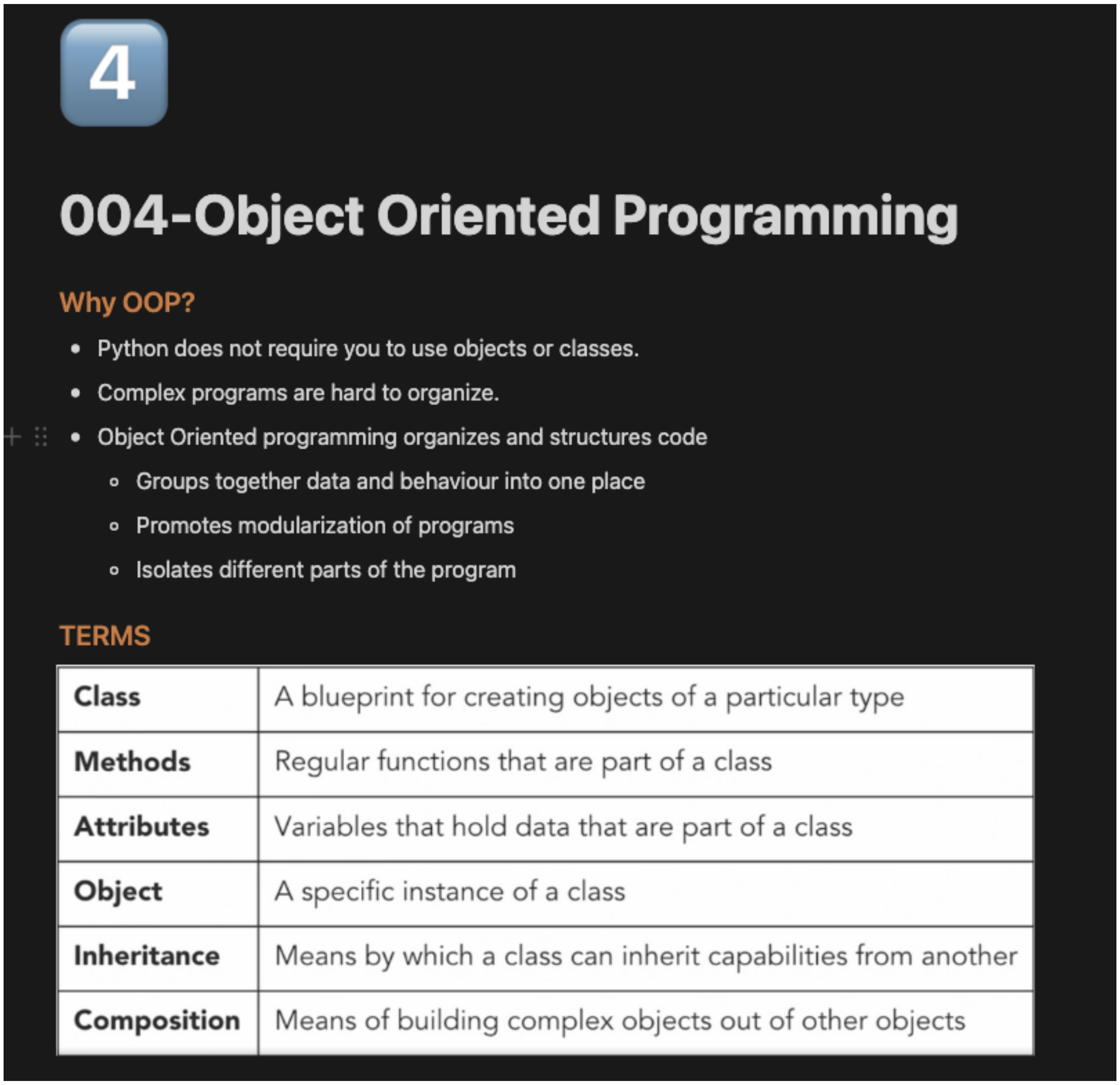


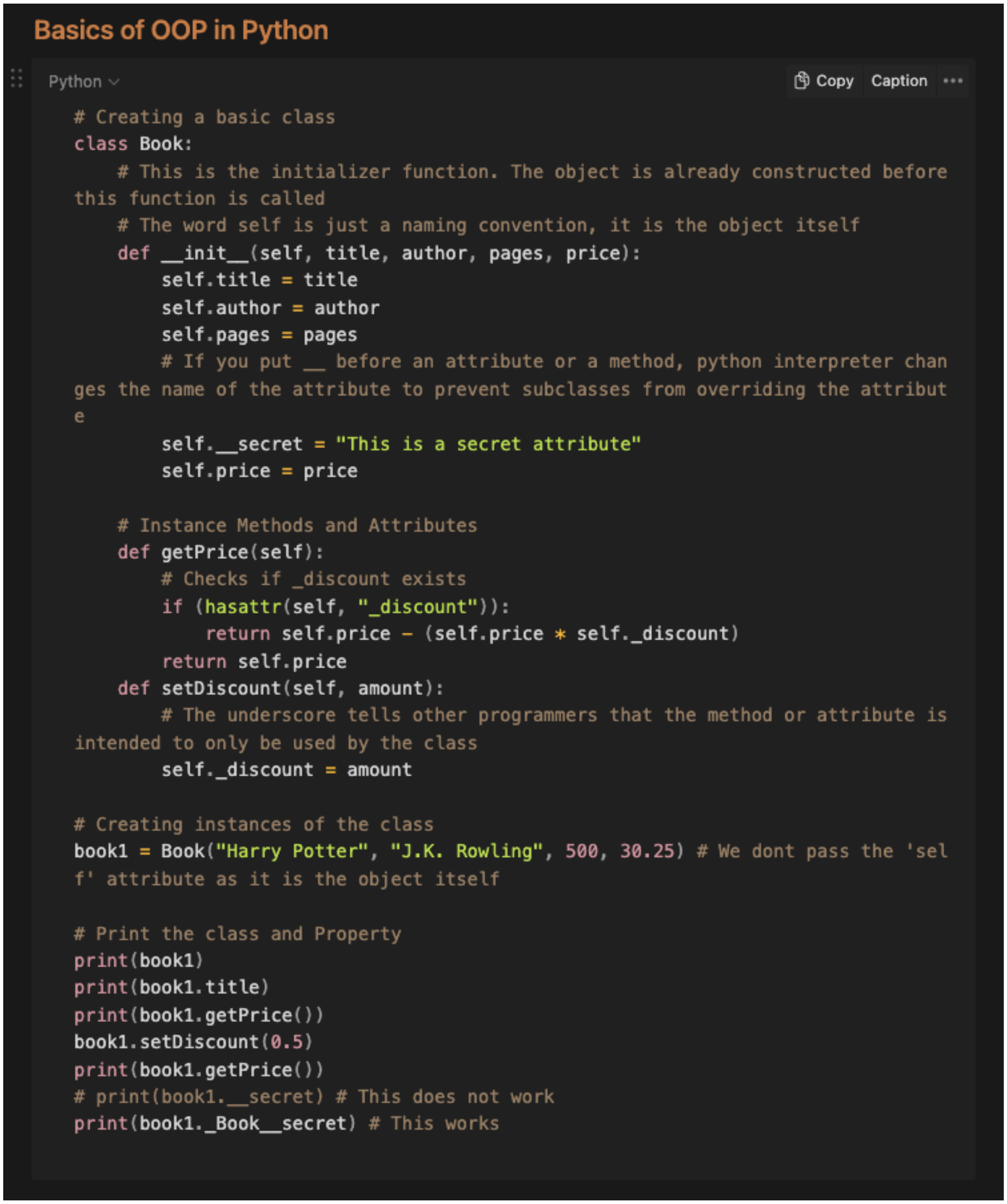
004-Object Oriented Programming

Why OOP?

- Python does not require you to use objects or classes.
- Complex programs are hard to organize.
- Object Oriented programming organizes and structures code
    - Groups together data and behaviour into one place
    - Promotes modularization of programs
    - Isolates different parts of the program

