Here's a **formatted roadmap** for your OOP learning journey, ready to be copied and converted into a PDF. I've made the structure and examples easy to differentiate.

Object-Oriented Programming (OOP) in Python: A Roadmap

Stage 1: Understanding the Basics of OOP

1. What is OOP?

- **Definition**: A programming paradigm based on "objects," which bundle data and methods together.
- **Key Principles**:
    1. **Encapsulation**: Restrict access to internal details.
    2. **Inheritance**: Reuse and extend existing code.
    3. **Polymorphism**: Same interface, different behavior.
    4. **Abstraction**: Hide unnecessary details and expose only essential features.

---

2. Why Use OOP?

- **Advantages**:
    - Reusability of code.
    - Modular structure for complex systems.
    - Easier to debug, extend, and maintain.

---

3. Classes and Objects

- **Class**: A blueprint for creating objects.
- **Object**: An instance of a class.

Example:

```
class MyClass:
    pass

obj = MyClass()  # Create an object
```

Practice:

- Create a class `Car` with attributes like `brand` and `model`.

- Create objects for different cars and print their details.

Stage 2: Diving into OOP Concepts

4. Attributes and Methods

- **Instance Attributes**: Specific to each object.
- **Class Attributes**: Shared across all objects.
- **Methods**: Functions defined in a class.

Example:

```
class Car:
    wheels = 4  # Class Attribute

    def __init__(self, brand, model):
        self.brand = brand  # Instance Attribute
        self.model = model

    def start(self):
        print(f"{self.brand} {self.model} is starting.")
```

Practice:

- Add attributes like `year` and `color` to `Car`.
- Define methods like `stop()` or `honk()`.

---

5. Encapsulation

- Restrict access to internal details of an object using private attributes (`__attribute`) and getters/setters.

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private Attribute

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

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

Practice:

- Create a class `Student` with private attributes for grades.
- Add methods to modify and access grades securely.

---

6. Inheritance

- **Definition**: Reuse and extend code from an existing class.

Example:

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

class Bike(Vehicle):
    def ride(self):
        print(f"Riding a {self.brand} bike.")
```

Practice:

- Create a `Truck` class inheriting from `Vehicle`.
- Add specific methods like `load()`.

---

7. Polymorphism

- **Definition**: Same interface with different behaviors (e.g., method overriding).

Example:

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("Bark!")

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

Practice:

- Add more animal classes and override their `speak()` methods.

---

8. Abstraction

- **Definition**: Hiding complex details using abstract classes or interfaces.

Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

Practice:

- Create an abstract class `Appliance` with methods like `turn_on()` and `turn_off()`.
- Implement it in classes like `WashingMachine` or `Refrigerator`.

Stage 3: Advanced OOP Techniques

9. Magic Methods and Dunder Methods

- **Definition**: Special methods prefixed and suffixed with double underscores (e.g., `__init__`, `__str__`, `__add__`).

Example:

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

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

Practice:

- Implement `__sub__` for subtraction of two objects.

---

10. Composition

- **Definition**: Combining objects of different classes.

Example:

```
class Engine:
    def start(self):
        print("Engine started.")
```

```
class Car:
    def __init__(self):
        self.engine = Engine()  # Composition

    def start(self):
        self.engine.start()
        print("Car is ready to go.")
```

Practice:

- Add `Battery` to `Car` and use it via composition.

---

11. Static and Class Methods

- **Static Method**: Use `@staticmethod` for utility methods.
- **Class Method**: Use `@classmethod` for methods affecting the class as a whole.

Example:

```
class Math:
    @staticmethod
    def add(x, y):
        return x + y

    @classmethod
    def info(cls):
        print(f"This is the {cls.__name__} class.")
```

Practice:

- Create static methods for calculations like multiplication and division.

Stage 4: Practical Applications

12. Build Small Projects

- **Library Management System**:
  - Use classes like `Book`, `Library`, and `Member`.
- **Banking System**:
  - Include account management and transactions.
- **Inventory System**:
  - Use OOP to track stock, sales, and purchases.

---

13. Test OOP Knowledge

- Use unit tests (`unittest` module) to test your OOP code.

Example:

```
import unittest

class TestCar(unittest.TestCase):
    def test_start(self):
        car = Car()
        self.assertTrue(car.start())
```

Stage 5: Explore Advanced Topics

14. Design Patterns

- Learn common OOP patterns:
    - Singleton, Factory, Observer, etc.

15. Mix OOP with Other Paradigms

- Combine OOP with Functional Programming for a hybrid approach.

Tips to Master OOP

1. Practice daily by solving small problems.
2. Read others' code on platforms like GitHub.
3. Build real-world projects to reinforce concepts.

This roadmap is structured for a smooth and logical progression into OOP. Would you like assistance creating examples for projects or additional practice problems?