§3 Object-oriented Programming in Python

Object-oriented programming (OOP) is a programming style that focuses on the data rather than the algorithm. Such programming approach is well-suited for programs that are large, complex, and actively updated. **Simula** is regarded as the first OOP language which was developed by the Norwegian computer scientists Ole-Johan Dahl and Kristen Nygaard in the 1960s. It introduced a number of key concepts of object-oriented programming that are now implemented in the most widely used programming languages such as **C++**, **Java**, and **Python**. In this chapter, we will discuss how to perform object-oriented programming in Python.

3.1 Basics of Object-oriented Programming

So far we have only worked on Python programs that are designed using a programming paradigm known as **procedural programming**. In this approach, a program is broken into small parts using code blocks and functions so that its structure likes a recipe providing a set of steps which are executed sequentially in order to complete a task. Procedural programming is simple, straight forward, and efficient. However, the maintenance of large and complex programs developed by this approach can be difficult and time consuming.

An alternative programming paradigm is object-oriented programming which is based on the concepts of **objects** that models real-world entities. An object is simply a collection of data and functions that act on those data. A **class** is a "blueprint" from which individual objects are made. It specifies the properties and behaviors that an object should have. Classes are defined with two types of **attributes**: **data attributes** and **methods**. Data attributes refer to data in classes while methods refer to functions in classes. We can create multiple objects from the same class. Each object is unique and it is an **instance** of a class. The process of creating an object from a class is called **instantiation**.

As we have already pointed out, everything in Python is an object. For example, a Python string is an instance of the **`str`** class. A **`str`** object possesses its own data (the sequence of characters making up the string) and provides a number of methods for manipulating those data. For instance, the `upper` method returns a new string object created from the original string by changing all letters to uppercase and the `split` method returns a list of strings by splitting up the original string:

```
>>> x = "Hello, Peter, goodbye, Mary"
>>> x.upper()
'HELLO, PETER, GOODBYE, MARY'
>>> x.split()
['Hello,', 'Peter,', 'goodbye,', 'Mary']
```

Even indexing a string is actually a call to the method `__getitem__`:

```
>>> y = [2, 4, 6, 8, 10, 12]
>>> y.__getitem__(5)
12
```

In other words, `y[5]` is equivalent to `y.__getitem__(5)`.

Part of the popularity of object-oriented programming, at least for larger projects, is due to the way helping us to conceptualize the problem that a program aims to solve. It is often possible to break a problem down into units of data and operations that are appropriate to carry out on that data. For example, a retail bank deals with people who have bank accounts. A natural object-oriented approach for managing a bank would be to define a `BankAccount` class with data such as an account number, balance, and owner and a `Customer` class with data such as a name, address, and date of birth. The `BankAccount` class might have methods for allowing or forbidding transactions depending on its balance and the `Customer` class might have methods for calculating the customer's age from their date of birth as shown in Figure 3.1.

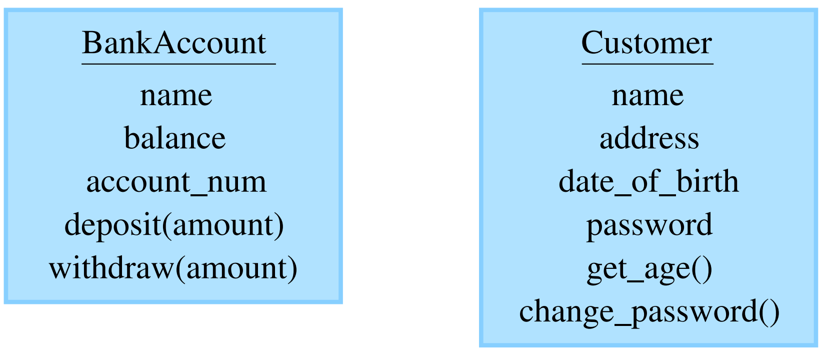

Figure 3.1: Classes representing a bank account and a customer.

