Chapter 1 - Review of Basic Concepts

1. Syntax of a Python Code

Python is an interpreted, high-level, dynamically typed programming language. It uses indentation instead of curly brackets for code blocks.

Example:

```python
# Basic Python syntax
name = "John Doe"  # Variable assignment
age = 25           # Integer variable

if age >= 18:
    print(f"{name} is an adult.")
else:
    print(f"{name} is a minor.")
```

2. Lists, Tuples, Sets, Dictionaries

Lists:

A **list** is an ordered, mutable collection of elements.

```python
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']
```

Tuples:

A **tuple** is an immutable, ordered collection of elements.

```python
tuple_example = (1, 2, 3, "Python")
print(tuple_example[1])  # 2
```

Sets:

A **set** is an unordered collection of unique elements.

```python
unique_numbers = {1, 2, 3, 3, 2, 1}
print(unique_numbers)  # {1, 2, 3}
```

Dictionaries:

A **dictionary** stores key-value pairs.

```python
student = {"name": "Alice", "age": 22, "grade": "A"}
print(student["name"])  # Alice
```

3. Functions

Functions allow code reusability.

```python
def greet(name):
    return f"Hello, {name}!"

print(greet("John"))  # Hello, John!
```

4. If-Elif-Else Statements

Conditional statements control the flow of execution.

```python
x = 10
if x > 10:
    print("Greater than 10")
elif x == 10:
    print("Equal to 10")
else:
    print("Less than 10")
```

5. For / While Loops

For Loop:

```python
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4
```

While Loop:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

6. Modules

A module is a file containing Python definitions and statements.

```
# math module example
import math
print(math.sqrt(25))  # 5.0
```

7. Package Managers

PyPI:

The **Python Package Index (PyPI)** hosts third-party Python packages.

Pip:

pip is Python's package manager.

```
pip install requests
```

Conda:

A package manager for Python and other languages.

```
conda install numpy
```

8. Regular Expressions

Regular expressions (regex) help in pattern matching.

```
import re
pattern = r"\d+"
result = re.findall(pattern, "User123 has 456 points")
print(result)  # ['123', '456']
```

9. Django Installation and Setup

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites.

Installing Django

Ensure you have Python installed (version 3.x recommended). Then install Django using pip:

```
pip install django
```

Verify the installation:

```
django-admin --version
```

Creating a Django Project

Run the following command to start a new project:

```
django-admin startproject myproject
```

Navigate into the project directory:

```
cd myproject
```

Run the development server:

```
python manage.py runserver
```

This will start the Django server, and you can access the default Django welcome page at `http://127.0.0.1:8000/`.

Creating a Django App

Inside your project directory, create a new app:

```
python manage.py startapp myapp
```

Register the app in `settings.py` by adding `'myapp'` to the `INSTALLED_APPS` list.

Defining a Model

In `myapp/models.py`, define a simple model:

```python
from django.db import models

class Item(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
```

Run migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Creating a View

In `myapp/views.py`, create a simple view:

```python
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello, Django!")
```

Configuring URLs

In `myapp/urls.py`, define a URL pattern:

```python
from django.urls import path
from .views import home

urlpatterns = [
    path('', home, name='home'),
]
```

Include this in the project's main `urls.py`:

```python
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),
]
```

Running the Server

Start the Django server again:

```
python manage.py runserver
```

Now visit `http://127.0.0.1:8000/` in your browser to see your first Django view in action.

Exercises

1. Create a list of numbers from 1 to 10 and print only even numbers.
2. Write a function that takes a name as input and returns "Hello, !".
3. Write a program that asks for a user's age and prints if they are a minor or an adult.
4. Write a while loop that prints numbers from 5 to 0.
5. Use the `math` module to compute the square root of a number entered by the user.
6. Use a regular expression to extract all email addresses from a given text.
7. Install Django and create a simple project with an app that returns "Welcome to Django!" on the homepage.

Chapter 2 - Classes and Objects

1. Introduction to Classes and Objects

Object-Oriented Programming (OOP) is a paradigm that helps structure programs into reusable and modular code. The fundamental building blocks of OOP are **classes** and **objects**.

1.1 What is a Class?

A **class** is a blueprint for creating objects. It defines attributes (data) and methods (functions) that describe the behavior of the objects.

Syntax of a Class

```python
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):
        return f"{self.year} {self.brand} {self.model}"
```

1.2 What is an Object?

An **object** is an instance of a class. It holds real data and interacts with other objects.

Creating an Object from a Class

```python
my_car = Car("Toyota", "Corolla", 2022)
print(my_car.display_info())  # Output: 2022 Toyota Corolla
```

---

2. Understanding the `__init__()` Method

The `__init__()` method is a special constructor method in Python used to initialize an object's attributes.

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

p1 = Person("Alice", 25)
print(p1.name)  # Output: Alice
print(p1.age)   # Output: 25
```

Key Points About `__init__()`

- It is automatically called when an object is created.
- It initializes attributes of the object.
- It allows each object to have different attribute values.

---

3. Instance and Class Attributes

3.1 Instance Attributes

Instance attributes are unique to each object. They are defined inside the `__init__()` method.

```python
class Dog:
    def __init__(self, name, breed):
        self.name = name  # Instance attribute
        self.breed = breed

dog1 = Dog("Buddy", "Labrador")
dog2 = Dog("Max", "Beagle")

print(dog1.name)  # Output: Buddy
print(dog2.name)  # Output: Max
```

3.2 Class Attributes

Class attributes are shared across all instances of a class.

```python
class Animal:
    species = "Mammal"  # Class attribute

a1 = Animal()
```

```python
a2 = Animal()

print(a1.species)  # Output: Mammal
print(a2.species)  # Output: Mammal
```

Difference Between Class and Instance Attributes

Feature	Instance Attribute	Class Attribute
Scope	Specific to an object	Shared across all instances
Defined in	`__init__()`	Outside any method
Modification	Only affects one instance	Affects all instances

---

4. Methods in a Class

4.1 Instance Methods

Instance methods work on individual objects.

```python
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def get_details(self):
        return f"{self.name} is in grade {self.grade}."

s1 = Student("John", 10)
print(s1.get_details())  # Output: John is in grade 10.
```

4.2 Class Methods (`@classmethod`)

Class methods work on the class rather than an instance.

