Chapter 14

How to define and use your own classes

Objectives

Applied

1. Code the constructor for a class that has attributes and methods.
2. Import a class, create objects from it, access the attributes of the objects, and call the methods of the objects.
3. Use object composition to combine simple objects into more complex data structures.
4. Use encapsulation to hide the data attributes of an object.

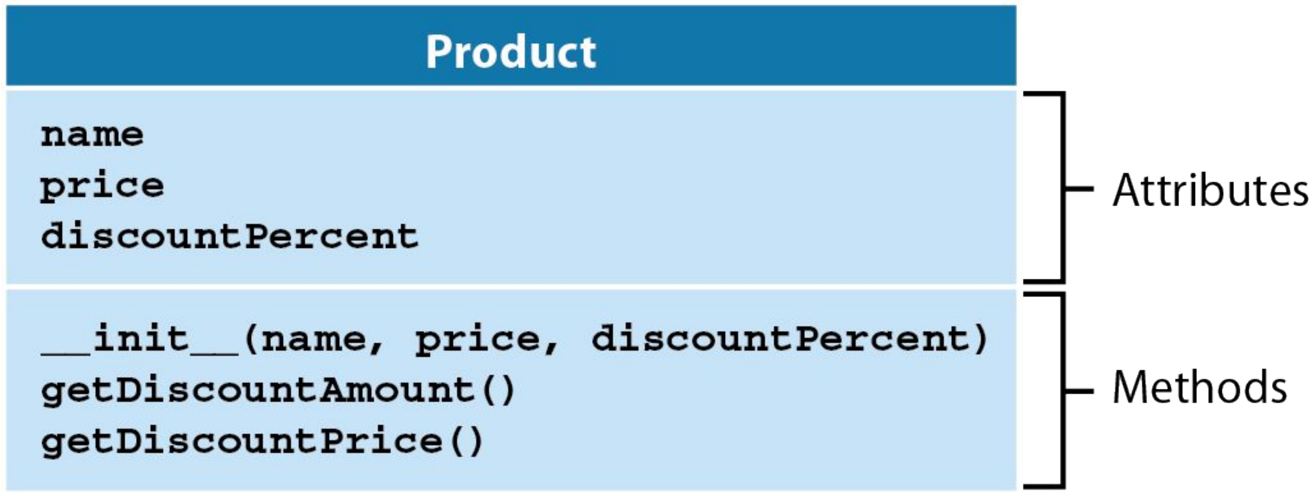
Knowledge

1. Describe a UML class diagram.
2. Describe the relationship between a class and an object.

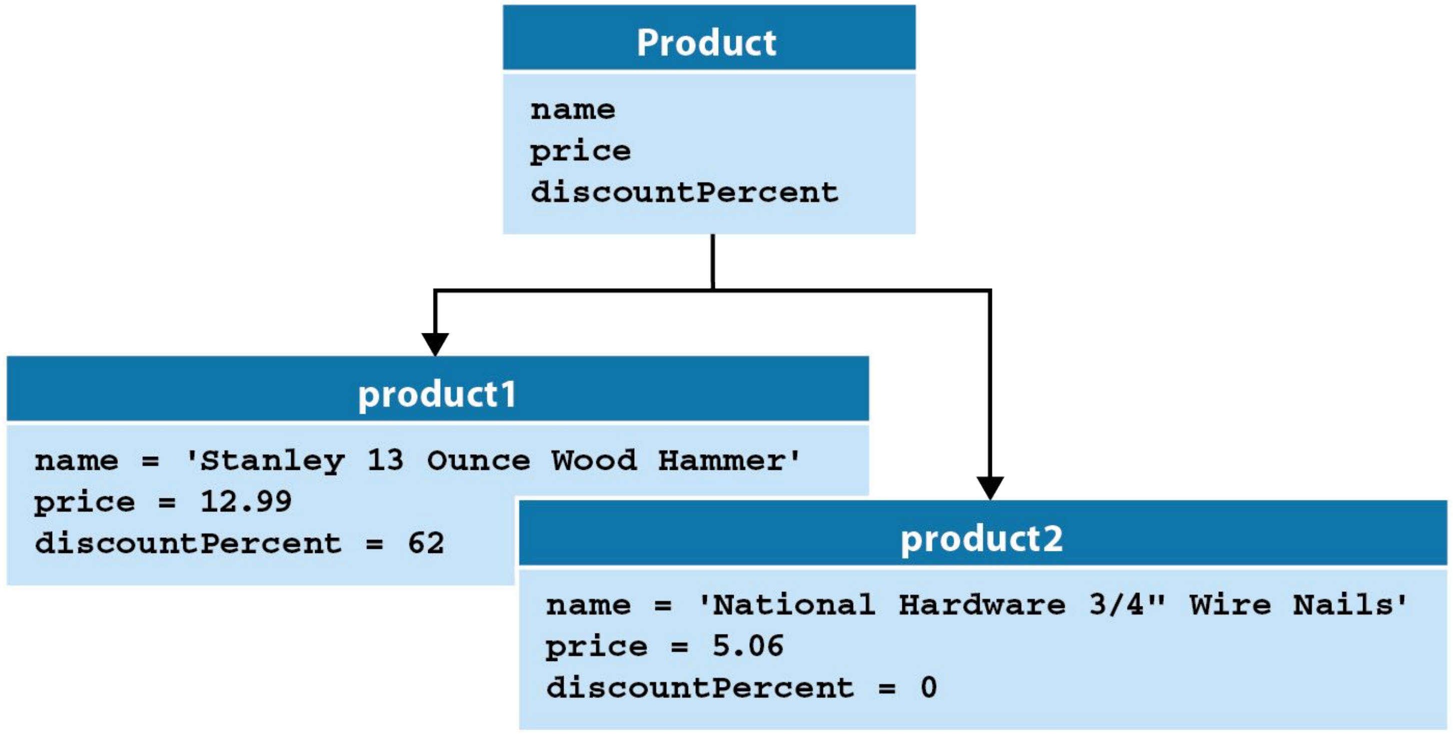
Objectives (cont.)