An important aspect of object-oriented programming is **inheritance**. The idea behind inheritance is that a new class can be defined to take over the properties and behaviors from an existing class. The existing class is called a **superclass** or **base class** and the new class is called its **subclass** or **derived class**. Let's consider our bank example again to illustrate how inheritance works. In a retail bank, there may be different kinds of bank accounts: savings accounts, current accounts, and so on. Each one is derived from a generic bank account which can be represented by a superclass that defines basic attributes such as a balance and an account number. The specialized bank accounts can be represented by subclasses that **inherit** the data attributes and methods of the superclass. These subclasses may also customize the superclass attributes by overriding the methods of the superclass as well as add their own data attributes and methods. Inheritance helps structure the program and encourages **code reuse**, e. g. there is no need to declare an account number separately for the specialized bank account classes since they all inherit one automatically from the superclass. If a superclass is not to be instantiated itself, but serves only as a template for the subclasses, then it is called an **abstract class**. Through inheritance, related classes can be put together in families so that each family can be viewed as one unit. A family of classes is known as a **class hierarchy**.

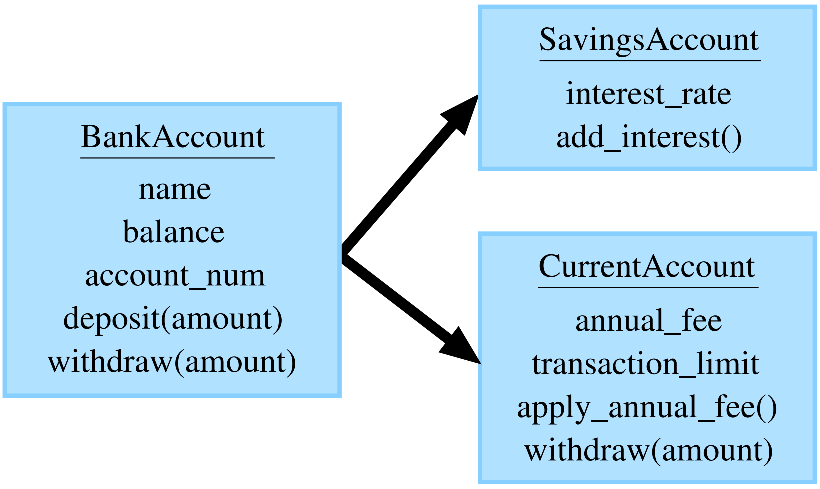

Figure 3.2: Two subclasses derived from an abstract superclass: `SavingsAccount` and `CurrentAccount`. They inherit data and methods from `BankAccount` but also customize and extend its functionality.

Figure 3.2 depicts the relationship between the superclass and two subclasses for our bank example. The superclass `BankAccount` defines some data attributes (`account_number`, `balance`, and `customer`) and methods (such as `deposit` and `withdraw`) which are common to all types of account, and these are inherited by the subclasses. The subclass `SavingsAccount` adds the attributes for handling interest payments on the account while the subclass `CurrentAccount` adds the attributes describing the annual account fee and transaction withdrawal limit as well as overrides the method `withdraw` defined in the superclass.

3.2 Creating Classes in Python

3.2.1 Class Definition

A Python class is defined using the **class** keyword and indenting the body of statements in a block following this declaration. The definition of a simple class looks like this:

```
class <ClassName>:
    <method1>
    <method2>
        ⋮
    <methodN>
```

where <*ClassName*> is the class name and <*method1*>, <*method2*>, ... , <*methodN*> are definitions of methods of the class. Like function definitions, class definitions must be executed before they have any effect. By convention, classes names are written in CamelCase, i. e. a compound word which uses capital letters to delimit the word parts. It's a good idea to follow the **class** statement by a docstring describing what does the class do. Methods are defined in a class using the **def** keyword in the way like a normal function definition. Each method has a first parameter named `self` which refers to the object on which the method is acting. Actually, we can use any name for this parameter. But the conventional name is `self` which will be always used here.

