UnCodemy

Python OOPs Notes PDF

Object-oriented programming in Python organises code into classes that bundle data and behaviour. These notes explain classes and objects, the __init__ method, instance and class attributes, methods, encapsulation, inheritance, polymorphism, special methods and dataclasses, using short examples that you can run and change.

Key takeaways

- A class is a blueprint; an object is one instance made from it.
- The **init** method sets up each new object, and self refers to that object.
- Inheritance reuses a parent class; polymorphism lets different classes share a method name.
- Python uses naming conventions for privacy, not strict enforcement.
- Dunder methods such as **str** and **eq** customise how objects behave.

What is Python OOPs?

Object-oriented programming (OOP) is a style of programming that models a problem as objects, each created from a class that defines its data (attributes) and actions (methods).

What is a class and an object?

A class describes what data something has and what it can do. An object is one real instance of that class. A `Student` class might store a name and marks, and have a method to compute the average. Each student object has its own values.

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

    def average(self):
        return sum(self.marks) / len(self.marks)

s1 = Student("Asha", [72, 85, 64])
print(s1.name, round(s1.average(), 2))
```

`__init__` runs when the object is created, and `self` is the object itself.

What is the difference between instance and class attributes?

Instance attributes belong to one object and are set with `self`. Class attributes are shared by all objects of the class.

```
class Course:
    provider = "Uncodemy"          # class attribute

    def __init__(self, title):
        self.title = title         # instance attribute

a = Course("SQL")
b = Course("Python")
print(a.provider, b.provider, a.title, b.title)
```

What are the kinds of methods?

Instance methods use `self`. **Class methods** use `@classmethod` and receive the class as `cls`, often as alternative constructors. **Static methods** use `@staticmethod` and receive neither; they are plain functions placed in the class for organisation.

```
class Temp:
    def __init__(self, celsius):
        self.celsius = celsius

    @classmethod
    def from_fahrenheit(cls, f):
        return cls((f - 32) * 5 / 9)

    @staticmethod
    def is_valid(c):
        return c >= -273.15

print(round(Temp.from_fahrenheit(212).celsius, 1), Temp.is_valid(-300))
```

What is encapsulation?

Encapsulation keeps data and the code that uses it together and controls access. Python relies on convention: a single underscore (`_balance`) means "internal", and a double underscore (`__balance`) triggers name mangling to avoid accidental clashes. Use properties to control access.

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

    @property
    def balance(self):
        return self._balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive")
        self._balance += amount

acc = Account(100)
acc.deposit(50)
print(acc.balance)
```

How does inheritance work?

A child class reuses and extends a parent class. `super()` calls the parent's version of a method.

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

    def role(self):
        return "Employee"

class Manager(Employee):
    def __init__(self, name, salary, team):
        super().__init__(name, salary)
        self.team = team

    def role(self):
        return "Manager of " + str(len(self.team))

m = Manager("Ravi", 50000, ["A", "B"])
print(m.name, m.role(), isinstance(m, Employee))
```

Python supports multiple inheritance; the method resolution order (`ClassName.__mro__`) decides which parent is used first.

What is polymorphism?

Different classes can provide the same method name, and calling code does not need to know which class it has. This is sometimes called duck typing: if it has the method you need, it works.

```
class Cat:
    def speak(self): return "Meow"

class Dog:
    def speak(self): return "Woof"

for animal in [Cat(), Dog()]:
    print(animal.speak())
```

What are dunder methods?

Special methods with double underscores customise behaviour: `__str__` for readable text, `__repr__` for debugging, `__eq__` for equality, `__len__` for `len()`, `__add__` for `+` and `__iter__` for loops.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __str__(self):
        return f"({self.x}, {self.y})"
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

print(Point(1, 2), Point(1, 2) == Point(1, 2))
```

What are dataclasses?

The `dataclasses` module generates `__init__`, `__repr__` and `__eq__` for classes that mainly hold data.

```
from dataclasses import dataclass

@dataclass
class Item:
    name: str
    price: float
    qty: int = 1

print(Item("Pen", 10.0))
```

When should I use OOP?

Use classes when you have data with related behaviour and several similar objects. For small scripts, plain functions are fine. Prefer composition (an object holding other objects) over deep inheritance, because deep hierarchies are hard to change.

What mistakes do beginners make?

Forgetting `self`, using a mutable default argument such as `def f(items=[])`, putting everything in one large class and confusing class and instance attributes.

What practice programs help?

Build a library system with Book and Member classes, a bank account with deposit and withdraw rules, a shape hierarchy with an area method, a simple inventory with a class that stores items and totals them, and a to-do manager with a Task class. For each, write the classes first on paper, listing attributes and methods, then code them and add three small tests. Explaining why you chose each class boundary is good interview practice.

How are OOP questions asked in interviews?

Common questions are the difference between a class and an object, what self means, how inheritance and super work, what method overriding is, what encapsulation and abstraction mean in Python, what `__str__` and `__repr__` do, and the difference between class, static and instance methods. Use short code examples when you answer.

Frequently asked questions

What is the difference between a class and an object in Python?

A class is a blueprint that defines attributes and methods, and an object is one instance created from that class.

What does self mean in Python?

self refers to the current object inside a method, so the method can read and change that object attributes.

How does inheritance work in Python?

A child class extends a parent class, reusing its methods and calling the parent version with super() when needed.

What are dunder methods?

They are special methods with double underscores, such as __init__ and __str__, that customise how objects are created, printed and compared.

When should I use a dataclass?

Use a dataclass when a class mainly stores data, because it generates the init, repr and equality methods automatically.

Sources and further reading

- Python tutorial: Classes
- Python data model
- dataclasses module

About this PDF

These PDFs are study material for learning and revision. They are not a certificate, a degree or a promise of a job, and technology changes, so check official documentation for the latest details. Spot an error? Write to info@uncodemy.com and the trainer team will review it.

More free resources and courses: uncodemy.com | info@uncodemy.com | +91 98183 66550