3. In general terms, describe the identity, state, and behavior of an object.
4. In general terms, describe the way Python code is used to define a constructor, its attributes, and its methods.
5. In general terms, describe the way Python code is used to create an object from a class.
6. Describe the concept of object composition.
7. Describe the concept of encapsulation.
8. Distinguish between public and private attributes.
9. Describe the use of getter and setter methods.

A diagram of the Product class



The relationship between a class and its objects



UML diagramming notes

- *UML* (*Unified Modeling Language*) is the industry standard used to describe the classes and objects of an object-oriented application.
- A UML *class diagram* describes the attributes and methods of one or more classes.

The Product class in the module named objects

```
class Product:
    # a constructor that initializes 3 attributes
    def __init__(self, name, price, discountPercent):
        self.name = name                            # attribute 1
        self.price = price                          # attribute 2
        self.discountPercent = discountPercent      # attribute 3

    # a method that uses two attributes
    def getDiscountAmount(self):
        return self.price * self.discountPercent / 100

    # a method that calls another method
    def getDiscountPrice(self):
        return self.price - self.getDiscountAmount()
```

A script that creates and uses a Product object

```
from objects import Product

# create two product objects
product1 = Product("Stanley 13 Ounce Wood Hammer", 12.99, 62)
product2 = Product('National Hardware 3/4" Wire Nails', 5.06, 0)

# print data for product1 to console
print("PRODUCT DATA")
print("Name:             {:s}".format(product1.name))
print("Price:            {:.2f}".format(product1.price))
print("Discount percent: {:d}%".format(product1.discountPercent))
print("Discount amount:  {:.2f}".format(
                                 product1.getDiscountAmount()))
print("Discount price:   {:.2f}".format(
                                 product1.getDiscountPrice()))
```

The console

```
PRODUCT DATA
Name:             Stanley 13 Ounce Wood Hammer
Price:            12.99
Discount percent: 62%
Discount amount:  8.05
Discount price:   4.94
```

How to import a class

The syntax

```
from module_name import ClassName1[, ClassName2]...
```

Import the Product class from the objects module

```
from objects import Product
```

How to create an object

The syntax

```
objectName = ClassName([parameters])
```

Create two Product objects

```
product1 = Product('Stanley 13 Ounce Wood Hammer',
                   12.99, 62)
product2 = Product('National Hardware 3/4" Wire Nails',
                   5.06, 0)
```