In Python, the data attributes defined within methods of a class are known as **instance variables** that are owned by instances of the class. Thus instance variables can take on different values for different instances of a class. Just like regular variables, instance variables are accessed by name. To access instance variables outside the class, we use the dot notation <*object*>.<*ivar*> where <*object*> is the name of the instance and <*ivar*> is the name of the instance variable. The power of instance variables is that we can use them to remember the state of a particular object of a class, and this information then gets passed around the program as part of the object. The values of instance variables can be referred to again in other methods or even in successive calls to the same method. This is different from regular local function variables, whose values disappear once the function terminates.

Some methods of Python classes have names starting and ending with a double underscore. These methods allow a special syntax in the program and are called **special methods**. The **constructor** `__init__` is a typical example. Python calls this method to create a new instance of a class. A call to a constructor is an expression that creates a brand new object. It has the general syntax

```
<ClassName>(<arg1>, <arg2>, ..., <argN>)
```

where <*ClassName*> is the name of the class from which a new instance is created and <*arg1*>, <*arg2*>, ..., <*argN*> are the arguments for initializing the data attributes of the instance. Note that the arguments are always passed on as parameters to the constructor __init__ after the self parameter. Inside the constructor, the argument self is a variable holding the new instance to be constructed. After the body of the constructor is executed, the self parameter is invisibly returned to the calling code. By convention, the constructor is used for initializing the instances variables in the class. To access instance variables inside the constructor, we use the dot notation self.<*ivar*> where <*ivar*> is the name of the instance variable.

Next we consider methods in Python classes apart from special methods. These methods are invoked using the dot notation as follows:

```
<object>.<methodname>(<arg1>, <arg2>, ..., <argN>)
```

where <*object*> is the name of the instance, <*methodname*> is the name of the method, and <*arg1*>, <*arg2*>, ..., <*argN*> are the arguments for the method call. Note that the self parameter is again dropped in the call of these methods. Just like that inside the constructor, we access an instance variable <*ivar*> inside these methods using the dot notation self.<*ivar*>.

To illustrate how a class works, let's consider the superclass BankAccount in our previous bank example. This class could be defined in the program bankaccount.py like this:

```
# backaccount.py
# This program defines the abstract superclass BankAccount which
# represents a generic bank account.
# Last Update on 19 Nov 2020 by F K Chow

class BankAccount:
    """ An abstract superclass representing a generic bank account """

    def __init__(self, name, account_num, opening_balance=0):
        """ Initialize the bank account with a name, an account
            number, and an opening balance whose default is 0 """
```

```
        self.name = name
        self.account_num = account_num
        self.balance = opening_balance

    def deposit(self, amount):
        """ Deposit amount into the bank account """
        if amount > 0:
            self.balance += amount
        else:
            print("Invalid deposit amount:", amount)

    def withdraw(self, amount):
        """ Withdraw amount from the bank account if there are
            sufficient funds """
        if amount <= 0:
            print("Invalid withdrawal amount:", amount)
        else:
            if amount > self.balance:
                print("Insufficient funds")
            else:
                self.balance -= amount

    def print_balance(self):
        """ Print a statement of the account balance """
        print("The balance of account number {0:d} is ${1:.2f}."
              .format(self.account_num, self.balance))
```

To use this simple class, we import the program `bankaccount.py` into a new program or the interactive Python shell using the statement:

```
from bankaccount import BankAccount
```

This program can now create `BankAccount` objects and manipulate them by calling the methods described in the previous section.