```python
class School:
    school_name = "Greenwood High"

    @classmethod
    def change_school_name(cls, new_name):
        cls.school_name = new_name

print(School.school_name)  # Output: Greenwood High
School.change_school_name("Sunrise Academy")
print(School.school_name)  # Output: Sunrise Academy
```

4.3 Static Methods (`@staticmethod`)

Static methods are independent of class and instance attributes.

```python
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

print(MathUtils.add(5, 3))  # Output: 8
```

Type	Purpose	Uses `self` ?	Uses `cls` ?
Instance Method	Works with object attributes	✅	❌
Class Method	Works with class attributes	❌	✅
Static Method	Independent utility function	❌	❌

Here’s a more detailed introduction to Django Models to provide better clarity and depth.

---

5. Working with Objects in Django

Introduction to Django Models

In Django, **models** are Python classes that define the structure and behavior of database tables. Each model maps directly to a single table in the database, and Django provides an **Object-Relational Mapping (ORM)** system to interact with the database using Python instead of SQL.

A model typically includes:

- **Fields**: Attributes that define the data structure (e.g., `CharField`, `IntegerField`, `DateField`).
- **Methods**: Functions that operate on model instances.
- **Meta Options**: Configurations like ordering, database table name, etc.

Defining a Model in Django

To define a model, create a class that inherits from `models.Model` and specify fields as class attributes.

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)  # Title of the book
    author = models.CharField(max_length=100)  # Author name
    published_date = models.DateField()  # Date of publication
    price = models.DecimalField(max_digits=6, decimal_places=2, null=True,
blank=True)  # Optional price field

    def __str__(self):
        return self.title  # Returns the book title when printed

    def get_book_info(self):
        return f"{self.title} by {self.author}"
```

Explanation of Fields

- `CharField(max_length=...)` → Stores short text data (e.g., title, author).
- `DateField()` → Stores date values (e.g., publication date).
- `DecimalField(max_digits=6, decimal_places=2)` → Stores decimal values (e.g., price of the book).
- `null=True, blank=True` → Allows a field to be empty.

Applying Migrations

Once a model is defined, you need to create and apply migrations to reflect changes in the database.

```
python manage.py makemigrations
python manage.py migrate
```

Creating an Object in Django Shell

Django provides an interactive Python shell to work with models.

```
python manage.py shell
```

```python
from myapp.models import Book

# Creating an object
book1 = Book(title="Django for Beginners", author="William S. Vincent",
published_date="2023-05-10", price=29.99)
book1.save()  # Saves the object to the database

# Retrieving the object
print(book1.get_book_info())  # Output: Django for Beginners by William S.
Vincent
```

Querying Objects

Once you have created objects, you can retrieve them using Django's ORM.

Retrieving All Objects

```python
books = Book.objects.all()
for book in books:
    print(book.title)
```

Filtering Objects

```python
books_by_author = Book.objects.filter(author="William S. Vincent")
```

Retrieving a Single Object

```
book = Book.objects.get(id=1)  # Retrieves the book with ID 1
```

Updating an Object

```
book.title = "Updated Django Book"
book.save()
```

Deleting an Object

```
book.delete()
```

6. Exercises

Exercise 1: Define a Class and Create an Object

Write a Python class called `Laptop` with attributes `brand`, `model`, and `price`. Create an object of the class and print the details.

Exercise 2: Implement Class Methods

Modify the `Laptop` class to include a class attribute `category = "Electronics"`. Add a class method to update the category and test it.

Chapter 3 - Constructors and Destructors

1. Introduction

In object-oriented programming, **constructors** and **destructors** are special methods that handle object initialization and cleanup.

- The **constructor** (`__init__()`) is called when an object is created.
- The **destructor** (`__del__()`) is called when an object is deleted or goes out of scope.

These methods allow us to manage resources, such as database connections, files, or memory, effectively.

---

2. Constructors (`__init__()`)

2.1 What is a Constructor?

A **constructor** is a special method that initializes an object's attributes when it is created. In Python, the constructor is the `__init__()` method.

2.2 Syntax and Example

```python
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        print(f"{self.brand} {self.model} created!")

car1 = Car("Toyota", "Corolla", 2022)
car2 = Car("Honda", "Civic", 2023)
```

Output:

```
Toyota Corolla created!
Honda Civic created!
```

2.3 Constructor Overloading (Using Default Values)

Python does not support **method overloading**, but we can achieve similar behavior using default values.

```
class Laptop:
    def __init__(self, brand="Unknown", model="Unknown", price=0):
        self.brand = brand
        self.model = model
        self.price = price

l1 = Laptop("Dell", "XPS", 1200)
l2 = Laptop()  # Uses default values

print(l1.brand, l1.model, l1.price)  # Output: Dell XPS 1200
print(l2.brand, l2.model, l2.price)  # Output: Unknown Unknown 0
```

2.4 Using `super().__init__()` in Inheritance

If a class inherits from another class, we use `super().__init__()` to call the parent class's constructor.

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

class Car(Vehicle):
    def __init__(self, brand, model, year):
        super().__init__(brand)  # Calls parent constructor
        self.model = model
        self.year = year

car1 = Car("Ford", "Mustang", 2022)
print(car1.brand, car1.model, car1.year)  # Output: Ford Mustang 2022
```

---

3. Destructors (`__del__()`)

3.1 What is a Destructor?

A **destructor** is a special method that is automatically called when an object is destroyed. It helps in **resource management**, like closing database connections or file handles.

3.2 Syntax and Example

```python
class Person:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} is created!")

    def __del__(self):
        print(f"{self.name} is deleted!")

p1 = Person("Alice")
del p1  # Manually deleting object
```

Output:

```
Alice is created!
Alice is deleted!
```

3.3 When is the Destructor Called?

- When an object **goes out of scope** (e.g., at the end of a function).
- When we **explicitly delete** an object using `del`.
- When **Python's garbage collector** removes an unreferenced object.

Example:

```python
def create_object():
    obj = Person("John")  # Object created
    return obj

p = create_object()
print("Function ended")
```

Output:

```
John is created!
Function ended
John is deleted!  # Destructor called automatically
```

3.4 Using Destructor to Release Resources

Destructors are useful for managing **files, database connections, or network resources**.

```python
class FileHandler:
    def __init__(self, filename):
        self.file = open(filename, "w")
        print(f"File {filename} opened.")

    def write_data(self, data):
        self.file.write(data)

    def __del__(self):
        self.file.close()
        print("File closed.")

