```
                self.__balance -= amount
                return True
            return False

        def get_balance(self):
            return self.__balance

    account = BankAccount("Alice", 1000)
    account.deposit(500)
    print(account.get_balance())  # Output: 1500
    account.withdraw(200)
    print(account.get_balance())  # Output: 1300
```

Exercise 4: Polymorphism

python

```
    class Animal:
        def make_sound(self):
            raise NotImplementedError("Subclass must implement abstract method")

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

    class Dog(Animal):
        def make_sound(self):
            return "Woof"

    def animal_sound(animal):
        print(animal.make_sound())

    cat = Cat()
    dog = Dog()
    animal_sound(cat)  # Output: Meow
    animal_sound(dog)  # Output: Woof
```

This guide provides a thorough introduction to Python's OOP concepts, complete with detailed explanations, numerous examples, and practice assignments with answers. It covers the fundamentals as well as more advanced topics to ensure a comprehensive understanding of Python's object-oriented programming capabilities.

include syntax and comments explaining every line of code. explain each concept in a more clear and exhaustive manner and provide suitable file names for each example or assignment given.

Comprehensive Guide to Python Object-Oriented Programming (OOP)

Table of Contents

---

1. Introduction to Object-Oriented Programming

What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design software. Objects are instances of classes, which can contain both data (attributes) and methods (functions).

Key Concepts of OOP

- **Class**: A blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.
- **Object**: An instance of a class. It is created from a class and can have its own values for the class's attributes.
- **Encapsulation**: The practice of keeping an object's data and methods private or protected, and only exposing what is necessary. This is done to prevent unauthorized access and modification.
- **Abstraction**: Simplifying complex systems by modeling classes appropriate to the problem, and working at the most relevant level of inheritance for a particular aspect of the problem.
- **Inheritance**: A mechanism for creating a new class that is based on an existing class. The new class inherits the attributes and methods of the existing class.

- **Polymorphism**: The ability to use a common interface for multiple forms (data types). It allows methods to do different things based on the object it is acting upon.

---

2. Creating Classes and Objects

Defining a Class

File Name: `example_class.py`

```python
# Define a class named Car
class Car:
    # Class attribute
    wheels = 4

    # Initializer / Instance attributes
    def __init__(self, brand, model):
        self.brand = brand  # Instance attribute
        self.model = model  # Instance attribute

    # Instance method to display information about the car
    def display_info(self):
        print(f"Car brand: {self.brand}, Model: {self.model}")

# End of class definition
```

Explanation:

- `class Car:`: Defines a class named `Car`.
- `wheels = 4`: A class attribute that is shared by all instances of the class.
- `def __init__(self, brand, model):`: The constructor method initializes the instance attributes.
- `self.brand = brand`: Assigns the `brand` parameter to the instance attribute `brand`.
- `self.model = model`: Assigns the `model` parameter to the instance attribute `model`.
- `def display_info(self):`: Defines an instance method that prints the car's brand and model.

Creating Objects

File Name: `example_create_objects.py`

```python
# Import the Car class from the example_class file
from example_class import Car

# Create an object of the Car class
my_car = Car("Toyota", "Corolla")

# Accessing object attributes
print(my_car.brand)  # Output: Toyota
print(my_car.model)  # Output: Corolla

# Calling an instance method
my_car.display_info()  # Output: Car brand: Toyota, Model: Corolla
```

Explanation:

- `my_car = Car("Toyota", "Corolla")`: Creates an instance of the `Car` class with the brand "Toyota" and model "Corolla".
- `print(my_car.brand)`: Prints the `brand` attribute of the `my_car` object.
- `print(my_car.model)`: Prints the `model` attribute of the `my_car` object.
- `my_car.display_info()`: Calls the `display_info` method to print the car's information.