The `BankAccount` class possesses three instance variables: `name`, `account_num`, and `balance`. These variables are all initialized in the constructor so that a `BankAccount` object is created with the name of the owner, an account number, and an optional opening balance (which defaults to 0 if it's not provided) as follows:

```
>>> acc1 = BankAccount("Peter Jackson", 246810)
```

Recall that the `self` parameter is invisibly returned to the calling code. So Python automatically translates the right hand side of the above statement to

```
BankAccount.__init__(acc1, "Peter Jackson", 246810)
```

In addition to the constructor `__init__`, the `BankAccount` class defines three methods: `deposit`, `withdraw`, and `print_balance`. The first two methods are used for depositing a positive amount of money and for withdrawing money if the amount to be withdrawn is both positive and not greater than the account balance. And the third method is used for printing the balance of the account. Here are some examples for using these methods:

```
>>> acc2 = BankAccount("Julie Anderson", 135791, 80)
>>> acc2.name
'Julie Anderson'
>>> acc2.account_num
135791
>>> acc2.deposit(120)
>>> acc2.balance
200
>>> acc2.print_balance()
The balance of account number 135791 is $200.00.
```

We can also use classes to solve problems in mathematical and physical sciences. For example, consider the class `Projectile` which simulates a projectile moving near the Earth's surface under negligible air resistance. Below is the Python program `projectile.py` which contains the definition of this class.

```
# projectile.py
# This program defines the class Projectile which simulates a
# projectile moving near the Earth's surface under negligible air
# resistance.
# Last Update on 19 Nov 2020 by F K Chow

class Projectile:
```

```python
    """ A class simulating a projectile moving near the Earth's
        surface under negligible air resistance """

    def __init__(self, theta, u, h):
        """ Initialize the projectile with the launching angle theta,
            the initial speed u, and the initial height h """
        import math
        self._xpos = 0
        self._ypos = h
        thetarad = math.radians(theta)
        self._xvel = u*math.cos(thetarad)
        self._yvel = u*math.sin(thetarad)

    def _getX(self):
        """ Return the x position of the projectile """
        return self._xpos

    def _getY(self):
        """ Return the y position of the projectile """
        return self._ypos

    def _update(self, delt):
        """ Update the status of the projectile to move it delt
            seconds farther into its flight """
        self._xpos = self._xpos + self._xvel*delt
        self._ypos = self._ypos + self._yvel*delt - 4.90*delt**2
        self._yvel = self._yvel - 9.80*delt

    def printrange(self, delt):
        """ Print the horizontal range and time of flight of the
            projectile """
        time = 0
        while self._getY() >= 0:
            self._update(delt)
            time = time + delt
```

```
        print("Horizontal range of the projectile = {0:0.2f} m".
              format(self._getX()))
        print("Time of flight of the projectile = {0:0.2f} s".
              format(time))
```

The `Projectile` class has four data attributes: `_xpos`, `_ypos`, `_xvel`, and `_yvel`. They all start with an underscore to indicate that they should never be accessed outside the class. Unlike other languages, Python does not have special keywords that restrict access to any data attribute or method. All attributes in a Python class can be accessed outside the class environment. By convention, any name starting with an underscore in Python represents an attribute that should be used freely only inside the methods of the class. However, this convention does not change how the attribute can be accessed.

The `Projectile` class also defines five methods including the constructor `__init__`. The methods `_getX` and `_getY` give the current x and y positions of the projectile. And the method `_update` updates the status of the projectile to account for the passage of the time interval `delt`. These three methods start with an underscore to indicate that they should never be called outside the class. On the other hand, `printrange` is a method that can be accessed outside the class. It prints the horizontal range and time of flight of the projectile. Here are the examples for working on this class:

```
>>> from projectile import Projectile
>>> ball = Projectile(30, 50, 100)
>>> ball.printrange(0.01)
Horizontal range of the projectile = 335.15 m
Time of flight of the projectile = 7.74 s
```

3.2.2 Special Methods

Besides the constructor `__init__`, Python also provides other special methods for calling instances as we call ordinary functions, performing arithmetic operations with instances, comparing instances with comparison operators, and so on. Here let us study these methods one by one in details.

