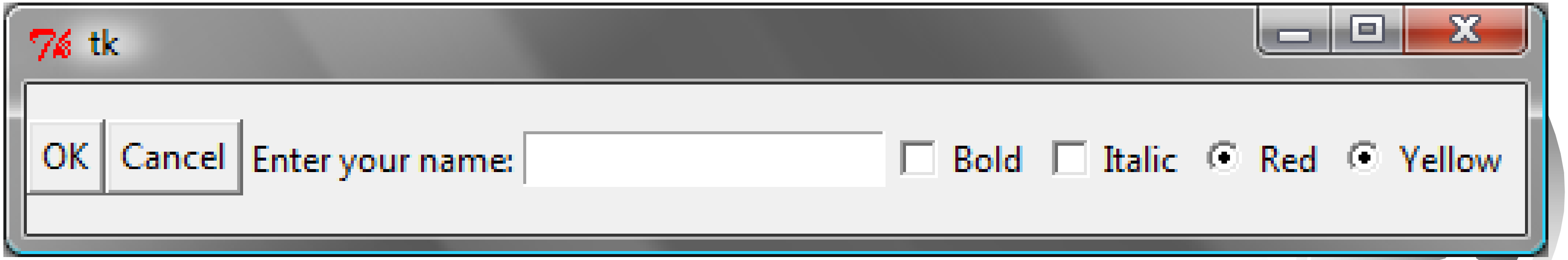
Chapter 9 Object-Oriented Programming

Motivations

After learning the preceding chapters, you are capable of solving many programming problems using selections, loops, and functions. However, these Python features are not sufficient for developing graphical user interfaces and large scale software systems. Suppose you want to develop a graphical user interface as shown below. How do you program it?



OO Programming Concepts

- Object-oriented programming (OOP) involves programming using *objects*

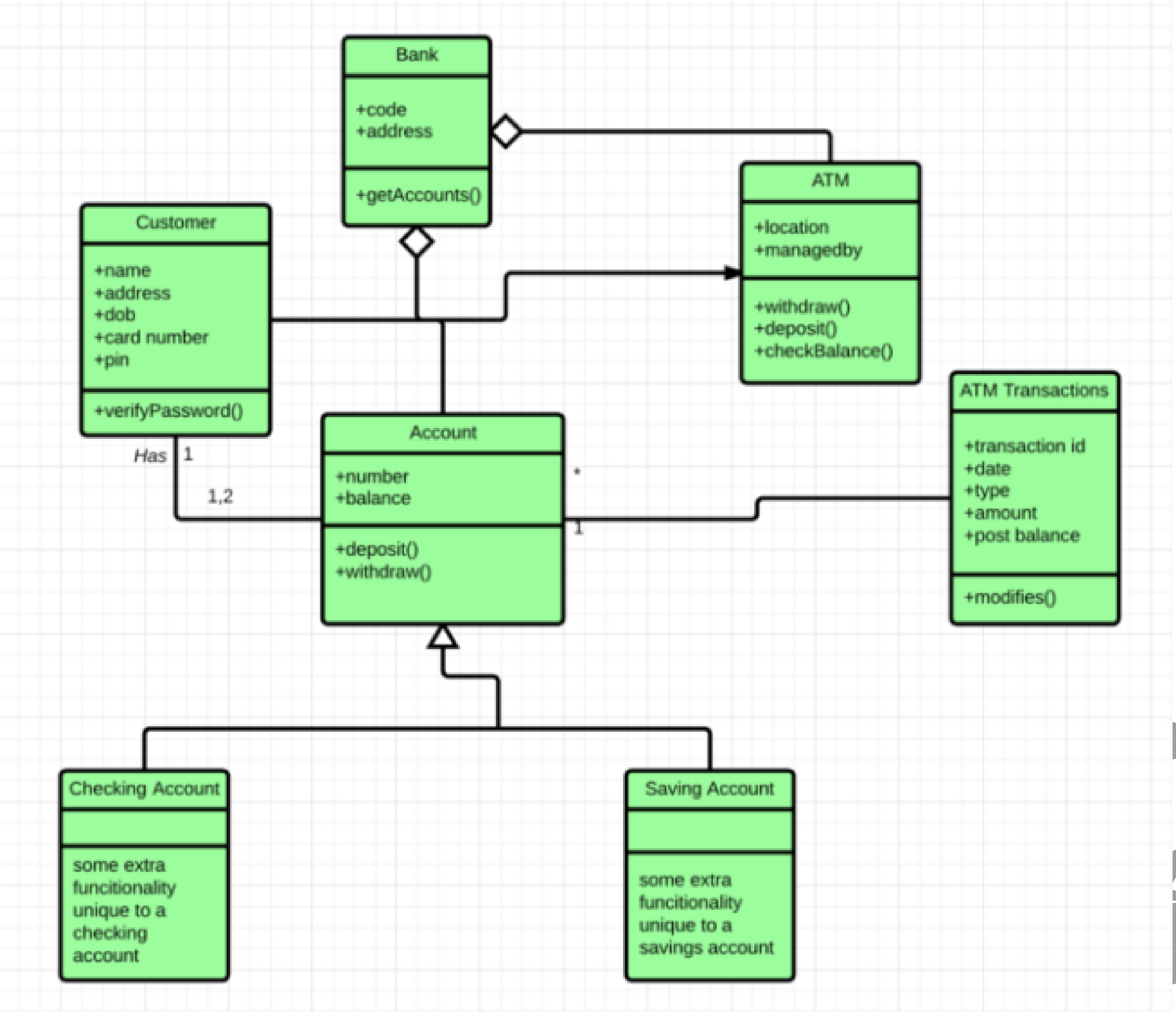
- An *object* represents an entity in the real world that can be distinctly identified

  - For example, a student, a circle, a button, and even a loan can all be viewed as objects

- An object has a unique identity, state, and behaviors

  - The *state* of an object consists of a set of ***data fields*** (also known as *properties*) with their current values

  - The *behavior* of an object is defined by a set of ***methods***



Bank

+code
+address

+getAccounts()

ATM

+location
+managedby

+withdraw()
+deposit()
+checkBalance()

Customer

+name
+address
+dob
+card number
+pin

+verifyPassword()

Has 1

1,2

Account

+number
+balance

+deposit()
+withdraw()

*

1

ATM Transactions

+transaction id
+date
+type
+amount
+post balance

+modifies()

