with initial distances set to infinity for each node in the graph. We set the distance for the starting node to 0, since it is the starting point. The queue list is initialized with a tuple containing the starting node and its distance, which is 0.

```
while queue:
 current_distance, current_node = heapq.heappop(queue)
 if current_distance > distances[current_node]:
     continue
 for neighbor, weight in graph[current_node].items():
     distance = current_distance + weight
     if distance < distances[neighbor]:
         distances[neighbor] = distance
         heapq.heappush(queue, (distance, neighbor))
```

This code is a loop that runs until there are no more nodes to visit. In each iteration of the loop, we pop the node with the smallest tentative distance from the queue.

We then check if the node has already been visited. If the node has already been visited, we skip it and move on to the next node.
return distances

Finally, we return the distances dictionary, which contains the shortest distance from the starting node to each node in the graph.

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a way of organizing code that focuses on creating "objects" which have both data and functions associated with them.

In the world of FAANG, OOP is a crucial part of software development. Here are some reasons why:

1. Code reusability

One of the key benefits of OOP is code reusability. By encapsulating data and behavior within objects, we can create reusable code that can be used across different projects and applications.

This saves time and effort by reducing the amount of code that needs to be written, tested, and maintained.

2. Modularity and maintainability

OOP promotes modularity by dividing code into smaller, more manageable chunks. This makes it easier to test, debug, and maintain code over time.

Changes to one part of the code can be made without affecting other parts of the code, which reduces the risk of unintended consequences.

3. Encapsulation

OOP's encapsulation allows for information hiding, which is an important security feature in software development.

By hiding the implementation details of an object from the outside world, we can ensure that data is only accessible in the ways that we want it to be.

This prevents unauthorized access to sensitive data and makes it harder for hackers to exploit vulnerabilities in the code.

4. Polymorphism

Polymorphism is a key feature of OOP that allows objects to take on multiple forms.

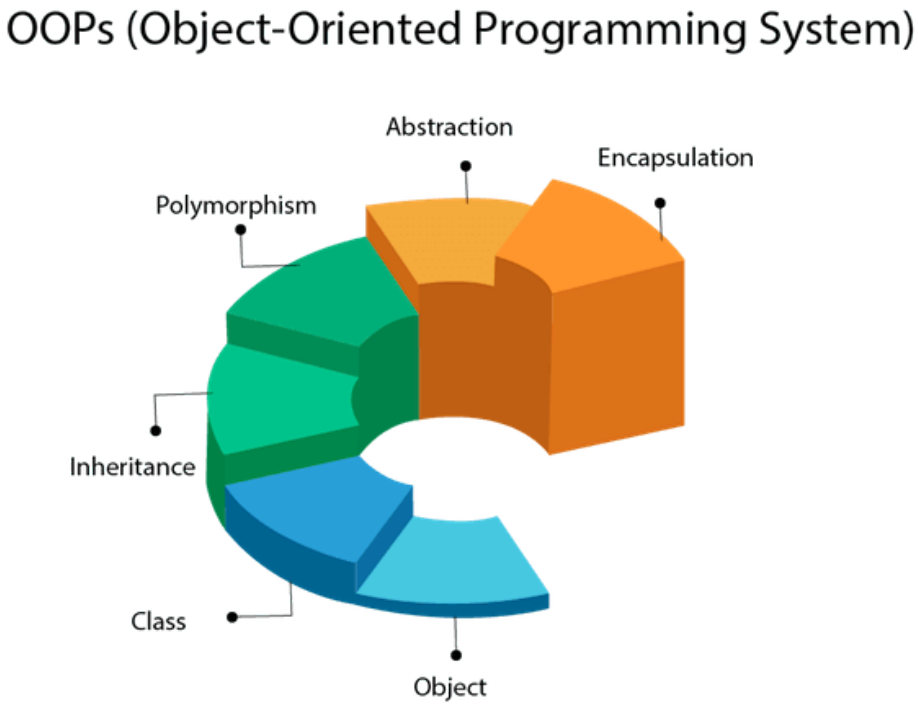
This allows for flexibility in code design and enables developers to write code that is easily extensible and adaptable to changing requirements.

Polymorphism is particularly important in FAANG, where companies need to quickly respond to changes in the market and new technologies.

5. Scalability

OOP allows for code to be scaled easily, which is important in FAANG where companies need to handle large amounts of data and traffic.

By breaking code into smaller modules, it becomes easier to add or remove functionality without disrupting the overall system. This makes it easier to scale applications and systems to handle increasing demand.



Implementation of OOP in Python

1. Classes and Objects

In Python, we create a class using the class keyword, followed by the name of the class. For example, we can define a class called Person:

```
class Person:
    pass
```

This creates an empty class that doesn't do anything yet. We can create objects of this class using the class name and parentheses:

```
person1 = Person()
person2 = Person()
```

These create two separate instances of the Person class.

2. Properties and Methods

We can add properties and methods to our class by defining them within the class definition. For example, we can add a name property to our Person class:

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

This defines a constructor method called __init__ that takes a parameter called name. The self parameter refers to the object that is being created.

We can also add methods to our class. For example, we can add a method called greet that prints a greeting:

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

    def greet(self):
        print("Hello, my name is", self.name)
```

3. Inheritance

We can create a new class that inherits from an existing class by including the name of the existing class in parentheses after the new class name. For example, we can create a Student class that inherits from our Person class:

```
class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)
        self.student_id = student_id

    def greet(self):
        print("Hello, my name is", self.name, "and my student ID is", self.student_id)
```

This defines a new class called Student that inherits from the Person class. It has a new property called student_id, and it overrides the greet method to include the student_id property.

We can now create objects of our Student class and call the greet method:

```
student1 = Student("Alice", 1234)
student2 = Student("Bob", 5678)

student1.greet() # Output: Hello, my name is Alice and my student ID is 1234
student2.greet() # Output: Hello, my name is Bob and my student ID is 5678
```

4. Encapsulation

In Python, we can use the underscore _ to indicate that a property or method is intended to be private.

This means that it is not intended to be accessed or modified directly from outside the class.

```python
class BankAccount:
    def __init__(self, balance):
        self._balance = balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if amount > self._balance:
            print("Insufficient funds")
        else:
            self._balance -= amount

    def get_balance(self):
        return self._balance
```

In this example, we have a BankAccount class that has a private property called _balance which ensures that the balance can only be accessed and modified through the methods provided by the class.

5. Polymorphism

Polymorphism is the ability of objects of different classes to be used interchangeably. In Python, this is achieved through the use of inheritance and method overriding.

```python
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

def make_animal_sound(animal):
    animal.make_sound()

dog = Dog()
cat = Cat()

make_animal_sound(dog)  # Output: Woof!
make_animal_sound(cat)  # Output: Meow!
```

In the previous example, we have an Animal class with a method called make_sound.

We also have two subclasses, Dog and Cat, that override the make_sound method with their own implementations.

Finally, we have a function called make_animal_sound that takes an Animal object as a parameter and calls its make_sound method.

This function can be used with any object that inherits from the Animal class, making it polymorphic.

Frequently asked OOP questions in FAANG interviews

1. What is the difference between a class and an object?

A class is a blueprint for creating objects, while an object is an instance of a class. In other words, a class defines the properties and methods that an object will have, but an object is a specific instance of that class.

2. What is method overloading, and does Python support it?

Method overloading is the ability to define multiple methods with the same name but different parameters. This allows you to create more flexible and intuitive APIs. However, Python does not support method overloading in the traditional sense. Instead, you can use default parameter values or variable-length argument lists to achieve similar functionality.

```python
class Calculator:
    def add(self, x, y, z=0):
        return x + y + z

    def multiply(self, *numbers):
        result = 1
        for number in numbers:
            result *= number
        return result
```

3. Implement a class hierarchy for vehicles

Create a class hierarchy for different types of vehicles, with a base class called Vehicle and subclasses for specific types of vehicles. Each class should have properties and methods that are appropriate for that type of vehicle. Here's an example hierarchy:

- Vehicle
  - Car
    - Sedan
    - SUV
  - Truck
  - Motorcycle

Solution:

```python
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        print("Engine started")

class Car(Vehicle):
    def __init__(self, make, model, year, num_doors):
        super().__init__(make, model, year)
        self.num_doors = num_doors

class Sedan(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year, num_doors=4)

class SUV(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year, num_doors=5)

class Truck(Vehicle):
    def __init__(self, make, model, year, payload_capacity):
        super().__init__(make, model, year)
        self.payload_capacity = payload_capacity

class Motorcycle(Vehicle):
    def __init__(self, make, model, year, engine_size):
        super().__init__(make, model, year)
        self.engine_size = engine_size
```

4. Check type of an object

Write a program to determine which class a given Bus object belongs to.

Given Input:

```
class Vehicle:
    def __init__(self, name, mileage, capacity):
        self.name = name
        self.mileage = mileage
        self.capacity = capacity

class Bus(Vehicle):
    pass

School_bus = Bus("School Volvo", 12, 50)
```

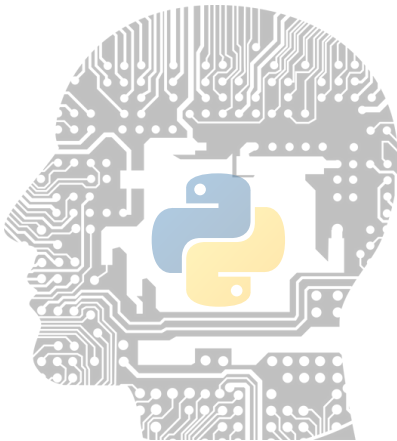
Solution:

```
# Python's built-in type()
print(type(School_bus))
```

5. Create a Class with instance attributes

Write a Python program to create a Vehicle class with max_speed and mileage instance attributes.

```
class Vehicle:
    def __init__(self, max_speed, mileage):
        self.max_speed = max_speed
        self.mileage = mileage

modelX = Vehicle(240, 18)
print(modelX.max_speed, modelX.mileage)
```



Advanced Learning Phase

In this Phase we'll go over Advanced Python Concepts required for FAANG Interview.

1. Decorators

Decorators are a powerful feature in Python that allows you to modify the behavior of a function or class without changing its source code.

Decorators are essentially functions that take another function as input and return a modified version of it.

They are commonly used to implement cross-cutting concerns like logging, authentication, and caching.

1.1 Function decorators

In Python, a function can be decorated by simply placing the decorator syntax (@decorator_name) before the function definition.

```python
def my_decorator(func):
    def wrapper():
        print("Before the function is called.")
        func()
        print("After the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello, world!")

say_hello()
```

In this example, the my_decorator function is used as a decorator for the say_hello function.

The my_decorator function takes the say_hello function as input, and returns a new function wrapper that wraps around the original function.

When the say_hello function is called, the wrapper function is executed instead. The wrapper function first prints "Before the function is called.", then calls the original say_hello function, and finally prints "After the function is called.".

The output of the code above would be:

```
Before the function is called.
Hello, world!
After the function is called.
```

1.2 Class decorators

In addition to function decorators, Python also supports class decorators.

Class decorators are similar to function decorators, but they are applied to classes instead of functions.

```python
def my_class_decorator(cls):
    class MyNewClass:
        def __init__(self, *args, **kwargs):
            print("Before the class is instantiated.")
            self.instance = cls(*args, **kwargs)
            print("After the class is instantiated.")

        def __getattr__(self, name):
            return getattr(self.instance, name)

    return MyNewClass

@my_class_decorator
class MyClass:
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        print(f"Hello, {self.name}!")

my_object = MyClass("John")
my_object.say_hello()
```

In this example, the my_class_decorator function is used as a decorator for the MyClass class.

The my_class_decorator function takes the MyClass class as input, and returns a new class MyNewClass that wraps around the original class.

When an instance of the MyClass class is created, the MyNewClass class is instantiated instead.

The MyNewClass class first prints "Before the class is instantiated.", then creates an instance of the original MyClass class, and finally prints "After the class is instantiated.".

The MyNewClass class also defines the __getattr__ method, which is called when an attribute is accessed on the instance of the MyNewClass class.

This method simply delegates the attribute access to the original MyClass instance.

Output:

```
Before the class is instantiated.
After the class is instantiated.
Hello, John!
```

1.3 Decorators with arguments

Decorators can also take arguments, allowing you to customize their behavior.

On next page you'll see an example of a decorator with arguments:

```python
def repeat(num_repeats):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for i in range(num_repeats):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(num_repeats=3)
def say_hello(name):
print(f"Hello, {name}!")
say_hello("John")
```

In this example, the `repeat` function takes an argument `num_repeats`, and returns a decorator function that takes the original function as input.

When the `say_hello` function is decorated with `@repeat(num_repeats=3)`, the `repeat` function is called with `num_repeats=3`, and the returned decorator function is applied to the `say_hello` function.

The decorated `say_hello` function now prints "**Hello, John!**" three times, since the `wrapper` function is executed three times.

Output:

```
Hello, John!
Hello, John!
Hello, John!
```

1.4 Chaining decorators

Decorators can also be chained together, allowing you to apply multiple decorators to a single function or class. Below is an example:

```python
def my_decorator1(func):
    def wrapper():
        print("Decorator 1: Before the function is called.")
        func()
        print("Decorator 1: After the function is called.")
    return wrapper

def my_decorator2(func):
    def wrapper():
        print("Decorator 2: Before the function is called.")
        func()
        print("Decorator 2: After the function is called.")
    return wrapper

@my_decorator1
@my_decorator2
def say_hello():
    print("Hello, world!")

say_hello()
```

In this example, the say_hello function is decorated with both @my_decorator1 and @my_decorator2.

When the say_hello function is called, the my_decorator2 function is applied first, followed by the my_decorator1 function.

The output of the code above would be:

```
Decorator 1: Before the function is called.
Decorator 2: Before the function is called.
Hello, world!
Decorator 2: After the function is called.
Decorator 1: After the function is called.
```

2. Metaclasses

Python Metaclasses are heavily used in Big Organisations.

Metaclasses in Python are classes that define the behavior of other classes. In other words, they are classes that define the way that classes should be created. It's a bit of a mind-bending concept, but stick with me and I'll try to explain it in simpler terms.

First, let's start with a basic example of a class in Python:

```
class MyClass:
    pass
```

This is a very simple class that doesn't do anything. But what if we wanted to add some behavior to all classes we define in our program? That's where **Metaclasses** come in.

A Metaclass is simply a class that defines the behavior of other classes. To create a Metaclass, we'll define a class that inherits from type:

```
class MyMeta(type):
    def __new__(cls, name, bases, attrs):
        print("Creating class", name)
        return super().__new__(cls, name, bases, attrs)
```

In this example, we defined a **Metaclass** called MyMeta that inherits from type.

The __new__ method is a special method that is called when a new class is being created.

It takes four arguments: cls (the Metaclass itself), name (the name of the class being created), bases (the base classes of the class being created), and attrs (the attributes of the class being created).

In this case, we simply print out a message saying that we're creating a new class, and then we call the __new__ method of the superclass (type) to actually create the class.

Now let's use our Metaclass to create a new class:

```
class MyClass(metaclass=MyMeta):
    pass
```

In this example, we define a new class called MyClass and specify MyMeta as its Metaclass using the metaclass keyword argument. When we run this code, we should see the message "Creating class MyClass" printed to the console.

So what's the point of all this? Well, Metaclasses can be used to add behavior to classes at the time they are defined.

For example, we could modify the __new__ method of our Metaclass to add a new attribute to each class that is created:

```
class MyMeta(type):
    def __new__(cls, name, bases, attrs):
        attrs['my_attribute'] = 42
        return super().__new__(cls, name, bases, attrs)
```

Now when we create a new class using our Metaclass, it will automatically have a new attribute called my_attribute with a value of 42:

```
class MyClass(metaclass=MyMeta):
    pass

print(MyClass.my_attribute)  # Output: 42
```

Metaclasses FAANG Interview Questions

Q. What is a metaclass in Python, and how does it differ from a regular class?

A. A metaclass in Python is the class of a class. It defines how a class should be instantiated and behaves during its creation. While a regular class defines the structure and behavior of its instances, a metaclass defines the structure and behavior of its classes.

Q. Explain the purpose and usage of the __new__ method in a metaclass.

A. The __new__ method in a metaclass is responsible for creating a new class object. It is called before the __init__ method of the metaclass. By overriding __new__, you can customize the class creation process, modify attributes, or perform additional operations.

Q. What are some practical use cases for metaclasses in software development?

A. Metaclasses can be used for various purposes, such as:

- Enforcing coding standards and guidelines.
- Implementing object-relational mapping (ORM) frameworks.
- Creating declarative programming interfaces.
- Implementing automatic resource management or context managers.
- Implementing class registration or plugin systems.

Q. How can metaclasses be used for class registration or automatic generation of class instances?

A. Metaclasses can maintain a registry of classes or automatically perform operations on classes during their creation. This allows for class registration, where classes are automatically added to a registry for easy lookup or instantiation.

Metaclasses can also intercept class creation and modify attributes or behavior based on certain criteria.

3. Magic Functions

Magic functions, also known as special methods or dunder methods, are a set of predefined methods in Python that have special behavior when called.

They are called magic functions because they start and end with double underscores, which gives them a special syntax and behavior.

Magic functions are used to define behavior for built-in operations like addition or comparison, implement iterators and generators, and perform various other tasks.

They provide a way to customize the behavior of built-in operations and create more expressive and powerful Python code.

Let's look at some examples of how magic functions work. Consider a simple class called Point that represents a point in 2D space:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

This class defines an __init__ method that initializes the x and y attributes of a Point object. Now let's add a magic function to this class to make it more expressive:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0}, {1})".format(self.x, self.y)
```