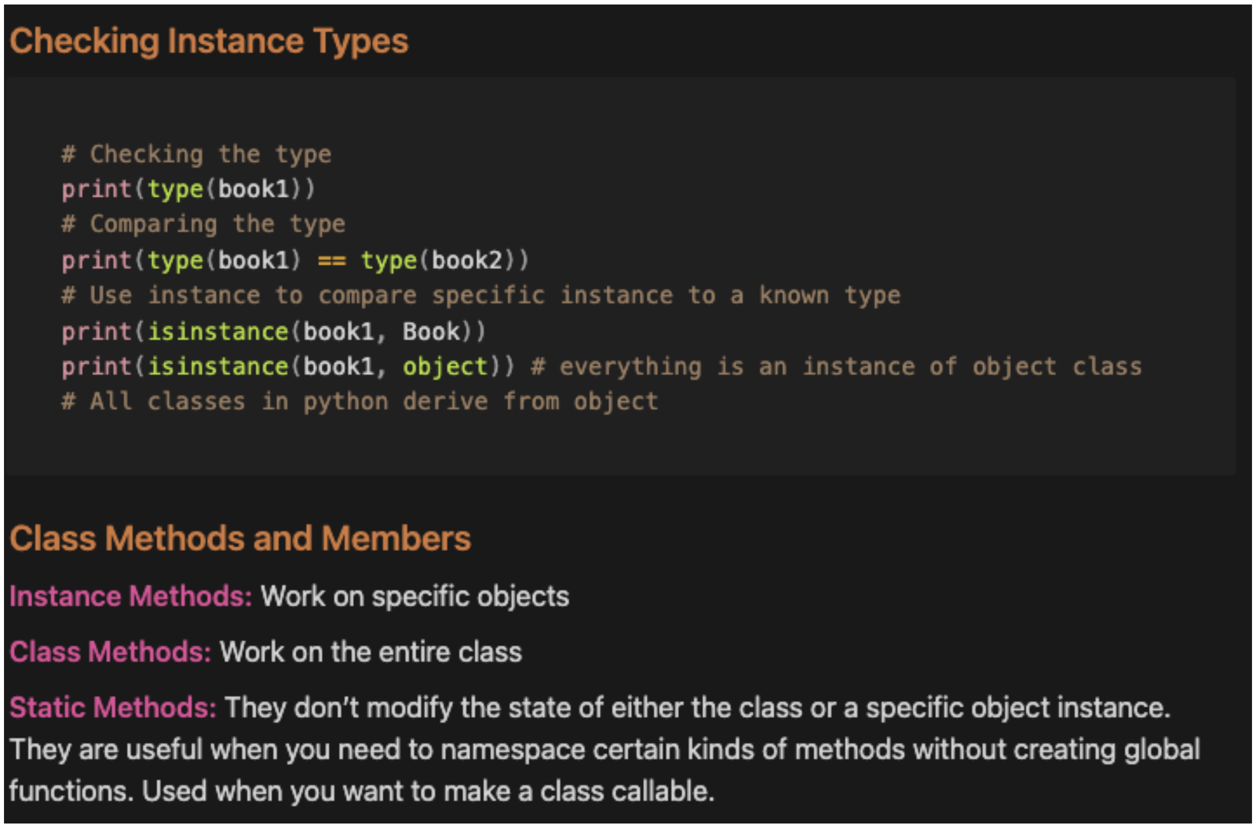
TERMS

Class	A blueprint for creating objects of a particular type
Methods	Regular functions that are part of a class
Attributes	Variables that hold data that are part of a class
Object	A specific instance of a class
Inheritance	Means by which a class can inherit capabilities from another
Composition	Means of building complex objects out of other objects



Basics of OOP in Python

```python
# Creating a basic class
class Book:
    # This is the initializer function. The object is already constructed before
this function is called
    # The word self is just a naming convention, it is the object itself
    def __init__(self, title, author, pages, price):
        self.title = title
        self.author = author
        self.pages = pages
        # If you put __ before an attribute or a method, python interpreter chan
ges the name of the attribute to prevent subclasses from overriding the attribut
e
        self.__secret = "This is a secret attribute"
        self.price = price

    # Instance Methods and Attributes
    def getPrice(self):
        # Checks if _discount exists
        if (hasattr(self, "_discount")):
            return self.price - (self.price * self._discount)
        return self.price
    def setDiscount(self, amount):
        # The underscore tells other programmers that the method or attribute is
intended to only be used by the class
        self._discount = amount

# Creating instances of the class
book1 = Book("Harry Potter", "J.K. Rowling", 500, 30.25) # We dont pass the 'sel
f' attribute as it is the object itself

# Print the class and Property
print(book1)
print(book1.title)
print(book1.getPrice())
book1.setDiscount(0.5)
print(book1.getPrice())
# print(book1.__secret) # This does not work
print(book1._Book__secret) # This works
```



Checking Instance Types

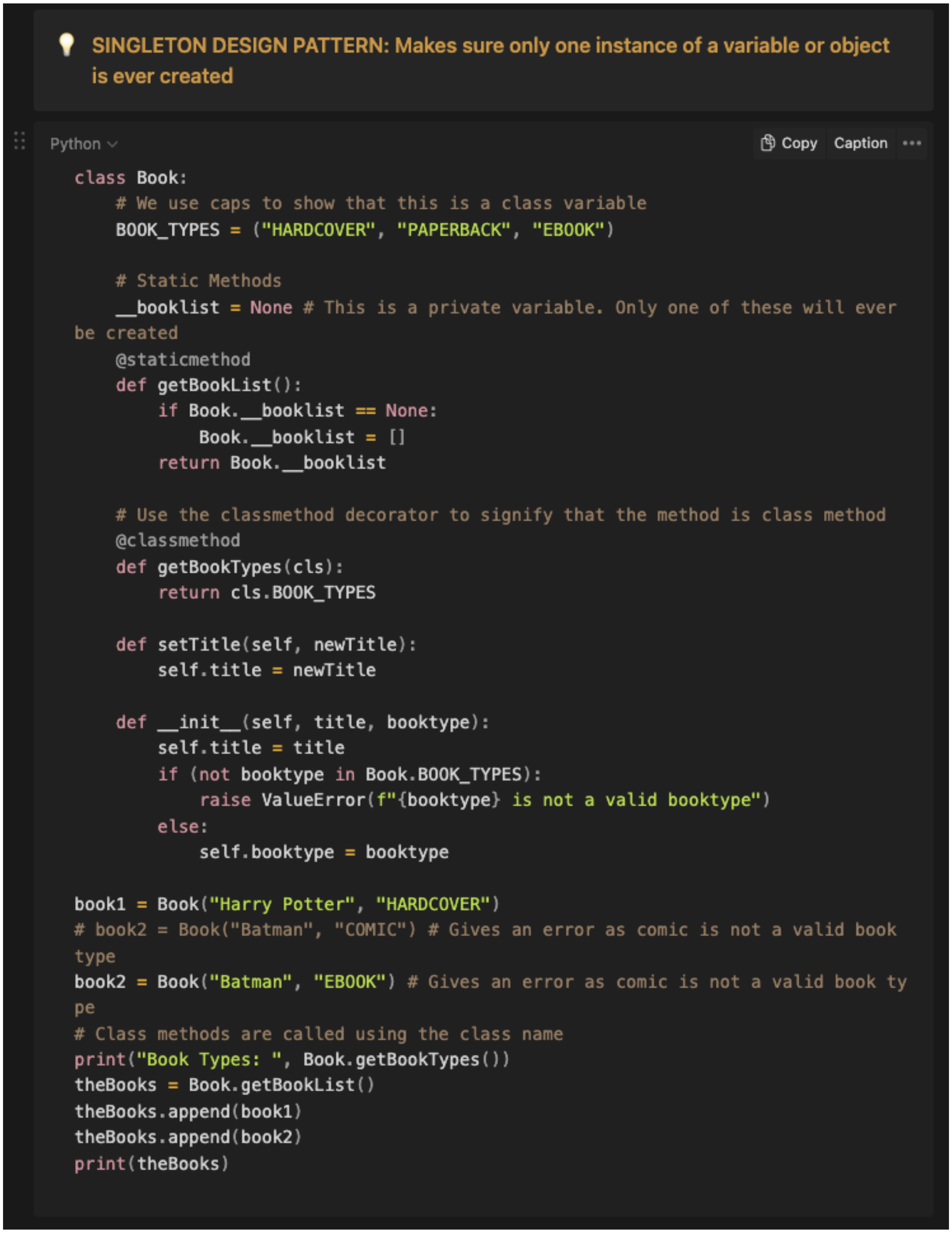
```python
# Checking the type
print(type(book1))
# Comparing the type
print(type(book1) == type(book2))
# Use instance to compare specific instance to a known type
print(isinstance(book1, Book))
print(isinstance(book1, object)) # everything is an instance of object class
# All classes in python derive from object
```