---

3. Attributes and Methods

Instance Attributes

Instance attributes are variables that are specific to each object created from a class.
File Name: `example_instance_attributes.py`

```python
# Define a class named Dog
class Dog:
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

# Create an object of the Dog class
my_dog = Dog("Buddy", 5)

# Accessing object attributes
print(my_dog.name)  # Output: Buddy
print(my_dog.age)   # Output: 5
```

Explanation:

- `self.name = name`: Assigns the `name` parameter to the instance attribute `name`.
- `self.age = age`: Assigns the `age` parameter to the instance attribute `age`.
- `my_dog = Dog("Buddy", 5)`: Creates an instance of the `Dog` class with the name "Buddy" and age 5.
- `print(my_dog.name)`: Prints the `name` attribute of the `my_dog` object.
- `print(my_dog.age)`: Prints the `age` attribute of the `my_dog` object.

Instance Methods

Instance methods operate on the instance attributes of the object.
File Name: `example_instance_methods.py`

```python
# Define a class named Dog
class Dog:
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    # Instance method to make the dog bark
    def bark(self):
        print(f"{self.name} is barking!")

# Create an object of the Dog class
my_dog = Dog("Buddy", 5)
```

```
# Calling an instance method
my_dog.bark()  # Output: Buddy is barking!
```

Explanation:

- `def bark(self):`: Defines an instance method named `bark`.
- `print(f"{self.name} is barking!")`: Prints a message that includes the dog's name.
- `my_dog.bark()`: Calls the `bark` method on the `my_dog` object.

Class Attributes and Methods

Class attributes and methods are shared among all instances of the class.
File Name: `example_class_attributes_methods.py`

```
python
```

```
# Define a class named Dog
class Dog:
    species = "Canis familiaris"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    @classmethod
    def species_info(cls):
        return f"All dogs are of species: {cls.species}"

# Calling a class method
print(Dog.species_info())  # Output: All dogs are of species: Canis familiaris
```

Explanation:

- `species = "Canis familiaris"`: Defines a class attribute named `species`.
- `@classmethod`: A decorator that defines a class method.
- `def species_info(cls):`: Defines a class method named `species_info`.
- `return f"All dogs are of species: {cls.species}"`: Returns a string that includes the class attribute `species`.
- `print(Dog.species_info())`: Calls the class method `species_info` and prints the result.

---

4. Encapsulation and Abstraction

Principles of Encapsulation

Encapsulation restricts direct access to some of the object's components, which can prevent the accidental modification of data.
File Name: `example_encapsulation.py`

```
python
```

```
# Define a class named BankAccount
class BankAccount:
    def __init__(self, owner, balance=0):
```

```
            self.owner = owner
            self.__balance = balance  # Private attribute

        # Method to deposit money
        def deposit(self, amount):
            if amount > 0:
                self.__balance += amount
                return True
            return False

        # Method to get the balance
        def get_balance(self):
            return self.__balance

    # Create an object of the BankAccount class
    account = BankAccount("Alice", 1000)

    # Deposit money
    account.deposit(500)

    # Get the balance
    print(account.get_balance())  # Output: 1500
```

Explanation:

- `self.__balance = balance`: Defines a private instance attribute named `__balance`.
- `def deposit(self, amount):`: Defines a method named `deposit`.
- `if amount > 0: self.__balance += amount`: Adds the amount to `__balance` if it is greater than 0.
- `def get_balance(self):`: Defines a method named `get_balance`.
- `return self.__balance`: Returns the value of `__balance`.
- `account = BankAccount("Alice", 1000)`: Creates an instance of the `BankAccount` class with an initial balance of 1000.
- `account.deposit(500)`: Calls the `deposit` method to add 500 to the balance.
- `print(account.get_balance())`: Calls the `get_balance` method and prints the balance.

Implementing Abstraction

Abstraction means exposing only the relevant parts of an object while hiding the unnecessary details.
File Name: `example_abstraction.py`

```
python
```

```
    # Define a class named Computer
    class Computer:
        def __init__(self, cpu, ram):
            self.__cpu = cpu  # Private attribute
            self.__ram = ram  # Private attribute

        # Method to perform calculations
        def perform_calculation(self):
            return "Calculating..."

    # Create an object of the Computer class
    my_computer = Computer("Intel i5", "16GB")

    # Call the perform_calculation method
    print(my_computer.perform_calculation())  # Output: Calculating...
```

Explanation:

- `self.__cpu = cpu`: Defines a private instance attribute named `__cpu`.