This magic function is called __str__ and is used to return a string representation of the Point object. When we call print() on a Point object, Python automatically calls the __str__ function to convert the object to a string:

```
p = Point(3, 4)
print(p) # prints "(3, 4)"
```

Now let's add another magic function to this class to define the behavior for addition:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0}, {1})".format(self.x, self.y)

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
```

This magic function is called __add__ and is used to define the behavior for addition. It takes two Point objects as arguments and returns a new Point object with the sum of their coordinates.

Now we can add two Point objects together using the + operator:

```
p1 = Point(3, 4)
p2 = Point(5, 6)
p3 = p1 + p2
print(p3) # prints "(8, 10)"
```

As you can see, the __add__ magic function allows us to add two Point objects together using the + operator, which makes the code more expressive and readable.

Magic functions are a fundamental part of Python's object-oriented programming model and are used extensively in many Python libraries and frameworks.

By defining behavior for built-in operations and implementing custom behavior for classes, magic functions allow for more powerful and expressive Python code.

Magic Functions Use Cases:

1. Operator Overloading: Magic functions allow you to define the behavior of operators such as addition, subtraction, and comparison for objects of custom classes.

For example, __add__ can be used to define the behavior of the + operator, and __lt__ can be used to define the behavior of the < operator.

2. String Representation: The __str__ and __repr__ magic functions allow you to define how an object is represented as a string. __str__ is used for the informal string representation (e.g., when you print the object), while __repr__ provides a more detailed and formal string representation.

3. Container Behavior: Magic functions such as __len__, __getitem__, __setitem__, and __delitem__ allow you to define the behavior of objects as containers. This enables you to use indexing and slicing operations on your custom objects.

4. Attribute Access: Magic functions such as __getattr__, __setattr__, and __delattr__ enable you to control attribute access and manipulation for objects. They are useful for implementing custom attribute behavior and validation.

5. Callable Objects: The __call__ magic function allows you to make an object callable like a function. When an object is called using parentheses, the __call__ function is invoked, allowing you to define custom behavior for invoking the object.

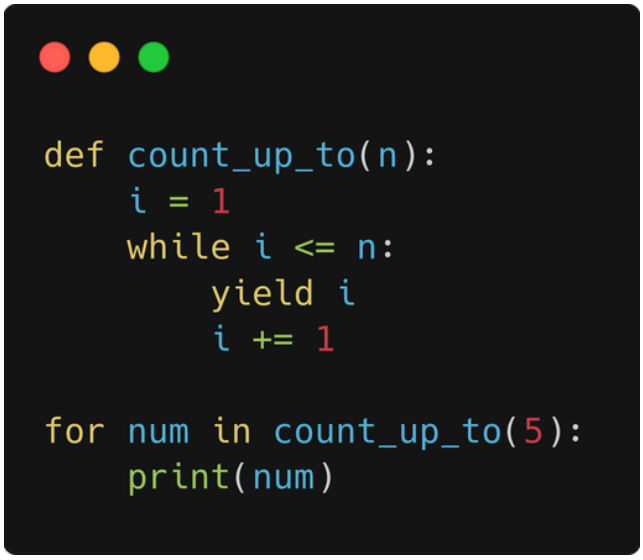
4. Generators

In Python, a generator is a special type of iterator that allows you to iterate over a sequence of values without creating the entire sequence in memory at once.

Generators are particularly useful for working with large datasets or infinite sequences.
In this chapter, we'll cover what generators are, how to create them, and some common use cases for generators.

Creating a generator
Generators are created using the yield keyword. When a function contains a yield statement, it becomes a generator function that returns a generator object.

Here's an example:



```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for num in count_up_to(5):
    print(num)
```

In this example, the count_up_to function is a generator function that generates the sequence of numbers from 1 up to n.

When the yield statement is reached, the current value of i is returned as the next value in the sequence. The function then suspends its execution and waits until the next value is requested.

The for loop then iterates over the generator object returned by count_up_to(5), printing each value in the sequence.

Using generators for infinite sequences

One of the most powerful features of generators is the ability to generate infinite sequences. Since generators generate values on-the-fly, you can create a generator that generates values forever.

Here's an example of a generator that generates an infinite sequence of Fibonacci numbers:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

for num in fibonacci():
    if num > 100:
        break
    print(num)
```

In this example, the fibonacci function generates an infinite sequence of Fibonacci numbers using a while loop.

The yield statement is used to return each value in the sequence as it is generated.

The for loop then iterates over the generator object returned by fibonacci, printing each value until the value exceeds 100.

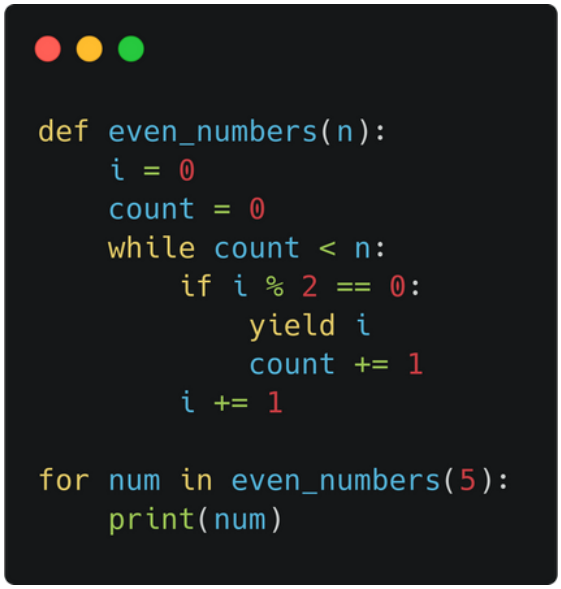
Output:

```
0
1
1
2
3
5
8
13
```

Using generators for lazy evaluation

Generators are also useful for lazy evaluation. In some cases, you may not need to generate all of the values in a sequence at once, and generating them on-the-fly can save time and memory.

Here's an example of a generator that generates the first n even numbers:



```
def even_numbers(n):
    i = 0
    count = 0
    while count < n:
        if i % 2 == 0:
            yield i
            count += 1
        i += 1

for num in even_numbers(5):
    print(num)
```

In this example, the even_numbers function generates the first n even numbers using a while loop.

The yield statement is used to return each even number as it is generated.

The for loop then iterates over the generator object returned by even_numbers, printing each even number.

Output:

```
0
2
4
6
8
```

Using generators with comprehensions

In addition to for loops, generators can also be used with comprehensions.

A generator comprehension is similar to a list comprehension, but instead of creating a list, it creates a generator.

Here's an example of a generator comprehension that generates the squares of the first 5 even numbers:

```
even_squares = (x**2 for x in range(0, 10, 2))
for square in even_squares:
    print(square)
```

In this example, the generator comprehension `(x**2 for x in range(0, 10, 2))` generates the squares of the even numbers from 0 to 8.

The for loop then iterates over the generator object returned by the comprehension, printing each square.

Output of the code:

```
0
4
16
36
64
```

Closing a generator

When you're done with a generator, you can close it by calling the `close` method.

This is useful if you're generating an infinite sequence and want to stop the generator.

Here's an example of a generator that generates an infinite sequence of random numbers:

```python
import random

def random_numbers():
    while True:
        yield random.randint(1, 100)

gen = random_numbers()
for i in range(10):
    print(next(gen))

gen.close()
```

In this example, the random_numbers function generates an infinite sequence of random numbers using the random module.

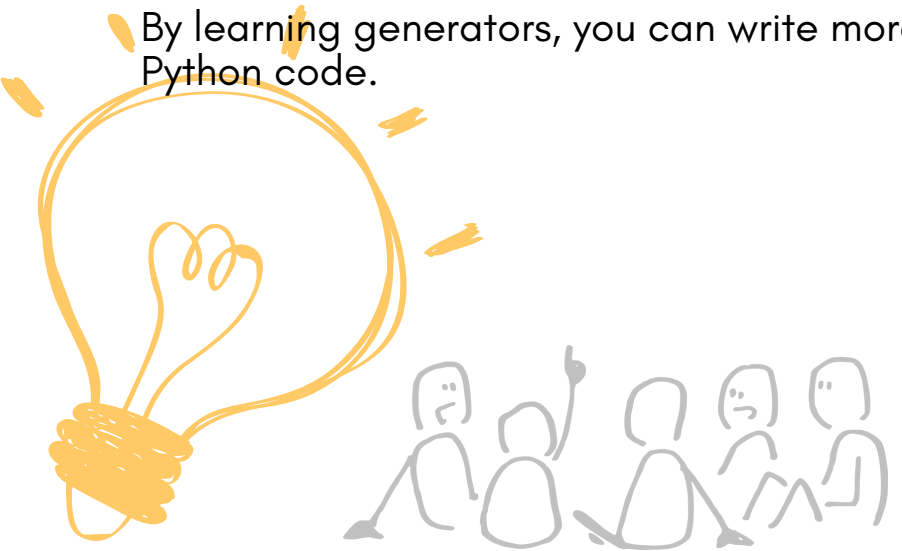
The next function is used to retrieve the next random number from the generator object returned by random_numbers, and the for loop iterates over the first 10 values in the sequence.

The close method is then called to stop the generator.

Generators are a powerful feature of Python that allow you to iterate over a sequence of values without creating the entire sequence in memory at once.

They're particularly useful for working with large datasets or infinite sequences, and can be used with for loops, comprehensions, and lazy evaluation.

By learning generators, you can write more efficient and elegant Python code.



5. Context Managers

Context managers are a powerful feature in Python that allow you to manage resources such as files, locks, and network connections in a clean and concise way.

They provide a convenient way to allocate and release resources automatically, while also handling any exceptions that might occur.

In this chapter, we'll cover what context managers are, how to create them, and how to use them.

What is a context manager?

A context manager is an object that defines the methods `__enter__` and `__exit__`.

The `__enter__` method is called when the block associated with the context manager is entered, and the `__exit__` method is called when the block is exited, whether normally or due to an exception.

Why do you need Context Managers?

Context managers have many use cases in Python, including:

1. File I/O: Opening and closing files using the with statement ensures that the file is properly closed, even if an exception is raised.

2. Database connections: Context managers can be used to open and close database connections, ensuring that resources are properly managed and released.

3. Threading: Context managers can be used to acquire and release locks, allowing multiple threads to safely access shared resources.

4. Network connections: Context managers can be used to manage network connections, ensuring that sockets are properly closed and resources are released.

5. Resource allocation: Context managers can be used to allocate and release any resource that requires cleanup, such as memory or hardware resources.

Here's an example of a simple context manager that prints a message when it's entered and exited:

```
class MyContextManager:
    def __enter__(self):
        print("Entering context...")

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context...")
```

In this example, the MyContextManager class defines the __enter__ and __exit__ methods. The __enter__ method prints a message when the block associated with the context manager is entered, and the __exit__ method prints a message when the block is exited.

Using a context manager

To use a context manager, you can use the with statement.

The with statement automatically calls the __enter__ method of the context manager when the block is entered, and calls the __exit__ method when the block is exited.

Here's an example of using the MyContextManager context manager:

```
with MyContextManager():
    print("Hello, world!")
```

In this example, the with statement creates an instance of MyContextManager, and calls its __enter__ method.

It then executes the code in the block associated with the with statement, which in this case simply prints "Hello, world!".

Finally, the with statement calls the __exit__ method of the context manager.

The output of the code above would be:

```
Entering context...
Hello, world!
Exiting context...
```

Using a context manager with resources

Context managers are particularly useful when working with resources that need to be allocated and released, such as files or network connections.

Here's an example of a context manager that opens a file, writes to it, and then closes it:

```
class FileHandler:
    def __init__(self, filename):
        self.filename = filename
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, 'w')
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        if self.file:
            self.file.close()
```

In this example, the FileHandler context manager opens a file specified by the filename parameter in the __init__ method when the __enter__ method is called.

It returns the file object so it can be used in the block associated with the with statement.

The __exit__ method is responsible for closing the file when the block is exited.

Here's an example of using the FileHandler context manager:

```
with FileHandler('example.txt') as file:
    file.write("Hello, world!")
```

In this example, the with statement creates an instance of FileHandler, and calls its __enter__ method.

It assigns the file object returned by the __enter__ method to the variable `file, which is used to write "Hello, world!" to the file.

Finally, the with statement calls the __exit__ method of the context manager, which closes the file.

Nesting context managers

You can also nest context managers inside other context managers. This is useful when you need to manage multiple resources in a single block of code.

Here's an example of a context manager that locks a mutex:

```
import threading

class Mutex:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        self.lock.acquire()
        return self.lock

    def __exit__(self, exc_type, exc_value, traceback):
        self.lock.release()
```

In the previous example, the Mutex context manager uses a threading.Lock object to acquire and release the mutex.

The __enter__ method acquires the mutex and returns the Lock object so it can be used in the block associated with the with statement.

The __exit__ method releases the mutex when the block is exited.

Below is an example of nesting the FileHandler and Mutex context managers:

```
with Mutex(), FileHandler('example.txt') as (lock, file):
    lock.acquire()
    file.write("Hello, world!")
    lock.release()
```

In this example, we created an instance of Mutex and FileHandler using with, and calls their __enter__ methods.

The as clause assigns the Lock object returned by the Mutex context manager to the variable lock, and the file object returned by the FileHandler context manager to the variable file.

The block associated with the with statement then acquires the lock, writes "Hello, world!" to the file, and releases the lock.

Finally, the with statement calls the __exit__ methods of the context managers in reverse order of their creation, releasing the lock and closing the file.

Context managers are a powerful feature in Python that allow you to manage resources in a clean and concise way.

They provide a convenient way to allocate and release resources automatically, while also handling any exceptions that might occur.

By defining the __enter__ and __exit__ methods in a class, you can create your own context managers for use with the with statement.

6. Multi-threading and Multi-processing

Multi-threading and **Multi-processing** are two techniques used for achieving concurrency in Python.

Concurrency is a property that enables different tasks to execute simultaneously, thus improving the overall performance of the application.

In this chapter, we will explore the concepts of Multi-threading and Multi-processing in Python, their differences, and how to use them in practice.

Multi-threading:

Multi-threading is a technique that enables multiple threads to run concurrently within a single process.

Each thread runs independently, but shares the same memory space and system resources. This makes it a powerful technique for achieving parallelism and improving performance.

To create a new thread in Python, you need to use the threading module. Here is an example of how to create a new thread:

```python
import threading

def my_thread_function():
    print("Starting my thread")
    # code to be executed by the thread
    print("Ending my thread")

# create a new thread
t = threading.Thread(target=my_thread_function)
# start the thread
t.start()
```

In the above example, we first defined a function my_thread_function that represents the code that will be executed by the thread.

We then created a new thread using the Thread class from the threading module, and specify the target parameter as the function we just defined.

Finally, we start the thread using the start method.

Multi-threading can be used for a wide range of applications, such as:

1. GUI applications: Multi-threading can be used to keep the GUI responsive while performing time-consuming tasks in the background.

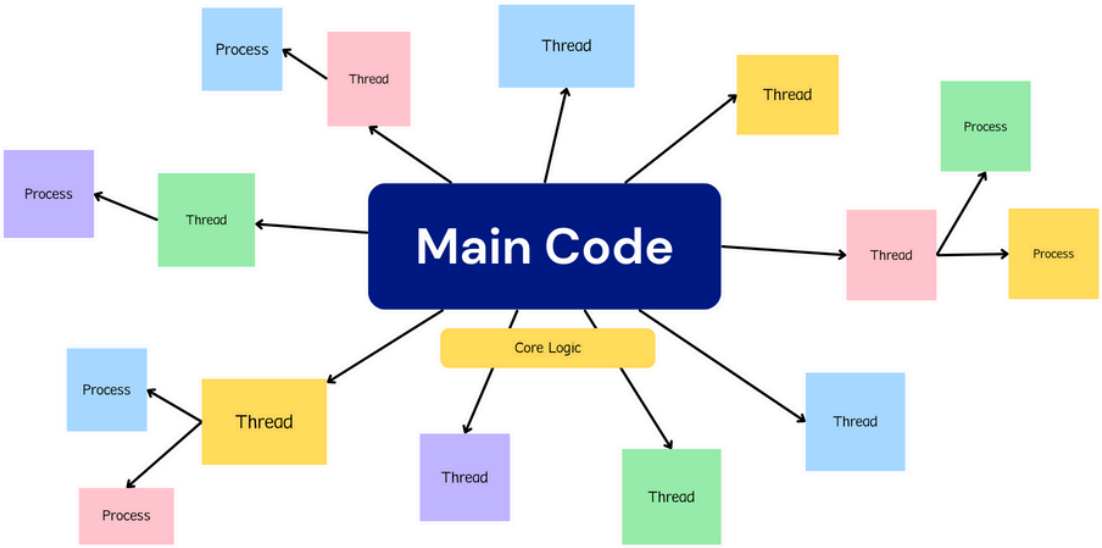
2. Network programming: Multi-threading can be used to handle multiple client requests concurrently.

