Chapter 15

How to work with inheritance

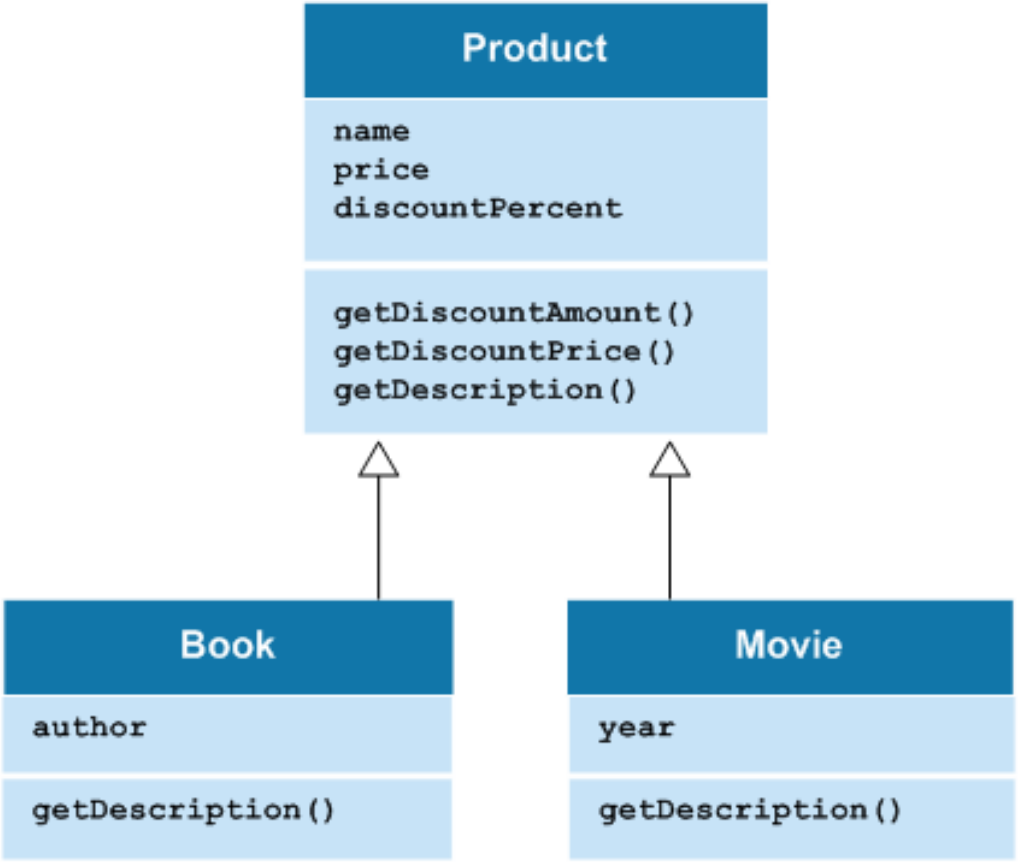
Applied objectives

1. Define and use a subclass that inherits a superclass and overrides one or more of the methods of the superclass.
2. Define and use a class that overrides one or more methods of the object class.

Knowledge objectives

1. Describe the way inheritance works.
2. In general terms, explain how to override a method in the superclass when you're defining a subclass.
3. Describe the concept of polymorphism.
4. Describe the use of the isinstance() method when working with objects.
5. Describe the use of the __str__() method of the object class for defining a string representation for an object.
6. Explain what an iterator is and how you create one using the __iter__() method of the object class and the yield keyword.
7. Describe three factors that help determine when it is appropriate to use inheritance

A UML diagram for three classes that use inheritance



UML diagramming note

- To indicate that a class inherits another class, a UML diagram typically uses an arrow with an open arrowhead, not a shaded arrowhead as shown in the previous slide.

The syntax for working with subclasses

To define a subclass

```
class SubClassName(SuperClassName):
```

To call a method or constructor of the superclass

```
SuperClassName.methodName(self[, argumentList])
```

The code for the Product superclass

```
@dataclass
class Product:
    name:str = ""
    price:float = 0.0
    discountPercent:int = 0

    def getDiscountAmount(self):
        return self.price * self.discountPercent / 100

    def getDiscountPrice(self):
        return self.price - self.getDiscountAmount()

    def getDescription(self):
        return self.name
```

The code for the Book subclass

```
@dataclass
class Book(Product):  # Python calls the constructor
                      # of the superclass

    author:str = ""   # add another attribute to the
                      # three in the superclass

    # override the getDescription method
    def getDescription(self):
        return f"{Product.getDescription(self)} by "
               f"{self.author}"
```

The code for the Book subclass with a constructor

```
class Book(Product):

def __init__(self, name="", price=0.0,
             discountPercent=0, author=""):
    # call the constructor of the superclass
    Product.__init__(self, name, price, discountPercent)

    # set the author
    self.author = author

...
```

When coding a subclass…

- You can directly access public attributes of the superclass.
- You can add new attributes and methods that aren't in the superclass.
- You can call methods of the superclass (including constructors and properties) by coding the name of the superclass, the dot operator, and the name of the method.
- You can override existing methods in the superclass by coding methods that have the same name.