Suppose a mathematical function y of time t is represented by the Python class `Y`. Moreover, it defines a method `value` so that writing `y.value(t)` gives the value of the function at time `t` for the instance `y`. If we could instead write `y(t)`, the instance `y` would look like an ordinary function. Such a syntax is indeed possible and offered by the special method named `__call__`. Writing `y(t)` implies a call

```
y.__call__(t)
```

if the method `__call__` is defined in the corresponding class `Y`. For instance, the following code defines the class `Y` representing the mathematical function $y(t)$ that gives the displacement of a free falling object.

```
class Y:
    """ A class representing the mathematical function for the
        displacement of a free falling object """

    def __init__(self, u):
        """ Initialize the function with the initial speed u and the
            acceleration due to gravity g """
        self.u = u
        self.g = 9.80

    def __call__(self, t):
        """ Compute the displacement of the object at time t """
        return self.u*t - 0.5*self.g*t**2

    def formula(self):
        """ Return the formula for the function """
        return "u*t - 0.5*g*t**2; u={:g}".format(self.u)
```

A good programming convention is to include a `__call__` method in all classes that represent a mathematical function. Instances with `__call__` methods are said to be **callable objects**, just as plain functions are callable objects as well. The call syntax for callable objects is the same, regardless of whether the object is a function or a class instance. Given an object `a`, the command

```
if callable(a)
```

tests whether `a` is callable, i. e. if `a` is a Python function or an instance with a `__call__` method.

Another useful special method is `__str__`. It is called when a class instance needs to be converted to a string. This occurs when we print an instance. Python will then look into this instance for a `__str__` method, which is supposed to return a string. If such a special method is found, then the returned string is printed; otherwise just the name of the class is printed. Let's illustrate this feature with an example. First we try to print an instance `y` of class `Y` from `freefall.py` which does not have a `__str__` method:

```
>>> print(y)
<__main__.Y object at 0x000001C9FE3B0EF0>
```

The output means that `y` is an instance of class `Y` in the `__main__` module (the main program or the interactive session). It also contains an address telling where the instance `y` is stored in the computer's memory.

If we want the command `print(y)` to print out the instance `y`, then we need to define the `__str__` method in class `Y`:

```
class Y:
    ⋮
    def __str__(self):
        """ Return the output for printing out the function. """
        return "u*t - 0.5*g*t**2; u={:g}".format(self.u)
```

Observe that `__str__` replaces the `formula` method and `__call__` replaces the `value` method. Python programmers with the experience that we now have gained will thus write class `Y` with special methods only:

```
class Y:
    """ A class representing the mathematical function for the
        displacement of a free falling object """

    def __init__(self, u):
        """ Initialize the function with the initial speed u and the
            acceleration due to gravity g """
        self.u = u
        self.g = 9.80

    def __call__(self, t):
        """ Compute the displacement of the object at time t """
        return self.u*t - 0.5*self.g*t**2

    def __str__(self):
```

```
        """ Return the output for printing out the function """
        return "u*t - 0.5*g*t**2; u={:g}".format(self.u)
```

Let us see the class in action:

```
>>> y = Y(5)
>>> y(0.4)
1.2159999999999997
>>> print(y)
u*t - 0.5*g*t**2; u=5
```

What have we gained by using special methods? Of course, we can still simply evaluate the formula and write it out. But many users will claim that the use of special methods makes the syntax more attractive since `y(t)` in code means $y(t)$ in mathematics and we can use `print(y)` to view the formula. The bottom line of using special methods is to achieve a more user-friendly syntax. The next example illustrate this point further.

Let `Person` be the Python class representing the personal data in a phone book. This class can be implemented as follows:

```
class Person:
    """ A class representing the record of personal data in a phone
        book """

    def __init__(self, name, mobile_phone=None, office_phone=None,
                 private_phone=None, email=None):
        """ Initialize the record with a name, a mobile phone
            number, a office phone number, a private phone number,
            and an email address """
        self.name = name
        self.mobile = mobile_phone
        self.office = office_phone
        self.private = private_phone
        self.email = email

    def add_mobile_phone(self, number):
        """ Add mobile phone number to the record """
        self.mobile = number
```

```
    def add_office_phone(self, number):
        """ Add office phone number to the record """
        self.office = number

    def add_private_phone(self, number):
        """ Add private phone number to the record """
        self.private = number

    def add_email(self, address):
        """ Add email address to the record """
        self.email = address

    def __str__(self):
        """ Return the output for printing out the record """
        s = self.name + "\n"
        if self.mobile is not None:
            s += "mobile phone: {0:s}\n".format(self.mobile)
        if self.office is not None:
            s += "office phone: {0:s}\n".format(self.office)
        if self.private is not None:
            s += "private phone: {0:s}\n".format(self.private)
        if self.email is not None:
            s += "email address: {0:s}\n".format(self.email)
        return s
```