# Using the class
file = FileHandler("test.txt")
file.write_data("Hello, World!")
del file  # File closed automatically
```

---

4. Constructors and Destructors in Django

4.1 Using `__init__()` in Django Models

In Django, the `__init__()` method is used to **customize** object creation.

```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        print(f"Product {self.name} created!")

# Creating an object in Django shell
# python manage.py shell
# >>> p = Product(name="Laptop", price=1200)
# >>> p.save()
# Output: Product Laptop created!
```

4.2 Destructor in Django

Django does not directly use destructors like in Python classes because Django models interact with the database. However, we can use **signals** to execute actions before deletion.

Example using the `pre_delete` signal:

```python
from django.db.models.signals import pre_delete
from django.dispatch import receiver

@receiver(pre_delete, sender=Product)
def product_deleting(sender, instance, **kwargs):
    print(f"Deleting product: {instance.name}")

# When deleting a product:
# >>> p.delete()
# Output: Deleting product: Laptop
```

---

5. Exercises

Exercise 1: Implement a Constructor

Create a class `Book` with attributes `title`, `author`, and `price`. Initialize these attributes using a constructor and create an object.

Exercise 2: Implement a Destructor

Modify the `Book` class by adding a destructor that prints `"Book deleted!"` when an object is destroyed.

Exercise 3: Using `super().__init__()`

Create a class `Electronic` with an attribute `category`. Create a subclass `Phone` that inherits from `Electronic` and uses `super().__init__()`.

Exercise 4: Manage a File with a Class

Create a Python class called `FileWriter` that:

1. **Opens a file** in write mode when an object is created. The filename should be passed as an argument to the constructor.

2. **Has a method** `write_data(self, data)` that writes the given text to the file.
3. **Closes the file automatically** when the object is deleted (in the destructor).
4. **Demonstrates the behavior** by creating an instance, writing some data, and then deleting the object.

Chapter 4 - Inheritance, Types of Inheritance, and Polymorphism

1. Introduction

In **Object-Oriented Programming (OOP)**, **inheritance** allows one class (child class) to acquire properties and methods from another class (parent class). This promotes **code reuse**, reducing redundancy.

Polymorphism enables objects of different classes to be treated as objects of a common superclass. This allows functions or methods to process different object types seamlessly.

---

2. Inheritance in Python

2.1 What is Inheritance?

Inheritance allows a class (**child class**) to derive properties and behaviors from another class (**parent class**).

Example of a parent class:

Python

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

    def make_sound(self):
        return "Some sound"
```

Example of a child class inheriting from `Animal`:

Python

```python
class Dog(Animal):
    def make_sound(self):  # Overriding the parent method
        return "Woof!"
```

```
dog = Dog("Buddy")
print(dog.name)          # Output: Buddy
print(dog.make_sound())  # Output: Woof!
```

---

3. Types of Inheritance

3.1 Single Inheritance

In single inheritance, a **child class** derives from **one parent class**.

Diagram:

```
     Vehicle
        |
        Car
```

Code:

Python

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

    def show_brand(self):
        print(f"Brand: {self.brand}")

class Car(Vehicle):  # Single Inheritance
    def __init__(self, brand, model):
        super().__init__(brand)
        self.model = model

    def show_model(self):
        print(f"Model: {self.model}")

car = Car("Toyota", "Corolla")
car.show_brand()  # Output: Brand: Toyota
car.show_model()  # Output: Model: Corolla
```

3.2 Multiple Inheritance

A class can inherit from **multiple parent classes**.

Diagram:

```
    Engine    Wheels
       \      /
        Truck
```

Code:

Python

```python
class Engine:
    def engine_type(self):
        return "Diesel Engine"

class Wheels:
    def wheel_count(self):
        return "Four Wheels"

class Truck(Engine, Wheels):  # Multiple Inheritance
    pass

truck = Truck()
print(truck.engine_type())  # Output: Diesel Engine
print(truck.wheel_count())  # Output: Four Wheels
```

⚠️ **Avoid multiple inheritance unless necessary**, as it can make debugging difficult.

3.3 Multilevel Inheritance

A class can inherit from another child class, forming a **chain of inheritance**.

Diagram:

```
    Animal
       |
     Mammal
```

```
        |
       Dog
```

Code:

Python

```python
class Animal:
    def make_sound(self):
        return "Some sound"

class Mammal(Animal):
    def has_fur(self):
        return True

class Dog(Mammal):  # Multilevel Inheritance
    def make_sound(self):
        return "Woof!"

dog = Dog()
print(dog.make_sound())  # Output: Woof!
print(dog.has_fur())     # Output: True
```

3.4 Hierarchical Inheritance

One parent class is inherited by **multiple child classes**.

Diagram:

```
       Animal
      /      \
     Cat      Dog
```

Code:

Python

```python
class Animal:
    def make_sound(self):
        return "Some sound"

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

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

cat = Cat()
dog = Dog()
print(cat.make_sound())  # Output: Meow!
print(dog.make_sound())  # Output: Woof!
```

3.5 Hybrid Inheritance

A combination of multiple inheritance types.

Diagram:

```
      A
     / \
    B   C
     \ /
      D
```

Code:

Python

```
class A:
    def method_A(self):
        return "Method A"

class B(A):
    def method_B(self):
        return "Method B"

class C(A):
    def method_C(self):
        return "Method C"

class D(B, C):  # Hybrid Inheritance
    def method_D(self):
```

```python
        return "Method D"

obj = D()
print(obj.method_A())  # Output: Method A
print(obj.method_B())  # Output: Method B
print(obj.method_C())  # Output: Method C
print(obj.method_D())  # Output: Method D
```

Hybrid inheritance should be used carefully as it can make debugging complex.

---

4. Polymorphism

4.1 What is Polymorphism?

Polymorphism allows **different classes** to be treated as **a single entity** by using a common interface.

4.2 Method Overriding

A **child class** provides a different implementation for a method already defined in the **parent class**.

```python
class Bird:
    def fly(self):
        return "I can fly!"

class Penguin(Bird):
    def fly(self):  # Overriding parent method
        return "I can't fly!"

bird = Bird()
penguin = Penguin()
print(bird.fly())    # Output: I can fly!
print(penguin.fly()) # Output: I can't fly!
```

4.3 Method Overloading (Achieved via Default Arguments)

Python does **not** support method overloading directly, but it can be achieved using default parameters.