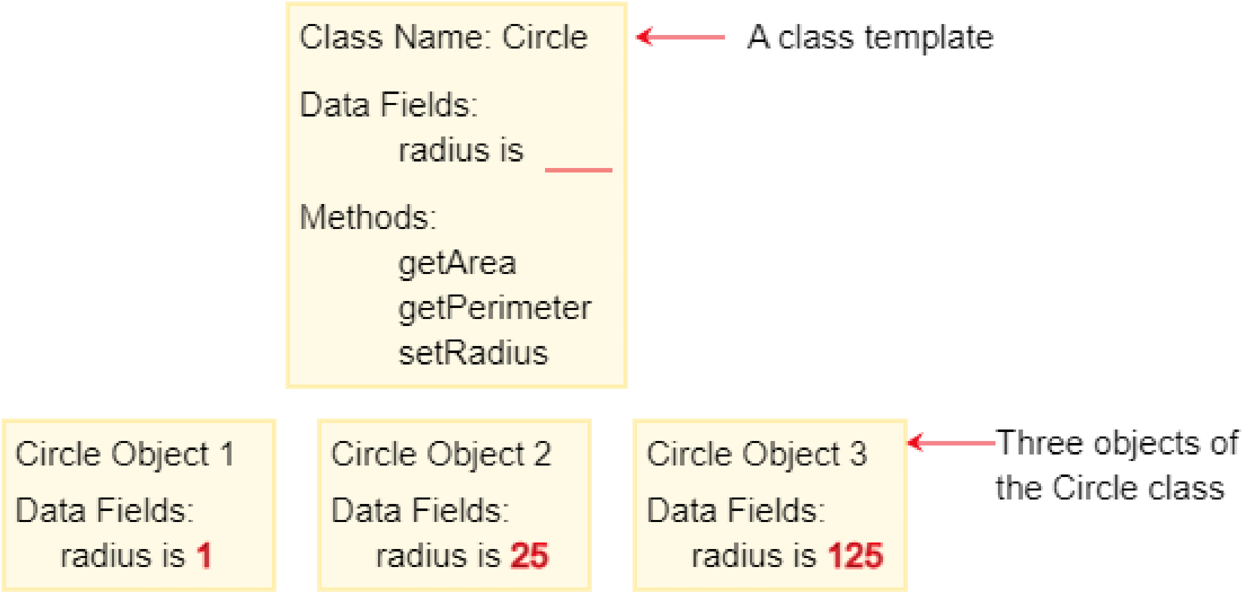
Checking Account

some extra funcitionality unique to a checking account

Saving Account

some extra funcitionality unique to a savings account

Objects



Class Name: Circle

Data Fields:
radius is ___

Methods:
getArea
getPerimeter
setRadius

A class template

Circle Object 1
Data Fields:
radius is **1**

Circle Object 2
Data Fields:
radius is **25**

Circle Object 3
Data Fields:
radius is **125**

Three objects of the Circle class

An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Classes

A Python class uses variables to store data fields and defines methods to perform actions. Additionally, a class provides a special type method, known as *initializer*, which is invoked to create a new object. An initializer can perform any action, but initializer is designed to perform initializing actions, such as creating the data fields of objects.

```
class ClassName:
    initializer
    methods
```

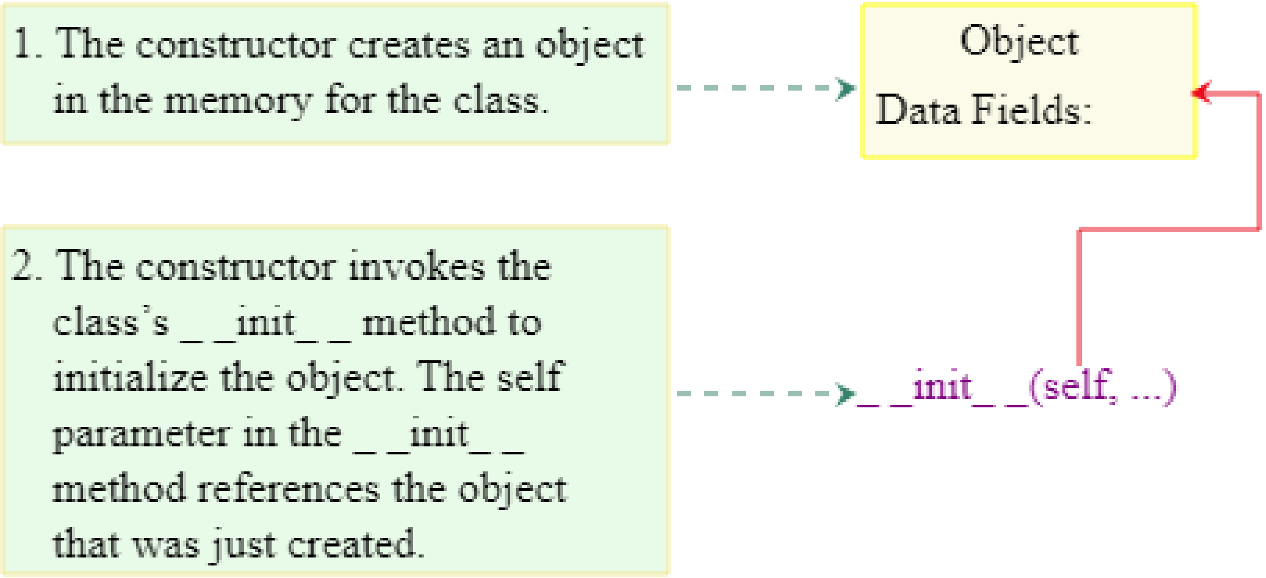
Circle

TestCircle

Constructing Objects

Once a class is defined, you can create objects from the class by using the following syntax, called a *constructor*:

```
an_object = ClassName(arguments)
```



1. The constructor creates an object in the memory for the class.

2. The constructor invokes the class's __init__ method to initialize the object. The self parameter in the __init__ method references the object that was just created.

Object

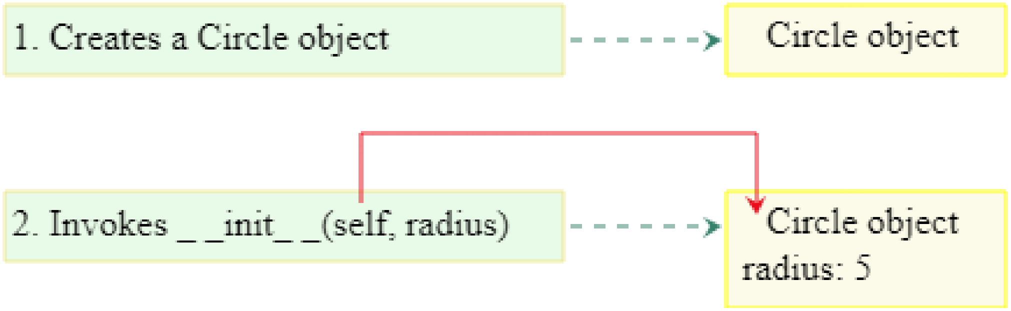
Data Fields:

__init__(self, ...)

Constructing Objects

The effect of constructing a Circle object using Circle(5) is shown below:

Figure 9.4



1. Creates a Circle object

Circle object

2. Invokes __init__(self, radius)

Circle object
radius: 5

A circle object is constructed using `Circle(5)`.