- `self.__ram = ram`: Defines a private instance attribute named `__ram`.
- `def perform_calculation(self):`: Defines a method named `perform_calculation`.
- `return "Calculating..."`: Returns a string indicating that a calculation is being performed.
- `my_computer = Computer("Intel i5", "16GB")`: Creates an instance of the `Computer` class with specific CPU and RAM.
- `print(my_computer.perform_calculation())`: Calls the `perform_calculation` method and prints the result.

---

5. Constructors and Destructors

The `__init__` Method

The `__init__` method is a special method that is called when an object is instantiated. It initializes the object's attributes.
File Name: `example_init_method.py`

```
python
```

```python
# Define a class named Person
class Person:
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    # Method to display person's info
    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Create an object of the Person class
person1 = Person("John", 30)

# Call the display_info method
person1.display_info()  # Output: Name: John, Age: 30
```

Explanation:

- `def __init__(self, name, age):`: Defines the constructor method.
- `self.name = name`: Assigns the `name` parameter to the instance attribute `name`.
- `self.age = age`: Assigns the `age` parameter to the instance attribute `age`.
- `person1 = Person("John", 30)`: Creates an instance of the `Person` class with the name "John" and age 30.
- `person1.display_info()`: Calls the `display_info` method to print the person's information.

The `__del__` Method

The `__del__` method is a special method that is called when an object is about to be destroyed. It is useful for cleanup activities.
File Name: `example_del_method.py`

```
python
```

```python
# Define a class named Person
class Person:
    def __init__(self, name, age):
```

```
            self.name = name  # Instance attribute
            self.age = age    # Instance attribute

        # Method to display person's info
        def display_info(self):
            print(f"Name: {self.name}, Age: {self.age}")

        # Destructor method
        def __del__(self):
            print(f"Person object with name {self.name} is being deleted")

    # Create an object of the Person class
    person1 = Person("John", 30)

    # Call the display_info method
    person1.display_info()  # Output: Name: John, Age: 30

    # Delete the person1 object
    del person1  # Output: Person object with name John is being deleted
```

Explanation:

- `def __del__(self):`: Defines the destructor method.
- `print(f"Person object with name {self.name} is being deleted")`: Prints a message when the object is about to be destroyed.
- `del person1`: Deletes the `person1` object, triggering the `__del__` method.

---

6. Inheritance and Polymorphism

Single Inheritance

Single inheritance allows a class to inherit attributes and methods from a single parent class.
File Name: `example_single_inheritance.py`

```
python
```

```
    # Define a base class named Animal
    class Animal:
        def __init__(self, name):
            self.name = name  # Instance attribute

        # Method to make the animal speak
        def speak(self):
            raise NotImplementedError("Subclass must implement abstract method")

    # Define a subclass named Dog that inherits from Animal
    class Dog(Animal):
        def speak(self):
            return f"{self.name} says Woof!"

    # Create an object of the Dog class
    dog = Dog("Buddy")

    # Call the speak method
    print(dog.speak())  # Output: Buddy says Woof!
```

Explanation:

- `class Animal:`: Defines a base class named `Animal`.
- `def speak(self):`: Defines a method that raises an exception if not overridden.

- `class Dog(Animal):`: Defines a subclass named `Dog` that inherits from `Animal`.
- `def speak(self):`: Overrides the `speak` method to provide specific functionality.
- `dog = Dog("Buddy")`: Creates an instance of the `Dog` class with the name "Buddy".
- `print(dog.speak())`: Calls the `speak` method and prints the result.