Class Methods and Members

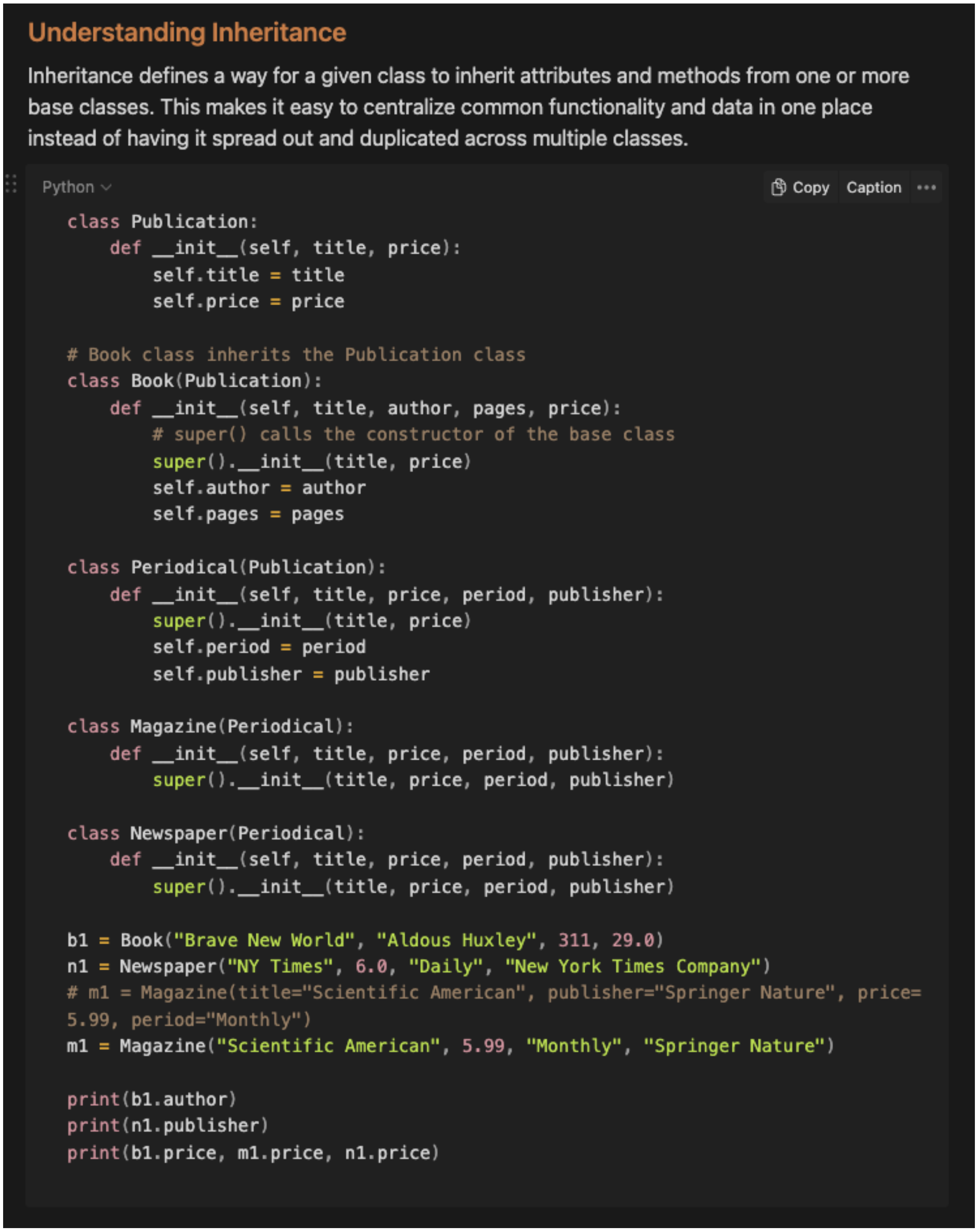
Instance Methods: Work on specific objects

Class Methods: Work on the entire class

Static Methods: They don't modify the state of either the class or a specific object instance. They are useful when you need to namespace certain kinds of methods without creating global functions. Used when you want to make a class callable.


💡 SINGLETON DESIGN PATTERN: Makes sure only one instance of a variable or object is ever created

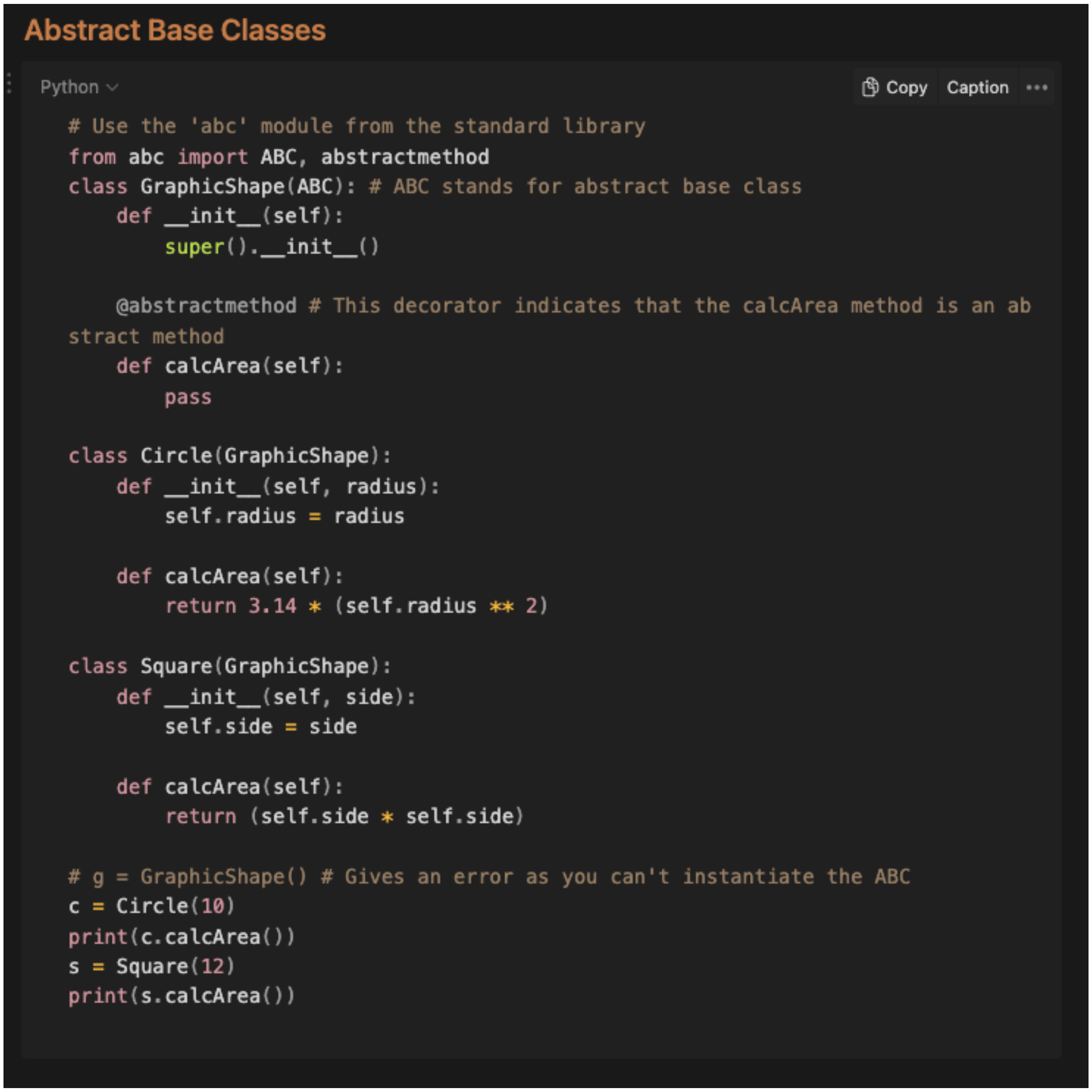
```python
class Book:
    # We use caps to show that this is a class variable
    BOOK_TYPES = ("HARDCOVER", "PAPERBACK", "EBOOK")

    # Static Methods
    __booklist = None # This is a private variable. Only one of these will ever
be created
    @staticmethod
    def getBookList():
        if Book.__booklist == None:
            Book.__booklist = []
        return Book.__booklist

    # Use the classmethod decorator to signify that the method is class method
    @classmethod
    def getBookTypes(cls):
        return cls.BOOK_TYPES

    def setTitle(self, newTitle):
        self.title = newTitle

    def __init__(self, title, booktype):
        self.title = title
        if (not booktype in Book.BOOK_TYPES):
            raise ValueError(f"{booktype} is not a valid booktype")
        else:
            self.booktype = booktype

book1 = Book("Harry Potter", "HARDCOVER")
# book2 = Book("Batman", "COMIC") # Gives an error as comic is not a valid book
type
book2 = Book("Batman", "EBOOK") # Gives an error as comic is not a valid book ty
pe
# Class methods are called using the class name
print("Book Types: ", Book.getBookTypes())
theBooks = Book.getBookList()
theBooks.append(book1)
theBooks.append(book2)
print(theBooks)
```



