Here are 10 advantages of using object-oriented programming (OOP) in Python, along with code examples for each advantage. Afterward, I'll provide 5 disadvantages with explanations and examples.

Advantages:

1. **Code Reusability**:
   - OOP allows you to create reusable classes and objects, reducing the need to write the same code multiple times.

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

   calc = Calculator()
   result = calc.add(3, 5)  # Result: 8
   ```

2. **Modularity**:
   - OOP encourages the organization of code into smaller, manageable modules (classes), making it easier to develop and maintain.

   ```python
   class FileManager:
       def read_file(self, filename):
           # Code to read a file

       def write_file(self, filename, data):
           # Code to write data to a file
   ```

3. **Abstraction**:
   - OOP allows you to abstract complex systems by modeling real-world objects and their interactions.

   ```python
   class Car:
       def start_engine(self):
           # Code to start the car's engine

   my_car = Car()
   my_car.start_engine()
   ```

4. **Encapsulation**:
   - OOP encapsulates data (attributes) and behavior (methods) into a single unit (class), protecting data integrity.

   ```python
   class BankAccount:
       def __init__(self):
           self.balance = 0

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

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

   account = BankAccount()
   account.deposit(1000)
   account.withdraw(500)
   ```

5. **Inheritance**:
   - OOP supports inheritance, allowing you to create new classes based on existing ones, promoting code reuse.

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

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

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

   my_dog = Dog()
   my_cat = Cat()
   ```

6. **Polymorphism**:
   - OOP enables objects of different classes to be treated as objects of a common superclass, simplifying code.

   ```python
   def animal_speak(animal):
       return animal.speak()

   my_dog = Dog()
   my_cat = Cat()

   print(animal_speak(my_dog))  # Output: Woof!
   print(animal_speak(my_cat))  # Output: Meow!
   ```

7. **Easy Maintenance**:
   - OOP's modular and organized structure makes it easier to maintain and extend code over time.

   ```python
   class Product:
       def calculate_price(self):
           # Code to calculate the product's price

   class DiscountedProduct(Product):
       def apply_discount(self):
           # Code to apply a discount to the product's price
   ```

8. **Flexibility**:
   - OOP allows you to create complex systems by combining and extending classes and objects.

   ```python
   class Shape:
       def area(self):
           pass

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

       def area(self):
   ```

```
        return 3.14159265359 * self.radius ** 2
```

9. **Security**:
   - OOP supports data hiding and access control, ensuring that data is only accessible through defined interfaces.

   ```python
   class User:
       def __init__(self, username, password):
           self.username = username
           self.__password = password  # Private attribute

       def authenticate(self, password):
           return self.__password == password
   ```

10. **Scalability**:
    - OOP is well-suited for building scalable applications by providing a structured and organized approach to code development.

    ```python
    class DatabaseConnection:
        def connect(self):
            # Code to establish a database connection

    class UserDatabase(DatabaseConnection):
        def get_user(self, username):
            # Code to retrieve user data from the database
    ```

Disadvantages:

1. **Complexity**:
   - OOP can introduce complexity, especially in small projects, where a simpler approach may be more suitable.

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

2. **Performance Overhead**:

- OOP can have a slight performance overhead due to the creation of objects and method calls.

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

    def distance(self, other_point):
        return ((self.x - other_point.x) ** 2 + (self.y - other_point.y) ** 2) ** 0.5
```

3. **Learning Curve**:
   - OOP concepts may be challenging for beginners to grasp initially, leading to a steeper learning curve.

4. **Memory Consumption**:
   - Creating multiple objects in OOP can lead to increased memory consumption.

```python
class MemoryHog:
    def __init__(self):
        self.data = [0] * 1000000
```

5. **Rigidity**:
   - OOP systems can become rigid if the initial class hierarchy is poorly designed, making it challenging to adapt to changing requirements.

```python
class Car:
    def drive(self):
        pass

# Later, if you want to add ElectricCar and HybridCar, the hierarchy may need restructuring.
```

While OOP has numerous advantages for organizing and structuring code, it's essential to consider the specific needs and complexity of your project to determine whether OOP is the most appropriate approach.