Instance Methods

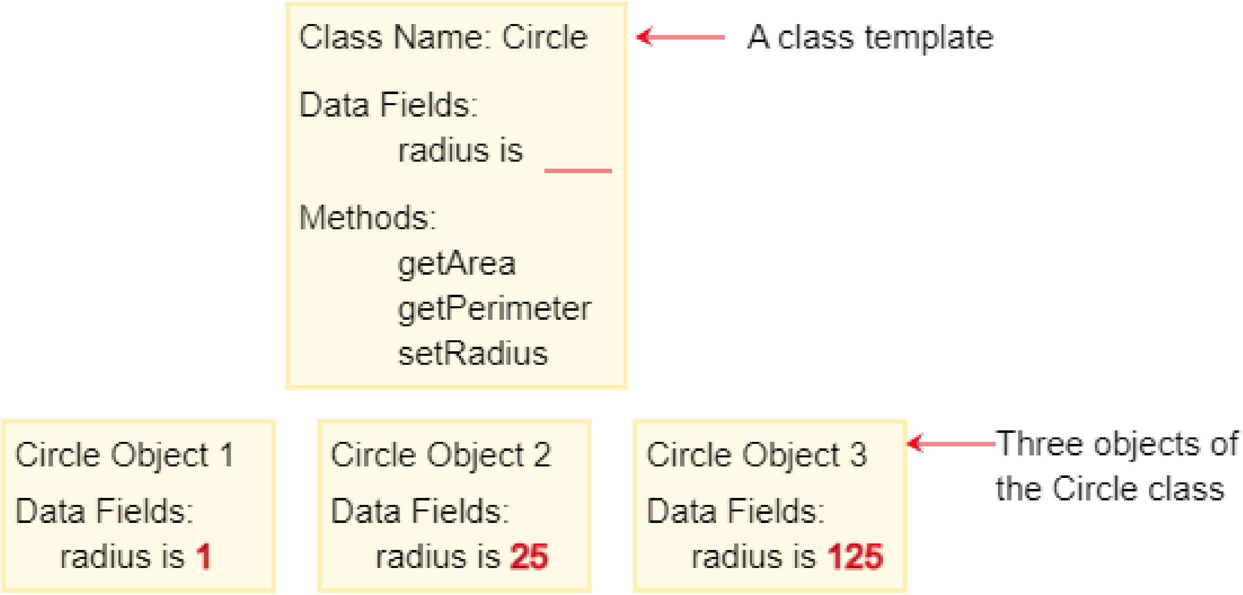
Methods are functions defined inside a class. They are invoked by objects to perform actions on the objects. For this reason, the methods are also called *instance methods* in Python. You probably noticed that all the methods including the __init__ have the first parameter **self**, which refers to the object that invokes the method. You can use any name for this parameter. But by convention, **self** is used.

Why self?

Note that the first parameter is special. It is used in the implementation of the method, *but not used when the method is called*. So, what is this parameter `self` for? Why does Python need it?

- `self` is a parameter that represents an object
- Using `self`, you can access instance variables in an object
- Instance variables are for storing data fields
- Each object is an instance of a class
- Instance variables are tied to specific objects
- Each object has its own instance variables
- You can use the syntax `self.x` to access the instance variable `x` for the object `self` in a method.

Objektene har felles kode



Class Name: Circle

Data Fields:
radius is ___

Methods:
getArea
getPerimeter
setRadius

A class template

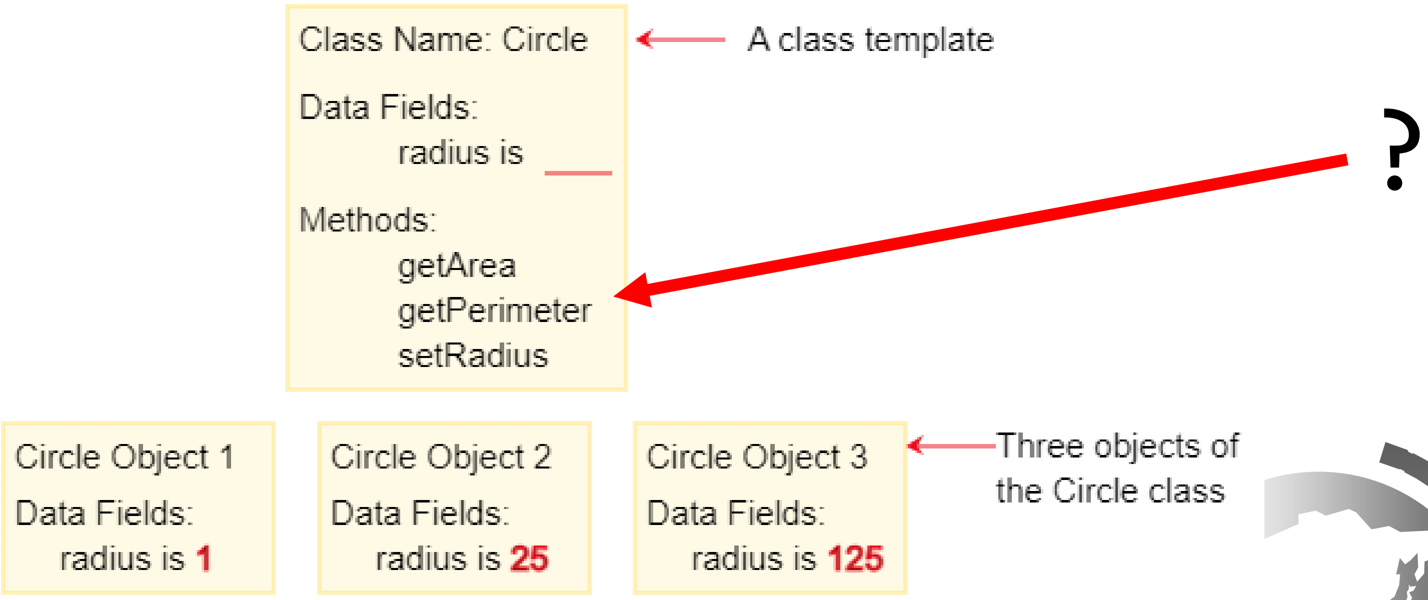
Circle Object 1
Data Fields:
radius is **1**

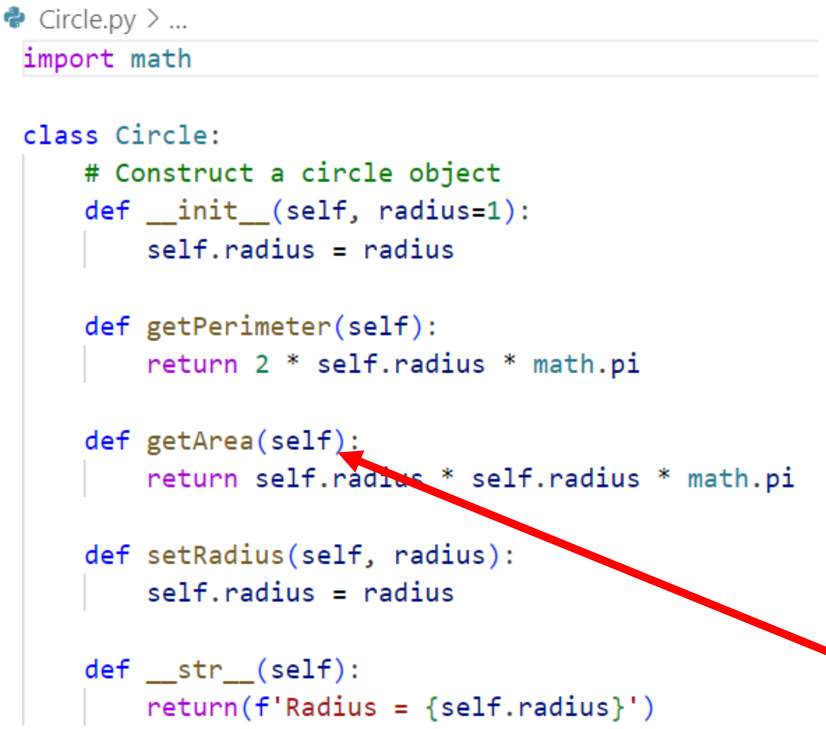
Circle Object 2
Data Fields:
radius is **25**