```python
class MathOperations:
    def add(self, a, b, c=0):  # Default argument for overloading behavior
        return a + b + c

math = MathOperations()
print(math.add(2, 3))       # Output: 5
print(math.add(2, 3, 4))    # Output: 9
```

4.4 Operator Overloading

We can define how operators (`+` , `-` , `*` , etc.) behave for objects.

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

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

v1 = Vector(2, 3)
v2 = Vector(1, 5)
v3 = v1 + v2  # Uses __add__()
print(v3.x, v3.y)  # Output: 3 8
```

---

5. Inheritance and Polymorphism in Django

5.1 Inheritance in Django Models

Django models support **model inheritance**, which helps in code reuse.

Abstract Base Class

Useful when you want to share fields across multiple models but don’t want to create instances of the base class.

```python
from django.db import models

class BaseModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    class Meta:
        abstract = True  # This makes it an abstract class

class Product(BaseModel):
    name = models.CharField(max_length=100)
    price = models.FloatField()
```

Multi-table Inheritance

Each subclass gets its own table.

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

class Admin(User):  # Multi-table inheritance
    access_level = models.IntegerField()
```

---

6. Exercises

Exercise 1: Implement Single Inheritance (Detailed)

Problem: Create a class `Person` with attributes `name` (string) and `age` (integer). Create a subclass `Student` that inherits from `Person` and adds an attribute `grade` (string). Implement methods to display all attributes for both classes.

Example:

```
person = Person("Alice", 30)
student = Student("Bob", 20, "A")
person.display() # Output: Name: Alice, Age: 30
student.display() # Output: Name: Bob, Age: 20, Grade: A
```

Exercise 2: Implement Method Overriding (Detailed)

Problem: Create a parent class `Shape` with a method `area()` that returns 0 (base area). Create subclasses `Rectangle` and `Circle` that override `area()` to calculate the respective areas.

Example:

```
rectangle = Rectangle(5, 10)
circle = Circle(7)
print(rectangle.area()) # Output: 50
print(circle.area()) # Output: 153.93804002589985 (approximately)
```

Exercise 3: Operator Overloading (Detailed)

Problem: Create a class `BankAccount` with an attribute `balance` (float). Overload the `+` operator to merge the balances of two `BankAccount` objects and return a new `BankAccount` object with the combined balance.

Example:

```
account1 = BankAccount(100)
account2 = BankAccount(200)
merged_account = account1 + account2
print(merged_account.balance) # Output: 300
```

Exercise 4: Hybrid Inheritance (Detailed)

Problem: Create classes `ElectronicDevice`, `PortableDevice`, and `GPSDevice`. `SmartPhone` should inherit from `PortableDevice` and `GPSDevice`. `Tablet` should inherit from `PortableDevice`. `SmartWatch` should inherit from `PortableDevice` and `GPSDevice`. Each class should have a unique method that prints a message indicating its functionality.

Example:

```
smartphone = SmartPhone()
tablet = Tablet()
smartwatch = SmartWatch()
smartphone.make_call() # Output: Making a call...
smartwatch.track_location() # Output: Tracking location...
```

Chapter 5 - Operator Overloading

1. Introduction

Operator Overloading is a feature in Python that allows objects of user-defined classes to interact with built-in operators (`+` , `-` , `*` , `/` , `==` , etc.). By **overloading operators**, we can define how they behave when applied to objects of a particular class.

For example, in mathematical operations, `+` is used for addition:

```
print(2 + 3)  # Output: 5
```

But for objects, Python does not understand how to perform `+` unless we define it explicitly.

1.1 Why Use Operator Overloading?

- **Enhances readability**: Writing `a + b` is more intuitive than calling `a.add(b)` .
- **Custom behaviors**: Allows user-defined classes to behave like built-in types.
- **Improves code reusability**: Enables reuse of operators in object operations.

---

2. Magic Methods (Dunder Methods)

Python provides special **"magic methods"** (also called **dunder methods**, meaning "double underscore") that allow us to define custom behavior for operators.

2.1 Common Operator Overloading Methods

Operator	Method
`+`	`__add__(self, other)`
`-`	`__sub__(self, other)`
`*`	`__mul__(self, other)`
`/`	`__truediv__(self, other)`
`//`	`__floordiv__(self, other)`
`%`	`__mod__(self, other)`
`**`	`__pow__(self, other)`

Operator	Method
`==`	`__eq__(self, other)`
`!=`	`__ne__(self, other)`
`<`	`__lt__(self, other)`
`<=`	`__le__(self, other)`
`>`	`__gt__(self, other)`
`>=`	`__ge__(self, other)`

---

3. Implementing Operator Overloading

3.1 Overloading the `+` Operator

Let's say we have a class `Vector`, and we want to add two vectors using `+`:

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

    def __add__(self, other):  # Overloading +
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2  # Calls __add__
print(v3)  # Output: Vector(6, 8)
```

3.2 Overloading the `*` Operator

We can overload `*` to scale a vector:

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

    def __mul__(self, scalar):  # Overloading *
        return Vector(self.x * scalar, self.y * scalar)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v = Vector(3, 4)
v_scaled = v * 2  # Calls __mul__
print(v_scaled)  # Output: Vector(6, 8)
```

3.3 Overloading Comparison Operators

We can overload comparison operators like `>`, `<`, `==`:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def __gt__(self, other):  # Overloading >
        return self.area() > other.area()

    def __eq__(self, other):  # Overloading ==
        return self.area() == other.area()

rect1 = Rectangle(5, 10)
rect2 = Rectangle(4, 12)

print(rect1 > rect2)  # Output: True
print(rect1 == rect2) # Output: False
```

3.4 Overloading `__str__` and `__repr__`

By default, printing an object gives an unreadable output:

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

p = Person("Alice", 25)
print(p)  # Output: <__main__.Person object at 0x...>
```

We can override `__str__` and `__repr__` for a readable output:

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

    def __str__(self):  # Readable output for users
        return f"Person(name={self.name}, age={self.age})"

    def __repr__(self):  # Debugging output for developers
        return f"Person({self.name}, {self.age})"

p = Person("Alice", 25)
print(str(p))   # Output: Person(name=Alice, age=25)
print(repr(p))  # Output: Person(Alice, 25)
```

---

4. Operator Overloading in Django Models

Django models often need custom operators, especially for comparison or mathematical operations.