Multiple Inheritance

Multiple inheritance allows a class to inherit attributes and methods from more than one parent class.
File Name: `example_multiple_inheritance.py`

```python
# Define a base class named Animal
class Animal:
    def __init__(self, name):
        self.name = name  # Instance attribute

# Define another base class named Pet
class Pet:
    def __init__(self, owner):
        self.owner = owner  # Instance attribute

# Define a subclass named Dog that inherits from both Animal and Pet
class Dog(Animal, Pet):
    def __init__(self, name, owner):
        Animal.__init__(self, name)
        Pet.__init__(self, owner)

    # Method to make the dog speak
    def speak(self):
        return f"{self.name} says Woof!"

# Create an object of the Dog class
dog = Dog("Buddy", "Alice")

# Accessing attributes and calling method
print(f"Dog: {dog.name}, Owner: {dog.owner}")  # Output: Dog: Buddy, Owner: Alice
print(dog.speak())  # Output: Buddy says Woof!
```

Explanation:

- `class Pet:`: Defines a base class named `Pet`.
- `class Dog(Animal, Pet):`: Defines a subclass named `Dog` that inherits from both `Animal` and `Pet`.
- `Animal.__init__(self, name)`: Initializes the `Animal` part of the `Dog` class.
- `Pet.__init__(self, owner)`: Initializes the `Pet` part of the `Dog` class.
- `dog = Dog("Buddy", "Alice")`: Creates an instance of the `Dog` class with the name "Buddy" and owner "Alice".
- `print(f"Dog: {dog.name}, Owner: {dog.owner}")`: Prints the dog's name and owner.
- `print(dog.speak())`: Calls the `speak` method and prints the result.

Method Overriding

Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
File Name: `example_method_overriding.py`

python

```python
# Define a base class named Animal
class Animal:
    def __init__(self, name):
        self.name = name  # Instance attribute

    # Method to make the animal speak
    def speak(self):
        return f"{self.name} makes a sound"

# Define a subclass named Dog that overrides the speak method
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

# Create objects of Animal and Dog classes
animal = Animal("Generic Animal")
dog = Dog("Buddy")

# Call the speak method
print(animal.speak())  # Output: Generic Animal makes a sound
print(dog.speak())     # Output: Buddy says Woof!
```

Explanation:

- `` `def speak(self):` ``: Defines a method in the base class `` `Animal` ``.
- `` `class Dog(Animal):` ``: Defines a subclass named `` `Dog` `` that inherits from `` `Animal` ``.
- `` `def speak(self):` ``: Overrides the `` `speak` `` method to provide specific functionality.
- `` `animal = Animal("Generic Animal")` ``: Creates an instance of the `` `Animal` `` class with the name "Generic Animal".
- `` `dog = Dog("Buddy")` ``: Creates an instance of the `` `Dog` `` class with the name "Buddy".
- `` `print(animal.speak())` ``: Calls the `` `speak` `` method on the `` `animal` `` object and prints the result.
- `` `print(dog.speak())` ``: Calls the `` `speak` `` method on the `` `dog` `` object and prints the result.

Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon.
File Name: `` `example_polymorphism.py` ``

python

```python
# Define a base class named Animal
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

# Define a subclass named Cat
class Cat(Animal):
    def speak(self):
        return "Meow"

# Define a subclass named Dog
class Dog(Animal):
    def speak(self):
        return "Woof"

# Define a function that takes an Animal object and calls speak
def animal_sound(animal):
    print(animal.speak())

# Create objects of Cat and Dog classes
```

```
cat = Cat()
dog = Dog()

# Call the animal_sound function with different objects
animal_sound(cat)  # Output: Meow
animal_sound(dog)  # Output: Woof
```

Explanation:

- `class Cat(Animal):`: Defines a subclass named `Cat` that inherits from `Animal`.
- `def speak(self):`: Overrides the `speak` method to provide specific functionality for cats.
- `def animal_sound(animal):`: Defines a function that takes an `Animal` object and calls the `speak` method.
- `animal_sound(cat)`: Calls the `animal_sound` function with a `Cat` object and prints the result.
- `animal_sound(dog)`: Calls the `animal_sound` function with a `Dog` object and prints the result.