Three versions of the getDescription() method

In the Product superclass

```
def getDescription(self):
    return self.name
```

In the Book subclass

```
def getDescription(self):
    return f"{Product.getDescription(self)} "
           f"by {self.author}"
```

In the Movie subclass

```
def getDescription(self):
    return f"{Product.getDescription(self)} "
           f"({self.year})"
```

Code that uses the overridden methods

```
from objects import Product, Book, Movie

def show_products(products):
    print("PRODUCTS")
    for product in products:
        print(product.getDescription())
    print()

def main():
    # a tuple of Product objects
    products = (
        Product("Stanley 13 Ounce Wood Hammer", 12.99, 62),
        Book("The Big Short", 15.95, 34, "Michael Lewis"),
        Movie("The Holy Grail - DVD", 14.99, 68, 1975))
    show_products(products)
```

The console

```
PRODUCTS
Stanley 13 Ounce Wood Hammer
The Big Short by Michael Lewis
The Holy Grail - DVD (1975)
```

A function for checking an object's type

```
isinstance(object, [modName.]ClassName)
```

Code that uses the isinstance() method

```
from objects import Product, Book, Movie

def show_product(product):
    print("PRODUCT DATA")
    print(f"Name:                {product.name}")
    if isinstance(product, Book):
        print(f"Author:              {product.author}")
    if isinstance(product, Movie):
        print(f"Year:                {product.year}")
    print(f"Discount price:      {product.getDiscountPrice():.2f}")
    print()

def main():
    product1 = Product("Stanley 13 Ounce Wood Hammer", 12.99, 62)
    show_product(product1)
    print()

    product2 = Movie("The Holy Grail - DVD", 14.99, 68, 1975)
    show_product(product2)

if __name__ == "__main__":
    main()
```

The console

```
PRODUCT DATA
Name:               Stanley 13 Ounce Wood Hammer
Discount price:     4.94

PRODUCT DATA
Name:               The Holy Grail - DVD
Year:               1975
Discount price:     4.80
```

The objects module (part 1)

```
from dataclasses import dataclass

@dataclass
class Product:
    name:str = ""
    price:float = 0.0
    discountPercent:int = 0

    def getDiscountAmount(self):
        return self.price * self.discountPercent / 100

    def getDiscountPrice(self):
        return self.price - self.getDiscountAmount()

    def getDescription(self):
        return self.name
```

The objects module (part 2)

```
@dataclass
class Book(Product):
    author:str = ""

    def getDescription(self):
        return f"{Product.getDescription(self)} by {self.author}"

@dataclass
class Movie(Product):
    year:int = 0

    def getDescription(self):
        return f"{Product.getDescription(self)} ({self.year})"
```

The user interface for the Product Viewer

```
PRODUCTS
1. Stanley 13 Ounce Wood Hammer
2. The Big Short by Michael Lewis
3. The Holy Grail - DVD (1975)

Enter product number: 2

PRODUCT DATA
Name:               The Big Short
Author:             Michael Lewis
Discount price:     10.53

View another product? (y/n)
```

The product_viewer module (part 1)

```
from objects import Product, Book, Movie

def show_products(products):
    print("PRODUCTS")
    for i, product in enumerate(products, start=1):
        print(f"{i}. {product.getDescription()}")
    print()

def show_product(product):
    w=18
    print("PRODUCT DATA")
    print(f"{'Name:':{w}}{product.name}")
    if isinstance(product, Book):
        print(f"{'Author:':{w}}{product.author}")
    if isinstance(product, Movie):
        print(f"{'Year:':{w}}{product.year}")
    print(f"{'Discount price:':{w}}{
        product.getDiscountPrice():.2f}")
    print()
```

The product_viewer module (part 2)

```
def get_products():
    # return a tuple of Product, Book, and Movie objects
    return (Product("Stanley 13 Ounce Wood Hammer", 12.99, 62),
            Book("The Big Short", 15.95, 34, "Michael Lewis"),
            Movie("The Holy Grail - DVD", 14.99, 68, 1975))

def get_product(products):

    # same as code in figure 14-6


def main():

    # same as code in figure 14-6

if __name__ == "__main__":
    main()
```