4.1 Overloading `__str__` in Django Models

Django **automatically** calls `__str__` when displaying objects in the admin panel.

```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.FloatField()
    def __str__(self):  # Overloading __str__
        return f"{self.name} - ${self.price}"
```

4.2 Overloading Comparison Operators in Django Models

For comparing products by price:

```python
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.FloatField()

    def __gt__(self, other):  # Overloading >
        return self.price > other.price
```

---

5. Exercises

Exercise 1: Overload the `+` operator

Create a class `BankAccount` that allows adding two accounts using `+` to merge their balances.

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

    def __add__(self, other):
        # Your code here

# Test your implementation
acc1 = BankAccount(500)
acc2 = BankAccount(300)
merged_acc = acc1 + acc2
print(merged_acc.balance)  # Expected Output: 800
```

Exercise 2: Overload the `>`, `<`, `==` operators

Create a class `Student` where students can be compared based on their grades.

```python
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    # Your code here (Overload >, <, ==)

# Test your implementation
s1 = Student("Alice", 85)
s2 = Student("Bob", 90)
print(s1 > s2)  # Expected Output: False
print(s1 == s2) # Expected Output: False
```

Exercise 3: Overload `__str__` and `__repr__`

Modify the `Student` class to print readable output when using `print(student)`.

Chapter 6 - Iterators and Generators

1. Introduction to Iterators

In Python, **iterators** are objects that allow us to traverse through a collection (like a list, tuple, dictionary) one element at a time. Iterators implement two key methods:

1. `__iter__()` : This method returns the iterator object itself. It's used in loops to initialize the iteration process.
2. `__next__()` : This method returns the next item in the collection. When there are no more items, it raises the `StopIteration` exception.

---

2. Creating an Iterator Class

An **iterator** must implement both `__iter__()` and `__next__()` methods. Let’s look at an example where we create a simple iterator to traverse through a custom range of numbers.

2.1 Example: Custom Range Iterator

```
class MyRange:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.current = start

    def __iter__(self):
        return self  # Return the iterator object itself

    def __next__(self):
        if self.current >= self.end:
            raise StopIteration  # Stop the iteration
        self.current += 1
        return self.current - 1

# Using the iterator
for number in MyRange(1, 5):
    print(number)
```

Output:

```
1
2
3
4
```

In this example, the class `MyRange` defines an iterator that starts from a given `start` and iterates until `end`. The `__next__()` method raises a `StopIteration` exception when there are no more elements to return.

---

3. Iterators in Django Models

In Django, iterators can be useful when handling large datasets, such as when fetching records from the database. Instead of loading all the records into memory, which might be inefficient for large datasets, iterators can be used to iterate through database querysets efficiently.

3.1 Example: Using `iterator()` with QuerySets

Django QuerySets support the `iterator()` method, which retrieves rows from the database one at a time, saving memory when dealing with large queries.

```python
from myapp.models import Product

# Using the iterator method to fetch products efficiently
for product in Product.objects.iterator():
    print(product.name, product.price)
```

This approach is more memory efficient because it avoids fetching all records at once.

---

4. Introduction to Generators

Generators are a simple and powerful tool in Python to create iterators. Instead of implementing `__iter__()` and `__next__()` methods manually, we can use Python's `yield` statement to make a function a generator. The `yield` statement produces a value and pauses the function's execution, which allows it to be resumed later when `next()` is called.

4.1 Example: Simple Generator

```python
def my_range(start, end):
    while start < end:
        yield start  # Yield the current value and pause
        start += 1

# Using the generator
for number in my_range(1, 5):
    print(number)
```

Output:

```
1
2
3
4
```

Here, the function `my_range` is a generator. Each time `yield` is called, the function's state is saved, and the value is returned to the caller. The function picks up where it left off the next time `next()` is called.

---

5. Advantages of Using Generators

- **Memory Efficient**: Generators produce items one at a time and only when needed. This makes them much more memory efficient than creating and returning entire lists.
- **Lazy Evaluation**: Values are produced on demand, which can be useful for handling large datasets or infinite sequences.
- **Cleaner Code**: Using `yield` makes the code more concise and avoids the need to explicitly manage an internal iterator.

5.1 Example: Generator with Infinite Sequence

```python
def infinite_counter():
    count = 0
    while True:
        yield count
        count += 1

# Using the generator
gen = infinite_counter()
```

```
for i in range(5):
    print(next(gen))
```

Output:

```
0
1
2
3
4
```

In this case, the generator produces an infinite sequence of numbers, but we can control how many numbers we want by using `next()`.

---

6. Using Generators for Django Tasks

Generators are particularly useful when dealing with tasks like:

- Streaming large datasets from a database
- Implementing paginated data fetching for APIs

For example, let's imagine you want to create a paginated API for fetching `Product` records from your database in chunks:

6.1 Example: Paginated Generator for API

```
from myapp.models import Product

def get_paginated_products(page_size):
    page = 0
    while True:
        products = Product.objects.all()[page * page_size: (page + 1) *
page_size]
        if not products:
            break
        for product in products:
            yield product
        page += 1

# Fetching products using the generator
```

```python
for product in get_paginated_products(10):
    print(product.name)
```

In this example, we use a generator to fetch products in chunks of 10. Each time `next()` is called, a new page of products is retrieved.

---

7. Exercise

Exercise 1: Implementing a Custom Iterator

Create a class `BookShelf` that holds a list of books and allows you to iterate over the books. Implement the `__iter__()` and `__next__()` methods.

```python
class BookShelf:
    def __init__(self, books):
        self.books = books
        self.current = 0

    def __iter__(self):
        return self  # Return the iterator object itself

    def __next__(self):
        if self.current >= len(self.books):
            raise StopIteration  # Stop the iteration
        book = self.books[self.current]
        self.current += 1
        return book

# Test your implementation
shelf = BookShelf(["Python Basics", "Advanced Django", "Data Science"])
for book in shelf:
    print(book)
```

Exercise 2: Generator for Fibonacci Sequence

Create a generator `fibonacci(n)` that generates the Fibonacci sequence up to the nth number.

```python
def fibonacci(n):
    a, b = 0, 1
    while n > 0:
        yield a
        a, b = b, a + b
        n -= 1

# Test your generator
for num in fibonacci(10):
    print(num)
```

Exercise 3: Django QuerySet Generator

Create a generator that fetches products from the database in chunks of 5 and prints their names.