Storing the instances of class `Person` in a dictionary to form a phone book is straightforward. But we can make the dictionary a bit easier to use if we wrap a class around it. That is to say, we make a class `PhoneBook` which holds the dictionary as a data attribute. The complete code for this class looks as follows in Python.

```
class PhoneBook:
    """ A class representing a phone book """

    def __init__(self):
        """ Initialize the phone book with an empty dictionary of the
            instances of class Person """
```

```
        self.contacts = {} # dict of Person instances

    def add(self, name, mobile=None, office=None, private=None,
            email=None):
        """ Add a new person to the phone book """
        p = Person(name, mobile, office, private, email)
        self.contacts[name] = p

    def __call__(self, name):
        """ Return the record of name in the phone book """
        return self.contacts[name]

    def __str__(self):
        """ Return the output for printing out the phone book """
        s = ""
        for p in sorted(self.contacts):
            s += str(self.contacts[p]) + "\n"
        s = s[:-1] # Remove the trailing new line character
        return s
```

In this class, the `add` method is used to add a new person to the the phone book. Moreover, the special method `__call__` retrieves a `Person` instance while the special method `__str__` prints the phone book in alphabetic order. The only advantage of the `__call__` method is simpler syntax. For a `PhoneBook` class instance `pb`, we can get data about `name` by calling `pb("name")` rather than accessing the internal dictionary `pb.contact("name")`. Below is a simple test for the `PhoneBook` class with three names:

```
>>> pb = PhoneBook()
>>> pb.add("Ole Olsen", office="767828292",
           email="olsen@somemail.net")
>>> pb.add("Hans Hanson", office="767828283", mobile="995320221")
>>> pb.add("Per Person", mobile="906849781")
>>> print(pb("Per Person"))
Per Person
mobile phone: 906849781
```

```
>>> print(pb)
Hans Hanson
mobile phone: 995320221
office phone: 767828283

Ole Olsen
office phone: 767828292
email address: olsen@somemail.net

Per Person
mobile phone: 906849781
```

Suppose `a` and `b` are two instances of a given Python class. The standard binary arithmetic operations with `a` and `b` are defined by the special methods as shown in Table 3.1. For example, the special method `__add__` can be defined in a class `C` like this:

```
class C:
    ⋮
    def __add__(self, other):
        ⋮
```

The `__add__` method would add the instances `self` and `other` together and return the result as another instance of the class. Whenever Python encounters `a + b` where `a` and `b` are the instances of a class, it will check whether the class has an `__add__` method. If the method is found, `a + b` will be interpreted as the call `a.__add__(b)`; otherwise, a `TypeError` will be yielded.

Table 3.1: Special Methods of Python Classes for Arithmetic Operations

Method	Result
a.__add__(b)	a + b
a.__sub__(b)	a − b
a.__mul__(b)	a*b
a.__truediv__(b)	a/b
a.__pow__(b)	a**b