Circle Object 3
Data Fields:
radius is **125**

Three objects of the Circle class

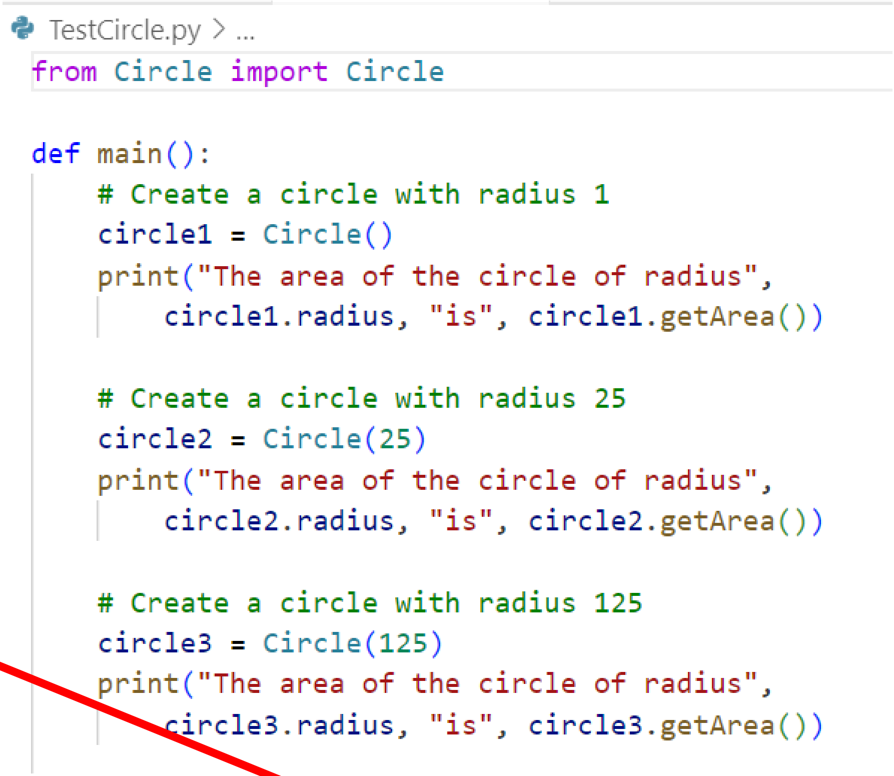
Hvordan vet koden hvilket objekt det jobber på?





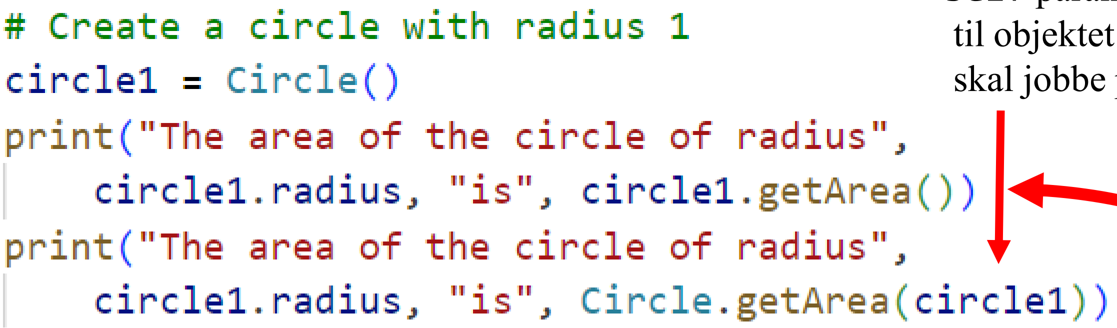
Circle.py > ...

```python
import math

class Circle:
    # Construct a circle object
    def __init__(self, radius=1):
        self.radius = radius

    def getPerimeter(self):
        return 2 * self.radius * math.pi

    def getArea(self):
        return self.radius * self.radius * math.pi

    def setRadius(self, radius):
        self.radius = radius

    def __str__(self):
        return(f'Radius = {self.radius}')
```



TestCircle.py > ...

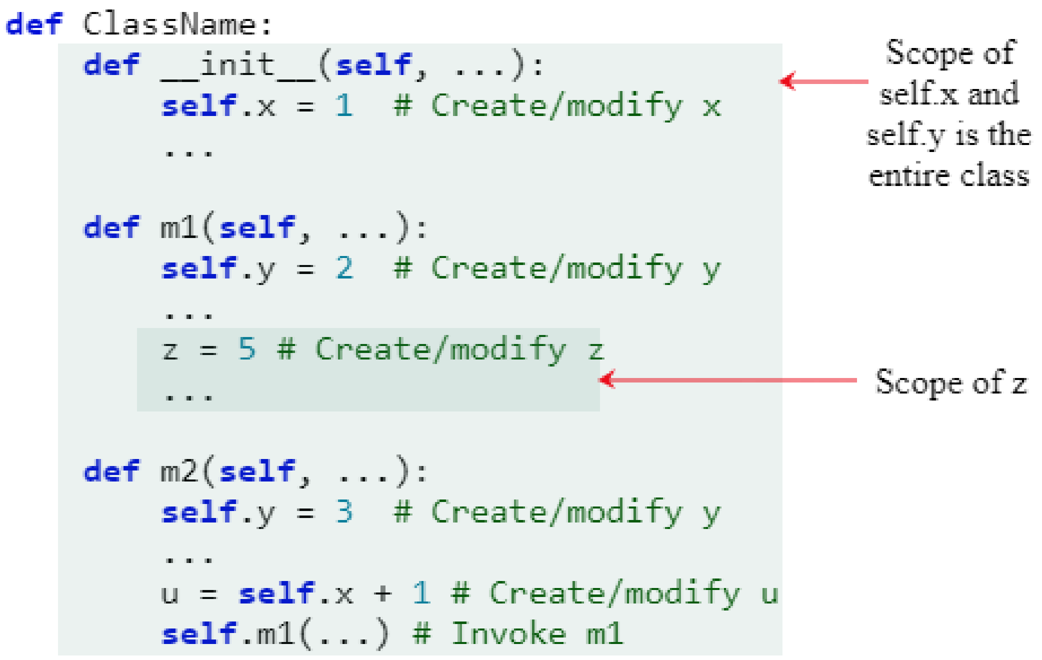
```python
from Circle import Circle

def main():
    # Create a circle with radius 1
    circle1 = Circle()
    print("The area of the circle of radius",
        circle1.radius, "is", circle1.getArea())

    # Create a circle with radius 25
    circle2 = Circle(25)
    print("The area of the circle of radius",
        circle2.radius, "is", circle2.getArea())

    # Create a circle with radius 125
    circle3 = Circle(125)
    print("The area of the circle of radius",
        circle3.radius, "is", circle3.getArea())
```



```python
# Create a circle with radius 1
circle1 = Circle()
print("The area of the circle of radius",
    circle1.radius, "is", circle1.getArea())
print("The area of the circle of radius",
    circle1.radius, "is", Circle.getArea(circle1))
```