How to access the attributes of an object

The syntax

```
objectName.attributeName
```

Set an attribute

```
product1.discountPercent = 40
```

Get an attribute

```
percent = product1.discountPercent     # percent = 40
```

How to call the methods of an object

The syntax

```
objectName.methodName([parameters])
```

Call the getDiscountAmount() method

```
discount = product1.getDiscountAmount()
```

Call the getDiscountPrice() method

```
salePrice = product1.getDiscountPrice()
```

The syntax for coding a constructor

```
def __init__(self[, parameters]):       # the constructor
    self.attrName1 = attrValue1         # first attribute
    self.attrName2 = attrValue2         # second attribute
    ...
```

A constructor with no parameters

```
def __init__(self):
    self.name = ""
    self.price = 0.0
    self.discountPercent = 0
```

Code that uses this constructor to create an object

```
product = Product()
```

Code that sets the attributes of the object

```
product.name = "Stanley 13 Ounce Wood Hammer"
product.price = 12.99
product.discountPercent = 62
```

A constructor with three parameters

```
def __init__(self, name, price, discountPercent):
    self.name = name
    self.price = price
    self.discountPercent = discountPercent
```

Code that creates an object and set its attributes

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

A constructor with default values for parameters

```
def __init__(self, name="", price="0.0",
             discountPercent=0):
    self.name = name
    self.price = price
    self.discountPercent = discountPercent
```

Code that supplies all three parameters

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

Code that supplies just two parameters so the default value is used for discountPercent

```
product = Product(name="Stanley 13 Ounce Wood Hammer",
                  price=12.99)
```

The syntax for coding a method

```
def methodName(self[, parameters]):
    statements
```

A method that returns a value

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

Code that calls this method

```
discountAmount = product.getDiscountAmount()
```

A more concise way to code this method

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

Code that calls this method

```
discountAmount = product.getDiscountAmount()
```

A method that calls another method of the class

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

Code that calls this method

```
discountPrice = product.getDiscountPrice()
```

A method of the Product class that accepts a parameter

```
def getPriceStr(self, country):
    priceStr = "{:.2f}".format(self.price)
    if country == "US":
        priceStr += " USD"
    elif country == "DE":
        priceStr = priceStr + " EUR"
    return priceStr
```

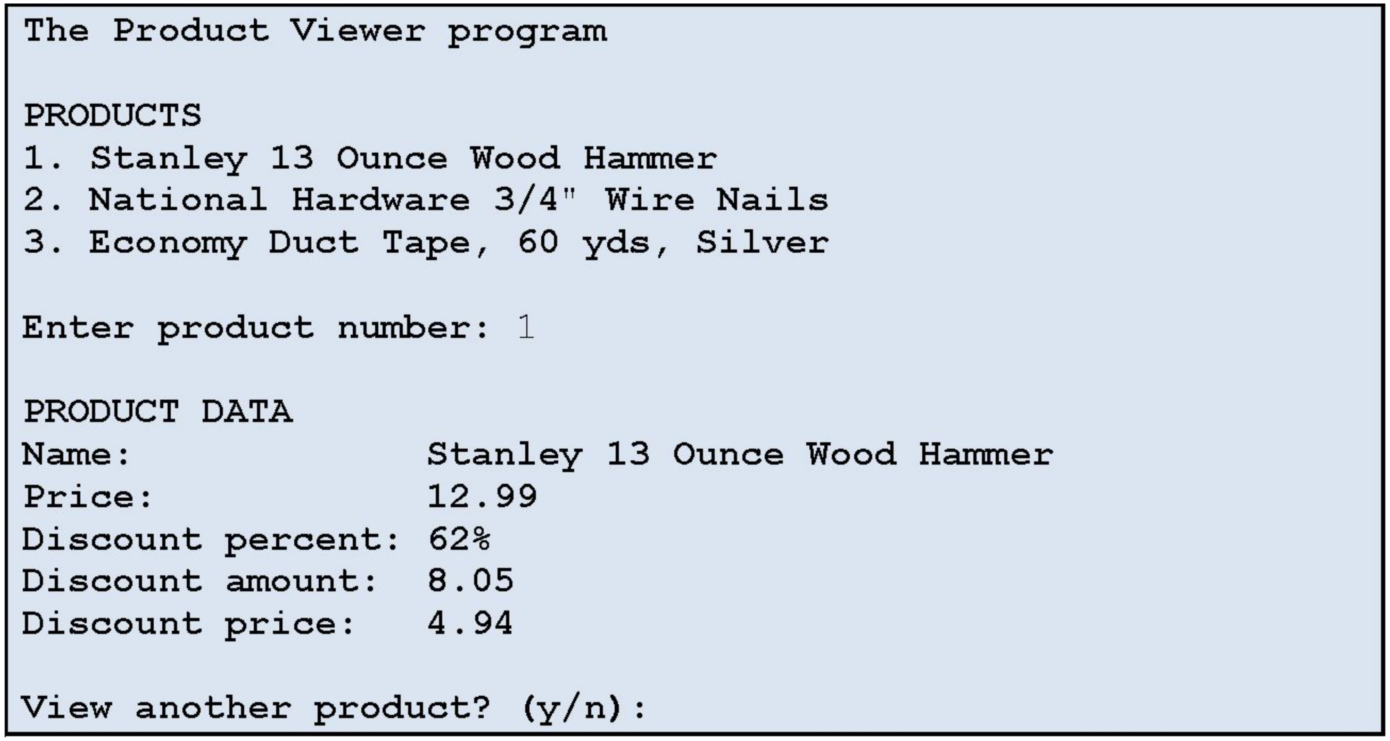
Code that calls this method

```
print("Price: " + product.getPriceStr("US"))
```