To study how the special methods for arithmetic operations works, let us create a class `Polynomial` which represents polynomials. The coefficients in the polynomial is given to the constructor as a list where the index number i in this list represents the coefficients of the x^i term in the polynomial. In other words, writing `Polynomial([1, 0, -1, 2])` defines the polynomial

$$1 + 0 \cdot x - 1 \cdot x^2 + 2 \cdot x^3 = 1 - x^2 + 2x^3$$

Polynomials can be added (simply by adding the coefficients) and so our class may have an `__add__` method. Besides, it's natural to use a `__call__` method to evaluate the polynomial for a given value of x. Here is the complete implementation of class `Polynomial`:

```
class Polynomial:
    """ A class representing polynomials """

    def __init__(self, coefficients):
        """ Initialize the polynomial with its coefficients """
        self.coeff = coefficients

    def __call__(self, x):
        """ Evaluate the polynomial for a given value of x """
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        """ Return the sum of two polynomials as a Polynomial
            object """
        # Start with the longest list and add in the other
        if len(self.coeff) > len(other.coeff):
            result_coeff = self.coeff[:]  # Copy!
            for i in range(len(other.coeff)):
                result_coeff[i] += other.coeff[i]
        else:
            result_coeff = other.coeff[:] # Copy!
            for i in range(len(self.coeff)):
                result_coeff[i] += self.coeff[i]
        return Polynomial(result_coeff)
```

Class `Polynomial` has only one data attribute: the list of coefficients in the polynomial. In the `__call__` method, the polynomial is evaluated by just summing up coefficient number i times x^i for $i = 0$ to the number of coefficients in the list.

The `__add__` method in class `Polynomial` looks more advanced. The idea is to add the two lists of coefficients. But it may happen that the lists have different length. We thus start with the longer list and add in the other list element by element. Observe that `result_coeff` starts out as a *copy* of `self.coeff`. Otherwise, `self` would be the sum of itself and the `other` instance. In such case, changes in `result_coeff` as we compute the sum will be reflected in `self.coeff`. That is to say, if `p1` and p2 are instances of the class, evaluating the sum `p1 + p2` would change `p1`. Of course this is not what we want!

A subtraction method `__sub__` can be implemented in the class `Polynomial` in the same way as `__add__`. But it is slightly more complicated and left as an exercise for you. A somewhat more complicated operation is the multiplication of two polynomials. Let $p(x) = \sum_{i=0}^{m} c_i x^i$ and $q(x) = \sum_{j=0}^{n} d_j x^j$ be two polynomials. Their product is given by

$$\left(\sum_{i=0}^{m} c_i x^i\right)\left(\sum_{j=0}^{n} d_j x^j\right) = \sum_{i=0}^{m}\sum_{j=0}^{n} c_i d_j x^{i+j}$$

The nested sum must be implemented as a double loop and the list for the resulting polynomial must be created with length $m+n+1$. So the implementation of the `__mul__` method in class `Polynomial` becomes

```
def __mul__(self, other):
    """ Return the product of two polynomials as a Polynomial
        object """
    c = self.coeff
    d = other.coeff
    m = len(c) - 1
    n = len(d) - 1
    result_coeff = [0]*(m+n+1)
    for i in range(0, m+1):
        for j in range(0, n+1):
            result_coeff[i+j] += c[i]*d[j]
    return Polynomial(result_coeff)
```

To demonstrate the functionality of class `Polynomial`, we introduce two polynomials

$$p_1(x) = 1 + x, \qquad p_2(x) = 2x + 3x^3 - x^4$$

Below are some examples for working on these polynomials with this class:

```
>>> p1 = Polynomial([1, -2])
>>> p2 = Polynomial([0, 2, 0, 3, -5])
>>> p3 = p1 + p2
>>> print(p3.coeff)
[1, 0, 0, 3, -5]
>>> p4 = p1*p2
>>> print(p4.coeff)
[0, 2, -4, 3, -11, 10]
```

One way to verify the implementation is comparing `p3` at $x = 0.5$ with $p_1(x) + p_2(x)$ as follows:

```
>>> x = 0.5
>>> p1_plus_p2_value = p1(x) + p2(x)
>>> p3_value = p3(x)
>>> print(p1_plus_p2_value - p3_value)
0.0
```

Beware that `p1 + p2` is different from `p1(x) + p2(x)`. In the former case, we add two instances of class `Polynomial`. But in the latter case we add two instances of class **`float`** as the `__call__` method returns a **`float`** object.