3. Parallel computation: Multi-threading can be used to speed up computation by distributing the workload across multiple threads.

Multi-processing:

Multi-processing is a technique that enables multiple processes to run concurrently. Each process runs independently and has its own memory space and system resources.

This makes it a powerful technique for achieving parallelism and improving performance.



To create a new process in Python, you need to use the multiprocessing module. Here is an example of how to create a new process:

```python
import multiprocessing

def my_process_function():
    print("Starting my process")
    # code to be executed by the process
    print("Ending my process")

# create a new process
p = multiprocessing.Process(target=my_process_function)
# start the process
p.start()
```

Here, we have a function my_process_function that represents the code that will be executed by the process.

Then a new process is created using the Process class from the multiprocessing module, and specify the target parameter as the function we just defined.

Then we start the process using the start method.

Multi-threading vs Multi-processing

Both Multi-threading and Multi-processing can be used to achieve concurrency and improve performance in Python, but they have different strengths and weaknesses.

Here are some of the differences between them:

1. Memory sharing: Multi-threading shares the same memory space between threads, which can lead to race conditions and other synchronization issues.

Multi-processing, on the other hand, has separate memory spaces for each process, which avoids these issues.

2. CPU-bound vs I/O-bound tasks: Multi-threading is better suited for I/O-bound tasks, where threads can perform other tasks while waiting for I/O operations to complete.

Multi-processing is better suited for CPU-bound tasks, where processes can run on separate CPU cores to achieve faster execution.

3. GIL: Multi-threading in Python is subject to the Global Interpreter Lock (GIL), which can limit the performance benefits of using multiple threads.

Multi-processing is not subject to the GIL, which makes it more suitable for CPU-bound tasks.

4. Overhead: Multi-processing has more overhead than Multi-threading, as each process needs to be created and managed by the operating system.

Multi-threading, on the other hand, has less overhead as threads are managed by the Python interpreter.

Example:

Let's take an example to illustrate the difference between Multi-threading and Multi-processing.

Suppose we have a program that needs to download a large number of files from a remote server. We can use Multi-threading or Multi-processing to achieve concurrency and improve performance.

Here's how to use Multi-threading to download files:

```
import threading
import urllib.request

def download_file(url, file_name):
    with urllib.request.urlopen(url) as response, open(file_name, 'wb') as out_file:
        data = response.read()
        out_file.write(data)

urls = ['https://example.com/file1.txt', 'https://example.com/file2.txt', 'https://example.com/file3.txt']

for url in urls:
    t = threading.Thread(target=download_file, args=(url, url.split('/')[-1]))
    t.start()
```

Code Breakdown:

1. The code begins by importing the required modules: threading for creating and managing threads, and urllib.request for making HTTP requests and downloading files.

2. The download_file function is defined to download a file from a given URL and save it with the specified file name.

It uses the urlopen function from urllib.request to open the URL and retrieve the response. The response is then read and written to a local file using the open function in binary write mode ('wb').

3. A list of URLs is defined in the urls variable. This list contains the URLs from which the files need to be downloaded.

4. A for loop is used to iterate over each URL in the urls list. For each URL, a new thread is created using the threading.

Thread class. The target argument is set to the download_file function, and the args argument is a tuple containing the URL and the file name (extracted from the URL using url.split('/')[-1]).

5. The start method is called on each thread to begin the execution of the download_file function in a separate thread.

Now, Lets see an example of how to use Multi-processing to download files:

```python
import multiprocessing
import urllib.request

def download_file(url, file_name):
    with urllib.request.urlopen(url) as response, open(file_name, 'wb') as out_file:
        data = response.read()
        out_file.write(data)

urls = ['https://example.com/file1.txt', 'https://example.com/file2.txt', 'https://example.com/file3.txt']

for url in urls:
    p = multiprocessing.Process(target=download_file, args=(url, url.split('/')[-1]))
    p.start()
```

The difference betwee 2 codes is that in multi-threading multiple threads are created, and they execute concurrently within the same process.

While in multi-processing multiple processes are created, and they execute independently in separate memory spaces.

7. Asynchronous programming (Async I/0)

Asynchronous programming is a programming method that allows for non-blocking I/O operations and concurrent execution of code.

It is particularly useful for applications that involve network I/O or other I/O-bound tasks, as it allows for faster and more efficient execution of these tasks.

In asynchronous programming, code can continue to execute while waiting for I/O operations to complete. This is achieved through the use of coroutines, which are functions that can be paused and resumed at certain points in their execution.

Coroutines allow for interleaving of multiple tasks in a single thread, which can lead to significant performance improvements over traditional synchronous programming.

In Python, asynchronous programming is typically implemented using the asyncio library, which provides a framework for writing asynchronous code.

What is coroutine?

A **coroutine** is a special type of function in Python that can pause its execution, let other tasks run, and then resume from where it left off.

It's like taking turns for different tasks to do their work cooperatively.

Coroutines are useful for handling tasks that involve waiting for something, like reading from a file or waiting for a network response.

They allow us to write asynchronous and concurrent code that doesn't block the whole program.

In Python, we define a coroutine using the async def syntax. When we call a coroutine, it returns a special object that we can "await" to pause the coroutine until a specific task is finished.

What is asyncio?

Asyncio is a Python library that helps you work with courtine. Asyncio provides the necessary infrastructure and tools to work with coroutines effectively.

Here is an example of a simple asynchronous program using asyncio and courtine:

```python
import asyncio

async def my_coroutine():
    print("Starting coroutine")
    await asyncio.sleep(1)
    print("Coroutine completed")

async def main():
    print("Starting program")
    task = asyncio.create_task(my_coroutine())
    await task
    print("Program completed")

asyncio.run(main())
```

In the above code, there's a coroutine my_coroutine() that prints a message, waits for 1 second using asyncio.sleep(), and then prints another message.

We then define another coroutine main() that creates a task to run my_coroutine() and waits for it to complete using await. Finally, the main() coroutine is run using asyncio.run().

When we run this program, we get the following output:

```
Starting program
Starting coroutine
Coroutine completed
Program completed
```

Understanding async and await

async and await are the most important keywords to handle asynchronous code in Python.

async keyword:
The async keyword is used to define a asynchronous function.

When a function is marked as async, it can contain await expressions and is allowed to yield control to the event loop while waiting for certain operations to complete.

await keyword:
The await keyword is used inside an async function to pause the execution of the function until a coroutine or an awaitable object is complete.

When encountering an await expression, the async function temporarily suspends its execution, allows other tasks to run, and waits for the awaited object to finish.

```python
import asyncio

async def async_function():
    print("Task 1")
    await asyncio.sleep(1)  # Pause execution for 1 second
    print("Task 2")

async def main():
    await asyncio.gather(async_function(), async_function())

asyncio.run(main())
```

In this example, the async_function is defined as an asynchronous coroutine function using the async keyword.

Within the function, await asyncio.sleep(1) pauses the execution for 1 second without blocking the event loop, allowing other tasks to run in the meantime.

The main function uses asyncio.gather to concurrently execute two instances of async_function.

Synchronous vs. Asynchronous

Synchronous Programming:

Synchronous programming is the traditional approach where code execution occurs sequentially, one statement at a time.

Each operation blocks the execution until completion, causing the program to wait.

Key characteristics of synchronous programming include:

1. Sequential Execution:

- In synchronous programming, the flow of execution proceeds line by line, following a linear path.
- Each statement or operation is executed before moving on to the next one, creating a predictable order of execution.

2. Blocking Operations:

- Synchronous operations are blocking, meaning that when an operation is initiated, the program halts until it completes.
- If an operation takes a significant amount of time, the program remains idle, unable to perform any other tasks.

3. Workflow in Synchronous Programs:

- Synchronous programs wait for each operation to complete before moving on to the next one.
- Delays occur when performing I/O operations, such as reading from files or making network requests.
- Resource-intensive operations can significantly slow down the program's execution, leading to poor performance.

Use Cases for Synchronous Programming:

- Simple and linear workflows with minimal I/O operations.
- Situations where simplicity and readability are prioritized over performance.
- Debugging and understanding program flow in a straightforward manner.

Asynchronous Programming:

Asynchronous programming offers an alternative approach that enables concurrent execution and non-blocking operations.

It allows multiple tasks to progress independently, resulting in improved efficiency and responsiveness.

Characteristics of asynchronous programming:

1. Concurrent Execution:

- Asynchronous programming allows multiple operations to progress concurrently, independently of each other.
- Tasks can overlap in their execution, leading to better utilization of system resources and reduced idle time.

2. Non-Blocking Operations:

- In asynchronous programming, operations can initiate and continue without waiting for completion.
- When an operation is initiated, the program doesn't block but instead moves on to the next task.
- Asynchronous operations leverage mechanisms such as callbacks, promises, or coroutines to handle the completion of operations.

3. Workflow in Asynchronous Programs:

- Asynchronous programs can initiate operations concurrently and proceed with other tasks while waiting for operations to complete.
- Non-blocking I/O operations and event-driven architecture enhance responsiveness and enable efficient handling of multiple tasks.
- An event loop manages the execution flow, efficiently switching between tasks and maximizing concurrency.

Use Cases for Asynchronous Programming:

- **I/O-bound Operations:** Asynchronous programming excels in handling operations involving I/O, such as network requests, file operations, and database queries.
- **GUI Applications:** Asynchronous programming helps maintain a responsive user interface while performing time-consuming operations in the background.
- **Real-time Applications:** Applications that require real-time updates, such as chat applications, streaming services, and event-driven systems.

Let's take a simple example program that shows the difference between synchronous and asynchronous execution.

We'll create a program that fetches data from multiple URLs.

Synchronous Execution: In synchronous execution, each URL will be fetched sequentially, one after another.

The program will wait for the response from one URL before moving on to the next.

Here's an example:

```python
import requests

def fetch_data(url):
    response = requests.get(url)
    return response.text

def main():
    urls = [
        'https://example.com',
        'https://google.com',
        'https://github.com'
    ]

    for url in urls:
        data = fetch_data(url)
        print(f"Fetched data from {url}: {len(data)} bytes")

main()
```

In this synchronous program, the main function iterates over the list of URLs and calls the fetch_data function for each URL.

The fetch_data function makes a synchronous HTTP request to fetch the data from the URL using the requests library.

The program waits for the response from each URL before moving on to the next, resulting in sequential execution.

As a result, fetching the data from each URL will take significant time because the program blocks while waiting for the response.

Asynchronous Execution:

In asynchronous execution, the program can fetch data from multiple URLs concurrently, allowing for efficient utilization of resources and responsiveness.

Here's an example using Python's asyncio module:

```python
import asyncio
import aiohttp

async def fetch_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            data = await response.text()
            return data

async def main():
    urls = [
        'https://example.com',
        'https://google.com',
        'https://github.com'
    ]

    tasks = []
    for url in urls:
        task = asyncio.create_task(fetch_data(url))
        tasks.append(task)

    results = await asyncio.gather(*tasks)
    for url, data in zip(urls, results):
        print(f"Fetched data from {url}: {len(data)} bytes")

asyncio.run(main())
```

In this asynchronous program, we defined the fetch_data function as an asynchronous function using the async keyword.

We used the aiohttp library, which provides asynchronous HTTP client functionality. Within the main function, we create a list of tasks, where each task corresponds to fetching data from a specific URL.

We then used asyncio.create_task to create an asynchronous task for each URL. The asyncio.gather function is used to await all the tasks concurrently, and the results are collected.

In the asynchronous program, the program doesn't block while waiting for the response from each URL. Instead, it can proceed with other tasks or URLs while waiting for the I/O operation (fetching data) to complete.

SQLite3 for Python

SQLite3 is a lightweight database management system that is built into Python's standard library. It provides a simple and efficient way to store and manipulate data using SQL (Structured Query Language).

In this chapter, we will cover the basics of SQLite3 and show you how to create, read, update, and delete data using Python.

Key Features of SQLite:

1. Zero Configuration: Unlike other databases, SQLite does not require complex setup or configuration. It works out of the box and is ready to use with minimal effort.

2. Serverless Architecture: SQLite doesn't rely on a separate database server. Instead, it directly accesses the database file stored on the local machine.

This makes it lightweight and eliminates the need for managing a separate database server.

3. High Performance: SQLite is designed for efficiency and speed. It uses a transactional write-ahead log mechanism that allows for fast and concurrent access to the database.

Additionally, SQLite employs various optimization techniques to execute queries quickly.

4. Extensibility: SQLite provides extensibility through user-defined functions, collations, and virtual tables.

This allows developers to enhance the functionality of SQLite by integrating custom logic and extending the SQL capabilities.

Installing SQLite

Step 1 - Go to SQLite download page, and download precompiled binaries for your OS.

Step 2 - Unzip the downloaded file in the same folder as your Python program.

Step 3 - Import SQLite3 into your program using the command below.

```
import sqlite3
```

The first step in using SQLite3 is to establish a connection to a database file. If the file does not exist, SQLite3 will create it for you.

To establish a connection, you can use the connect() method:

```
conn = sqlite3.connect('example.db')
```

This creates a new database file called example.db and establishes a connection to it.

If you want to connect to an existing database file, simply provide the filename as an argument to the connect() method.

Once you have established a connection, you can create a cursor object to execute SQL commands:

```
c = conn.cursor()
```

Now let's create a table in our database. Here is an example of how to create a users table with three columns:

```
c.execute('''CREATE TABLE users
             (id INTEGER PRIMARY KEY,
              name TEXT,
              age INTEGER)''')
```

This SQL command creates a table with three columns: id, name, and age. The id column is defined as the primary key, which means that each row in the table will have a unique value for id.

Now let's insert some data into the table:

```
c.execute("INSERT INTO users (name, age) VALUES ('John Doe', 30)")
c.execute("INSERT INTO users (name, age) VALUES ('Jane Smith', 25)")
c.execute("INSERT INTO users (name, age) VALUES ('Bob Johnson', 40)")
```

These SQL commands insert three rows into the users table with different values for name and age.

To retrieve data from the table, you can use the SELECT statement:

```
c.execute("SELECT * FROM users")
rows = c.fetchall()
for row in rows:
    print(row)
```

This SQL command retrieves all rows from the users table, and the fetchall() method returns a list of tuples containing the data.

The for loop then prints each row to the console.

To update data in the table, you can use the **UPDATE** statement:

```
c.execute("UPDATE users SET age = 35 WHERE name = 'John Doe'")
```

This SQL command updates the age column of the row with name equal to 'John Doe' to 35 using **WHERE** statement.

To delete data from the table, you can use the **DELETE** statement:

```
c.execute("DELETE FROM users WHERE name = 'Bob Johnson'")
```

This SQL command deletes the row with name equal to 'Bob Johnson' from the users table.

Sorting Data using **ORDER BY** Clause: The ORDER BY clause is used to sort the result set based on one or more columns in ascending or descending order. Here's an example:

```
SELECT name, age FROM users ORDER BY name ASC, age DESC;
```

Joining Multiple Tables:

In relational databases, you often have multiple tables with relationships between them. The **JOIN** operation allows you to combine rows from different tables based on a related column. Here's an example of an **INNER JOIN**:

```
SELECT users.name, orders.order_number FROM users INNER JOIN orders ON users.id = orders.user_id;
```

The above query executes an INNER JOIN operation to combine rows from two tables, "users" and "orders." based on a shared column, which in this case is the user ID.

Aggregating Data using GROUP BY Clause: The GROUP BY clause is used to group rows based on one or more columns and perform aggregate functions on the grouped data. Here's an example:

```
SELECT age, COUNT(id) FROM users GROUP BY age;
```

This query Groups the users data based on age, COUNT is used to count the number of occurrences of a column value within each group.

Using Subqueries: Subqueries are queries nested within another query. They allow you to perform complex operations by using the result of one query as input to another query. Here's an example:

```
SELECT name FROM users WHERE age IN (SELECT age FROM users WHERE name = 'John');
```

WHERE name = 'John': The subquery selects the age of the user with the name 'John'.
age IN (SELECT age FROM users WHERE name = 'John'): The main query selects the names of all users whose age matches the values returned by the subquery.