The error that's displayed if you forget to code the self parameter

```
TypeError: getPriceStr() takes 1 positional argument but 2
were given
```

The console for the Product Viewer



```
The Product Viewer program

PRODUCTS
1. Stanley 13 Ounce Wood Hammer
2. National Hardware 3/4" Wire Nails
3. Economy Duct Tape, 60 yds, Silver

Enter product number: 1

PRODUCT DATA
Name:              Stanley 13 Ounce Wood Hammer
Price:             12.99
Discount percent: 62%
Discount amount:   8.05
Discount price:    4.94

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

The objects module

```
class Product:
    def __init__(self, name, price, discountPercent):
        self.name = name
        self.price = price
        self.discountPercent = discountPercent

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

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

The product_viewer module

```
from objects import Product

def show_products(products):
    print("PRODUCTS")
    for i in range(len(products)):
        product = products[i]
        print(str(i+1) + ". " + product.name)
    print()

def show_product(product):
    print("PRODUCT DATA")
    print("Name:             {:s}".format(
          product.name))
    print("Price:            {:.2f}".format(
          product.price))
    print("Discount percent: {:d}%".format(
          product.discountPercent))
    print("Discount amount:  {:.2f}".format(
          product.getDiscountAmount()))
    print("Discount price:   {:.2f}".format(
          product.getDiscountPrice())
    print()
```

The product_viewer module (cont.)

```
def main():
    print("The Product Viewer program")
    print()

    # a tuple of Product objects
    products = (Product("Stanley 13 Ounce Wood Hammer",
                        12.99, 62),
                Product('National Hardware 3/4" Wire Nails',
                        5.06, 0),
                Product("Economy Duct Tape, 60 yds, Silver",
                        7.24, 0))
    show_products(products)
```

The product_viewer module (cont.)

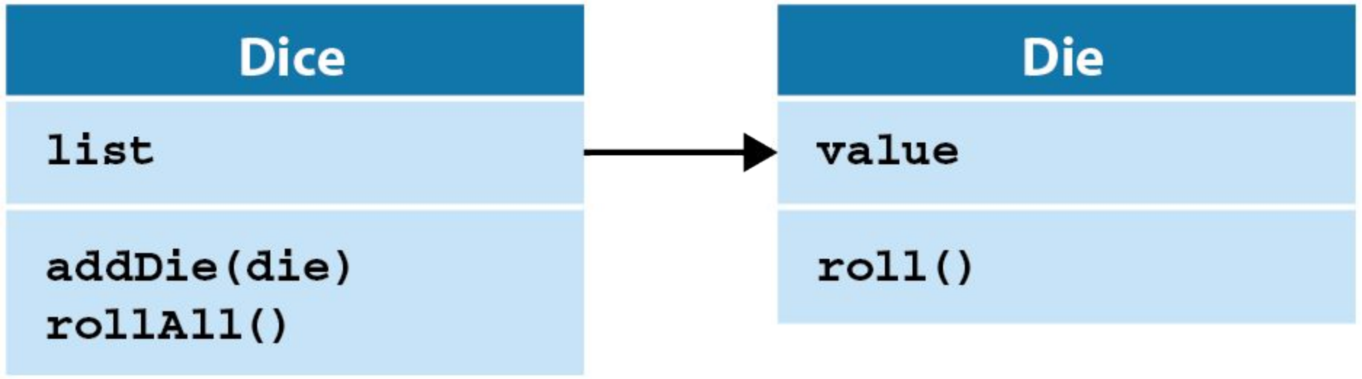
```
    while True:
        number = int(input("Enter product number:  "))
        print()

        product = products[number-1]
        show_product(product)

        choice = input("View another product? (y/n): ")
        print()
        if choice != "y":
            print("Bye!")
            break


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

A UML diagram for two classes that use composition



The dice module