`self` parameteren er referansen til objektet som koden skal jobbe på

Figure 9.5



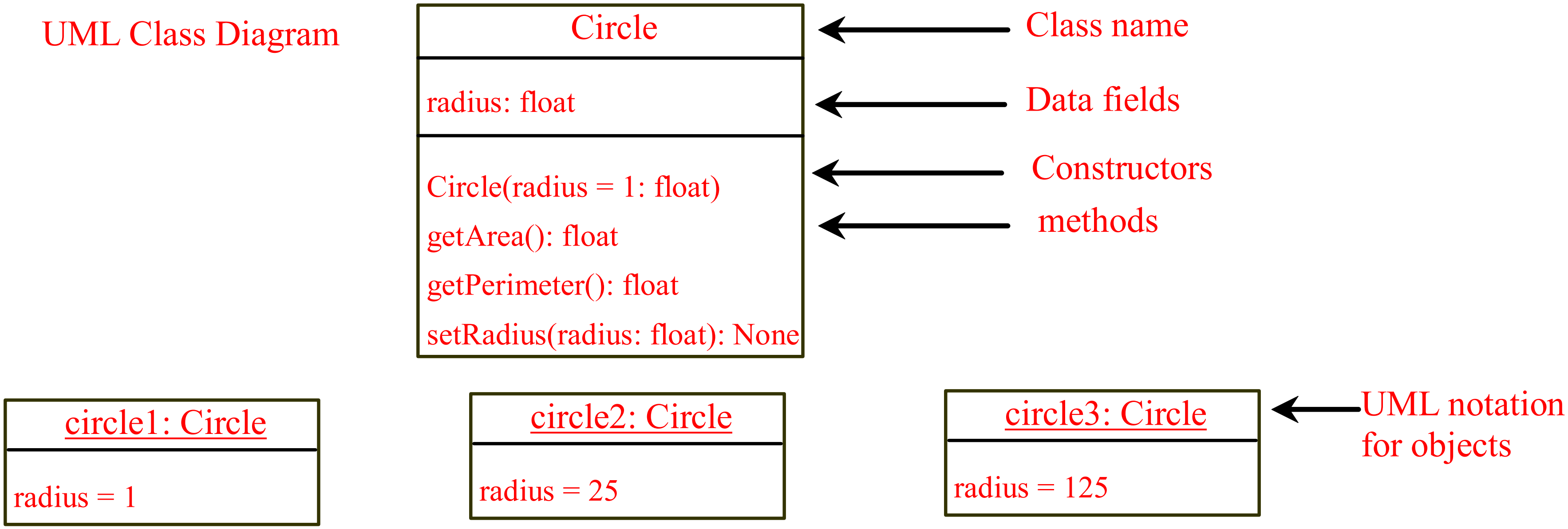
```
def ClassName:
    def __init__(self, ...):
        self.x = 1  # Create/modify x
        ...

    def m1(self, ...):
        self.y = 2  # Create/modify y
        ...
        z = 5 # Create/modify z
        ...

    def m2(self, ...):
        self.y = 3  # Create/modify y
        ...
        u = self.x + 1 # Create/modify u
        self.m1(...) # Invoke m1
```

Scope of self.x and self.y is the entire class

Scope of z

The scope of an instance variable is the entire class.

UML Class Diagram



UML Class Diagram

Circle

radius: float

Circle(radius = 1: float)

getArea(): float

getPerimeter(): float

setRadius(radius: float): None

Class name

Data fields

Constructors

methods

circle1: Circle

radius = 1

circle2: Circle

radius = 25

circle3: Circle

radius = 125

UML notation for objects

Example: Defining Classes and Creating Objects

TV	
channel: int	The current channel (1 to 120) of this TV.
volumeLevel: int	The current volume level (1 to 7) of this TV.
on: bool	Indicates whether this TV is on/off.
TV()	Constructs a default TV object.
turnOn(): None	Turns on this TV.
turnOff(): None	Turns off this TV.
getChannel(): int	Returns the channel for this TV.
setChannel(channel: int): None	Sets a new channel for this TV.
getVolume(): int	Gets the volume level for this TV.
setVolume(volumeLevel: int): None	Sets a new volume level for this TV.
channelUp(): None	Increases the channel number by 1.
channelDown(): None	Decreases the channel number by 1.
volumeUp(): None	Increases the volume level by 1.
volumeDown(): None	Decreases the volume level by 1.

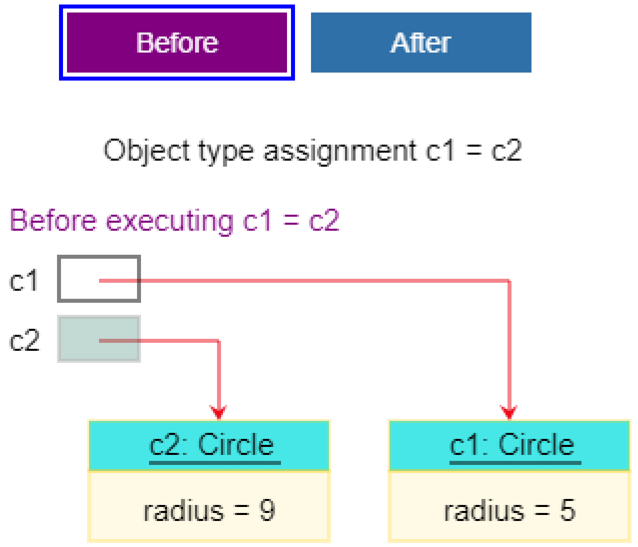
TV

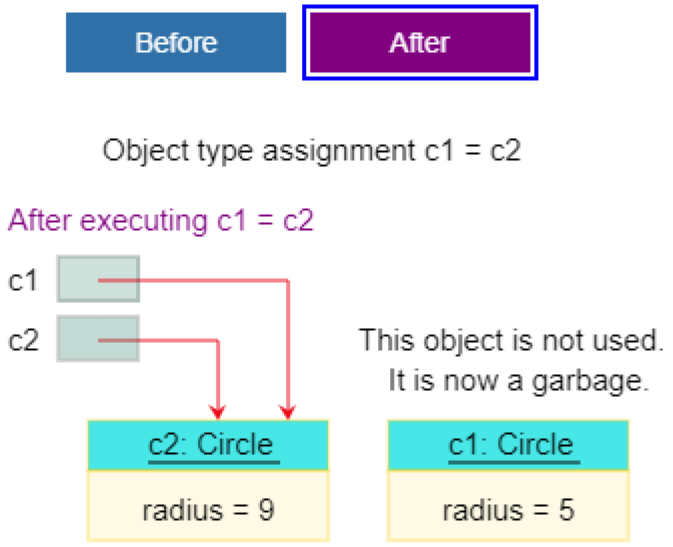
TestTV

Immutable Objects vs. Mutable Objects

TestPassMutableObject

Suppose `c1` and `c2` are two `Circle` objects. Can you copy the contents of `c2` to `c1` using `c1 = c2`? No. This assignment statement makes `c1` reference the same object as referenced by `c2`, as illustrated in **Figure 9.6**.





The datetime Class

```
from datetime import datetime
d = datetime.now()
print("Current year is " + str(d.year))
print("Current month is " + str(d.month))
print("Current day of month is " + str(d.day))
print("Current hour is " + str(d.hour))
print("Current minute is " + str(d.minute))
print("Current second is " + str(d.second))
```

Data Field Encapsulation

To protect data.

To make class easy to maintain.

To prevent direct modifications of data fields, don’t let the client directly access data fields. This is known as *data field encapsulation*. This can be done by defining private data fields. In Python, the private data fields are defined with two leading underscores. You can also define a private method named with two leading underscores.