Finally, don't forget to commit your changes and close the connection when you're done:

```
conn.commit()
conn.close()
```

Advanced SQLite Features

In addition to its basic querying capabilities, SQLite offers a range of advanced features that can enhance your database operations.

These features provide powerful functionality and flexibility for managing data efficiently.

In this chapter, we will explore some of the advanced features of SQLite, including transactions, indexes, triggers, and views.

Transactions: Ensuring Data Integrity

Transactions are a fundamental concept in database management systems that ensure data integrity and consistency.

A transaction represents a sequence of database operations that are executed as a single unit.

The ACID properties (Atomicity, Consistency, Isolation, Durability) guarantee that either all operations in a transaction are completed successfully, or none of them are applied to the database.

Let's consider an example of transferring funds between two bank accounts using transactions:

```python
import sqlite3

# Establish a connection to the database
conn = sqlite3.connect('database.db')

try:
    # Begin a transaction
    conn.execute('BEGIN')

    # Perform the debit operation
    conn.execute("UPDATE accounts SET balance = balance - 500 WHERE account_number = 'A123'")

    # Perform the credit operation
    conn.execute("UPDATE accounts SET balance = balance + 500 WHERE account_number = 'B456'")

    # Commit the transaction
    conn.execute('COMMIT')
except:
    # Rollback the transaction in case of any error
    conn.execute('ROLLBACK')

# Close the connection
conn.close()
```

In the previous example, we use the BEGIN, COMMIT, and ROLLBACK statements to manage the transaction.

The debit operation decreases the balance of account 'A123' by 500 units, while the credit operation increases the balance of account 'B456' by 500 units.

If any error occurs during the transaction, the ROLLBACK statement ensures that all changes made within the transaction are rolled back, maintaining the consistency of the data.

Indexes: Optimizing Query Performance

Indexes in SQLite are data structures that improve the efficiency of queries by allowing faster data retrieval based on specific columns.

An index provides a quick lookup mechanism, similar to the index of a book, which helps SQLite locate relevant data more efficiently.

Indexes are particularly useful when working with large datasets or frequently queried columns.

To create an index on a column, consider the following example:

```
CREATE INDEX idx_users_age ON users (age);
```

In this case, we create an index named idx_users_age on the age column of the users table. This index improves the performance of queries that involve filtering or sorting by the age column.

Triggers: Automating Actions on Database Events

Triggers are special types of stored procedures that automatically execute in response to specific database events, such as inserts, updates, or deletes on a table.

Triggers enable you to define custom logic that automatically executes when certain conditions are met, providing automation and enforcing business rules within the database.

Let's take an example of a trigger that updates a timestamp column whenever a new row is inserted into a table:

```
CREATE TRIGGER update_timestamp AFTER INSERT ON events
BEGIN
    UPDATE events SET created_at = DATETIME('now') WHERE rowid = new.rowid;
END;
```

In this example, the trigger named update_timestamp is created after an insert operation on the events table.

The trigger executes the SQL statement within the BEGIN and END block, which updates the created_at column of the inserted row with the current timestamp.

Views: Simplifying Complex Queries

Views are virtual tables that present the results of a query as a named table. They allow you to simplify complex queries, encapsulate logic, and provide a convenient way to retrieve specific data subsets.

Lets create a view to retrieve the names and ages of users older than 30 from the "users" table:

```
CREATE VIEW older_users AS
SELECT name, age FROM users WHERE age > 30;
```

In this example, we create a view named "older_users" that selects the "name" and "age" columns from the "users" table, filtering for users with an age greater than 30.

Once the view is created, you can query it just like any other table:

```
SELECT * FROM older_users;
```

This query will retrieve the names and ages of users who are older than 30 from the "older_users" view.

Views offer several advantages. They provide an abstraction layer that hides the underlying complexity of the query, making it easier to work with specific subsets of data.

Additionally, views can be used to enforce security measures by restricting access to sensitive columns or rows.

SQLite3 FAANG Interview Questions

Q. List out the standard SQLite commands.

A. SELECT, CREATE, INSERT, UPDATE, DROP, DELETE

Q. Explain what is SQLite transactions?

A. Transactions are a fundamental concept in database management systems that ensure data integrity and consistency.

A transaction represents a sequence of database operations that are executed as a single unit.

4. How will you Join multiple tables in SQLite?

A. By using INNER JOIN

5. What are Triggers in SQLite?

A. Triggers are special types of stored procedures that automatically execute in response to specific database events, such as inserts, updates, or deletes on a table.

Q. What does ACID stands for?

Atomicity: Transactions are atomic, meaning that either all the operations within a transaction are executed successfully, or none of them are applied to the database.

If any part of a transaction fails, the entire transaction is rolled back, and the database remains unchanged.

Consistency: Transactions maintain the consistency of the data. The database transitions from one consistent state to another consistent state, ensuring that all integrity constraints and business rules are satisfied.

Isolation: Transactions provide isolation, meaning that each transaction is executed independently and does not interfere with other transactions.

Concurrent transactions are isolated from each other to avoid conflicts and ensure data integrity.

Durability: Once a transaction is committed successfully, its changes are durable and will persist even in the event of system failure.

The changes are saved to the database's permanent storage.

Q. Mention what are the SQLite storage classes?

A. SQLite storage classes include

- **Null:** The value is a NULL value
- **Integer:** The value is a signed integer (1,2,3, etc.)
- **Real:** The value is a floating point value, stored as an 8 byte IEEE floating point number
- **Text:** The value is a text string, stored using the database encoding (UTF-8, UTF-16BE)
- **BLOB (Binary Large Object):** The value is a blob of data, exactly stored as it was input

Natural Language Processing (NLP)

Natural Language Processing, or NLP, is a field of artificial intelligence that focuses on the interaction between computers and human language.

NLP can be used to analyze, interpret, and generate natural language text or speech, and it has many practical applications such as chatbots, language translation, sentiment analysis, and more.

In this chapter, we will introduce you to the basics of NLP and show you how to perform some common NLP tasks using Python.

One of the most important steps in NLP is tokenization, which involves breaking down text into individual words or phrases.

To perform tokenization in Python, we can use the nltk library:

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize

text = "Hello, how are you today?"
tokens = word_tokenize(text)
print(tokens)
```

This code imports the nltk library and downloads the necessary punkt data for tokenization.

The word_tokenize function is then used to break down the text into individual words, which are stored in the tokens list.

The output of the code will be:

```
['Hello', ',', 'how', 'are', 'you', 'today', '?']
```

Another important NLP task is part-of-speech tagging, which involves labeling each word in a text with its corresponding part of speech (e.g., noun, verb, adjective, etc.).

To perform part-of-speech tagging in Python, we can again use the nltk library:

```
import nltk
nltk.download('averaged_perceptron_tagger')
from nltk import pos_tag

text = "I like to eat pizza"
tokens = word_tokenize(text)
tags = pos_tag(tokens)
print(tags)
```

This code imports the necessary averaged_perceptron_tagger data for part-of-speech tagging.

The pos_tag function is then used to label each word in the tokens list with its corresponding part of speech, and the resulting list of tuples is stored in the tags variable.

The output of the code will be:

```
[('I', 'PRP'), ('like', 'VBP'), ('to', 'TO'), ('eat', 'VB'), ('pizza', 'NN')]
```

Here, PRP stands for personal pronoun, VBP stands for verb in present tense, TO stands for the word "to", VB stands for verb, and NN stands for noun.

Sentiment analysis is another popular NLP task that involves determining the emotional tone of a text.

In Python, we can use the TextBlob library for sentiment analysis:

```
from textblob import TextBlob

text = "I love going to the beach!"
blob = TextBlob(text)
sentiment = blob.sentiment.polarity
print(sentiment)
```

This code imports the TextBlob library and creates a TextBlob object from the text string.

The sentiment.polarity attribute is then used to calculate the sentiment score of the text, which ranges from -1 (negative) to 1 (positive).

The output of the code will be:

0.5

Here, the sentiment score of the text is positive, which indicates a positive emotional tone.

Mastering Text Processing with NLTK

Text preprocessing is a crucial step in NLP, and NLTK offers a plethora of tools to tackle it with finesse.

Tokenization, the process of breaking text into individual words or tokens, is one of the first steps.

NLTK has various tokenizers to cater to different needs, such as word tokenization, sentence tokenization, and more.

For example, let's tokenize a sentence using NLTK:

```
from nltk import pos_tag
from nltk.tokenize import word_tokenize

sentence = "I love using NLTK!"
tokens = word_tokenize(sentence)
tags = pos_tag(tokens)
print(tags)
```

Output:

```
['NLTK', 'is', 'amazing', '!']
```

Impressive, isn't it? But NLTK doesn't stop there. It offers lemmatization, stemming, and other text processing techniques to refine your data further.

Whether you want to reduce words to their base forms or trim them down to their roots, NLTK has got you covered.

Unveiling Linguistic Secrets with NLTK's Models

NLTK not only equips you with data and processing tools but also offers various pre-trained models that can unravel linguistic mysteries. One of the most popular applications is part-of-speech tagging.

NLTK provides ready-to-use models to determine the grammatical category of each word in a sentence.

Let's unleash the power of part-of-speech tagging using NLTK:

```
from nltk import pos_tag
from nltk.tokenize import word_tokenize

sentence = "I love using NLTK!"
tokens = word_tokenize(sentence)
tags = pos_tag(tokens)
print(tags)
```

Output:

```
[('I', 'PRP'), ('love', 'VBP'), ('using', 'VBG'), ('NLTK', 'NNP'), ('!', '.')]
```

With NLTK, you can go beyond parts of speech and explore other models like named entity recognition and sentiment analysis.

It's like having an expert linguist at your beck and call!

Pandas

Pandas is the most popular data manipulation and analysis library in Python. It provides support for working with structured data, such as tables and time series.

In this chapter, we'll explore the basics of Pandas and how it can be used to work with data.

Pandas provides a powerful and flexible framework for data manipulation and analysis, and it is widely used in data science, finance, and other industries that rely on data analysis.

Its popularity is mainly due to its ease of use, flexibility, and speed.

The library provides two primary data structures: **Series** and **DataFrame**. The Series is a one-dimensional labeled array that can hold any data type.

A DataFrame, on the other hand, is a two-dimensional labeled data structure with columns of potentially different types. It is similar to a spreadsheet or a SQL table.

Pandas is also compatible with other Python libraries, such as NumPy, Matplotlib, and Scikit-learn, which makes it an integral part of the Python data analysis ecosystem.

Installation

To use Pandas, you first need to install it. You can install Pandas using pip, the Python package manager:

```
pip install pandas
```

Once you have installed Pandas, you can import it into your Python program using the following command:

```
import pandas as pd
```

Getting Started with Pandas.

1. Series

A Series is a one-dimensional array-like object in Pandas. It can hold any data type, such as integers, strings, or even Python objects.

Each element in a Series has a label, which can be used to access the element.

Example of how to create a Series:

```
import pandas as pd

s = pd.Series([1, 2, 3, 4, 5])
print(s)
```

Output:

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

In this example, we created a Series with five elements, each containing a different integer. Pandas automatically assigned an index to each element, starting from 0.

You can also create a Series with custom labels:

```
s = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
print(s)
```

Output:

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

In this example, we created a Series with the same data as before, but with custom labels.

2. DataFrames

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types.

You can think of it like a spreadsheet or SQL table, or a dictionary of Series objects. It is the most commonly used object in pandas.

You can create a DataFrame in many ways, but the most common is to pass a dictionary of equal-length lists or NumPy arrays.

Creating DataFrames

Here are a few ways to create a DataFrame:

1. From a dictionary of lists

You can create a DataFrame from a dictionary of lists where each key in the dictionary represents a column, and the list contains the data for the column.

```
import pandas as pd

data = {'name': ['John', 'Sara', 'David', 'Jessica'],
        'age': [25, 33, 18, 42],
        'country': ['USA', 'Canada', 'France', 'Spain']}

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

Output:

```
     name  age country
0    John   25     USA
1    Sara   33  Canada
2   David   18  France
3  Jessica   42   Spain
```

2. From a list of dictionaries

You can create a DataFrame from a list of dictionaries where each dictionary represents a row in the DataFrame.

```
import pandas as pd

data = [{'name': 'John', 'age': 25, 'country': 'USA'},
        {'name': 'Sara', 'age': 33, 'country': 'Canada'},
        {'name': 'David', 'age': 18, 'country': 'France'},
        {'name': 'Jessica', 'age': 42, 'country': 'Spain'}]

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

Output:

```
      name  age country
0     John   25     USA
1     Sara   33  Canada
2    David   18  France
3  Jessica   42   Spain
```

3. From a CSV file

You can also create a DataFrame from a CSV file using the read_csv function.

```
import pandas as pd

df = pd.read_csv('example.csv')
print(df)
```

Accessing Data in DataFrames

1. Accessing Columns

You can access a column in a DataFrame using the [] operator.

```
import pandas as pd

data = {'name': ['John', 'Sara', 'David', 'Jessica'],
        'age': [25, 33, 18, 42],
        'country': ['USA', 'Canada', 'France', 'Spain']}

df = pd.DataFrame(data)
print(df['name'])
```

Output:

```
0        John
1        Sara
2       David
3     Jessica
Name: name, dtype: object
```

2. Accessing Rows

You can access a row in a DataFrame using the loc[] method. The loc[] method accepts the index label of the row to be accessed.

```
import pandas as pd

data = {'name': ['John', 'Sara', 'David', 'Jessica'],
        'age': [25, 33, 18, 42],
        'country': ['USA', 'Canada', 'France', 'Spain']}

df = pd.DataFrame(data)
print(df.loc[1])
```

Output:

```
name         Sara
age            33
country    Canada
Name: 1, dtype: object
```

Viewing Data in Dataframes

Once we have created a dataframe, we can view its contents using various methods.

The head() method returns the first n rows of the dataframe (by default, n=5).

```
print(df.head())
```

Output:

```
      name  age           city
0    Alice   25       New York
1      Bob   30  San Francisco
2  Charlie   35    Los Angeles
3    David   40        Chicago
```

The tail() method returns the last n rows of the dataframe.

```
print(df.tail())
```

Output:

```
      name  age           city
0    Alice   25       New York
1      Bob   30  San Francisco
2  Charlie   35    Los Angeles
3    David   40        Chicago
```

We can also use the describe() method to get summary statistics for the numerical columns in the dataframe.

```
print(df.describe())
```

Output:

```
             age
count   4.000000
mean   32.500000
std     6.454972
min    25.000000
25%    28.750000
50%    32.500000
75%    36.250000
max    40.000000
```

Indexing and Selecting Data

One of the most important features of dataframes is the ability to select and manipulate subsets of the data.

Pandas provides many methods for selecting data based on various criteria.

We can select a single column by its name using bracket notation.

```
data = {'name': ['John', 'Alice', 'Bob'], 'age': [25, 30, 35], 'gender': ['male', 'female', 'male']}
df = pd.DataFrame(data)
print(df['name'])
```

Output:

```
0       Alice
1         Bob
2     Charlie
3       David
Name: name, dtype: object
```

We can select multiple columns by passing a list of column names.

print(df[['name', 'age']])

```
      name  age
0    Alice   25
1      Bob   30
2  Charlie   35
3    David   40
```

We can also select rows based on a particular condition using boolean indexing.

print(df[df['age'] > 30])

Output:

```
      name  age           city
2  Charlie   35    Los Angeles
```

Let's take another example where we have to filter data based on multiple conditions.

Let's say we want to filter out the data for the year 2020, and only for the 'California' state.

```
filtered_data = data[(data['year'] == 2020) & (data['state'] == 'California')]
```

This code creates a new DataFrame that contains only the rows where the 'year' column is equal to 2020 and the 'state' column is equal to 'California'.

The & operator is used to combine the two conditions.
We can also group the data based on a particular column and perform some operations on it.

For example, let's say we want to find the average 'price' of each 'product' in our data.

```
grouped_data = data.groupby('product').mean()['price']
```

This code groups the data by the 'product' column and calculates the mean of the 'price' column for each group. The result is a new Series that contains the average price for each product.

Applying Functions

Pandas enables us to apply functions to our data, either to individual columns or the entire DataFrame.

For example, suppose we have a DataFrame with a column of prices, and we want to apply a function that calculates the discounted price for each item.

We can use the apply() method along with a custom function to achieve this:

```
def calculate_discounted_price(price):
    return price * 0.8  # Applying 20% discount

df['discounted_price'] = df['price'].apply(calculate_discounted_price)
```

Here, we define a custom function calculate_discounted_price() that takes a price as input and returns the discounted price.

Then, the apply() method is used to apply this function to the 'price' column, creating a new column 'discounted_price' with the calculated values.

Handling Missing Data in Pandas

Missing data is a common issue in real-world datasets. Pandas provides various methods to handle missing data effectively.

For example, we can use the dropna() method to remove rows or columns with missing values:

```
cleaned_data = df.dropna()  # Drop rows with any missing values

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

Alternatively, we can use the fillna() method to fill missing values with appropriate substitutes.