```
import random

class Die:
    def __init__(self):
        self.value = 1

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


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()
```

The console for the Dice Roller

```
The Dice Roller program

Enter the number of dice to roll: 5
YOUR ROLL: 1 5 1 2 6

Roll again? (y/n): y
YOUR ROLL: 1 1 4 3 4

Roll again? (y/n): y
YOUR ROLL: 5 4 6 2 2

Roll again? (y/n): n
Bye!
```

The dice_roller module

```
from dice import Dice, Die

def main():
    print("The Dice Roller program")
    print()

    # get number of dice from user
    count = int(input("Enter the number of dice to roll: "))

    # Dice object and add Die objects to it
    dice = Dice()
    for i in range(count):
        die = Die()
        dice.addDie(die)
```

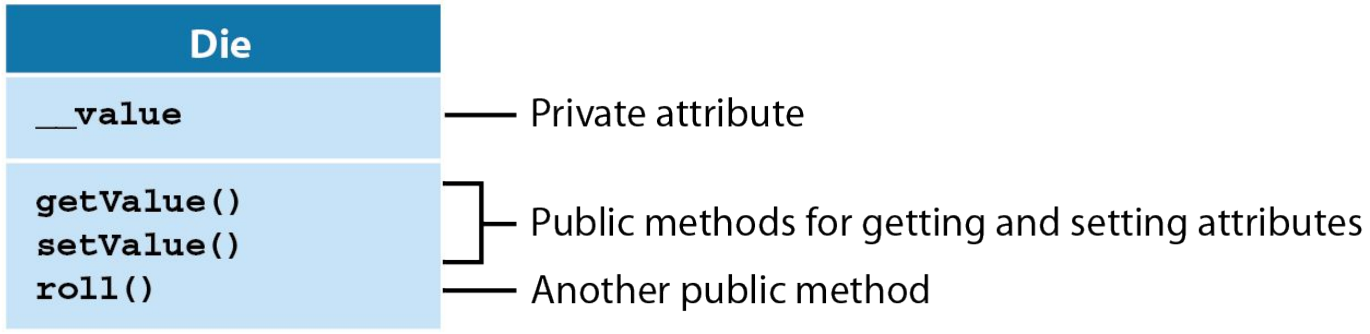
The dice_roller module (cont.)

```
        while True:
            # roll the dice
            dice.rollAll()
            print("YOUR ROLL: ", end="")
            for die in dice.list:
                print(die.value, end=" ")
            print("\n")

            choice = input("Roll again? (y/n): ")
            if choice != "y":
                print("Bye!")
                break

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

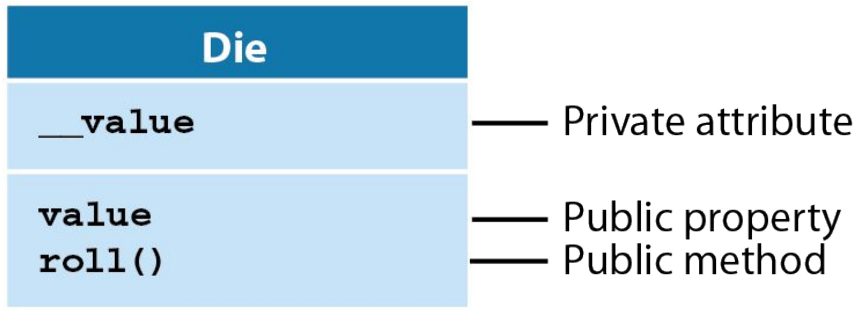
A Die class that uses methods to provide encapsulation



UML diagramming note

- The double underscores (__) identify the attributes that are private.

