examples demonstrating object-oriented programming (OOP) concepts in Python:

1. **Class and Object Creation**:
   - Creating a simple class and objects from that class.

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

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

   dog1 = Dog("Buddy")
   dog2 = Dog("Rex")

   print(dog1.bark())  # Output: Buddy says Woof!
   ```

2. **Inheritance**:
   - Creating a base class and a subclass that inherits attributes and methods.

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

       def speak(self):
           pass

   class Dog(Animal):
       def speak(self):
           return f"{self.name} says Woof!"

   dog = Dog("Buddy")
   print(dog.speak())  # Output: Buddy says Woof!
   ```

3. **Encapsulation**:
   - Demonstrating private and public attributes.

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

```
        self.name = name
        self.__id = 123456  # Private attribute

student = Student("Alice")
print(student.name)  # Accessing public attribute
# print(student.__id)  # This would raise an error
```

4. **Polymorphism**:
   - Implementing polymorphism with different classes that have a common method.

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

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

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

   dog = Dog()
   cat = Cat()

   print(animal_sound(dog))  # Output: Woof!
   print(animal_sound(cat))  # Output: Meow!
   ```

5. **Abstraction**:
   - Using abstraction to create a base class with placeholder methods.

   ```python
   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 3.14159 * self.radius ** 2

circle = Circle(5)
print(circle.area())  # Output: 78.53975
```

6. **Composition**:
   - Demonstrating composition by creating a complex object from simpler objects.

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

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

   my_car = Car()
   print(my_car.engine.start())  # Output: Engine started.
   ```

7. **Getter and Setter Methods**:
   - Using getter and setter methods for controlled attribute access.

   ```python
   class Temperature:
       def __init__(self):
           self._temperature = 0  # Protected attribute

       def get_temperature(self):
           return self._temperature

       def set_temperature(self, value):
           if value < -273.15:
               print("Temperature cannot be below absolute zero.")
           else:
               self._temperature = value

   temp = Temperature()
   temp.set_temperature(25)
   print(temp.get_temperature())  # Output: 25
```

```

8. **Multiple Inheritance**:
   - Creating a class that inherits from multiple parent classes.

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

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

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

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

9. **Class Variables**:
   - Using class variables to share data among all instances of a class.

   ```python
   class Employee:
       raise_amount = 1.04  # Class variable

       def __init__(self, name, salary):
           self.name = name
           self.salary = salary

       def apply_raise(self):
           self.salary *= self.raise_amount

   emp1 = Employee("Alice", 50000)
   emp2 = Employee("Bob", 60000)

   emp1.apply_raise()
   print(emp1.salary)  # Output: 52000.0
   print(emp2.salary)  # Output: 60000.0
```
```

```

10. **Duck Typing**:
    - Demonstrating duck typing where the type of an object is determined by its behavior rather than its class.

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

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

    def animal_sound(animal):
        return animal.speak()

    cat = Cat()
    dog = Dog()

    print(animal_sound(cat))  # Output: Meow!
    print(animal_sound(dog))  # Output: Woof!
    ```

These code examples showcase various OOP concepts in Python, including class creation, inheritance, encapsulation, polymorphism, abstraction, composition, and more.
    ```