Understanding Inheritance

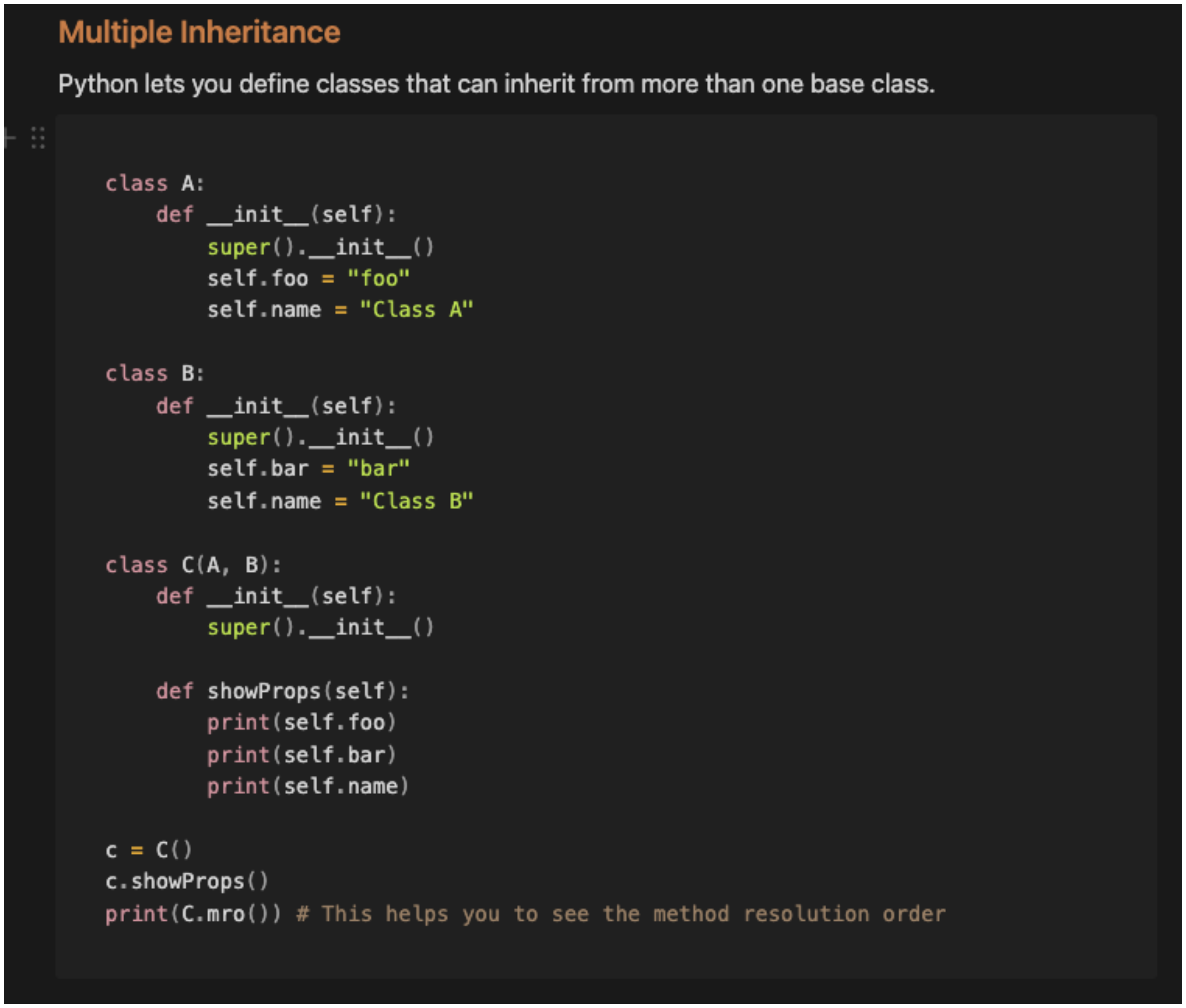
Inheritance defines a way for a given class to inherit attributes and methods from one or more base classes. This makes it easy to centralize common functionality and data in one place instead of having it spread out and duplicated across multiple classes.

```python
class Publication:
    def __init__(self, title, price):
        self.title = title
        self.price = price

# Book class inherits the Publication class
class Book(Publication):
    def __init__(self, title, author, pages, price):
        # super() calls the constructor of the base class
        super().__init__(title, price)
        self.author = author
        self.pages = pages

class Periodical(Publication):
    def __init__(self, title, price, period, publisher):
        super().__init__(title, price)
        self.period = period
        self.publisher = publisher

class Magazine(Periodical):
    def __init__(self, title, price, period, publisher):
        super().__init__(title, price, period, publisher)

class Newspaper(Periodical):
    def __init__(self, title, price, period, publisher):
        super().__init__(title, price, period, publisher)

b1 = Book("Brave New World", "Aldous Huxley", 311, 29.0)
n1 = Newspaper("NY Times", 6.0, "Daily", "New York Times Company")
# m1 = Magazine(title="Scientific American", publisher="Springer Nature", price=
5.99, period="Monthly")
m1 = Magazine("Scientific American", 5.99, "Monthly", "Springer Nature")

print(b1.author)
print(n1.publisher)
print(b1.price, m1.price, n1.price)
```



Abstract Base Classes

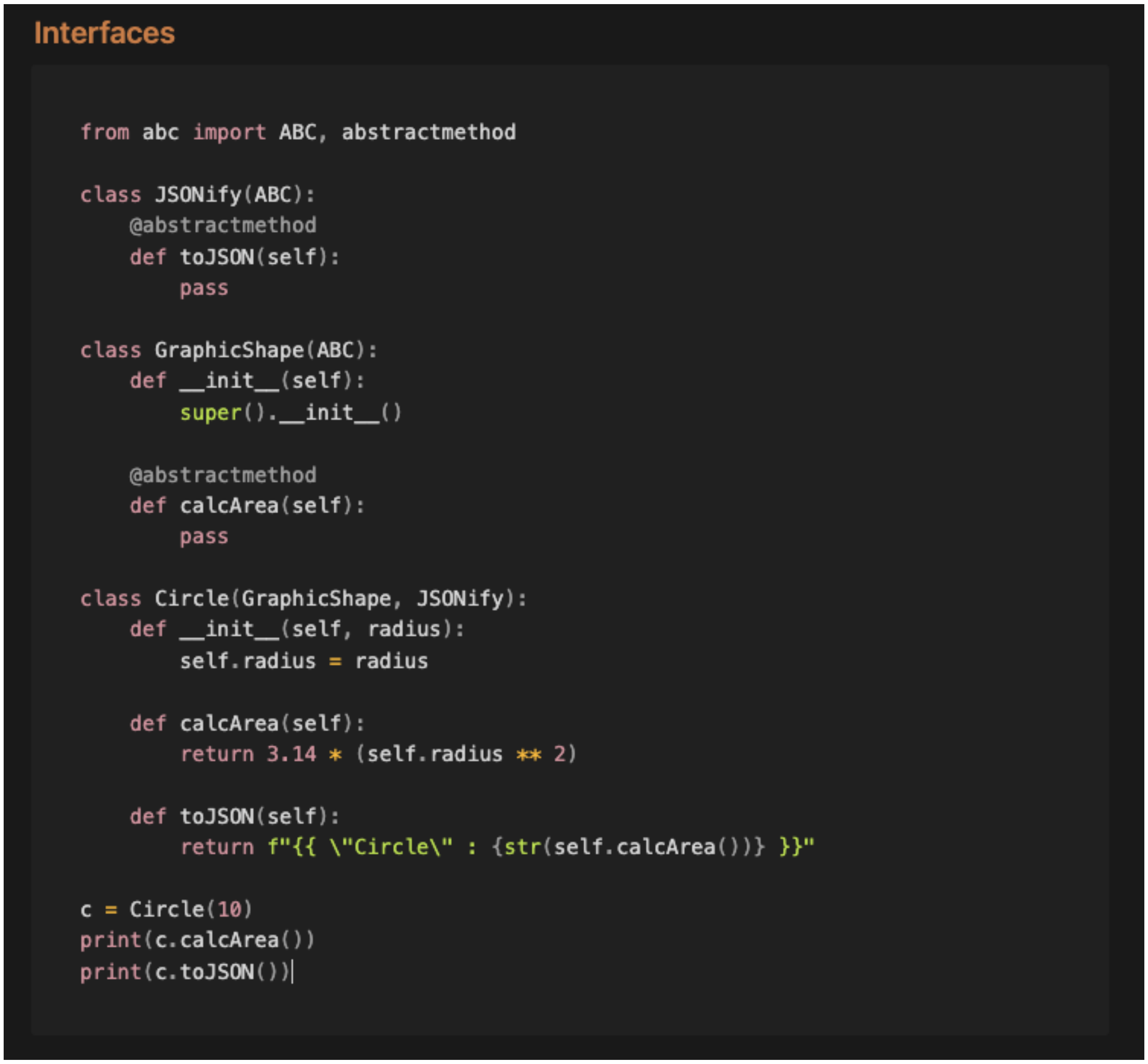
```python
# Use the 'abc' module from the standard library
from abc import ABC, abstractmethod
class GraphicShape(ABC): # ABC stands for abstract base class
    def __init__(self):
        super().__init__()

    @abstractmethod # This decorator indicates that the calcArea method is an ab
stract method
    def calcArea(self):
        pass

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

    def calcArea(self):
        return 3.14 * (self.radius ** 2)

class Square(GraphicShape):
    def __init__(self, side):
        self.side = side

    def calcArea(self):
        return (self.side * self.side)

# g = GraphicShape() # Gives an error as you can't instantiate the ABC
c = Circle(10)
print(c.calcArea())
s = Square(12)
print(s.calcArea())
```