A Die class that uses properties to provide encapsulation



The Die class with a public attribute named value

```
class Die:
    def __init__(self):
        self.value = 1

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

Code that directly sets and gets the public attribute

```
die = Die()
die.value = 10     # illegal value!
print("Die:", die.value)
```

The message that's displayed on the console

```
Die: 10
```

The Die class with a private attribute named __value

```
class Die:
    def __init__(self):
        self.__value = 1

    def getValue(self):
        return self.__value

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

Code that attempts to directly access a private attribute

```
die = Die()
die.__value = 10
```

The error message that's displayed on the console

```
AttributeError: 'Die' object has no attribute '__value'
```

Code that indirectly sets and gets the private attribute

```
die = Die()
die.roll()
print("Die:", die.getValue())
```

The Die class with methods that access a private attribute

```
class Die:
    def __init__(self):
        self.__value = 1

    def getValue(self):
        return self.__value

    def setValue(self, value):
        if value < 1 or value > 6:
            raise ValueError(
                "Die value must be from 1 to 6.")
        else:
            self.__value = value

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

Code that uses the getter and setter methods

```
die = Die()
die.setValue(6)
print("Die:", die.getValue())
```

The message that's displayed on the console

```
Die:  6
```

Code that attempts to use the setValue() method to set invalid data

```
die = Die()
die.setValue(-1)
```

The error message that's displayed on the console

```
ValueError: Die value must be from 1 to 6.
```

Two annotations for getting and setting properties

```
@property
@propertyName.setter
```

A Die class that uses a property to access a private attribute

```
class Die:
    def __init__(self):
        self.__value = 1

    @property
    def value(self):
        return self.__value

    @value.setter
    def value(self, value):
        if value < 1 or value > 6:
            raise ValueError(
                "Die value must be from 1 to 6.")
        else:
            self.__value = value
```

Code that uses the value property to get and set data

```
die = Die()
die.value = 6
print("Die:", die.value)
```

The message that's displayed on the console

```
Die:  6
```

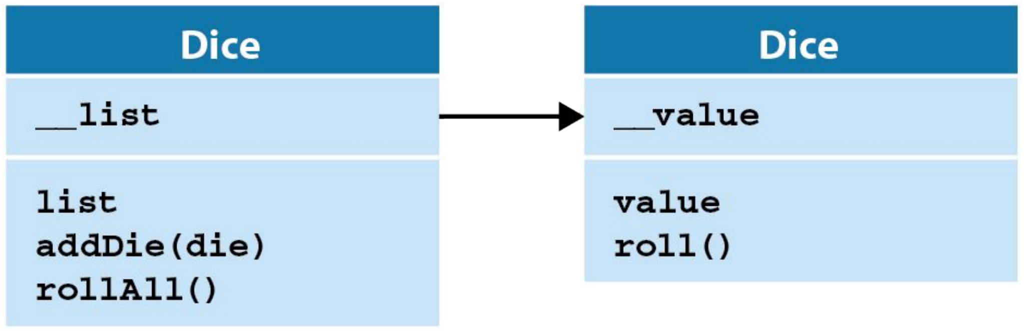
Code that attempts to use the value property to set invalid data

```
die = Die()
die.value = -1
```

The error message that's displayed on the console

```
ValueError: Die value must be from 1 to 6.
```

A UML diagram for two classes that use encapsulation



The dice module

```
import random

class Die:
    def __init__(self):
        self.__value = 1

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

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

The dice module (cont.)