CircleWithPrivateRadius

Data Field Encapsulation

CircleWithPrivateRadius

```
>>> from CircleWithPrivateRadius import Circle
>>> c = Circle(5)
>>> c.__radius = 2
AttributeError: 'Circle' object has no attribute
'__radius'
>>> c.getRadius()
5
```

«private» datamedlemmer

- ***en*** underscore i navnet
- ikke slik:

  ```
  self.__radius = radius
  ```

- men slik:

  ```
  self._radius = radius
  ```

Python har ikke private attributter eller metoder

- En underscore etter «.» er kun en konvensjon, du kan alltid få tak i / endre datamedlemmet fra klientkode
- For å «simulere» private datamedlemmer lager vi *properties*

..altså ikke slik

```
class Circle:
    # Construct a circle object
    def __init__(self, radius=1):
        self.__radius = radius

    def get_perimeter(self):
        return 2 * self.__radius * math.pi

    def get_area(self):
        return self.__radius * self.__radius * math.pi

    def set_radius(self, radius):
        self.__radius = radius

    def get_radius(self):
        return self.__radius
```

..men slik

```
class Circle:
    def __init__(self, radius: float = 1):
        self._radius = radius

    def get_perimeter(self) -> float:
        return 2 * self._radius * math.pi

    def get_area(self) -> float:
        return self._radius * self._radius * math.pi


    @property
    def radius(self) -> float:
        return self._radius

    @radius.setter
    def radius(self, value: float):
        if value > 0:
            self._radius = value
```

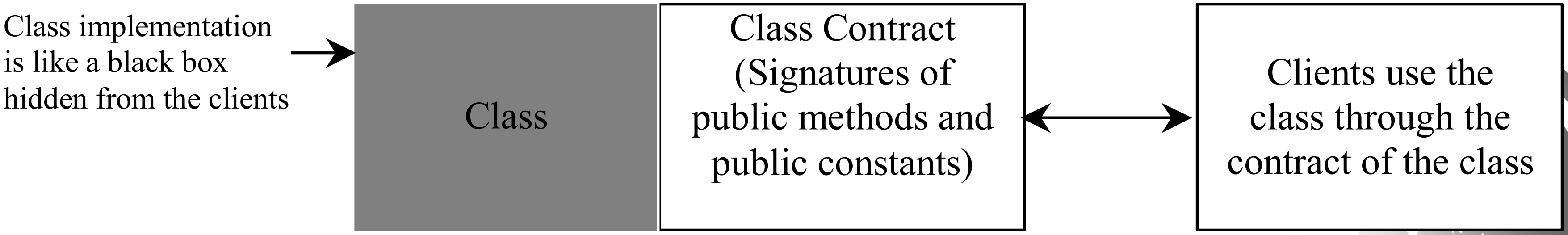
```
def main():
    circle = Circle(5)
    print(circle.radius)
    circle.radius = 4
    print(circle.radius)
```

Properties

- Ikke lag hvis du ikke behøver å endre / lese ut verdi av attributt
- Lag hvis du har behov for å sette en attributt, spesielt der det skal legges til logikk

Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.

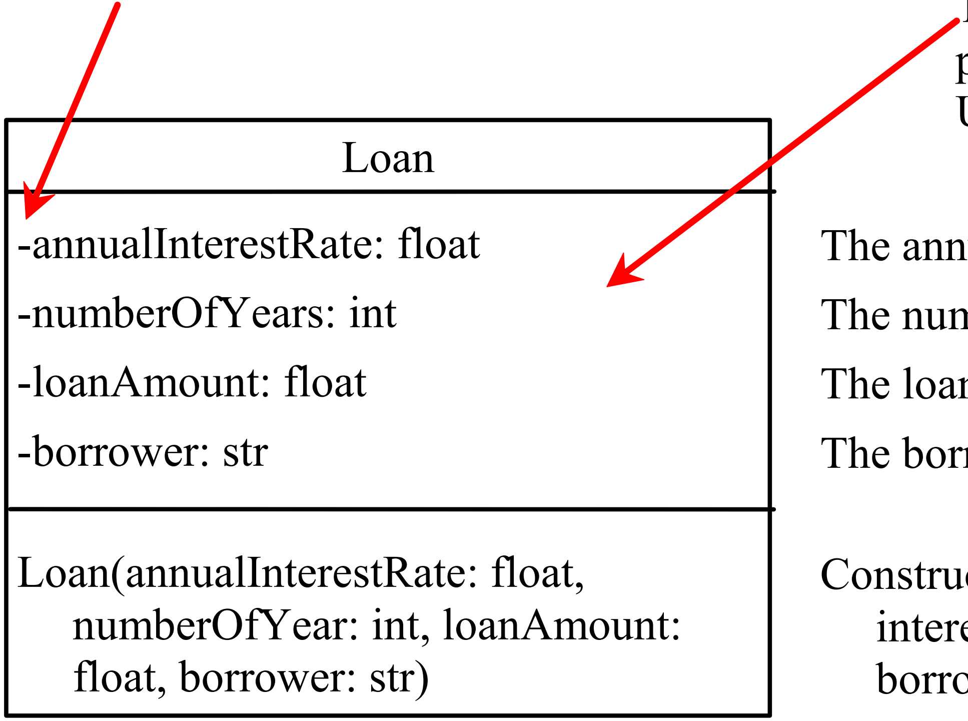

Class implementation is like a black box hidden from the clients
Class
Class Contract (Signatures of public methods and public constants)
Clients use the class through the contract of the class

Listing 2.8, ComputeLoan.py, presented a program for computing loan payments. The program as it is currently written cannot be reused in other programs. One way to fix this problem is to define functions for computing monthly payment and total payment. However, this solution has limitations. Suppose you wish to associate a borrower with the loan. There is no good way to tie a borrower with a loan without using objects. The traditional procedural programming paradigm is action-driven; data are separated from actions. The object-oriented programming paradigm focuses on objects, so actions are defined along with the data in objects. To tie a borrower with a loan, you can define a loan class with borrower along with other properties of the loan as data fields. A loan object would then contain data and actions for manipulating and processing data, with loan data and actions integrated in one object. **Figure 9.11** shows the UML class diagram for the `Loan` class. Note that the – (dash) in the UML class diagram denotes a private data field or method of the class.

Designing the Loan Class

The – sign denotes a private data field.

The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.



Loan

-annualInterestRate: float

-numberOfYears: int

-loanAmount: float

-borrower: str

Loan(annualInterestRate: float, numberOfYear: int, loanAmount: float, borrower: str)

The annual interest rate of the loan (default: 2.5).

The number of years for the loan (default: 1)

The loan amount (default: 1000).

The borrower of this loan.

Constructs a Loan object with the specified annual interest rate, number of years, loan amount, and borrower.