Multiple Inheritance

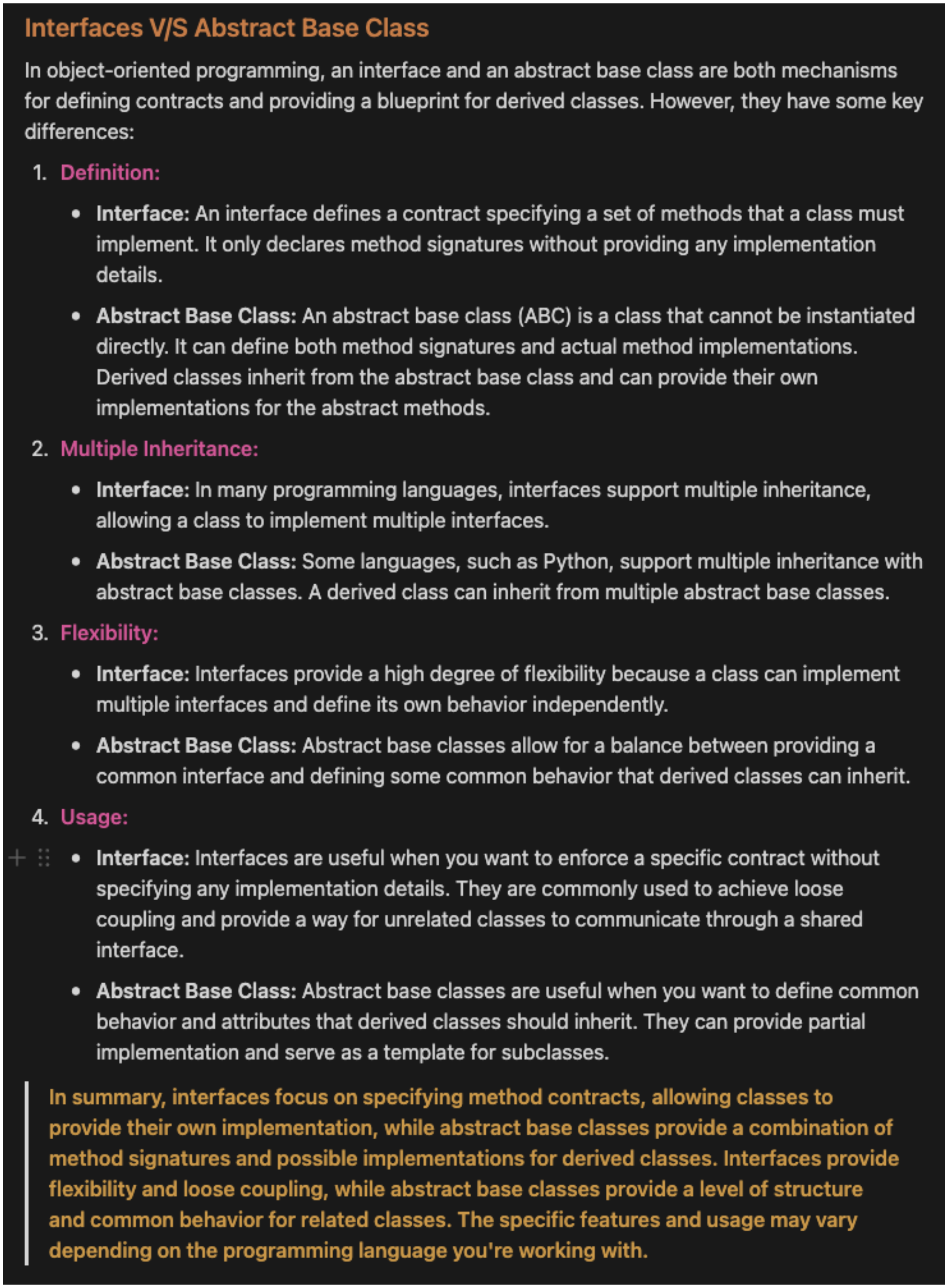
Python lets you define classes that can inherit from more than one base class.

```
class A:
    def __init__(self):
        super().__init__()
        self.foo = "foo"
        self.name = "Class A"

class B:
    def __init__(self):
        super().__init__()
        self.bar = "bar"
        self.name = "Class B"

class C(A, B):
    def __init__(self):
        super().__init__()

    def showProps(self):
        print(self.foo)
        print(self.bar)
        print(self.name)

c = C()
c.showProps()
print(C.mro()) # This helps you to see the method resolution order
```



Interfaces

```
from abc import ABC, abstractmethod

class JSONify(ABC):
    @abstractmethod
    def toJSON(self):
        pass

class GraphicShape(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def calcArea(self):
        pass

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

    def calcArea(self):
        return 3.14 * (self.radius ** 2)

    def toJSON(self):
        return f"{{ \"Circle\" : {str(self.calcArea())} }}"

c = Circle(10)
print(c.calcArea())
print(c.toJSON())
```



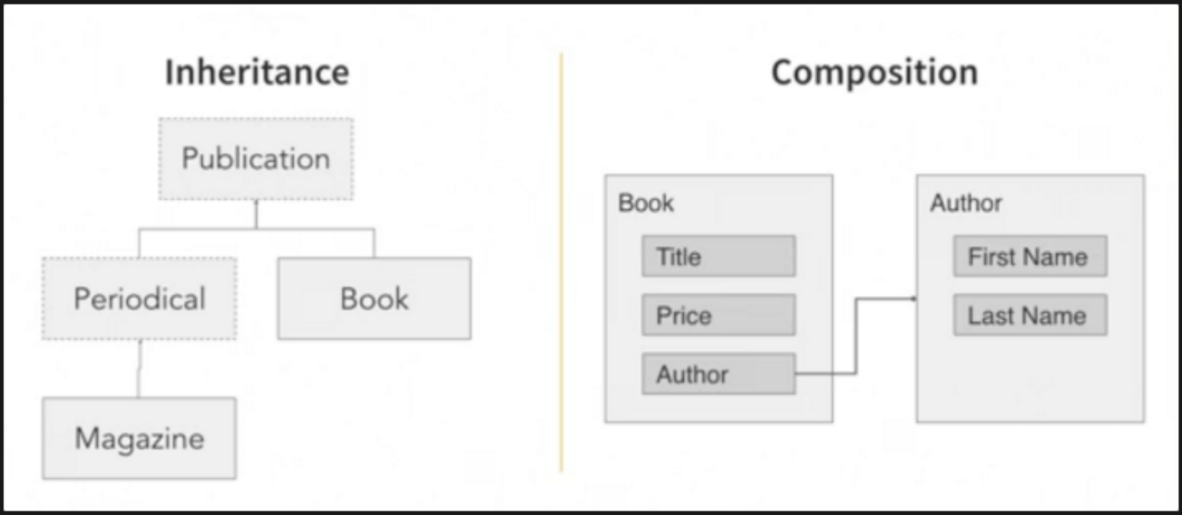
Interfaces V/S Abstract Base Class

In object-oriented programming, an interface and an abstract base class are both mechanisms for defining contracts and providing a blueprint for derived classes. However, they have some key differences:

1. **Definition:**
   - **Interface:** An interface defines a contract specifying a set of methods that a class must implement. It only declares method signatures without providing any implementation details.
   - **Abstract Base Class:** An abstract base class (ABC) is a class that cannot be instantiated directly. It can define both method signatures and actual method implementations. Derived classes inherit from the abstract base class and can provide their own implementations for the abstract methods.
2. **Multiple Inheritance:**
   - **Interface:** In many programming languages, interfaces support multiple inheritance, allowing a class to implement multiple interfaces.
   - **Abstract Base Class:** Some languages, such as Python, support multiple inheritance with abstract base classes. A derived class can inherit from multiple abstract base classes.
3. **Flexibility:**
   - **Interface:** Interfaces provide a high degree of flexibility because a class can implement multiple interfaces and define its own behavior independently.
   - **Abstract Base Class:** Abstract base classes allow for a balance between providing a common interface and defining some common behavior that derived classes can inherit.
4. **Usage:**
   - **Interface:** Interfaces are useful when you want to enforce a specific contract without specifying any implementation details. They are commonly used to achieve loose coupling and provide a way for unrelated classes to communicate through a shared interface.
   - **Abstract Base Class:** Abstract base classes are useful when you want to define common behavior and attributes that derived classes should inherit. They can provide partial implementation and serve as a template for subclasses.