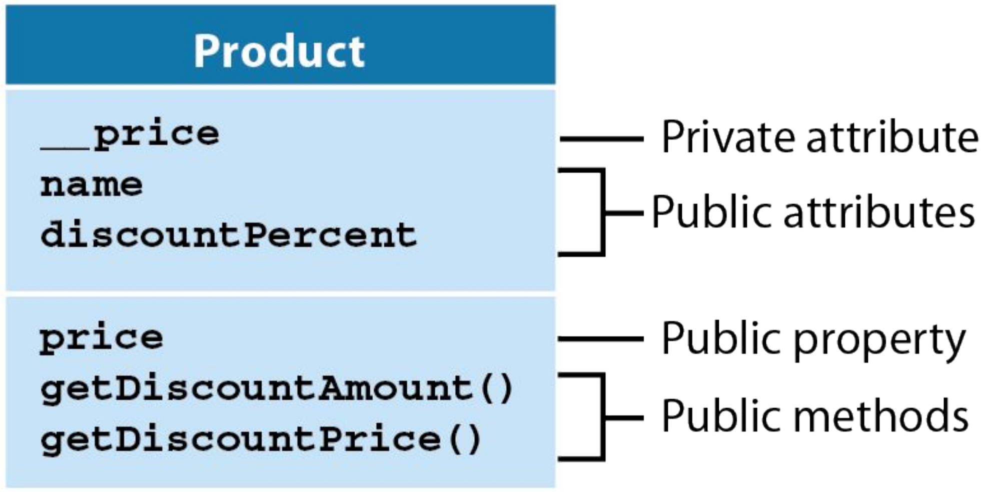
```
class Dice:
    def __init__(self):
        self.__list = []

    @property                              # read-only
    def list(self):
        dice_tuple = tuple(self.__list)
        return dice_tuple

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

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

A UML diagram for a Product class that uses some encapsulation



The code for the Product class

```
class Product:
    def __init__(self, name="", price=0.0,
                 discountPercent=0):
        self.name = name
        self.price = price   # passes param to setter
        self.discountPercent = discountPercent

    @property
    def price(self):
        return self.__price

    @price.setter
    def price(self, price):
        if price < 0:
            raise ValueError(
                "Price can't be less than 0")
        else:
            self.__price = price
```

The code for the Product class (cont.)

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

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

Code that attempts to use the price property to set invalid data

```
product = Product()
product.price = -11.50
```

Code that attempts to use the constructor to set invalid data

```
product = Product("Hammer", -11.50)
```

The error message that's displayed on the console

```
ValueError: Price can't be less than 0
```

The console for the Pig Dice game

```
Let's Play PIG!

* See how many turns it takes you to get to 20.
* Turn ends when you hold or roll a 1.
* If you roll a 1, you lose all points for the turn.
* If you hold, you save all points for the turn.

TURN 1
Roll or hold? (r/h): r
Die: 5
Roll or hold? (r/h): r
Die: 4
Roll or hold? (r/h): r
Die: 5
Roll or hold? (r/h): h
Score for turn: 14
Total score: 14

TURN 2
Roll or hold? (r/h): r
Die: 6
Roll or hold? (r/h): h
Score for turn: 6
Total score: 20

You finished in 2 turns!
Play again? (y/n):
```

The game module

```
from dice import Die

class Game:
    def __init__(self):
        self.turn = 1
        self.score = 0
        self.scoreThisTurn = 0
        self.isTurnOver = False
        self.isGameOver = False
        self.die = Die()
```

The pig_dice module

```
from game import Game

def display_welcome():
    print("Let's Play PIG!")
    print()
    print("* See how many turns it takes you to get to 20.")
    print("* Turn ends when you hold or roll a 1.")
    print("* If you roll a 1, you lose all points for the turn.")
    print("* If you hold, you save all points for the turn.")
    print()
```

The pig_dice module (continued)

```
def play_game():
    game = Game()
    while not game.isGameOver:
        take_turn(game)

def take_turn(game):
    print("TURN", game.turn)
    game.scoreThisTurn = 0
    game.isTurnOver = False
    while not game.isTurnOver:
        choice = input("Roll or hold?  (r/h): ")
        if choice == "r":
            roll_die(game)
        elif choice == "h":
            hold_turn(game)
        else:
            print("Invalid choice. Try again.")
```

The pig_dice module (continued)

```
def roll_die(game):
    game.die.roll()
    print("Die:", game.die.value)
    if game.die.value == 1:
        game.scoreThisTurn = 0
        game.turn += 1
        game.isTurnOver = True
        print("Turn over. No score.\n")
    else:
        game.scoreThisTurn += game.die.value

def hold_turn(game):
    game.score += game.scoreThisTurn
    game.isTurnOver = True
    print("Score for turn:", game.scoreThisTurn)
    print("Total score:", game.score, "\n")
    if game.score >= 20:
        game.isGameOver = True
        print("You finished in", game.turn, "turns!")
    else:
        game.turn += 1
```

The pig_dice module (continued)

```
def main():
    display_welcome()
    while True:
        play_game()
        choice = input("Play again?  (y/n): ")
        print()
        if choice != "y":
            print("Bye!")
            break

# if started as the main module, call the main function
if __name__ == "__main__":
    main()
```