A method of the object class

```
__str__(self)
```

The syntax for overriding the __str__() method

```
def __str__(self):
    return stringForObject
```

A __str__() method in the Product class

```
def __str__(self):
    return f"{self.name} | {self.price} | {self.discountPercent}"
```

Code that automatically calls the __str__() method

```
product = Product("Stanley 13 Ounce Wood Hammer", 12.99, 62)
print(product)
```

The console if the Product class overrides the __str__() method

```
Stanley 13 Ounce Wood Hammer | 12.99 | 62
```

The console if the __str__() method isn't overridden (data class)

```
Product(name='Stanley 13 Ounce Wood Hammer',
price=12.99, discountPercent=62)
```

The console if the __str__() method isn't overridden (explicit constructor)

```
<objects.Product object at 0x03769930>
```

The __iter__() method of the object class

```
__iter__(self)
```

The yield keyword

```
yield
```

The constructor for a Dice class

```
class Dice:
    def __init__(self):
        self.__list = []
```

A method that defines an iterator for the class

```
def __iter__(self):
    for die in self.__list:
        yield die
```

A Dice object that contains five Die objects

```
dice = Dice()
for i in range(5):
    die = Die()
    dice.addDie(die)
```

Code that automatically calls the __iter__() method

```
for die in dice:
    print(die.value, end=" ")
```

The console if the Dice class defines an iterator

```
1 1 1 1 1
```

The console if the Dice class doesn't define an iterator

```
TypeError: 'Dice' object is not iterable
```

The dice module (part 1)

```
import random
from dataclasses import dataclass

@dataclass
class Die:
    __value:int = 1

    @property  # read-only!
    def value(self):
        return self.__value

    def roll(self):
        self.__value = random.randrange(1, 7)

    # make it easier to get the value
    def __str__(self):
        return str(self.__value)
```

The dice module (part 2)

```
class Dice:
    def __init__(self):
        self.__list = []

    def addDie(self, die):
        self.__list.append(die)

    def rollAll(self):
        for die in self.__list:
            die.roll()

    def __iter__(self):
        for die in self.__list:
            yield die
```

Code that displays the value of each die

```
for die in dice:
    print(die, end=" ")
```

The hierarchy for six common exceptions

```
Exception
    NameError
    OSError
        FileExistsError
        FileNotFoundError
    ValueError
```

The syntax for creating your own exceptions

```
class CustomErrorName(ExceptionClassName):
    pass
```

A class that defines a custom exception

```
class DataAccessError(Exception):
    pass
```

A module that uses the DataAccessError class

```
from objects import DataAccessError

def read_movies():
    try:
        movies = []
        with open("movies.csv", newline="") as file:
            reader = csv.reader(file)
            for row in reader:
                movies.append(row)
        return movies
    except FileNotFoundError:
        raise DataAccessError("Data source not found.")
    except Exception:
        raise DataAccessError("Error accessing data source.")
```

Code that handles a custom exception

```
from objects import DataAccessError

try:
    movies = db.read_movies()
except DataAccessError as e:
    print("DataAccessError:", e)
```

The console when the FileNotFoundError occurs

```
DataAccessError: Data source not found.
```

The console if you don't use the DataAccessError class

```
FileNotFoundError: [Errno 2] No such file or directory:
'movies.csv'
```

It makes sense to use inheritance when…

- One object *is a* type of another object.
- Both classes are part of the same logical domain.
- The subclass primarily adds features to the superclass.

A Dice class that inherits the list class (not recommended)

```
class Dice(list):
    def rollAll(self):
        for die in self:
            die.roll()
```

Code that uses this Dice class

```
dice = Dice()
dice.append(Die())     # uses method from list class
dice.append(Die())     # uses method from list class
dice.rollAll()
die = dice[0]          # uses operators from list class
dice.insert(0, Die()) # uses method from list class
dice.pop()             # uses method from list class
print("Die value:", die.value)
print("Dice count:", len(dice))
```

A few of the problems with this approach

- **The Dice object *is not a* type of list object.** A list object stores any type of object and provides a wide variety of methods for working with those objects. A Dice object stores Die objects and provides specialized methods for working with them.
- **Both classes are *not* part of the same logical domain.** The list class is an implementation class that provides a general-purpose object that all Python programmers can use to work with lists of objects. The Dice class is part of a specific logical domain that creates a model that programmers can use to store and roll multiple dice.
- **The interface is too complex.** The Dice object should only provide the methods necessary to use it. This makes it easy for other programmers to use.
- **It violates encapsulation.** The Dice object allows other programmers to access the list that stores the Die objects. But the list is an implementation choice that should be hidden from other programmers in case you want or need to change the implementation later.