---

7. Advanced OOP Concepts

Class Methods vs. Static Methods

Class methods are methods that are bound to the class and not the instance of the class. Static methods do not have access to `self` or `cls`.
File Name: `example_class_static_methods.py`

```
python


# Define a class named MathOperations
class MathOperations:
    @classmethod
    def add(cls, a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b

# Calling class method
print(MathOperations.add(5, 3))  # Output: 8

# Calling static method
print(MathOperations.multiply(5, 3))  # Output: 15
```

Explanation:

- `@classmethod`: A decorator that defines a class method.
- `@staticmethod`: A decorator that defines a static method.
- `def add(cls, a, b):`: Defines a class method that adds two numbers.
- `def multiply(a, b):`: Defines a static method that multiplies two numbers.
- `print(MathOperations.add(5, 3))`: Calls the `add` class method and prints the result.
- `print(MathOperations.multiply(5, 3))`: Calls the `multiply` static method and prints the result.

The `super()` Function

The `super()` function is used to call a method from the parent class.
File Name: `example_super_function.py`

python

```python
# Define a base class named Animal
class Animal:
    def __init__(self, name):
        self.name = name  # Instance attribute

    def speak(self):
        return f"{self.name} makes a sound"

# Define a subclass named Dog that uses super()
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call the parent class's constructor
        self.breed = breed      # Instance attribute

    def speak(self):
        return f"{self.name}, the {self.breed}, says Woof!"

# Create an object of the Dog class
dog = Dog("Buddy", "Golden Retriever")

# Call the speak method
print(dog.speak())  # Output: Buddy, the Golden Retriever, says Woof!
```

Explanation:

- `super().__init__(name)`: Calls the constructor of the parent class `Animal`.
- `dog = Dog("Buddy", "Golden Retriever")`: Creates an instance of the `Dog` class with the name "Buddy" and breed "Golden Retriever".
- `print(dog.speak())`: Calls the `speak` method and prints the result.

Abstract Base Classes

Abstract Base Classes (ABCs) are classes that cannot be instantiated and are meant to be subclassed.
File Name: `example_abc.py`

python

```python
from abc import ABC, abstractmethod

# Define an abstract base class named Animal
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# Define a subclass named Dog that inherits from Animal
class Dog(Animal):
    def speak(self):
        return "Woof"

# Create an object of the Dog class
dog = Dog()

# Call the speak method
print(dog.speak())  # Output: Woof
```

Explanation:

- `from abc import ABC, abstractmethod`: Imports the necessary components for creating abstract base classes.
- `class Animal(ABC):`: Defines an abstract base class named `Animal`.
- `@abstractmethod`: A decorator that defines an abstract method.
- `def speak(self):`: Defines an abstract method named `speak`.
- `class Dog(Animal):`: Defines a subclass named `Dog` that inherits from `Animal`.
- `def speak(self):`: Provides a specific implementation for the `speak` method.
- `dog = Dog()`: Creates an instance of the `Dog` class.
- `print(dog.speak())`: Calls the `speak` method and prints the result.

Magic Methods and Operator Overloading

Magic methods allow you to define how objects of your class behave with built-in Python operations.
File Name: `example_magic_methods.py`

```python
# Define a class named Vector
class Vector:
    def __init__(self, x, y):
        self.x = x  # Instance attribute
        self.y = y  # Instance attribute

    # Overloading the addition operator
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # Overloading the string representation
    def __str__(self):
        return f"Vector({self.x}, {self.y})"

# Create objects of the Vector class
v1 = Vector(2, 3)
v2 = Vector(4, 5)

# Add the vectors
v3 = v1 + v2

# Print the result
print(v3)  # Output: Vector(6, 8)
```

Explanation:

- `def __add__(self, other):`: Defines the behavior of the `+` operator for `Vector` objects.
- `def __str__(self):`: Defines the string representation of `Vector` objects.
- `v1 = Vector(2, 3)`: Creates an instance of the `Vector` class with coordinates (2, 3).
- `v2 = Vector(4, 5)`: Creates an instance of the `Vector` class with coordinates (4, 5).
- `v3 = v1 + v2`: Adds the two vectors using the overloaded `+` operator.
- `print(v3)`: Prints the result using the overloaded `__str__` method.