For instance, we can fill missing values in a column with the mean value of that column:

```
mean_value = df['column_name'].mean()
df['column_name'] = df['column_name'].fillna(mean_value)
```

Merging and Joining Data in Pandas

Merging and joining data involve combining multiple datasets based on common columns or indices.

This is particularly useful when we have related information spread across different datasets. Pandas provides functions to perform merges and joins between DataFrames.

Let's explore these operations with examples.

Consider two DataFrames, df1 and df2, representing customer orders and customer information, respectively:

```
df1:

   order_id  customer_id  order_date  total_amount
0         1            1  2022-01-01           100
1         2            2  2022-01-02            80
2         3            1  2022-01-03           150

df2:

   customer_id    customer_name
0            1         John Doe
1            2     Jane Johnson
2            3  Michael Jackson
```

Concatenation

Concatenation allows us to combine datasets along a particular axis, either horizontally (adding columns) or vertically (adding rows).

Let's concatenate df1 and df2 vertically:

```
concatenated_data = pd.concat([df1, df2], axis=0)
```

The resulting DataFrame, concatenated_data, will have the rows from df1 followed by the rows from df2.

The columns will be aligned based on their labels.

Merging and Joining 2 datasets

Merging and joining involve combining datasets based on common columns or indices.

For example, let's merge df1 and df2 based on the 'customer_id' column:

```
merged_data = pd.merge(df1, df2, on='customer_id')
```

The resulting DataFrame, merged_data, will have all the columns from both DataFrames, with rows matched based on the common 'customer_id' column.

Pandas FAANG Interview Questions

Q. Explain the difference between iloc and loc in Pandas

A. iloc and loc are both indexing techniques in Pandas used to select subsets of data from a DataFrame. However, they differ in how they reference and access the data.

iloc is used for integer-based indexing. It allows you to select rows and columns based on their integer positions, similar to how indexing works in Python lists.

The syntax for using iloc is df.iloc[row_index, column_index], where row_index and column_index can be single integers, slices, or boolean arrays.

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
print(df.iloc[0])  # Selects the first row
print(df.iloc[:, 1])  # Selects the second column
print(df.iloc[0:2, 1:3])  # Selects a subset of rows and columns
```

loc is used for label-based indexing. It allows you to select rows and columns based on their labels (index and column names). The syntax for using loc is df.loc[row_label, column_label], where row_label and column_label can be single labels, slices, or boolean arrays.

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}, index=['X', 'Y', 'Z'])
print(df.loc['X'])  # Selects the row with label 'X'
print(df.loc[:, 'B'])  # Selects the column with label 'B'
print(df.loc['X':'Y', 'B':'C'])  # Selects a subset of rows and columns
```

Q. What is the purpose of the groupby() function in Pandas?

A. The purpose of the groupby() function in Pandas is to group data in a DataFrame based on one or more columns, allowing for efficient and flexible data aggregation and analysis.

It enables the application of various aggregate functions, such as sum, mean, count, max, min, etc., on grouped subsets of data.

Q. How to create new columns derived from existing columns in Pandas?

A. We create a new column by assigning the output to the DataFrame with a new column name in between the [].

Let's say we want to create a new column 'C' whose values are the multiplication of column 'B' with column 'A'. The operation will be easy to implement and will be element-wise, so there's no need to loop over rows.

```
df = pd.DataFrame({
  "A": [420, 380, 390],
  "B": [50, 40, 45]
})

df["C"] = df["A"] * df["B"]
```

Q. What are some methods for handling duplicates in Pandas?

A. duplicated(): The duplicated() method identifies duplicate rows in a DataFrame. It returns a boolean Series where True represents a duplicate row. You can then use this Series to filter and remove the duplicate rows.

drop_duplicates(): The drop_duplicates() method removes duplicate rows from a DataFrame. By default, it keeps the first occurrence of each duplicated row and removes the subsequent duplicates. You can also specify specific columns to consider when identifying duplicates.

What is Flask?

Flask is a popular web framework for building web applications using Python. It's a lightweight and flexible framework that allows developers to create web applications quickly and easily.

Flask provides a simple way to define routes, handle requests and responses, and interact with databases.

Flask Use Cases:

1. Building web applications
2. Building RESTful APIs
3. Building web services
4. Building microservices
5. Building real-time apps

Companies Using Flask:

As you can see **FLASK** is pretty popular among Tech companies.

Getting Started with Flask

Now that you have a basic understanding of what Flask is and what it can be used for, it's time to get started with building your first Flask application!

In this chapter, we'll walk you through the process of installing Flask, creating a new Flask application, and getting started with routing and views.

Installing Flask

Before you can start building your Flask application, you'll need to install Flask on your computer.

To do this, you'll need to have Python installed on your computer. Flask can be installed using pip, the package installer for Python.

Here's how you can install Flask using pip:

1. Open your command prompt or terminal.
2. Type "pip install Flask" and press Enter.
3. Wait for pip to download and install Flask.

That's it! Flask is now installed on your computer and you're ready to start building your application.

Creating a New Flask Application

The next step is to create a new Flask application. To do this, you'll need to create a new Python file and import Flask. Here's how you can do it:

Congratulations! You've just created a new Flask application.

Routing and Views

Now that you've created a new Flask application, it's time to get started with routing and views.

Routing is the process of mapping URLs to view functions, while views are the functions that handle HTTP requests and return HTTP responses.

To create a route, you'll need to use the `@app.route` decorator.

```
@app.route('/')
def index():
    return 'Hello, World!'
```

To run your Flask application, you'll need to add the following lines of code at the end of your file:

```
if __name__ == '__main__':
    app.run()
```

This code checks if the current file is being run as the main program (as opposed to being imported by another program) and then starts the Flask development server.

The Flask Development Server

By default, the Flask development server runs on localhost (127.0.0.1) and port 5000.

You can view your application in a web browser by navigating to http://localhost:5000/. If you've defined any routes in your application, you'll be able to access them by appending the route URL to the base URL.

It's important to note that the Flask development server is not suitable for use in production environments.

It's designed for use during development only and is not optimized for performance or security.

When you're ready to deploy your Flask application to a production environment, you'll need to use a production-ready web server like Apache or Nginx. *(More on this in the Deployment Section)*

Working with Templates in Flask

In the previous chapter, I showed you how to create a basic Flask application and get started with routing and views.

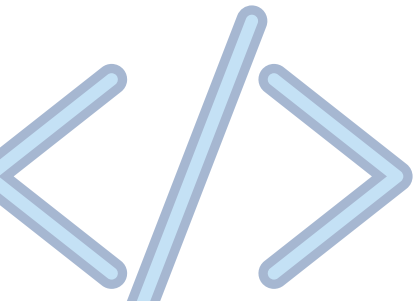
In this chapter, I'll show you how to work with templates in Flask.

Templates are used to generate dynamic HTML pages that can be customized based on data from your application.

Creating a Template

To create a new template in Flask, you'll need to create a new directory called "templates" in your project directory.

Inside the templates directory, create a new HTML file and name it something like "index.html".



```html
<!DOCTYPE html>
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ heading }}</h1>
    <p>{{ message }}</p>
</body>
</html>
```

In this file, we've defined a simple HTML page with a title, heading, and message.

Notice the double curly braces around the variable names - these indicate that the values should be replaced with actual data when the template is rendered.

Rendering a Template

To render a template in Flask, you'll need to use the render_template function.

This function takes the name of the template file and any variables that should be passed to the template.

```python
from flask import render_template

@app.route('/')
def index():
    title = 'Home Page'
    heading = 'Welcome to My Flask App'
    message = 'This is a sample message.'

    return render_template('index.html', title=title, heading=heading, message=message)
```

Here, we've defined a route for the root URL ('/') and created a view function called index.

The view function defines three variables - title, heading, and message - and passes them to the index.html template using the render_template function.

When the index.html template is rendered, the variables are replaced with their corresponding values, resulting in a dynamic HTML page that can be customized based on data from your application.

Passing Data to a Template

To pass data to a template, you'll need to define variables in your view function and pass them to the template using the render_template function.

```
from flask import render_template

@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

In this example, we've defined a route for URLs that include a username ('/user/<name>') and created a view function called user.

The view function takes the username as a parameter and passes it to the user.html template using the render_template function.

When the user.html template is rendered, the variable "name" is replaced with the actual username, resulting in a dynamic HTML page that displays personalized content for each user.

Flask Extensions and Libraries

Libraries are what makes any Framework attractive. Flask has many libraries developed by the Flask community to make developers life easier.

Flask-WTF

Flask-WTF is a Flask extension that provides integration with the WTForms library, which is a popular form handling and validation library for Python web applications.

Flask-WTF makes it easy to handle form validation and rendering in your Flask applications.

Installation

To install Flask-WTF, you can use pip, the Python package manager:

```
pip install Flask-WTF
```

Once you've installed Flask-WTF, you can import it into your Flask application:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField
from wtforms.validators import DataRequired, Email, Length
```

Creating a Form

To create a form using Flask-WTF, you need to define a class that extends FlaskForm and defines the form fields. Here's an example of a login form:

```
class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired(), Length(min=6)])
```

In this example, we've defined a LoginForm class that has two fields: email and password.

The email field is a StringField that requires a valid email address using the Email validator.

The password field is a PasswordField that requires a minimum length of 6 characters using the Length validator.

Handling Form Submission

Once you've defined a form, you can use it to handle form submission in your Flask application.

Here's an example of a login view function that handles form submission using Flask-WTF:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # Handle successful form submission
        email = form.email.data
        password = form.password.data
        # ...
    return render_template('login.html', form=form)
```

Rendering Form Fields

To render form fields in your Flask application, you can use Flask-WTF's template filters.

Here's an example of how to render a login form by passing dynamic values.

```
{% extends 'base.html' %}

{% block content %}
  <h1>Login</h1>
  <form method="POST">
    {{ form.hidden_tag() }}
    <div class="form-group">
      {{ form.email.label(class='control-label') }}
      {{ form.email(class='form-control') }}
    </div>
    <div class="form-group">
      {{ form.password.label(class='control-label') }}
      {{ form.password(class='form-control') }}
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>
{% endblock %}
```

In this example, we've defined a login.html template that extends a base template and includes a login form.

We're using the hidden_tag function to include a CSRF token to protect against cross-site request forgery (CSRF) attacks.

We're also using the label and form controls provided by Flask-WTF to render the form fields.

Flask-SQLAlchemy: Integrating Databases with Flask

Flask-SQLAlchemy is a powerful Flask extension that simplifies the integration of databases into Flask applications.

It provides an easy-to-use interface for interacting with databases using SQLAlchemy, a popular Object-Relational Mapping (ORM) library for Python.

In this chapter, we'll explore how to use Flask-SQLAlchemy to work with databases in your Flask application.

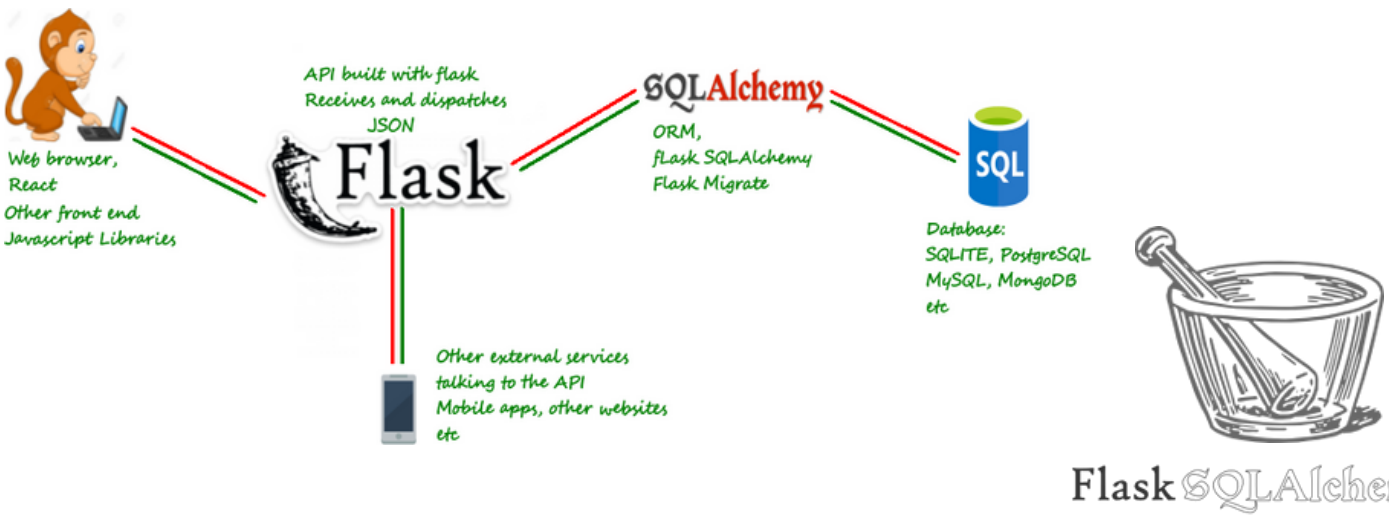
Installation

To get started with Flask-SQLAlchemy, you need to install it using pip, the Python package manager:

```
pip install Flask-SQLAlchemy
```

Once Flask-SQLAlchemy is installed, you can import it into your Flask application:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
```



Setting up the Database

Before you can use Flask-SQLAlchemy, you need to configure your database connection.

Flask-SQLAlchemy supports a variety of databases, including SQLite, MySQL, PostgreSQL, and more.

We'll configure Flask-SQLAlchemy to use SQLite.

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///mydatabase.db'

db = SQLAlchemy(app)
```

we created a Flask application and configure the database URI to use a SQLite database file called mydatabase.db.

You can replace the URI with the appropriate connection string for your chosen database.

Defining Database Models

With Flask-SQLAlchemy, you define database models as Python classes. Each model class represents a table in the database.

A simple User model will look like:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    def __repr__(self):
        return f'<User {self.username}>'
```

In the previous code, we defined a User model that represents a user table in the database.

The id, username, and email columns are defined using SQLAlchemy's Column class.

The __repr__ method provides a string representation of the User object, which is useful for debugging.

Creating the Database

To create the database tables based on your defined models, you can use Flask-SQLAlchemy's create_all method.

You typically call this method from your Flask application:

the create_all method creates the necessary tables in the database based on the defined models.

The if __name__ == '__main__': block ensures that the create_all method is only called when the script is run directly (not imported).

Performing Database Operations

Once you've defined your models and created the database tables, you can use Flask-SQLAlchemy to perform various database operations, such as inserting, updating, querying, and deleting records.

Inserting a Record:

```
user = User(username='john_doe', email='john@example.com')
db.session.add(user)
db.session.commit()
```

Updating a Record:

```
user = User.query.filter_by(username='john_doe').first()
user.email = 'john_new@example.com'
db.session.commit()
```

Querying Records:

```
users = User.query.all()  # Retrieve all users
user = User.query.filter_by(username='john_doe').first()  # Retrieve a specific user
```

Deleting a Record:

```
user = User.query.filter_by(username='john_doe').first()
db.session.delete(user)
db.session.commit()
```

Relationships between Models

Flask-SQLAlchemy also allows you to define relationships between different models, enabling you to establish connections and associations between tables in the database.

For example, let's consider a simple relationship where a User can have multiple Posts:

```python
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    content = db.Column(db.Text, nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    user = db.relationship('User', backref=db.backref('posts', lazy=True))

    def __repr__(self):
        return f'<Post {self.title}>'
```

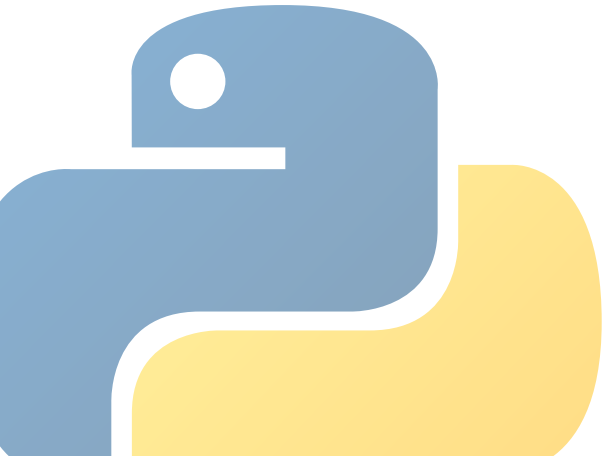
we've added a new Post model that has a foreign key user_id referencing the id column of the User model.