```python
from myapp.models import Product

def fetch_products_in_chunks():
    page = 0
    while True:
        products = Product.objects.all()[page * 5 : (page + 1) * 5]
        if not products:
            break
        for product in products:
            yield product
        page += 1

# Test the generator
for product in fetch_products_in_chunks():
    print(product.name)
```

Chapter 7 - Closures and Decorators

1. Introduction to Closures

A **closure** is a function that has access to variables from its enclosing scope, even after the outer function has finished execution. It allows a function to remember and access the environment in which it was created.

1.1 How Closures Work

A closure occurs when a nested function refers to a variable from its enclosing function, and the outer function has finished execution. This is often used to implement functions with state or configuration that should be remembered across multiple calls.

1.2 Example of a Closure

Here’s a simple example to demonstrate how closures work:

```python
def outer_function(x):
    def inner_function(y):
        return x + y  # 'x' is captured by the closure
    return inner_function

# Create a closure where x is remembered
add_five = outer_function(5)
print(add_five(10))  # Output: 15
```

In this example, `inner_function` is a closure because it references `x`, which is defined in the outer scope of `outer_function`. Even after `outer_function` finishes execution, `inner_function` still "remembers" the value of `x`.

---

2. Practical Use of Closures

Closures can be particularly useful for tasks such as:

- **Function Configuration**: You can create customized functions with different behaviors by passing parameters to the outer function and returning a function with those parameters fixed.

- **Data Encapsulation**: Closures help encapsulate data and prevent it from being accessed or modified directly from outside.

2.1 Example: Using Closures for Configuration

```python
def make_multiplier(factor):
    def multiplier(number):
        return number * factor
    return multiplier

# Create a multiplier that always multiplies by 3
multiply_by_3 = make_multiplier(3)
print(multiply_by_3(5))  # Output: 15

# Create a multiplier that always multiplies by 5
multiply_by_5 = make_multiplier(5)
print(multiply_by_5(5))  # Output: 25
```

In this example, `make_multiplier` is a factory function that returns a closure `multiplier` which remembers the `factor` it was created with.

---

3. Introduction to Decorators

A **decorator** is a design pattern in Python that allows you to modify or extend the behavior of functions or methods without changing their actual code. Decorators are often used to wrap a function and add functionality to it, like logging, timing, or authorization checks.

3.1 How Decorators Work

A decorator is essentially a function that takes another function as an argument and returns a new function that typically extends the behavior of the original function.

3.2 Syntax of a Decorator

The basic syntax of a decorator in Python is as follows:

```python
def decorator_function(original_function):
    def wrapper_function():
        print("Wrapper executed this before
{}".format(original_function.__name__))
```

```
        return original_function()
    return wrapper_function
```

You can apply the decorator to a function using the `@` symbol:

```
@decorator_function
def display():
    return "Display function executed"

# Calling the decorated function
print(display())
```

In this case, `decorator_function` wraps the `display` function, modifying its behavior.

---

4. Practical Use of Decorators

Decorators are commonly used in:

- **Logging**: Adding logging to functions automatically without modifying their code.
- **Authentication**: Checking if a user is authorized before executing a function (e.g., in web applications).
- **Memoization**: Caching the results of function calls to improve performance.

4.1 Example: Simple Logging Decorator

Here's an example of a simple decorator that logs the function name and its arguments:

```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Function {func.__name__} was called with arguments {args} and
keyword arguments {kwargs}")
        return func(*args, **kwargs)
    return wrapper

# Applying the decorator
@log_function_call
def add(a, b):
    return a + b

# Calling the decorated function
print(add(5, 3))
```

Output:

```
Function add was called with arguments (5, 3) and keyword arguments {}
8
```

The decorator `log_function_call` adds logging functionality to the `add` function without modifying its original code.

---

5. Decorators with Arguments

Sometimes, you might want to pass arguments to your decorators. To do this, you need to create a decorator factory function that takes arguments and returns a decorator.

5.1 Example: Parameterized Decorator

```python
def repeat(n):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator_repeat

@repeat(3)
def greet(name):
    print(f"Hello, {name}")

greet("Alice")
```

Output:

```
Hello, Alice
Hello, Alice
Hello, Alice
```

In this example, the `repeat` decorator takes an argument `n` and repeats the execution of the decorated function `n` times.

---

6. Using Decorators in Django

In Django, decorators are widely used for tasks like:

- **View permissions**: Check if the user has permission to access a view.
- **Caching**: Cache the response of a view function.
- **CSRF protection**: Automatically add protection to POST requests.

6.1 Example: Django View Decorators

Here's how decorators are used in Django views:

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

@login_required
def my_view(request):
    return HttpResponse("This is a private view")

# The @login_required decorator ensures that only authenticated users can
access the view.
```

In this example, the `@login_required` decorator checks whether the user is logged in before executing the view function.

---

7. Exercise

Exercise 1: Implementing a Simple Closure

Create a closure called `make_adder` that takes an integer `x` and returns a function that adds `x` to another number `y`.

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

# Test the closure
add_10 = make_adder(10)
print(add_10(5))  # Output: 15
```

Exercise 2: Creating a Decorator

Write a decorator that logs the time a function takes to execute.

```python
import time

def log_execution_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Function {func.__name__} took {end_time - start_time:.4f}
seconds to execute.")
        return result
    return wrapper

@log_execution_time
def some_long_function():
    time.sleep(2)  # Simulate a delay

some_long_function()
```

Exercise 3: Applying Decorators in Django

Create a custom decorator that checks if a user is an admin before allowing access to a view. If the user is not an admin, redirect them to the login page.

```python
from django.shortcuts import redirect
from django.http import HttpResponse

def admin_required(view_func):
    def wrapper(request, *args, **kwargs):
        if not request.user.is_superuser:
            return redirect('login')
        return view_func(request, *args, **kwargs)
    return wrapper

# Applying the decorator to a view
@admin_required
```

```python
def admin_dashboard(request):
    return HttpResponse("Welcome to the admin dashboard!")
```

Chapter 8 - Exceptions

1. Introduction to Exceptions

An **exception** is an event that disrupts the normal flow of a program's execution. When a function detects a problem, it raises an exception, which can be caught and handled to prevent the program from crashing.

1.1 What is an Exception?

An exception occurs when the program encounters an unexpected situation, like a division by zero, file not found, or an invalid operation. Python provides a robust way to handle such issues gracefully using `try`, `except`, `else`, and `finally` blocks.

---

2. Raising and Catching Exceptions

2.1 Raising Exceptions

In Python, you can raise exceptions using the `raise` keyword. This is often done when a certain condition occurs that makes it impossible to continue the program's execution.

Syntax:

```python
raise Exception("This is a custom error message")
```

2.2 Catching Exceptions

The `try` block allows you to test a block of code for errors. If an error occurs, Python will move to the `except` block to handle it.

Syntax:

```python
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error occurred: {e}")
```

In the above example, the `ZeroDivisionError` exception is caught and handled, preventing the program from crashing.

---

3. Types of Exceptions

Python has several built-in exceptions, and you can define custom exceptions as well. Some common built-in exceptions are:

- `ZeroDivisionError`: Raised when dividing by zero.
- `FileNotFoundError`: Raised when trying to open a file that does not exist.
- `ValueError`: Raised when a function receives an argument of the correct type but an inappropriate value.
- `IndexError`: Raised when trying to access an element of a list using an invalid index.
- `KeyError`: Raised when trying to access a dictionary with a key that does not exist.
- `TypeError`: Raised when an operation or function is applied to an object of inappropriate type.

---

4. The `try`, `except`, `else`, and `finally` Blocks

4.1 `try` Block

The `try` block is where you write the code that might raise an exception.

4.2 `except` Block

The `except` block is used to catch and handle exceptions that are raised in the `try` block.

4.3 `else` Block

The `else` block executes if no exception is raised in the `try` block.

4.4 `finally` Block

The `finally` block is always executed, no matter what, even if an exception is raised or not. It’s used for cleanup activities, such as closing files or releasing resources.

Syntax:

```python
try:
    # Code that may raise an exception
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
else:
    print(f"Result is: {result}")
finally:
    print("Execution completed.")
```

In this example:

- The `try` block attempts to get a number from the user and perform a division.
- If the user inputs an invalid number or tries to divide by zero, the `except` block handles the exception.
- The `else` block executes only if no exception occurred.
- The `finally` block always executes, ensuring that any final actions, like cleanup, are performed.

---

5. Custom Exceptions

Python allows you to create your own custom exceptions. This can be useful when you want to raise an exception specific to your application.

5.1 Defining Custom Exceptions

To create a custom exception, you define a new class that inherits from the `Exception` class.

Example:

```python
class InvalidAgeError(Exception):
    def __init__(self, message="Age must be a positive number"):
        self.message = message
        super().__init__(self.message)

# Raising a custom exception
def check_age(age):
    if age < 0:
        raise InvalidAgeError("Age cannot be negative!")
    return f"Your age is {age}"

try:
    print(check_age(-5))
except InvalidAgeError as e:
    print(f"Error: {e}")
```

In this example:

- `InvalidAgeError` is a custom exception that inherits from Python's built-in `Exception` class.
- The `check_age` function raises this custom exception if the age is negative.

---

6. Using Exceptions in Django

In Django, exceptions are used to handle errors gracefully, such as missing data, incorrect user input, or database connection issues. Django provides built-in exceptions to handle common scenarios.

6.1 Django's Built-In Exceptions

Some common exceptions in Django include:

- `Http404` : Raised when a requested page is not found.
- `ValidationError` : Raised when form validation fails.
- `PermissionDenied` : Raised when the user does not have the required permissions.

Example: Handling Http404 in Views

```
from django.http import Http404

def get_article(request, article_id):
    try:
        article = Article.objects.get(id=article_id)
    except Article.DoesNotExist:
        raise Http404("Article not found")
    return render(request, "article_detail.html", {"article": article})
```

In this example, the `Http404` exception is raised if the requested article does not exist in the database.

6.2 Custom Django Exceptions

You can define custom exceptions in Django, just like in regular Python. Custom exceptions can be used to handle application-specific errors.

Example: Custom Exception in Django

```python
class CustomValidationError(Exception):
    pass

def validate_data(data):
    if not data:
        raise CustomValidationError("Data cannot be empty")

# Using the custom exception
try:
    validate_data("")
except CustomValidationError as e:
    print(f"Validation failed: {e}")
```

---

7. Best Practices for Exception Handling

7.1 Avoiding Empty `except` Blocks

Catching all exceptions with a general `except` block can hide bugs. Always catch specific exceptions whenever possible.

7.2 Using Exceptions for Control Flow

While exceptions should be used for handling errors, avoid using them for regular control flow, as this can make your code harder to read and maintain.

7.3 Logging Exceptions

It's important to log exceptions for debugging and troubleshooting. In Python, the `logging` module can be used to log errors.

Example: Logging Exceptions

```python
import logging
```

```python
# Setting up logging
logging.basicConfig(filename="app.log", level=logging.ERROR)

try:
    1 / 0
except ZeroDivisionError as e:
    logging.error(f"Error occurred: {e}")
```

---

8. Exercise

Exercise 1: Handling Multiple Exceptions

Write a Python function `divide_numbers(a, b)` that divides two numbers. The function should:

- Raise a `ZeroDivisionError` if `b` is 0.
- Raise a `ValueError` if either `a` or `b` is not a number. Handle the exceptions in the `try` and `except` blocks and print appropriate error messages.

```python
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero")
    except ValueError:
        print("Error: Both inputs must be numbers")
    else:
        print(f"Result: {result}")
```

Exercise 2: Custom Exception for Invalid Input

Create a custom exception called `InvalidInputError`. Write a function `get_age()` that prompts the user for an age. If the user enters a negative age, raise an `InvalidInputError` with the message "Age cannot be negative".

```python
class InvalidInputError(Exception):
    pass

def get_age():
    try:
        age = int(input("Enter your age: "))
        if age < 0:
            raise InvalidInputError("Age cannot be negative")
        print(f"Your age is {age}")
    except InvalidInputError as e:
        print(f"Error: {e}")
```

Exercise 3: Django Exception Handling

Create a Django view that retrieves a `Product` by its ID. If the product is not found, raise a `Http404` exception with a custom message "Product not found".