---

8. Assignments and Practice

Exercises for Each Section

1. **Creating Classes and Objects**

Assignment File Name: `assignment_create_objects.py`
Create a class `Book` with the following:

- Instance attributes: `title`, `author`, `year`.
- Instance method: `get_book_info` that returns a string with the book's title, author, and year.

Answer:

python

```python
# Define a class named Book
class Book:
    def __init__(self, title, author, year):
        self.title = title  # Instance attribute
        self.author = author  # Instance attribute
        self.year = year  # Instance attribute

    # Instance method to get book info
    def get_book_info(self):
        return f"{self.title} by {self.author}, published in {self.year}"

# Create an object of the Book class
book = Book("1984", "George Orwell", 1949)

# Call the get_book_info method
print(book.get_book_info())  # Output: 1984 by George Orwell, published in 1949
```

2. **Encapsulation and Abstraction**

Assignment File Name: `assignment_encapsulation_abstraction.py`
Create a class `Account` with the following:

- Private instance attributes: `__account_number`, `__balance`.
- Methods: `deposit`, `withdraw`, and `get_balance` to encapsulate account operations.

Answer:

python

```python
# Define a class named Account
class Account:
    def __init__(self, account_number, balance=0):
        self.__account_number = account_number  # Private attribute
        self.__balance = balance  # Private attribute

    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return True
        return False

    # Method to withdraw money
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return True
        return False

    # Method to get the balance
    def get_balance(self):
        return self.__balance
```

```
# Create an object of the Account class
account = Account("12345678", 1000)

# Deposit money
account.deposit(500)

# Withdraw money
account.withdraw(200)

# Get the balance
print(account.get_balance())  # Output: 1300
```

3. **Inheritance and Polymorphism**

Assignment File Name: `assignment_inheritance_polymorphism.py`
Create a base class `Shape` with a method `area` (that raises a `NotImplementedError`) and subclasses `Circle` and `Rectangle` that implement the `area` method.
Answer:

python

```
# Define a base class named Shape
class Shape:
    def area(self):
        raise NotImplementedError("Subclass must implement abstract method")

# Define a subclass named Circle that inherits from Shape
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius  # Instance attribute

    # Method to calculate the area of the circle
    def area(self):
        return 3.14 * self.radius ** 2

# Define a subclass named Rectangle that inherits from Shape
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width  # Instance attribute
        self.height = height  # Instance attribute

    # Method to calculate the area of the rectangle
    def area(self):
        return self.width * self.height

# Create objects of Circle and Rectangle classes
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Call the area method
print(f"Circle area: {circle.area()}")  # Output: Circle area: 78.5
print(f"Rectangle area: {rectangle.area()}")  # Output: Rectangle area: 24
```

4. **Magic Methods and Operator Overloading**

Assignment File Name: `assignment_magic_methods.py`
Create a class `Complex` with `__add__` and `__str__` methods to handle addition of complex numbers and their string representation.
Answer:

python

```python
# Define a class named Complex
class Complex:
    def __init__(self, real, imag):
        self.real = real  # Instance attribute
        self.imag = imag  # Instance attribute

    # Overloading the addition operator
    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)

    # Overloading the string representation
    def __str__(self):
        return f"{self.real} + {self.imag}i"

# Create objects of the Complex class
c1 = Complex(2, 3)
c2 = Complex(1, 4)

# Add the complex numbers
c3 = c1 + c2

# Print the result
print(c3)  # Output: 3 + 7i
```

Conclusion

This comprehensive guide covers fundamental and advanced concepts of object-oriented programming in Python, with detailed explanations, numerous examples, and assignments with answers. By working through these sections, you will gain a solid understanding of OOP principles and their application in Python.