The user attribute is defined as a relationship, specifying that each Post is associated with a User object.

The backref argument allows us to access the user's posts from the User object.

With this relationship defined, we can easily access a user's posts:

```python
user = User.query.filter_by(username='john_doe').first()
posts = user.posts  # Access the posts associated with the user
```



Advanced Database Queries

Flask-SQLAlchemy provides a powerful query interface that allows you to perform complex database queries using SQLAlchemy's query API.

You can filter, sort, join tables, and apply various conditions to retrieve the desired data.

Here's an example of a more advanced query that retrieves all users who have posted in the last 7 days:

```
from datetime import datetime, timedelta

week_ago = datetime.now() - timedelta(days=7)
recent_users = User.query.join(Post).filter(Post.timestamp >= week_ago).all()
```

In this example, we've used the join method to combine the User and Post tables and applied a filter condition to retrieve only those users who have posts with a timestamp in the last 7 days.

Another Example:

```
from flask_sqlalchemy import SQLAlchemy

# Initialize SQLAlchemy
db = SQLAlchemy(app)

# Define models
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    email = db.Column(db.String(100))
    orders = db.relationship('Order', backref='user', lazy=True)

class Order(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    product = db.Column(db.String(100))
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

# Perform a join query
result = db.session.query(User, Order).join(Order).filter(User.name == 'John').all()
```

Here, we've defined two models, User and Order, where User has a one-to-many relationship with Order. We can use SQLAlchemy's join() method to perform a join query between the User and Order tables based on a specific condition.

Flask-Login: User Authentication and Session Management

Flask-Login is a Flask extension that provides user authentication and session management capabilities to your Flask applications.

It simplifies the process of handling user logins, managing user sessions, and restricting access to certain routes based on user authentication status.

In this chapter, we'll explore how to use Flask-Login to implement user authentication in your Flask application.

Installation

To get started with Flask-Login, you need to install it using pip, the Python package manager:

```
pip install Flask-Login
```

Once Flask-Login is installed, you can import it into your Flask application:

```
from flask import Flask
from flask_login import LoginManager
```

Initializing Flask-Login

You can initialize Flask-Login in your Flask application.

```
app = Flask(__name__)
app.secret_key = 'your_secret_key'

login_manager = LoginManager()
login_manager.init_app(app)
```

Here's a Flask application and set a secret key. The secret key is used to encrypt session cookies and should be kept secure.

We then defined the LoginManager object and associate it with our Flask application using the init_app method.

Defining a User Model

Flask-Login requires a user model that represents your application's users.

This model should implement certain methods and attributes that Flask-Login uses for authentication and session management.

```python
from flask_login import UserMixin

class User(UserMixin):
    def __init__(self, id):
        self.id = id

    def get_id(self):
        return str(self.id)
```

our User model extends the **UserMixin** provided by Flask-Login. The UserMixin provides default implementations for the required methods.

We defined an id attribute and implement the get_id method, which returns a unique identifier for the user.

User Loader Function

To load a user from the user model, Flask-Login requires a user loader function that retrieves a user object based on the user ID.

```python
@login_manager.user_loader
def load_user(user_id):
    # Retrieve user from the database or other data source
    return User(user_id)
```

Protecting Routes

Routes protection is a major issue in Web Apps. Flask-Login allows you to protect routes that require user authentication.

You can use the `@login_required` decorator to restrict access to certain routes. Here's an example:

```
from flask_login import login_required

@app.route('/dashboard')
@login_required
def dashboard():
    # Route accessible only to authenticated users
    return 'Welcome to the dashboard!'
```

In this example, the `/dashboard` route is protected using the `@login_required` decorator.

If a user tries to access the route without being authenticated, Flask-Login will redirect them to the login page or a specified login view.

Login and Logout Routes

To implement the login and logout functionality, you need to define routes that handle the login and logout requests.

```
from flask_login import login_user, logout_user

@app.route('/login', methods=['GET', 'POST'])
def login():
    # Handle login request
    if request.method == 'POST':
        # Validate user credentials
        # If valid, load the user and login
        user_id = request.form['username'] user = User.query.filter_by(username=username).first() if user and
check_password_hash(user.password, request.form['password']): login_user(user) return redirect(url_for('dashboard'))
else: flash('Invalid username or password. Please try again.', 'error') return render_template('login.html')
@app.route('/logout') @login_required def logout():
logout_user()
return redirect(url_for('login'))
```

Flask-RESTful: Building RESTful APIs with Flask

Flask-RESTful is a Flask extension that simplifies the creation of RESTful APIs (Application Programming Interfaces) in your Flask applications.

It provides a straightforward way to define resources, handle HTTP methods, and serialize data in a format that is easily consumable by clients.

In this chapter, we'll explore how to use Flask-RESTful to build powerful and scalable APIs with Flask.

Installation

To get started with Flask-RESTful, you need to install it using pip, the Python package manager:

```
pip install Flask-RESTful
```

Once Flask-RESTful is installed, you can import it into your Flask application:

```
from flask import Flask
from flask_restful import Api, Resource
```

Initializing Flask-RESTful

To use Flask-RESTful, you need to initialize it in your Flask application.

Here's an example of how to initialize Flask-RESTful:

```
app = Flask(__name__)
api = Api(app)
```

Defining API Resources

In Flask-RESTful, a resource represents a specific entity or object in your API.

Each resource corresponds to a URL endpoint and handles HTTP methods such as GET, POST, PUT, and DELETE.

To define a resource, you need to create a class that extends the Resource class provided by Flask-RESTful.

```
class HelloWorld(Resource):
    def get(self):
        return {'message': 'Hello, World!'}

api.add_resource(HelloWorld, '/hello')
```

In this example, we define a resource called HelloWorld that handles the /hello endpoint.

The get method is invoked when a GET request is made to the endpoint, and it returns a dictionary containing a simple message.

Handling HTTP Methods

Flask-RESTful provides methods in the resource class to handle different HTTP methods.

For example, you can define a post method to handle POST requests, a put method to handle PUT requests, and so on.

Here's an example that demonstrates handling different HTTP methods for a resource:

```
class Item(Resource):
    def get(self, item_id):
        # Retrieve the item with the given ID
        ...

    def post(self):
        # Create a new item
        ...

    def put(self, item_id):
        # Update the item with the given ID
        ...

    def delete(self, item_id):
        # Delete the item with the given ID
        ...

api.add_resource(Item, '/items/<int:item_id>')
```

The Item resource handles different HTTP methods for managing items.

The get method retrieves an item based on the provided item ID, the post method creates a new item, the put method updates an existing item, and the delete method deletes an item.

Serializing Data

Flask-RESTful provides convenient ways to serialize and deserialize data in different formats, such as JSON, XML, or others.

You can use Flask-RESTful's built-in data parsing and formatting capabilities to handle data in a format that suits your API.

```
from flask_restful import reqparse

class Item(Resource):
    def put(self, item_id):
        parser = reqparse.RequestParser()
        parser.add_argument('name', type=str, required=True)
        parser.add_argument('price', type=float, required=True)
        data= parser.parse_args() # Update the item with the given ID using the provided data ...
api.add_resource(Item, '/items/int:item_id')
```

In this example, we use the `reqparse` module from Flask-RESTful to define a request parser.

We add arguments to the parser to specify the expected data fields, their types, and whether they are required or not.

In the `put` method, we use the parser to parse the request data and retrieve the values for the `name` and `price` fields.

Flask-RESTful simplifies the process of creating robust and scalable APIs, allowing you to focus on designing and implementing your API endpoints.

FLASK Interview Questions

Q. What are Flask blueprints? How do you use them in your application?

A. Flask Blueprints allow you to define routes, views, templates, static files, and other application components in a separate module or package.

This promotes code separation and modularity, making it easier to manage large Flask applications with multiple features or modules.

To use a blueprint in a Flask application, you typically follow these steps:

1. Define the blueprint: Create a blueprint object using the flask.Blueprint class, specifying the blueprint's name, import name, and any other necessary configuration options.

```
from flask import Blueprint

auth_bp = Blueprint('auth', __name__, url_prefix='/auth')
```

2. Define routes and views: Within the blueprint, you can define routes and views using the blueprint's route decorator (auth_bp.route).

```
@auth_bp.route('/login')
def login():
    return 'Login page'

@auth_bp.route('/register')
def register():
    return 'Registration page'
```

Q. Explain the role of Flask-SQLAlchemy.

A. Flask-SQLAlchemy is an extension for Flask that adds support for SQLAlchemy to your application.

Flask-SQLAlchemy combines the flexibility of Flask with the powerful features of SQLAlchemy, making it easier to work with databases in Flask applications.

Q. Describe the concept of Flask templates.

A. Flask templates allow you to add dynamic content to HTML files. They utilize the Jinja templating engine, which enables the insertion of Python code and variables into HTML templates.

Q. How to get logged user id in flask?

A. To get the logged-in user ID in Flask, you can make use of Flask's session object or a user authentication extension like Flask-Login.

```
from flask_login import current_user

@app.route('/dashboard')
if current_user.is_authenticated():
        g.user = current_user.get_id()
```

Q. How to get visitors IP in Flask?

```
from flask import request
from flask import jsonify

@app.route("/get_user_ip", methods=["GET"])
def get_user_ip():
    return jsonify({'ip': request.remote_addr}), 200
```

Q. How do you handle asynchronous tasks in Flask?

A. Async tasks can be handled by:

1. Using Celery: Celery is a distributed task queue system that integrates well with Flask. It allows you to offload time-consuming tasks to background workers.

2. Using Flask's before_first_request decorator: Flask provides a before_first_request decorator, allowing you to run a function or a task before the first request is processed.

3. Using async and await: async await allows you to handle asynchronous program easily.

Q. How do you handle static files (e.g., CSS, JavaScript) in Flask?

1. Create a static folder: In your Flask project directory, create a folder named static. Flask automatically looks for static files in this folder.

2. Place static files (CSS, JavaScript, images) in the static folder.

3. Link static files in your templates: In your HTML templates, you can reference static files using the url_for('static', filename='...') function.

```
<link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
```

Q. How to add the mailing feature in the Flask Application?

```
from flask_mail import Mail, Message
from flask import Flask

app = Flask(__name__)
mail = Mail(app)

@app.route("/mail")
def email():
    msg = Message( "Hello", sender="sender@mail.com", recipients=["receiver@mail.com"])
   mail.send(msg)
```

Python MARS Project

In this project we'll use **NASA's API** to get Images from **Mars**.

For this, your can grab your FREE API key from NASA
https://api.nasa.gov/

Let's Get Started

First, we'll import the requests module, which allows us to send HTTP requests to the NASA API.

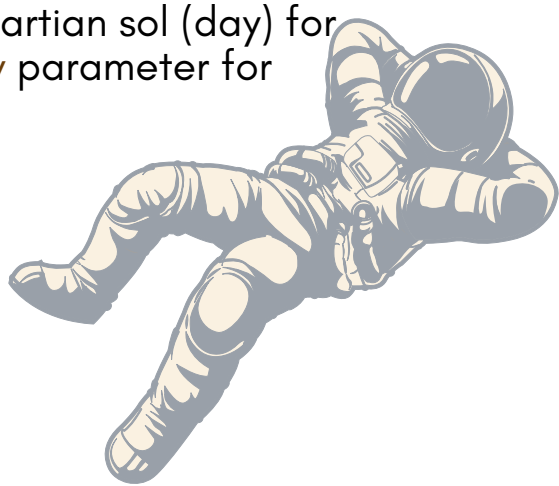
import requests

Then we'll create fetch_rover_photos function that takes three parameters: rover, sol, and api_key. It is responsible for fetching photos from the NASA API.

```python
def fetch_rover_photos(rover, sol, api_key):
    url = f"https://api.nasa.gov/mars-photos/api/v1/rovers/{rover}/photos"
    params = {
        "sol": sol,
        "api_key": api_key
    }
    response = requests.get(url, params=params)
    data = response.json()
    return data["photos"]
```

The function creates a params dictionary that contains the query parameters required by the API.

It includes the sol parameter, representing the Martian sol (day) for which you want to fetch photos, and the api_key parameter for authentication with the **NASA API**.



The received data will be stored in "photos" array.

We will iterate through each of the photo in the photos array using for-in loop, and print the URL of each image.

```
def display_photos(photos):
    for photo in photos:
        img_src = photo["img_src"]
        print(img_src)  # Modify this to display the image or perform other actions
```

You can use this image URL to display image on your own website if you want. For that you will use the HTML *<img>* tag.

<img src="https://yourURLhere.com">

Passing the API keys:

The main function serves as the entry point of the program.

It sets the values for the api_key, rover, and sol variables. You need to replace "YOUR_API_KEY" with your actual NASA API key to authenticate and make successful requests to the API.

The fetch_rover_photos function is called with the specified rover, sol, and api_key parameters to fetch the photos from the NASA API.

```
def main():
    api_key = "YOUR_API_KEY"  # Replace with your NASA API key
    rover = "curiosity"
    sol = 1000  # The Martian sol (day) for which you want to fetch photos

    photos = fetch_rover_photos(rover, sol, api_key)
    display_photos(photos)

if __name__ == "__main__":
    main()
```

In the above code we are using the **rover = "curiosity"** , because it's the latest rover sent to Mars by NASA.

You can change it to other rovers like "**opportunity**" or "**spirit**".

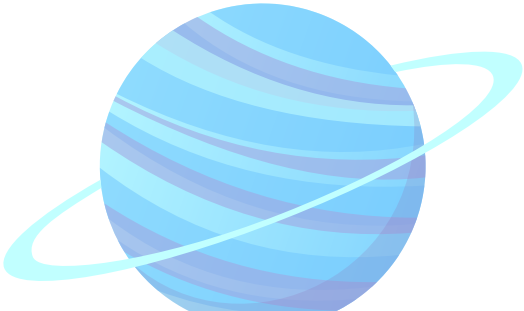
Result:

Image taken by Curiosity Rover

This is probably one of the most creative Projects that you can Add to your Portfolio.

You can find the whole code from our Github repository:

github.com/rishabhsdev/hello-mars-py



Deploy your code to the Cloud

Just creating awesome Projects is not Enough. Making amazing Projects just to keep them in your Laptop won't land you a Job.

You need to make those Projects LIVE for the public to see, and for that you need a Python Hosting.

How to choose a Python Hosting?

There are many Hosting providers on the internet, but you need to find one that is Developer friendly.

Below are the few things you should check while choosing a host:

Python Support: Obviously you'll need a host that supports Python code.

has git, ssh access: You need a host that offers git and ssh access, you'll need them while deploying your code, or making updates.

high inode limit: To make sure it can support large amount of files.

Here's a Python Hosting that I use for my personal Projects. It's super cheap, and has Support for all of the features mentioned above.

It's meant to be used by Developers, offers Python support out-of-the box, and gives you a Free **.com** Domain. You can check it out here

Once you have signed up for hosting, the next step is to Deploy your code.

Create a Python app in your Hosting:

There are 2 ways to do this:

1. Initiate your Python app to the Host using cPanel.

Here's the Link to complete Tutorial

2. Initiate your Python app via Terminal.

Link to the complete tutorial.