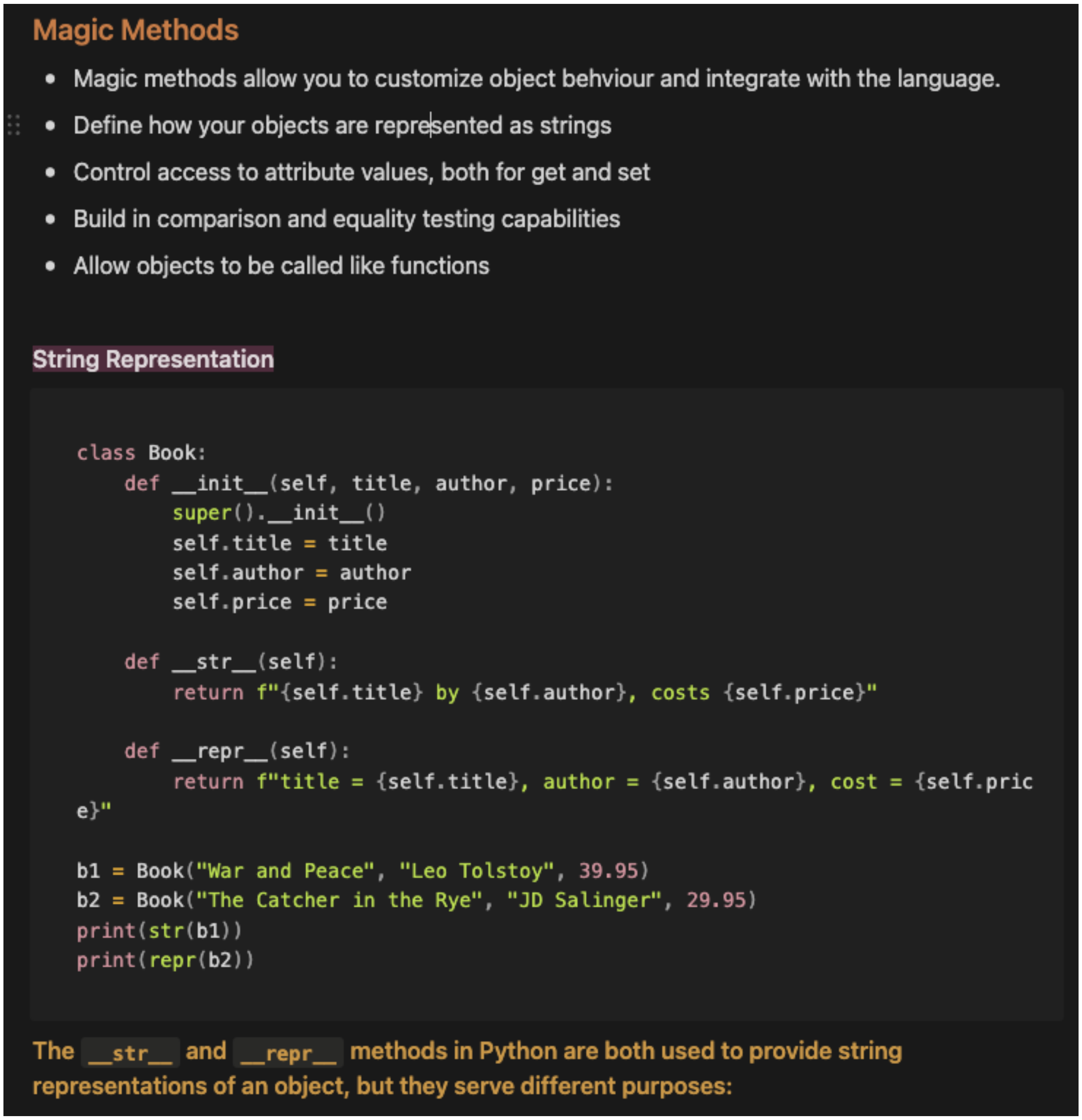
> **In summary, interfaces focus on specifying method contracts, allowing classes to provide their own implementation, while abstract base classes provide a combination of method signatures and possible implementations for derived classes. Interfaces provide flexibility and loose coupling, while abstract base classes provide a level of structure and common behavior for related classes. The specific features and usage may vary depending on the programming language you're working with.**

Composition



Python

```python
class Book:
    def __init__(self, title, price, author = None):
        self.title = title
        self.price = price
        self.author = author
        self.chapters = []

    def addChapter(self, chapter):
        self.chapters.append(chapter)

    def getBookPageCount(self):
        result = 0
        for ch in self.chapters:
            result += ch.pageCount
        return result

class Author:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
    def __str__(self):
        return f"{self.fname} {self.lname}"

class Chapter:
    def __init__(self, name, pageCount):
        self.name = name
        self.pageCount = pageCount

auth = Author("Leo", "Tolstoy")
b1 = Book("War and Peace", 39.0, auth)
b1.addChapter(Chapter("Chapter 1", 125))
b1.addChapter(Chapter("Chapter 2", 97))
```



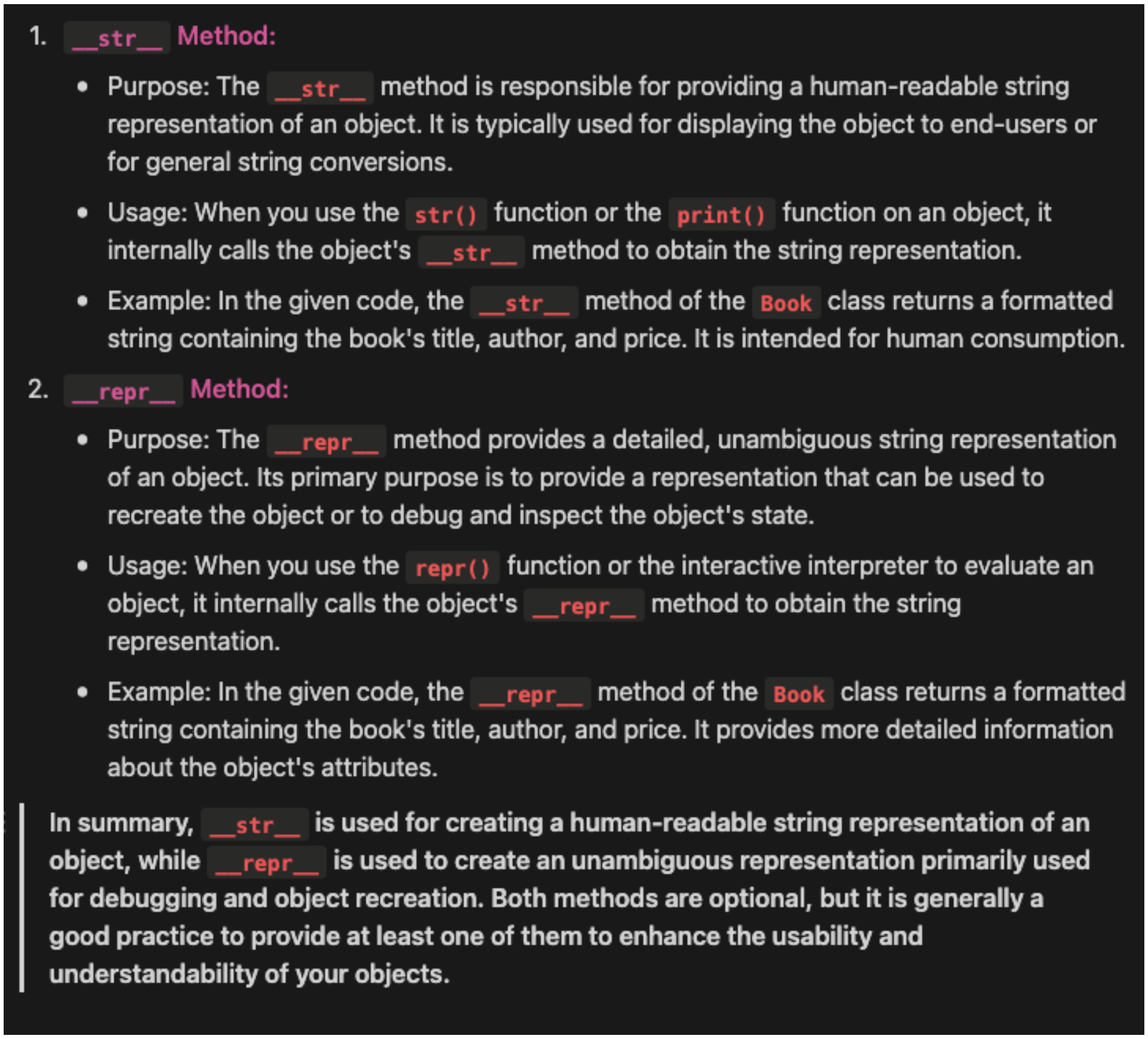
Magic Methods

- Magic methods allow you to customize object behviour and integrate with the language.
- Define how your objects are represented as strings
- Control access to attribute values, both for get and set
- Build in comparison and equality testing capabilities
- Allow objects to be called like functions