Loan

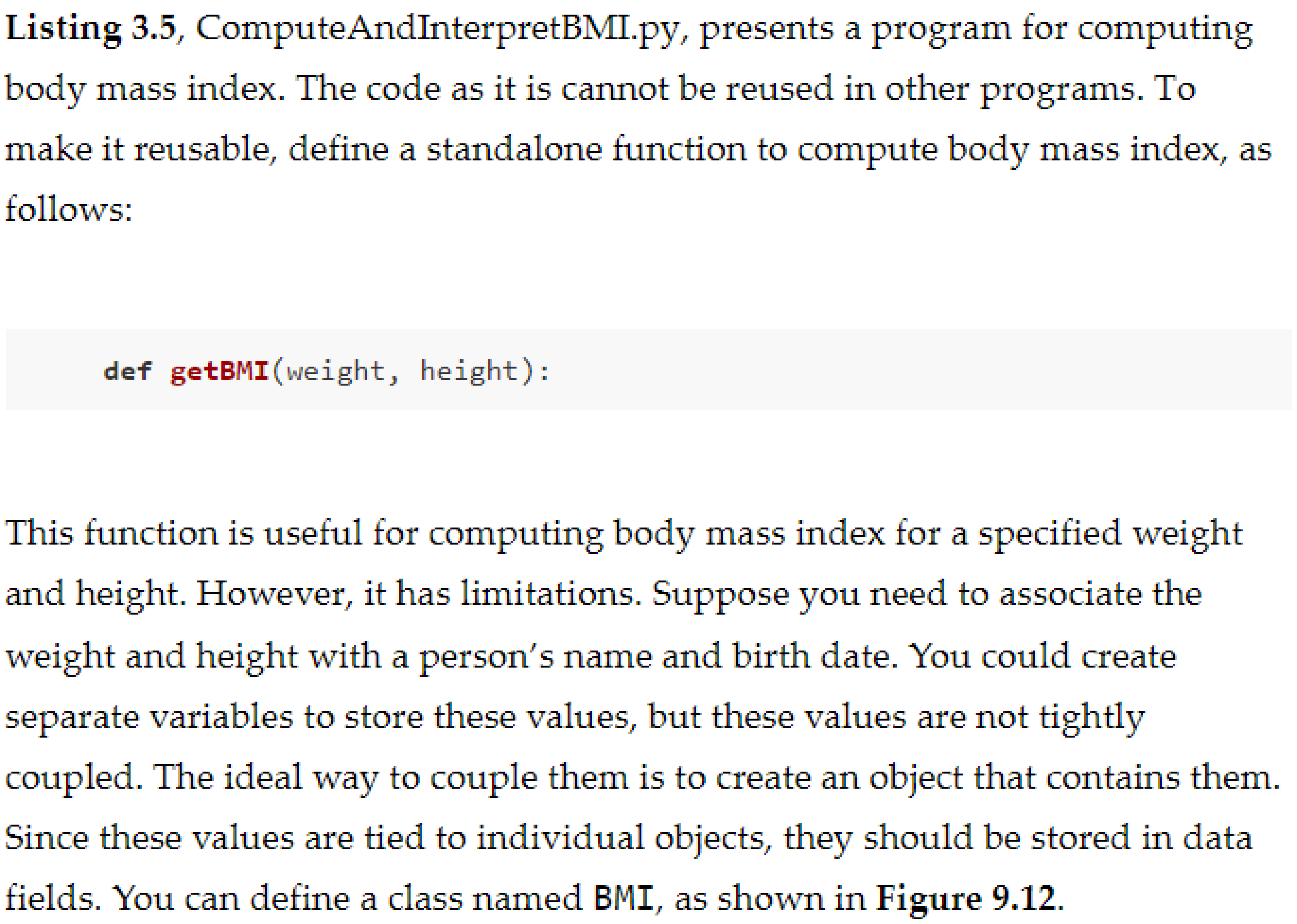
TestLoanClass

9.8 Object-Oriented Thinking

Key Point

The procedural paradigm for programming focuses on designing functions. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects.

This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming. This section shows how procedural and object-oriented programming differ. You will see the benefits of object-oriented programming and learn to use it effectively. We will improve the solution for the BMI problem introduced in **Section 3.8** by using the object-oriented approach. From the improvements, you will gain insight into the differences between procedural and object-oriented programming and see the benefits of developing reusable code using objects and classes.

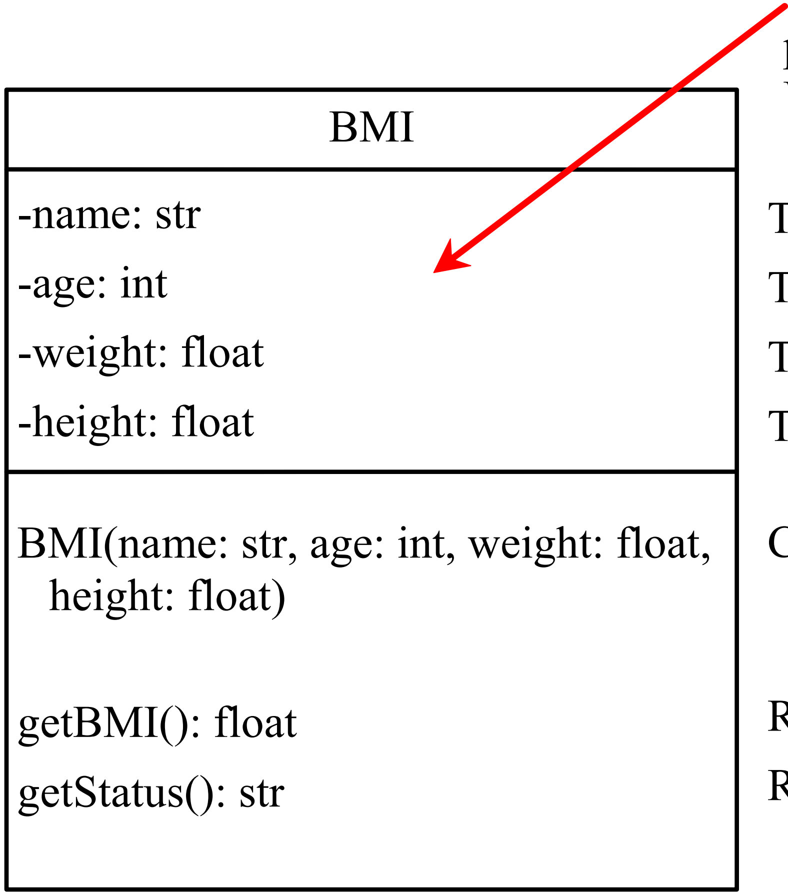


Listing 3.5, ComputeAndInterpretBMI.py, presents a program for computing body mass index. The code as it is cannot be reused in other programs. To make it reusable, define a standalone function to compute body mass index, as follows:

```
    def getBMI(weight, height):
```

This function is useful for computing body mass index for a specified weight and height. However, it has limitations. Suppose you need to associate the weight and height with a person's name and birth date. You could create separate variables to store these values, but these values are not tightly coupled. The ideal way to couple them is to create an object that contains them. Since these values are tied to individual objects, they should be stored in data fields. You can define a class named `BMI`, as shown in **Figure 9.12**.

The BMI Class



The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

BMI

-name: str — The name of the person.

-age: int — The age of the person.

-weight: float — The weight of the person in pounds.

-height: float — The height of the person in inches.

BMI(name: str, age: int, weight: float, height: float) — Creates a BMI object with the specified name, weight, height, and a default age 20.

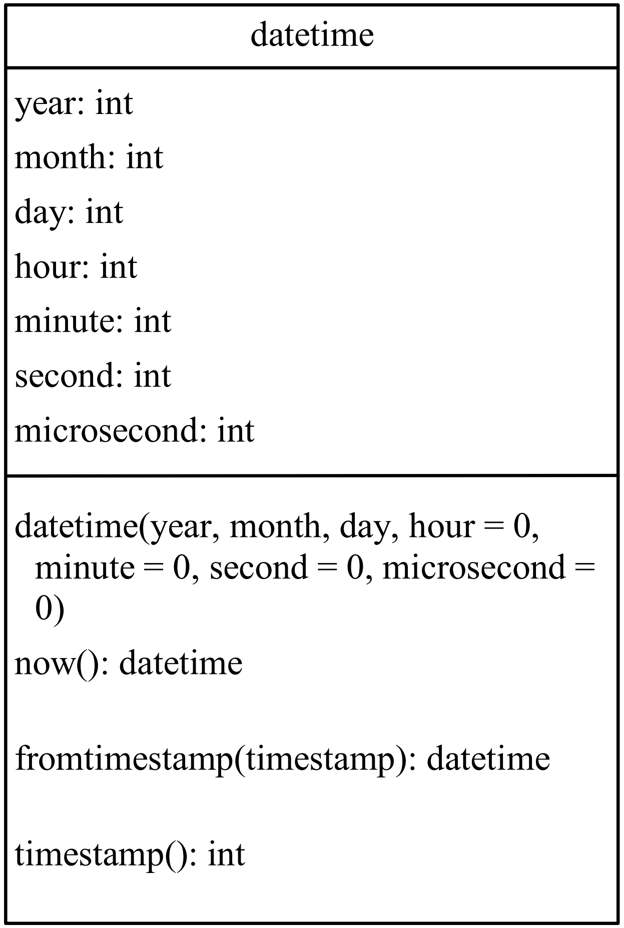
getBMI(): float — Returns the BMI

getStatus(): str — Returns the BMI status (e.g., normal, overweight, etc.)