Upload your Code Files:

Now you need to upload your Python code files to the hosting, you can do this by using FileZilla (a free file upload tool). You can download it from here filezilla-project.org

Click on **"Download FileZilla Client"**

Open FileZilla and enter your Login Details:

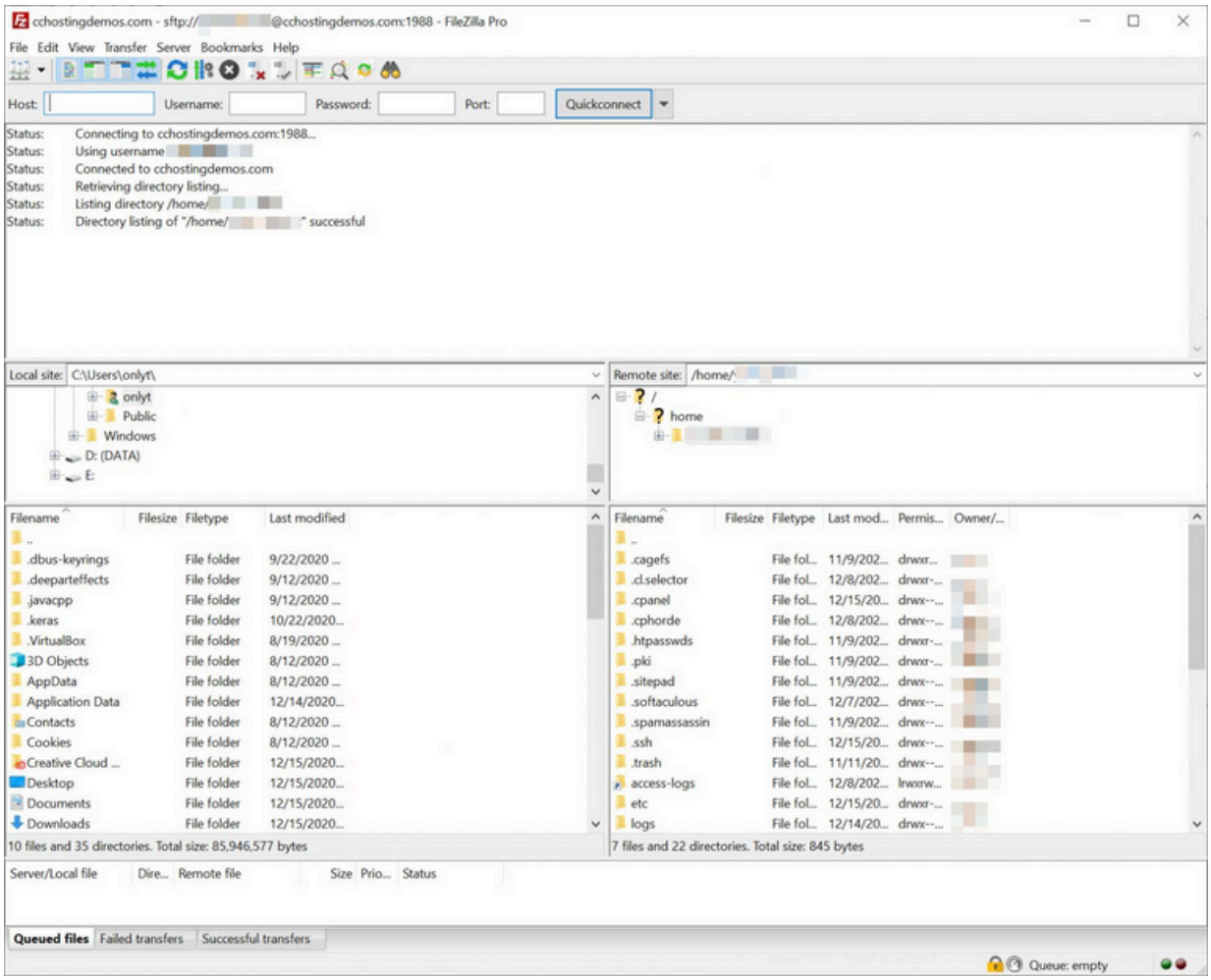
Host: this can be your domain name;

Username: your cPanel username;

Password: your cPanel password;

Port: 21; (or do not fill that field, it will go by default on 21)

Once connected, you will see a Screen like this:



Left size is your PC, and right side is your Hosting server.

Simply, drag the code file/folder from Left column (your PC) to the Right column (Hosting server).

If you've done all the previous steps correctly, you Python code should be Live and visible to the Public via your Domain.

E-Book PART - 2

PYTHON

&

AI

Why should you Learn about AI?

1. FAANG Companies Love AI: AI plays a pivotal role in FAANG (Facebook, Apple, Amazon, Netflix, and Google).

Whether it's improving recommendation algorithms, developing self-driving cars, or enabling natural language processing, AI is deeply embedded in the fabric of these tech giants.

2. AI Loves Python: Python has emerged as the de facto language for AI and machine learning, thanks to its simplicity, readability, and an array of powerful libraries.

Many AI frameworks, like TensorFlow, PyTorch, and scikit-learn, are primarily designed for Python.

3. Stand out of the Competition: As a Python developer, having AI knowledge under your belt opens up exciting opportunities within FAANG organizations.

AI is not limited to specialized roles; it permeates various departments, including research, data analysis, product development, and engineering.

4. Data-Driven Decision-Making: FAANG companies thrive on data-driven decision-making, and AI is the driving force behind extracting valuable insights from vast datasets.

Most widely used AI Libraries:

NumPy
Tensorflow
scikit-learn

NumPy

NumPy is a powerful Python library used for numerical computing with arrays and matrices.

It's an essential tool for data science and scientific computing, allowing you to perform complex mathematical operations with ease.

Where Do you Need NumPy?

1. Numerical Analysis: NumPy provides efficient and convenient tools for numerical analysis such as Fourier transforms, linear algebra, and numerical integration.

It is often used in fields such as physics, engineering, and mathematics.

2. Data Analysis: NumPy is frequently used in data analysis tasks to handle large datasets.

It allows for quick and easy manipulation of data in various formats such as CSV, Excel, and SQL databases. Many data analysis libraries, such as Pandas and SciPy, are built on top of NumPy.

3. Machine Learning: NumPy is an integral part of the Python machine learning ecosystem. It provides a foundation for many popular machine learning libraries such as Scikit-Learn, TensorFlow, and PyTorch.

Machine learning algorithms often involve matrix operations, which NumPy is designed to handle efficiently.

4. Image Processing: NumPy is also used for image processing tasks, such as manipulating images and performing various mathematical operations on image data.

It is often used together with libraries like OpenCV for computer vision tasks.

Installing NumPy

To get started with NumPy, you first need to install it. You can do this by using a package manager like pip.

```
pip install numpy
```

Once you have NumPy installed, you can import it into your Python script using the following code:

```
import numpy as np
```

NumPy Arrays

One of the main features of NumPy is its ability to work with arrays. In NumPy, arrays are objects that represent multi-dimensional, homogeneous, and fixed-sized collections of elements.

Here's an example of how to create a 1-dimensional NumPy array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
```

You can also create multi-dimensional arrays in NumPy.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
```

Once you have an array, you can perform various operations on it using NumPy's built-in functions.

For example, you can perform arithmetic operations on arrays, such as addition, subtraction, multiplication, and division.

Here's an example of how to add two arrays together:

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

arr3 = arr1 + arr2

print(arr3)
```

NumPy also provides many other useful functions for working with arrays, such as reshaping arrays, slicing arrays, and performing mathematical operations on arrays.

For example, you can use the 'reshape' function to change the shape of an array, as shown in this example:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

new_arr = np.reshape(arr, (2, 3))

print(new_arr)
```

Output:

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

The np.reshape() function is used to reshape the arr array into a new array called new_arr.

The desired shape is specified as (2, 3), indicating 2 rows and 3 columns.

Array Indexing and Slicing

Slicing allows us to extract subsets of elements from a NumPy array. We specify the start index, end index (exclusive), and optionally the step size using the colon : notation.

Here's an example:

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr[1:4])  # Slicing from index 1 to 4 (exclusive)
print(arr[::2])  # Slicing with a step size of 2
```

Output:

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

Boolean Indexing

NumPy also allows indexing arrays using boolean masks. A boolean mask is a binary array that has the same shape as the original array and consists of True and False values.

Elements corresponding to True values in the mask are selected, while those corresponding to False values are excluded.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

mask = np.array([True, False, True, False, True])

print(arr[mask])
```

Output:

[1 3 5]

Masking

Masking allows us to filter and extract specific elements from an array based on certain conditions.

We can create a boolean mask that identifies which elements meet the given condition, and then use that mask to select the corresponding elements.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Create a boolean mask
mask = arr > 2

# Use the mask to select elements
result = arr[mask]
print(result)  # Output: [3 4 5]
```

NumPy and Data Analysis

NumPy plays a crucial role in data analysis, providing powerful tools for manipulating, analyzing, and processing numerical data. In this chapter, we will explore how NumPy can be used in various aspects of data analysis, including data cleaning, transformation, aggregation, and computation. Let's dive in!

Data Cleaning and Transformation

When working with real-world datasets, data cleaning and transformation are essential steps to ensure data quality and prepare the data for analysis.

NumPy provides several functions and techniques to handle missing data, remove outliers, and transform data.
One common task is handling missing values.

NumPy offers the np.nan value to represent missing or undefined data. We can use functions like np.isnan() and np.nanmean() to identify and handle missing values. For example:

```
data = np.array([1, 2, np.nan, 4, 5])

# Check for missing values
missing_values = np.isnan(data)
print(missing_values)  # Output: [False False  True False False]

# Replace missing values with the mean
mean_value = np.nanmean(data)
data[missing_values] = mean_value
print(data)  # Output: [1. 2. 3. 4. 5.]
```

Here, we have an array data with a missing value represented by np.nan. We use np.isnan() to identify the missing value, and then calculate the mean value using np.nanmean().

Finally, we replace the missing value with the mean value.

Vectorized Computation

One of the key advantages of NumPy is its ability to perform vectorized computations.
Vectorization allows us to apply operations and functions to entire arrays or large portions of data at once, instead of looping over individual elements.

This leads to faster and more efficient computations.

```
import numpy as np

data = np.array([1, 2, 3, 4, 5])

# Perform vectorized computation
result = data * 2 + 3
print(result)  # Output: [5 7 9 11 13]
```

NumPy and Image Processing

NumPy provides functions to read and write image files, allowing us to work with images as multidimensional arrays.

The matplotlib.image module is commonly used in conjunction with NumPy for image I/O.

To load an image, we can use the imread() function:

```
import matplotlib.pyplot as plt

# Load an image
image = plt.imread('image.jpg')

# Display the image
plt.imshow(image)
plt.axis('off')
plt.show()
```

NumPy Interview Questions

Q. How do you create a NumPy array from a Python list?

A. You can create a NumPy array from a Python list by using the np.array() function.

```
import numpy as np

python_list = [1, 2, 3, 4, 5]
numpy_array = np.array(python_list)
```

Q. What is the difference between a NumPy array and a Python list?

A. The main difference between a NumPy array and a Python list is that a NumPy array is homogeneous, meaning it can only contain elements of the same data type.

A Python list can hold elements of different data types. NumPy arrays also offer more efficient storage and operations for numerical computations, making them preferable for mathematical and scientific tasks.

Q. How do you perform element-wise multiplication of two NumPy arrays?

A. You can perform element-wise multiplication of two NumPy arrays using the * operator or the np.multiply() function.

```
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

result = array1 * array2  # or np.multiply(array1, array2)
```

Q. How do you calculate the mean, median, and standard deviation of a NumPy array?

A. You can calculate the mean, median, and standard deviation of a NumPy array using the np.mean(), np.median(), and np.std() functions, respectively.

```
array = np.array([1, 2, 3, 4, 5])

mean = np.mean(array)
median = np.median(array)
std_dev = np.std(array)
```

Q. How can you load and save images using NumPy?

A. NumPy can work in conjunction with image processing libraries like PIL (Python Imaging Library) or opencv to load and save images.

For example, to load an image using PIL:

```
import numpy as np
from PIL import Image

image = np.array(Image.open('image.jpg'))
```

To save an image using PIL:

```
import numpy as np
from PIL import Image

image = np.array([[255, 0, 0], [0, 255, 0], [0, 0, 255]], dtype=np.uint8)
image = Image.fromarray(image)
image.save('image.jpg')
```

What is TensorFlow?

First things first, what exactly is TensorFlow? Well, TensorFlow is an open-source library developed by **Google** that makes it easier for us to build and deploy machine learning models.

It's like having a superpower in your coding toolkit! With TensorFlow, we can create and train neural networks, handle large datasets, and perform complex mathematical computations with ease.

Companies using TensorFlow:

As you can see that Tensor is Quite in Demand among big Tech Giants.

Installing TensorFlow and Setting Up the Python Environment

In this chapter, we'll go through the process of installing TensorFlow and setting up our Python environment. So, let's get started!

1. Choosing the Python Version: Before we dive into installing TensorFlow, it's essential to decide which version of Python we want to use.

TensorFlow supports Python 3.7 and above, so it's recommended to use the latest stable version of Python.

You can download Python from the official Python website (python.org) and follow the installation instructions for your operating system.

2. Installing TensorFlow: Once we have Python up and running, it's time to install TensorFlow.

Luckily, TensorFlow provides a convenient installation process using Python's package manager, pip.

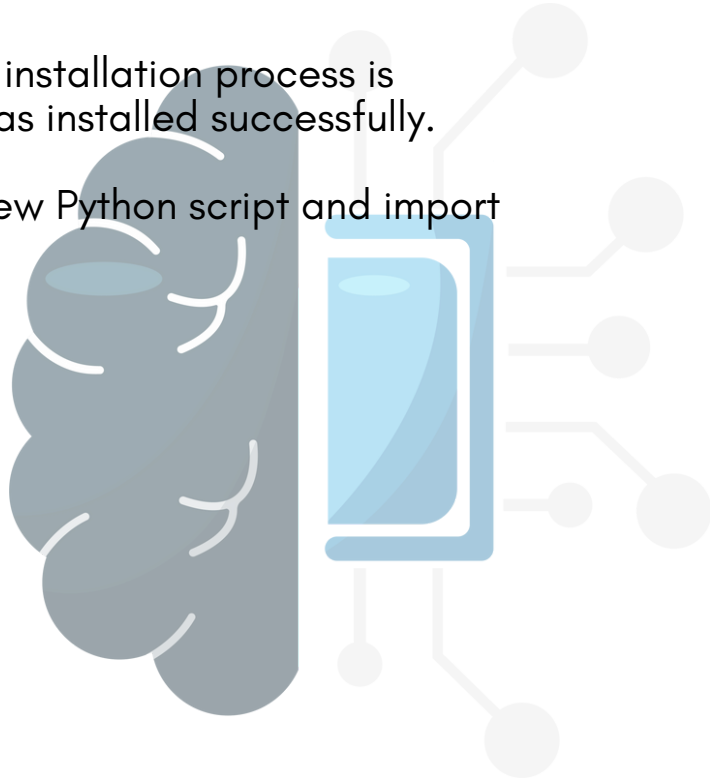
Open up your terminal or command prompt and run the following command to install TensorFlow:

```
pip install tensorflow
```

3. Verifying the Installation: Once the installation process is complete, we can verify if TensorFlow was installed successfully.

Open a Python interpreter or create a new Python script and import TensorFlow:

```
pip show tensorflow
# or
pip3 show tensorflow
```



4. Virtual Environments: It's always a good practice to work within a virtual environment to keep our Python projects isolated.

Virtual environments allow us to create a separate environment with its own dependencies.

We can create a virtual environment specifically for our TensorFlow projects by using a tool like venv or conda.

```
python -m venv tensorflow-env
```

This will create a new directory named tensorflow-env that contains our virtual environment.

Activate the virtual environment by running the appropriate command for your operating system:

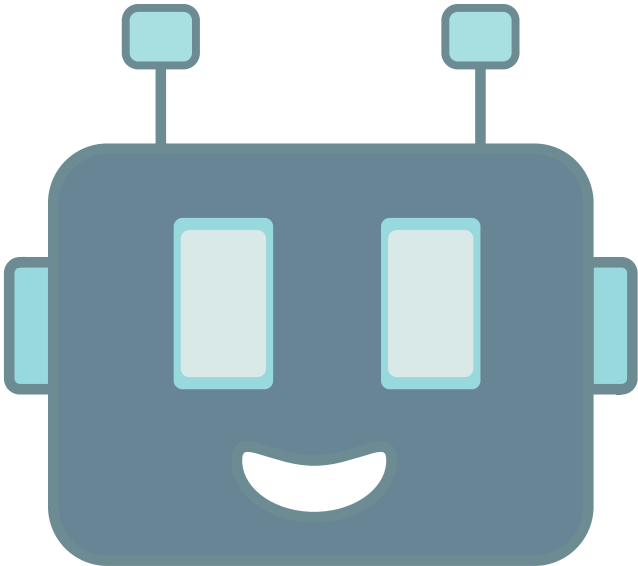
For Windows:

tensorflow-env\Scripts\activate

For macOS/Linux:

source tensorflow-env/bin/activate

Once activated, any packages installed or modifications made will be isolated within this virtual environment.



Building a Simple TensorFlow Program

Now that we have TensorFlow installed and our Python environment all set up, it's time to get our hands dirty and build our first TensorFlow program.

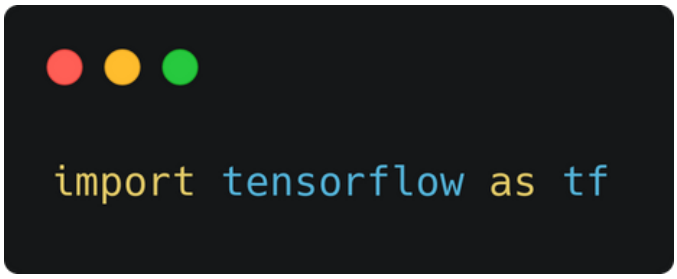
In this chapter, we'll walk through the process of creating a simple TensorFlow program step by step. So, let's roll up our sleeves and get started!

Our goal is to build a program that performs a basic linear regression task. Don't worry if you're not familiar with linear regression—we'll explain it along the way.

So, grab your favorite code editor and let's dive in!

Step 1: Importing the TensorFlow Library First things first, we need to import the TensorFlow library into our Python script.

Open up a new Python file and add the following line at the beginning:

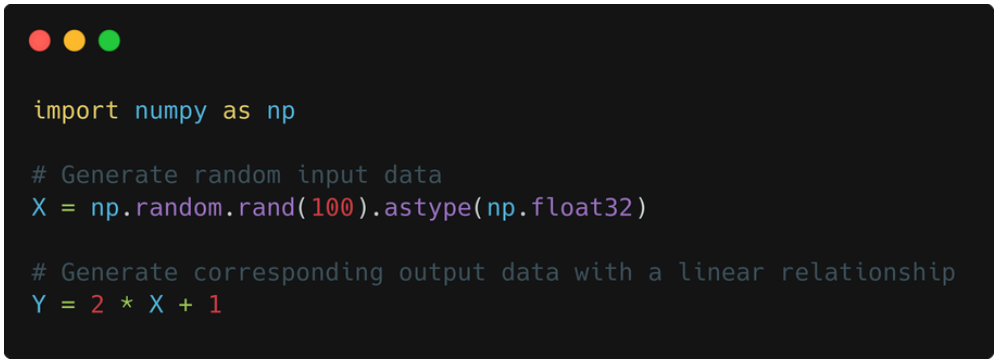


```
import tensorflow as tf
```

Step 2: Generating Data for Linear Regression In our linear regression task, we need some data to work with.

Let's generate some dummy data using NumPy, a popular library that we learned earlier.

Add the following lines to your script:



```
import numpy as np

# Generate random input data
X = np.random.rand(100).astype(np.float32)

# Generate corresponding output data with a linear relationship
Y = 2 * X + 1
```

In the above code, we import NumPy and generate 100 random floating-point numbers as our input data, stored in the variable X.

We also generate the corresponding output data Y using a simple linear relationship of Y = 2X + 1.

Step 3: Creating Variables and a Model Now, let's define the variables and model for our linear regression.

In TensorFlow, variables are used to hold and update the parameters of our model during training.