String Representation

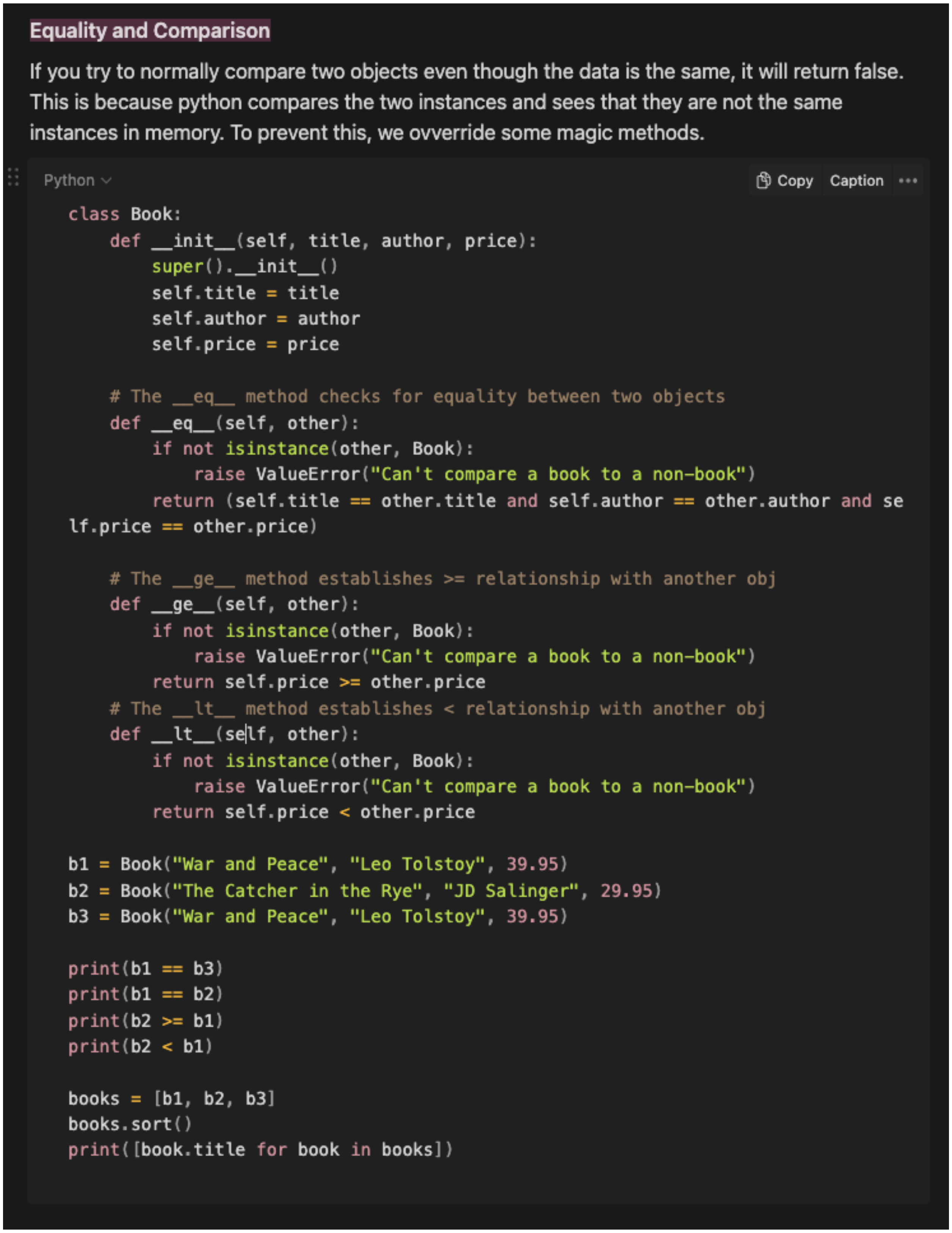
```
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price

    def __str__(self):
        return f"{self.title} by {self.author}, costs {self.price}"

    def __repr__(self):
        return f"title = {self.title}, author = {self.author}, cost = {self.pric
e}"

b1 = Book("War and Peace", "Leo Tolstoy", 39.95)
b2 = Book("The Catcher in the Rye", "JD Salinger", 29.95)
print(str(b1))
print(repr(b2))
```

The `__str__` and `__repr__` methods in Python are both used to provide string representations of an object, but they serve different purposes:



1. `__str__` **Method:**
   - Purpose: The `__str__` method is responsible for providing a human-readable string representation of an object. It is typically used for displaying the object to end-users or for general string conversions.
   - Usage: When you use the `str()` function or the `print()` function on an object, it internally calls the object's `__str__` method to obtain the string representation.
   - Example: In the given code, the `__str__` method of the `Book` class returns a formatted string containing the book's title, author, and price. It is intended for human consumption.
2. `__repr__` **Method:**
   - Purpose: The `__repr__` method provides a detailed, unambiguous string representation of an object. Its primary purpose is to provide a representation that can be used to recreate the object or to debug and inspect the object's state.
   - Usage: When you use the `repr()` function or the interactive interpreter to evaluate an object, it internally calls the object's `__repr__` method to obtain the string representation.
   - Example: In the given code, the `__repr__` method of the `Book` class returns a formatted string containing the book's title, author, and price. It provides more detailed information about the object's attributes.

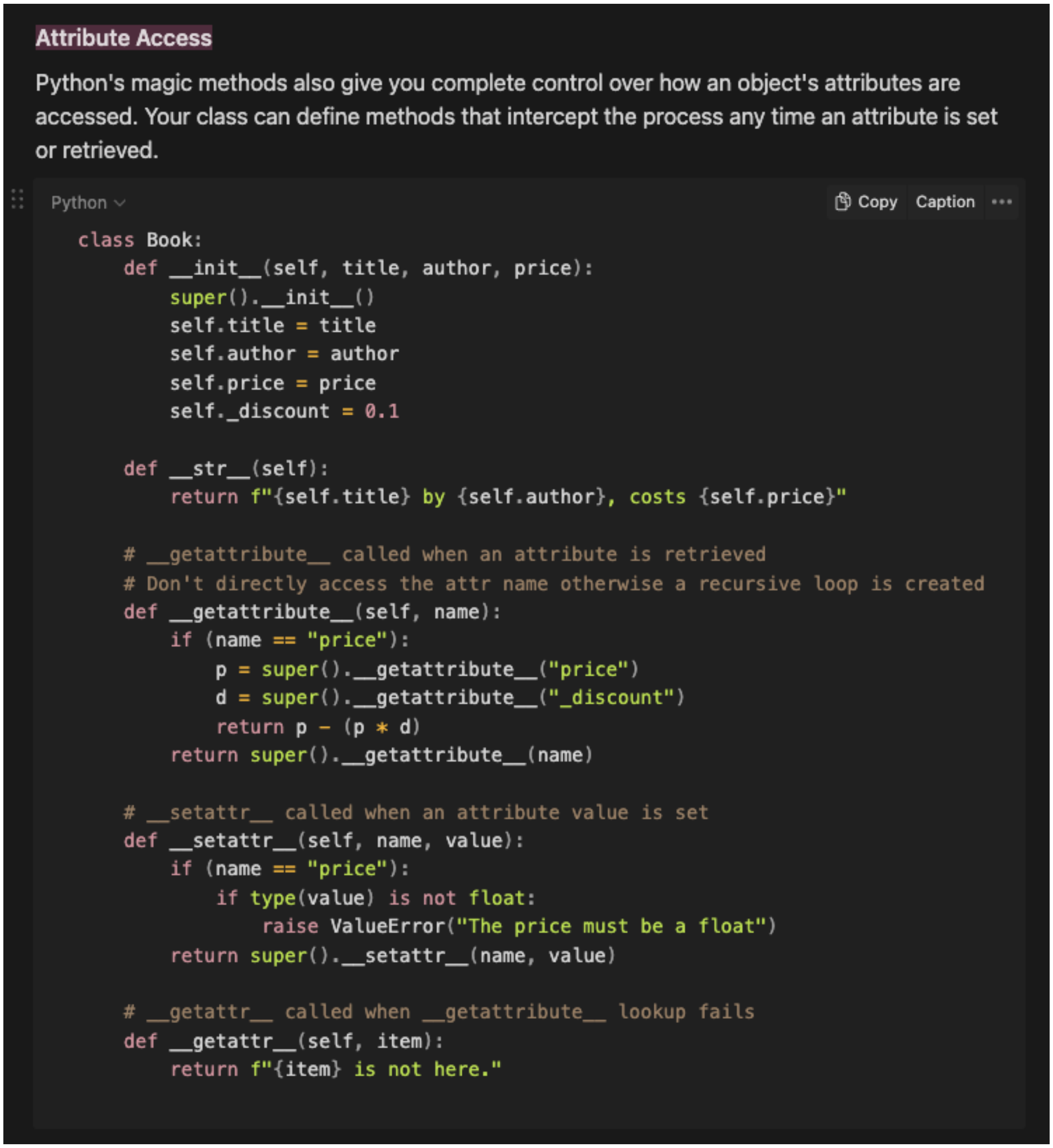
> **In summary, `__str__` is used for creating a human-readable string representation of an object, while `__repr__` is used to create an unambiguous representation primarily used for debugging and object recreation. Both methods are optional, but it is generally a good practice to provide at least one of them to enhance the usability and understandability of your objects.**



Equality and Comparison

If you try to normally compare two objects even though the data is the same, it will return false. This is because python compares the two instances and sees that they are not the same instances in memory. To prevent this, we ovverride some magic methods.