BMI

UseBMIClass

The datetime Class


datetime

year: int

month: int

day: int

hour: int

minute: int

second: int

microsecond: int

datetime(year, month, day, hour = 0, minute = 0, second = 0, microsecond = 0)

now(): datetime

fromtimestamp(timestamp): datetime

timestamp(): int

The year, month, day, hour, minute, second, and microsecond in this datatime object.

Creates a datetime object using the specified year, month, day, hour, minute, second, and microsecond.

Returns a datetime object for the current time.

Returns a datetime object from the specified timestamp in seconds.

Returns the timestamp in seconds in this datetime object.

DatetimeDemo

9.10 Case Study: The Rational Class

Key Point

This section shows how to design the `Rational` class for representing and processing rational numbers.

A rational number has a numerator and a denominator in the form `a/b`, where `a` is the numerator and `b` is the denominator. For example, `1/3`, `3/4`, and `10/4` are rational numbers.

A rational number cannot have a denominator of `0`, but a numerator of `0` is fine. Every integer `i` is equivalent to a rational number `i/1`. Rational numbers are used in exact computations involving fractions—for example, `1/3` = `0.33333....` This number cannot be precisely represented in floating-point format using data type `float`. To obtain the exact result, we must use rational numbers.

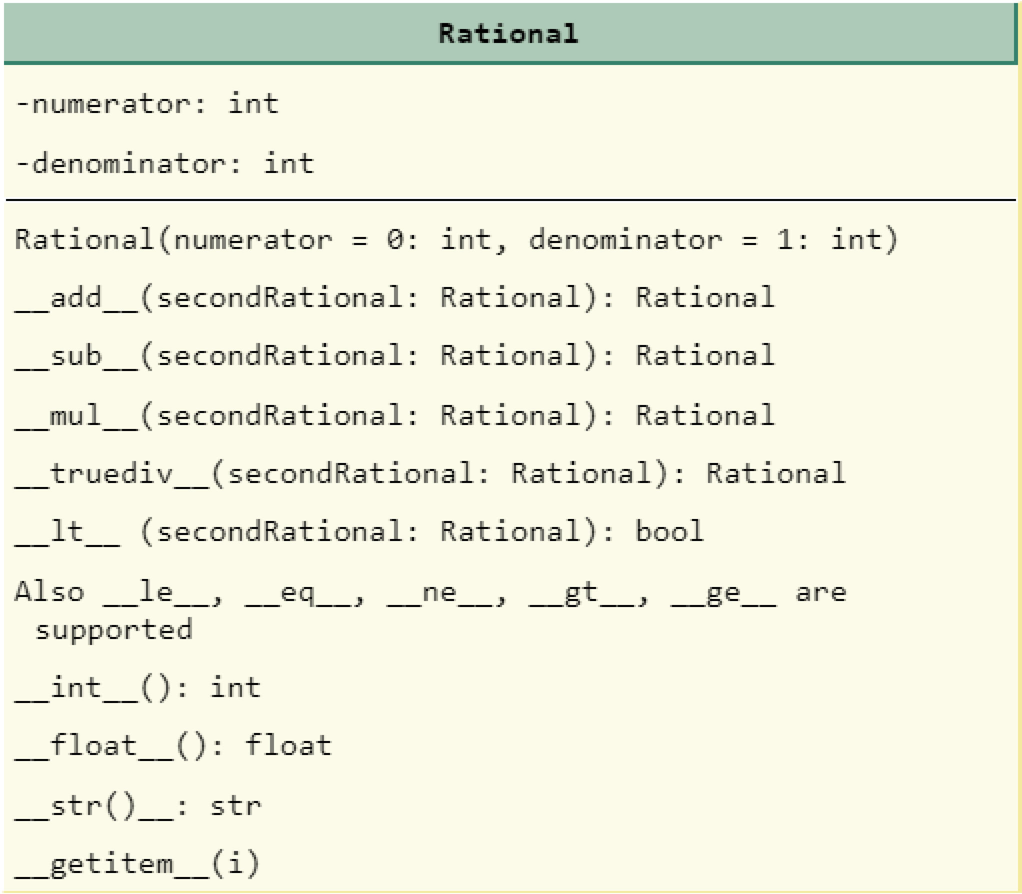
Tallene…

- Naturlige tall
  - Heltall inkl 0 og negative tall
- Rasjonale tall
  - Et heltall (teller/nominator) over et annet (nevner / denominator)
  - Et desimaltall med et endelig antall desimaler er alltid et rasjonalt tall
- Irrasjonale tall
  - Har ikke et endelig antall desimaler (roten av 2, pi)

Python provides data types for integers and floating-point numbers but not for rational numbers. This section shows how to design a class for rational numbers.

A rational number can be represented using two data fields: `numerator` and `denominator`. You can create a rational number with a specified numerator and denominator or create a default rational number with the numerator `0` and denominator `1`. You can add, subtract, multiply, divide, and compare rational numbers. You can also convert a rational number into an integer, floating-point value, or string. The UML class diagram for the `Rational` class is given in **Figure 9.13**.

Figure 9.13



```
Rational
-numerator: int
-denominator: int
Rational(numerator = 0: int, denominator = 1: int)
__add__(secondRational: Rational): Rational
__sub__(secondRational: Rational): Rational
__mul__(secondRational: Rational): Rational
__truediv__(secondRational: Rational): Rational
__lt__ (secondRational: Rational): bool
Also __le__, __eq__, __ne__, __gt__, __ge__ are
 supported
__int__(): int
__float__(): float
__str()__: str
__getitem__(i)
```

The UML diagram for the properties, initializer, and methods of the Rational class.

There are many equivalent rational numbers; for example, `1/3` = `2/6` = `3/9` = `4/12`. For convenience, `1/3` is used to represent all rational numbers that are equivalent to `1/3`. The numerator and the denominator of `1/3` have no common divisor except `1`, so `1/3` is said to be in *lowest terms*.

To reduce a rational number to its lowest terms, you need to find the greatest common divisor (GCD) of the absolute values of its numerator and denominator and then divide both numerator and denominator by this value. You can use the function for computing the GCD of two integers `n` and `d`, as suggested in **Listing 5.8**, GreatestCommonDivisor.py. The numerator and denominator in a `Rational` object are reduced to their lowest terms.

Regneregler rasjonale tall

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

Procedural vs. Object-Oriented

In procedural programming, data and operations on the data are separate, and this methodology requires sending data to methods. Object-oriented programming places data and the operations that pertain to them in an object. This approach solves many of the problems inherent in procedural programming.

Procedural vs. Object-Oriented

The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Python involves thinking in terms of objects; a Python program can be viewed as a collection of cooperating objects.