```
# Initialize model parameters
W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")

# Define the linear regression model
def linear_regression(x):
    return W * x + b
```

We created two TensorFlow variables, W and b, to hold the weight and bias of our linear regression model. We initialize them with random values.

Then, we defined the linear_regression() function, which takes an input x and returns the predicted output based on the current weight and bias.

Step 4: Defining Loss and Optimization To train our model, we need to define a loss function that measures how well our predictions match the actual output.

In this case, we'll use the mean squared error (MSE) as our loss function.

Additionally, we'll need an optimization algorithm to update the model parameters based on the calculated loss.

```python
# Define loss function (mean squared error)
def mean_squared_error(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_true - y_pred))

# Define the optimizer
optimizer = tf.optimizers.SGD(learning_rate=0.01)
```

Here, we define the mean_squared_error() function, which takes the true output y_true and predicted output y_pred and calculates the mean squared error between them.

We also create an optimizer using stochastic gradient descent (SGD) with a learning rate of 0.01.

The optimizer will update the model parameters to minimize the loss during training.

Step 5: Training the Model It's finally time to train our model! We'll run a loop to iteratively update the model parameters based on our training data.

```python
# Training loop
num_epochs = 100

for epoch in range(num_epochs):
    # Perform forward pass
    with tf.GradientTape() as tape:
        y_pred = linear_regression(X)
        loss = mean_squared_error(Y, y_pred)

    # Calculate gradients
    gradients = tape.gradient(loss, [W, b])

    # Update model parameters
    optimizer.apply_gradients(zip(gradients, [W, b]))

    # Print progress
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch + 1}: Loss = {loss.numpy():.4f}")
```

In this training loop, we iterate over a fixed number of epochs (iterations) and perform the following steps:

Forward Pass: We use a tf.GradientTape context to track the operations and calculate the predicted output y_pred using our linear_regression() function. We also compute the mean squared error loss.

Calculate Gradients: We use the tape to automatically calculate the gradients of the loss with respect to the variables W and b.

Update Model Parameters: The optimizer applies the gradients to update the values of W and b, minimizing the loss.

Print Progress: Every 10 epochs, we print the current epoch number and the value of the loss to monitor the training progress.

Step 6: Testing the Trained Model Now that our model is trained, we can test it on some new data to see how well it performs.

```
# Generate test data
X_test = np.array([0.2, 0.5, 0.8], np.float32)

# Predict the outputs for the test data
Y_test = linear_regression(X_test)

# Print the predictions
print("Test Predictions:")
for i in range(len(X_test)):
    print(f"Input: {X_test[i]:.2f}, Predicted Output: {Y_test[i]:.2f}")
```

In this code, we create a separate set of test data X_test, and then we used our trained model to predict the outputs Y_test for this test data.

Finally, we print the input values and their corresponding predicted outputs.

Step 7: Running the Program Congratulations! You've built your first **TensorFlow** program. Save your script with a meaningful name, such as linear_regression.py, and run it using your Python environment.

You should see the training progress printed on the console, followed by the predicted outputs for the test data.

Working with Tensors

Introducing Tensors: The Building Blocks of TensorFlow

Let's start by talking about tensors. In TensorFlow, tensors are the fundamental data structures used to represent and manipulate data. You can think of a tensor as a multi-dimensional array or matrix.

Tensors can have different ranks, which represent their number of dimensions.

For instance, a scalar is a tensor of rank 0, which represents a single value.

A vector is a tensor of rank 1, consisting of a sequence of values. A matrix is a tensor of rank 2, organized in rows and columns.

You can have tensors of higher ranks, such as 3D tensors, 4D tensors, and so on.

TensorFlow provides a rich set of functions and operations to create and manipulate tensors. Let's look at a simple example:

```python
import tensorflow as tf

# Create a scalar tensor
scalar_tensor = tf.constant(42)

# Create a vector tensor
vector_tensor = tf.constant([1, 2, 3, 4, 5])

# Create a matrix tensor
matrix_tensor = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(scalar_tensor)
print(vector_tensor)
print(matrix_tensor)
```

In this code, tf.constant() function is used to create tensors with different ranks.

The tf.constant() function allows us to define tensors with fixed values that cannot be changed. We then print the tensors to see their contents.

Performing Operations on Tensors

Now that we know how to create tensors, let's explore how we can perform operations on them.

TensorFlow provides a wide range of mathematical operations that can be applied to tensors, such as addition, subtraction, multiplication, and more.

```
import tensorflow as tf

# Create tensors
tensor1 = tf.constant([1, 2, 3])
tensor2 = tf.constant([4, 5, 6])

# Perform tensor addition
addition = tf.add(tensor1, tensor2)

# Perform tensor multiplication
multiplication = tf.multiply(tensor1, tensor2)

print(addition)
print(multiplication)
```

In this example, We used the tf.add() function to add the two tensors element-wise, and the tf.multiply() function to multiply them element-wise.

The results are stored in the addition and multiplication tensors, respectively.

TensorFlow Sessions: Running Computations

To actually execute the operations and evaluate tensors, we need to create a TensorFlow session.

A session encapsulates the environment in which TensorFlow operations are executed.

```
import tensorflow as tf

# Create tensors
tensor1 = tf.constant([1, 2, 3])
tensor2 = tf.constant([4, 5, 6])

# Perform tensor addition
addition = tf.add(tensor1, tensor2)

# Create a session
with tf.compat.v1.Session() as sess:
    result = sess.run(addition)
    print(result)
```

In this example, we created tensors and perform an addition operation as before. However, we wrap the evaluation of the addition tensor within a TensorFlow session using the tf.compat.v1.Session() context manager.

The sess.run() function is used to execute the operation and retrieve the result.

Finally, we print the result, which will be the sum of the two tensors.

TensorFlow Variables: Modifiable Tensors

While tensors created with tf.constant() are immutable and their values cannot be changed, TensorFlow provides another type of tensor called tf.Variable(), which allows for modifiable tensors.

Variables are typically used to store and update the parameters of machine learning models during training.

Let's see an example of how to create and update a TensorFlow variable:

```python
import tensorflow as tf

# Create a TensorFlow variable
my_variable = tf.Variable(2)

# Define an assignment operation
assign_op = tf.compat.v1.assign(my_variable, 3)

# Create a session
with tf.compat.v1.Session() as sess:
    # Initialize variables
    sess.run(tf.compat.v1.global_variables_initializer())

    # Print initial value of my_variable
    print(sess.run(my_variable))

    # Update the value of my_variable
    sess.run(assign_op)

    # Print updated value of my_variable
    print(sess.run(my_variable))
```

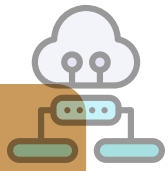
In this example, we created a TensorFlow variable called my_variable and initialize it with the value 2.

We then defined an assignment operation using tf.compat.v1.assign() to update the variable's value to 3.

Inside the session, we first initialized the variables using tf.compat.v1.global_variables_initializer().

Then, we used sess.run() to execute the assignment operation and print the initial and updated values of my_variable.

Handling Data with TensorFlow



TensorFlow, provides powerful tools and utilities to efficiently handle and preprocess data.

This chapter will explore various techniques and functionalities offered by TensorFlow for data handling.

We will cover data loading, preprocessing, augmentation, and batching, among other essential aspects of working with data in TensorFlow.

Section 1: Data Loading Loading data is the first step in any machine learning project. TensorFlow offers flexible options to load data from different sources such as files, databases, or APIs.

The following techniques are commonly used for data loading in TensorFlow:

1. TensorFlow Datasets (TFDS): TensorFlow Datasets is a library that provides access to various public datasets.

It offers a convenient way to download and load datasets directly into TensorFlow.

```
import tensorflow_datasets as tfds

# Load CIFAR-10 dataset
dataset, info = tfds.load('cifar10', split='train', with_info=True)
```

2. File I/O: TensorFlow provides functions to read data from files, such as CSV, TFRecord, or image files.

These functions enable you to parse and preprocess the data before feeding it to the model.

```
# Read CSV file using TensorFlow
dataset = tf.data.experimental.CsvDataset('data.csv', [tf.int32, tf.float32], header=True)
```

Section 2: Data Preprocessing Data preprocessing plays a vital role in preparing the data for training.

TensorFlow offers a wide range of preprocessing functions to transform and manipulate the data.

Some common preprocessing techniques include:

1. Normalization: Scaling the data to a common range to improve model convergence and performance.

```
# Normalize data using TensorFlow
normalized_data = tf.image.per_image_standardization(data)
```

2. Resizing: Resizing images or sequences to a consistent shape to ensure compatibility with the model.

```
# Resize images using TensorFlow
resized_images = tf.image.resize(images, size=(224, 224))
```

3. One-Hot Encoding: Converting categorical labels into one-hot encoded vectors for multi-class classification problems.

```
# Convert labels to one-hot encoding
one_hot_labels = tf.one_hot(labels, depth=num_classes)
```

Section 3: Data Augmentation Data augmentation is a technique used to artificially expand the training dataset by applying random transformations to the existing data.

TensorFlow provides numerous built-in functions for data augmentation, including rotation, flipping, zooming, and more.

```
# Apply random image rotation using TensorFlow
augmented_images = tf.image.rot90(images, k=tf.random.uniform(shape=[], minval=0, maxval=4, dtype=tf.int32))
```

Section 4: Batching and Shuffling To efficiently train models, data is typically processed in batches and shuffled during training.

TensorFlow provides functions to create batches of data and shuffle the data during each epoch.

```
# Create batches and shuffle data using TensorFlow
dataset = dataset.batch(batch_size)
dataset = dataset.shuffle(buffer_size=1000)
```

Architecture of Big Businesses depends heavely on data, which makes learning Data Handling crucial to crack Interviews.

Image Recognition Using TensorFlow

Image recognition is a fascinating field of computer vision, and TensorFlow provides a powerful platform for building and training image recognition models.

In this chapter, we will explore the process of image recognition using TensorFlow and delve into key concepts and code examples to help you understand and implement image recognition models effectively.

Convolutional Neural Networks (CNNs) for Image Recognition

Convolutional Neural Networks (CNNs) are the cornerstone of image recognition models.

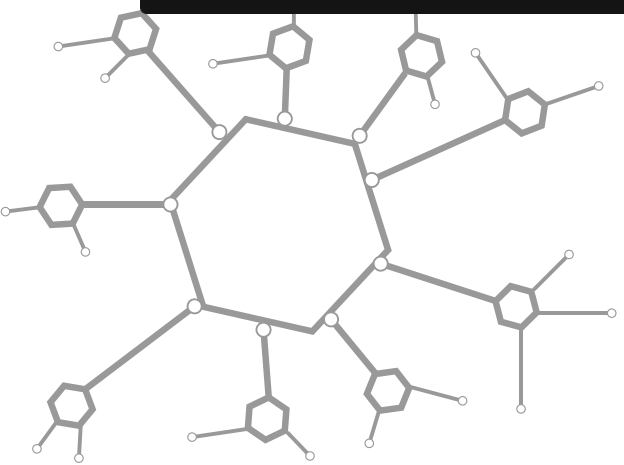
They excel at capturing local patterns and hierarchical structures within images.

Let's dive into some essential concepts related to CNNs:

Convolutional Layers: Convolutional layers apply filters to input images to extract meaningful features.

Each filter performs convolutions across the image, resulting in feature maps that highlight different patterns and textures.

```
# Example Convolutional Layer in TensorFlow
import tensorflow as tf

# Create a convolutional layer
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(224, 224, 3))
```



Pooling Layers: Pooling layers reduce the spatial dimensions of feature maps, preserving the most important information.

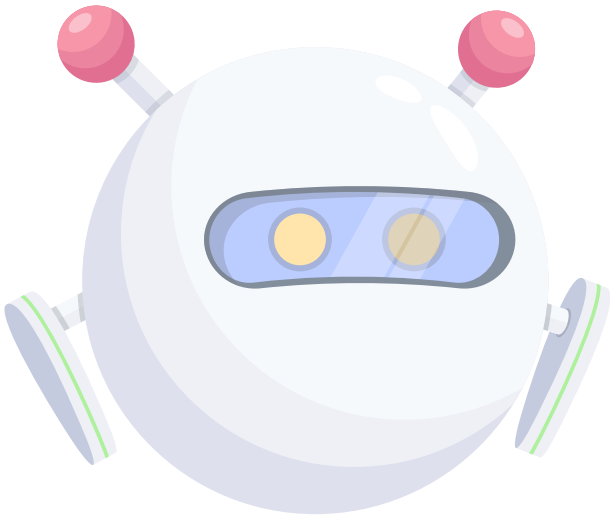
Max pooling is a common technique that retains the maximum value within a defined window.

```
# Example Max Pooling Layer in TensorFlow
import tensorflow as tf

# Create a max pooling layer
pooling_layer = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))
```

Activation Functions: Activation functions introduce non-linearity into the model, enabling the learning of complex relationships between input and output.

ReLU (Rectified Linear Unit) is a popular activation function used in CNNs.

```
# Example ReLU Activation Function in TensorFlow
import tensorflow as tf

# Apply ReLU activation
activation = tf.keras.activations.relu(x)
```



Fully Connected Layers: Fully connected layers, also known as dense layers, are the traditional neural network layers that connect every neuron in one layer to every neuron in the subsequent layer.

In TensorFlow, we can implement fully connected layers using the Dense class from the tf.keras.layers module. Let's explore how to define and use fully connected layers in code.

```
import tensorflow as tf

# Define the model architecture
model = tf.keras.models.Sequential()

# Add a flatten layer to convert the 2D feature maps into a 1D vector
model.add(tf.keras.layers.Flatten())

# Add the fully connected layers
model.add(tf.keras.layers.Dense(units=256, activation='relu'))
model.add(tf.keras.layers.Dense(units=128, activation='relu'))

# Add the output layer with the appropriate number of units
model.add(tf.keras.layers.Dense(units=num_classes, activation='softmax'))
```

Once the model is defined, we can proceed with the training process, including compiling the model with an appropriate optimizer, specifying the loss function, and fitting the model to the training data.

What are datasets in tensorflow?

A dataset in TensorFlow represents a collection of data instances that can be used for training, validation, or testing models. TensorFlow provides various built-in APIs to work with datasets effectively.

Datasets in TensorFlow offer several advantages, including:

Data Preparation: TensorFlow datasets allow you to apply transformations such as shuffling, batching, normalization, and augmentation to the data.

Efficiency: TensorFlow datasets provide mechanisms for loading data in parallel, prefetching data for faster processing, and caching data to minimize disk I/O.

Flexibility: TensorFlow datasets can can be easily integrated with different input pipelines, such as reading data from files, parsing records from TFRecord files, or generating data on the fly.

Congratulations! 🎉

You have successfully Finished the E-Book.

Whats Next?

Practice.
You need to practice what your learned, make new Projects, play around with the APIs.

Here's one Practice Project for You.

Q. Use NASAs API to find Weather on Mars.

It very similar to the Mars Project that we built earlier in this ebook, where we used NASA API to get Mars images.

Try and Build this Project.

Thanks