OOP for Python

Absolutely! Here's a practical set of **Python OOP code examples** for each item in your **Object-Oriented Programming (OOP)** checklist — perfect for review or muscle-memory practice.

Object-Oriented Programming (OOP) in Python — With Code Examples

---

1. Create a class with `__init__()`

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


d = Dog("Fido")
print(d.name)  # Fido
```

---

2. Add a method to the class

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


    def bark(self):
        return f"{self.name} says woof!"


d = Dog("Rex")
```

```
print(d.bark())  # Rex says woof!
```

□3. Use `@property` for a read-only attribute

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

    @property
    def area(self):
        return 3.14 * self._radius ** 2

c = Circle(3)
print(c.area)  # 28.26
# c.area = 50  # □ Error: can't set attribute
```

□4. Inherit from another class

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

class Dog(Animal):
    def speak(self):
        return "Bark"

d = Dog()
print(d.speak())  # Bark
```

□5. Override a method

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

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

c = Cat()
print(c.speak())  # Meow
```

---

6. Use `super()` to call base class method

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

class Dog(Animal):
    def speak(self):
        base = super().speak()
        return base + " + Bark"

d = Dog()
print(d.speak())  # Some sound + Bark
```

---

7. Use `__repr__()` or `__str__()`

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

    def __repr__(self):
        return f"Dog('{self.name}')"
    def __str__(self):
        return f"My dog's name is {self.name}"


d = Dog("Buddy")
print(repr(d))  # Dog('Buddy')
print(str(d))   # My dog's name is Buddy
```

Absolutely! Here's the full **ChatGPT response on polymorphism**, polished and formatted so you can save it, review it, or turn it into a markdown or flashcard:

□□ What is Polymorphism in Python?

□□ Definition:

> **Polymorphism** means "**many forms**" — the same function or method behaves differently depending on the object it’s acting on.

□□ Real-World Analogy:

- You call `draw()` on a `Circle`, `Square`, or `Triangle` and each one handles it differently.
- You don’t care *how* it draws — you just know the object will handle it correctly.

□□ Python Polymorphism Example 1 **Method Overriding**

This is classic *runtime polymorphism* through inheritance:

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

```
class Dog(Animal):
    def speak(self):
        return "Bark"

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

animals = [Dog(), Cat()]

for animal in animals:
    print(animal.speak())  # Bark, Meow
```

□ You call the same method `speak()`), but each class handles it differently.

□□ Python Polymorphism Example 2 **Duck Typing**

Python doesn’t require a shared parent class — it just cares if the object has the right method.

```
class Bird:
    def fly(self):
        print("Flapping wings")

class Plane:
    def fly(self):
        print("Engines on")

def lift_off(flier):
    flier.fly()

lift_off(Bird())   # Flapping wings
lift_off(Plane())  # Engines on
```

□ As long as the object has a `.fly()` method, it can be passed in — **“if it quacks like a duck...”**

TL;DR:

Concept	Meaning
Polymorphism	Same method name, different behavior
Method override	Subclasses implement their own version
Duck typing	Type doesn't matter, behavior does
Benefit	Cleaner, extensible, decoupled code

Would you like me to export this explanation as a `.md`, `.txt`, or add it to a reference doc for your interview prep?

Revision #2
Created 5 April 2025 22:30:26 by victor
Updated 5 April 2025 22:46:03 by victor