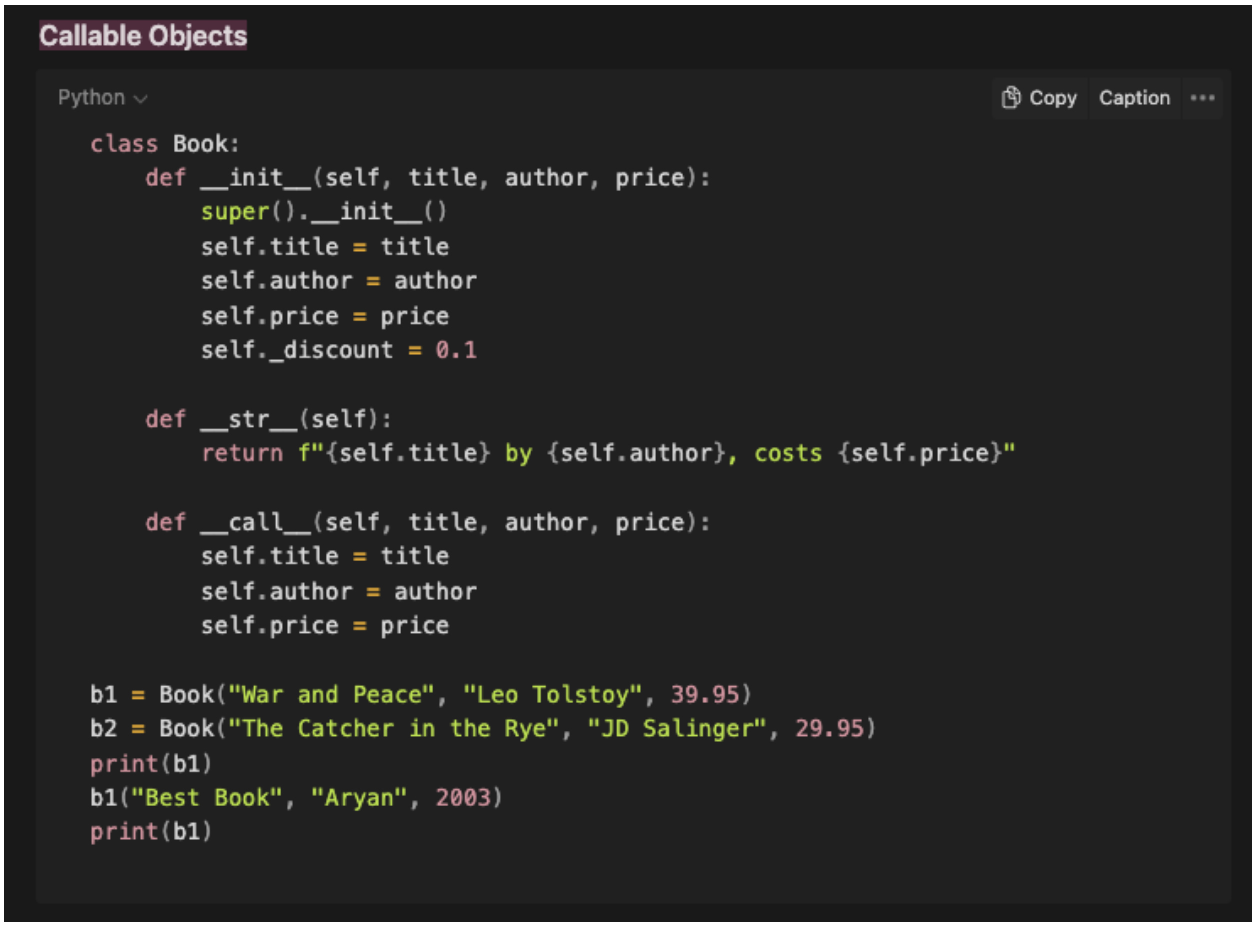
```python
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price

    # The __eq__ method checks for equality between two objects
    def __eq__(self, other):
        if not isinstance(other, Book):
            raise ValueError("Can't compare a book to a non-book")
        return (self.title == other.title and self.author == other.author and se
lf.price == other.price)

    # The __ge__ method establishes >= relationship with another obj
    def __ge__(self, other):
        if not isinstance(other, Book):
            raise ValueError("Can't compare a book to a non-book")
        return self.price >= other.price
    # The __lt__ method establishes < relationship with another obj
    def __lt__(self, other):
        if not isinstance(other, Book):
            raise ValueError("Can't compare a book to a non-book")
        return self.price < other.price

b1 = Book("War and Peace", "Leo Tolstoy", 39.95)
b2 = Book("The Catcher in the Rye", "JD Salinger", 29.95)
b3 = Book("War and Peace", "Leo Tolstoy", 39.95)

print(b1 == b3)
print(b1 == b2)
print(b2 >= b1)
print(b2 < b1)

books = [b1, b2, b3]
books.sort()
print([book.title for book in books])
```



Attribute Access

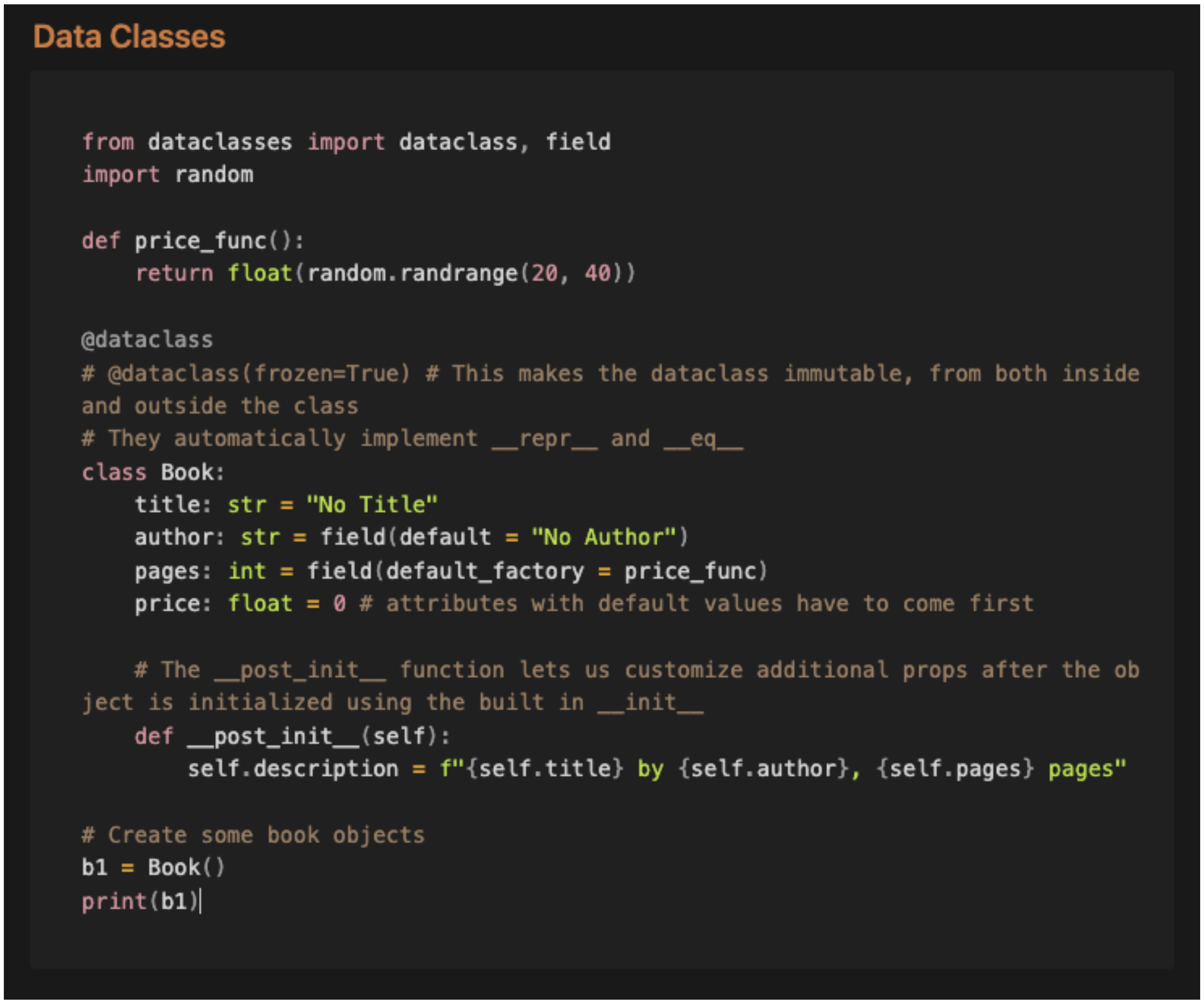
Python's magic methods also give you complete control over how an object's attributes are accessed. Your class can define methods that intercept the process any time an attribute is set or retrieved.

```python
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price
        self._discount = 0.1

    def __str__(self):
        return f"{self.title} by {self.author}, costs {self.price}"

    # __getattribute__ called when an attribute is retrieved
    # Don't directly access the attr name otherwise a recursive loop is created
    def __getattribute__(self, name):
        if (name == "price"):
            p = super().__getattribute__("price")
            d = super().__getattribute__("_discount")
            return p - (p * d)
        return super().__getattribute__(name)

    # __setattr__ called when an attribute value is set
    def __setattr__(self, name, value):
        if (name == "price"):
            if type(value) is not float:
                raise ValueError("The price must be a float")
        return super().__setattr__(name, value)

    # __getattr__ called when __getattribute__ lookup fails
    def __getattr__(self, item):
        return f"{item} is not here."
```



Callable Objects

```python
class Book:
    def __init__(self, title, author, price):
        super().__init__()
        self.title = title
        self.author = author
        self.price = price
        self._discount = 0.1

    def __str__(self):
        return f"{self.title} by {self.author}, costs {self.price}"

    def __call__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price

b1 = Book("War and Peace", "Leo Tolstoy", 39.95)
b2 = Book("The Catcher in the Rye", "JD Salinger", 29.95)
print(b1)
b1("Best Book", "Aryan", 2003)
print(b1)
```



Data Classes

```python
from dataclasses import dataclass, field
import random

def price_func():
    return float(random.randrange(20, 40))

@dataclass
# @dataclass(frozen=True) # This makes the dataclass immutable, from both inside
and outside the class
# They automatically implement __repr__ and __eq__
class Book:
    title: str = "No Title"
    author: str = field(default = "No Author")
    pages: int = field(default_factory = price_func)
    price: float = 0 # attributes with default values have to come first

    # The __post_init__ function lets us customize additional props after the ob
ject is initialized using the built in __init__
    def __post_init__(self):
        self.description = f"{self.title} by {self.author}, {self.pages} pages"

# Create some book objects
b1 = Book()
print(b1)
```