```
from django.http import Http404
from .models import Product

def get_product(request, product_id):
    try:
        product = Product.objects.get(id=product_id)
    except Product.DoesNotExist:
        raise Http404("Product not found")
    return render(request, "product_detail.html", {"product": product})
```

Chapter 9 - Testing

Testing is an essential part of software development that ensures the code behaves as expected. This chapter will focus on different testing methodologies and tools, including **Pytest**, **Doctests**, **UnitTest**, and **nose**, and how they can be used to test Python applications, especially Django projects.

---

1. Introduction to Testing in Python

Testing helps identify bugs, verify the correctness of the code, and ensure that changes don't break existing functionality. In Python, testing is typically done using frameworks that provide tools for writing and running tests.

---

2. Pytest

Pytest is one of the most popular and powerful testing frameworks for Python. It's known for its simplicity and ease of use, but it also offers advanced features for complex test suites.

2.1 Installing Pytest

To install Pytest, you can use `pip`:

```
pip install pytest
```

2.2 Writing Tests with Pytest

A basic Pytest test function starts with the keyword `test_` in the function name. This helps Pytest identify which functions are test cases.

Example:

```python
# test_example.py
def test_addition():
    assert 1 + 1 == 2
```

```python
def test_subtraction():
    assert 2 - 1 == 1
```

2.3 Running Tests

You can run Pytest from the command line by navigating to your test folder and running the command:

```
pytest
```

Pytest will automatically discover and run all tests in files starting with `test_` and containing functions starting with `test_`.

2.4 Pytest Features

- **Assertions**: Pytest uses simple Python `assert` statements for testing conditions.
- **Fixtures**: Pytest allows the use of fixtures for setting up test data or test environments.
- **Parameterized Tests**: Pytest can run the same test with different input data using `@pytest.mark.parametrize`.

Example: Parametrized Test

```python
import pytest

@pytest.mark.parametrize("a, b, expected", [(1, 1, 2), (2, 3, 5), (0, 0, 0)])
def test_add(a, b, expected):
    assert a + b == expected
```

---

3. Doctests

Doctests are a simple way to test Python code embedded within docstrings. They allow you to write test cases as part of your documentation.

3.1 Using Doctests

Doctests are written within the docstrings of Python functions. The test cases are written as if they were part of an interactive Python session.

Example:

```python
def add(a, b):
    """
    Add two numbers together.

    Example:
    >>> add(2, 3)
    5
    >>> add(-1, 1)
    0
    """
    return a + b
```

To run doctests, use the following command:

```
python -m doctest -v <your_module>.py
```

3.2 Advantages of Doctests

- It's simple to use and keeps documentation and tests in sync.
- Great for small scripts, utilities, or educational purposes.
- Keeps your documentation interactive and testable.

---

4. UnitTest / pyUnit

UnitTest is Python's built-in testing framework, inspired by Java's JUnit. It provides a rich set of tools for testing, organizing, and running tests. `unittest` comes bundled with Python, so no installation is needed.

4.1 Writing Tests with UnitTest

UnitTest uses classes to organize tests, and test methods must start with `test_`. The test methods include assertions to check whether the code produces the expected output.

Example:

```python
import unittest

class TestMathOperations(unittest.TestCase):

    def test_addition(self):
        self.assertEqual(1 + 1, 2)

    def test_subtraction(self):
        self.assertEqual(2 - 1, 1)

    def test_multiplication(self):
        self.assertEqual(2 * 3, 6)

    def test_division(self):
        self.assertEqual(6 / 2, 3)

if __name__ == '__main__':
    unittest.main()
```

4.2 Running UnitTests

UnitTest tests are run using the following command:

```
python -m unittest test_module.py
```

4.3 UnitTest Assertions

- `assertEqual(a, b)` : Checks if `a` is equal to `b` .
- `assertNotEqual(a, b)` : Checks if `a` is not equal to `b` .
- `assertTrue(a)` : Checks if `a` is true.
- `assertFalse(a)` : Checks if `a` is false.
- `assertRaises(ExceptionType)` : Checks if an exception is raised.

---

5. nose

nose is another testing framework that extends `unittest` . It offers features like test discovery, result reporting, and handling of fixtures, which make it a great option for larger projects.

5.1 Installing nose

Install `nose` using pip:

```
pip install nose
```

5.2 Writing Tests with nose

Nose can discover and run tests in modules and classes that start with `test_`. The tests themselves can use `unittest` assertions.

Example:

```
def test_addition():
    assert 1 + 1 == 2

def test_subtraction():
    assert 2 - 1 == 1
```

5.3 Running nose Tests

Run nose tests using:

```
nosetests
```

Nose will automatically find all the test functions in your project and execute them.

---

6. Using Testing with Django

Django supports testing through the `unittest` module, but it also offers its own testing framework that extends `unittest`. Django's testing framework provides tools like test clients to simulate user requests and model tests.

6.1 Django Test Case

Django’s `TestCase` class extends `unittest.TestCase` and provides additional methods like `setUp()` for setting up data before each test and `client` to simulate HTTP requests.

Example:

```
from django.test import TestCase
from .models import Product

class ProductTestCase(TestCase):

    def setUp(self):
        Product.objects.create(name="Test Product", price=10.99)

    def test_product_name(self):
        product = Product.objects.get(name="Test Product")
        self.assertEqual(product.name, "Test Product")

    def test_product_price(self):
        product = Product.objects.get(name="Test Product")
        self.assertEqual(product.price, 10.99)
```

6.2 Running Django Tests

To run Django tests, use the following command:

```
python manage.py test
```

Django will automatically discover all test cases within your app and execute them.

---

7. Best Practices for Writing Tests

7.1 Write Testable Code

Ensure that your code is modular and each component is easy to test. This makes it easier to write tests and ensures that individual components can be tested independently.

7.2 Keep Tests Small and Focused

Each test should test one specific piece of functionality. Keeping tests small and focused makes them easier to maintain and understand.

7.3 Use Mocking and Fixtures

Use mock objects and fixtures to isolate the functionality being tested. This is especially useful when working with databases, external APIs, or other systems that are difficult to test directly.

7.4 Automate Your Tests

Automate your test suite to run tests on every change, especially when working with version control. Use Continuous Integration (CI) services like Travis CI, CircleCI, or GitHub Actions to run tests automatically when code is pushed.

8. Exercise

Exercise 1: Pytest Practice

Write a Pytest function that tests the following functions:

```python
def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

Write tests to:

1. Check that multiplication works correctly.
2. Check that division works correctly, and raises a `ValueError` when dividing by zero.

Exercise 2: Doctests

Write a Python function with a doctest:

```python
def square(x):
    """
    Returns the square of a number.

    Example:
    >>> square(2)
    4
    >>> square(3)
    9
    """
    return x * x
```

Run the doctests and ensure they pass.

Exercise 3: UnitTest Practice

Create a `Calculator` class with methods to add, subtract, multiply, and divide. Write unit tests for these methods using the `unittest` framework to check if the methods perform correctly.

```python
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b
```