We can also define a `__str__` method for the `Polynomial` class to print the polynomial on the screen. A straight forward implementation could simply add up strings in the form of + `self.coeff[i]*x^i`:

```
class Polynomial:
      ⋮
    def __str__(self):
        """ Return the output for printing out the polynomial """
        s = ""
        for i in range(0, len(self.coeff)):
            s += " + {0:g}*x^{1:d}".format(self.coeff[i], i)
        return s
```

However, this implementation gives rise to ugly output from a mathematical viewpoint. For example, an instance of `Polynomial` class with coefficients `[1, 0, -2, 1, 5]` would be printed as

```
 + 1*x^0 + 0*x^1 + -2*x^2 + 1*x^3 + 5*x^4
```

A more desired output would be

```
1 - 2*x^2 + x^3 + 5*x^4
```

It can be obtained by making the following adjustments to the previous output:

1. Terms with a zero coefficient is dropped.
2. A part "+ -" of the output string is replaced by "- ".
3. Unit coefficients is dropped, i. e. " 1*" is replaced by space " ".
4. Zero power is dropped and replaced by its coefficient.
5. Unit power is dropped by replacing "x^1" with "x".
6. Initial space is fixed.

These adjustments can be implemented using the `replace` method in string objects and by composing slices of the strings. The new version of the `__str__` method below contains the necessary adjustments.

```
class Polynomial:
    ⋮
    def __str__(self):
        """ Return the output for printing out the polynomial """
        s = ""
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += " + {0:g}*x^{1:d}".format(self.coeff[i], i)
        # Adjust the layout
        s = s.replace("+ -", "- ")
        s = s.replace(" 1*", " ")
        s = s.replace(" x^0", " 1")
        s = s.replace("*x^0", "")
        s = s.replace("x^1", "x")
        if s[0:3] == " + ":  # Remove initial +
            s = s[3:]
        if s[0:3] == " - ":  # Fix spaces for initial -
            s = "-" + s[3:]
        return s
```

Here is an interactive Python session demonstrating the pretty print of `Polynomial` instances using the new `__str__` method:

```
>>> p1 = Polynomial([1, -2])
>>> print(p1)
1 - 2*x
>>> p2 = Polynomial([0, 5, -1, 0, 3, -4])
>>> print(p2)
5*x - x^2 + 3*x^4 - 4*x^5
```

We can also implement the special methods of Python classes for comparing two instances of `Polynomial` class. Table 3.2 shows these special methods which are used in a similar manner as those for arithmetic operations. On the other hand, Python also provides some useful special methods for an instance of a class as shown in Table 3.3.

Table 3.2: Special Methods of Python Classes for Comparison Operations

Method	**Result**
`a.__eq__(b)`	`a == b`
`a.__ne__(b)`	`a != b`
`a.__gt__(b)`	`a > b`
`a.__ge__(b)`	`a >= b`
`a.__lt__(b)`	`a < b`
`a.__le__(b)`	`a <= b`

Table 3.3: Some Useful Special Methods of Python Classes for a class instance

Method	**Description**
`a.__len__()`	Return the length of `a`, i. e. `len(a)`
`a.__abs__()`	Return the absolute value of `a`, i. e. `abs(a)`
`a.__neg__()`	Return the negation of `a`
`a.__bool__()`	Evaluate `a` as a Boolean expression

Suppose the class `MyClass` is defined with a `__str__` method like this:

```
class MyClass:
    def __init__(self):
        self.data = 2

    def __str__(self):
        return "In __str__: {0:s}".format(str(self.data))
```

Printing an instance `a` of this class leads to the following output:

```
>>> a = MyClass()
>>> print(a)
In __str__: 2
```

Obviously, the `__str__` method is called when `a` is printed.

But what will happen if we write just `a` at the command prompt in an interactive Python shell?

```
>>> a
<__main__.MyClass object at 0x00000202C9C8F6D8>
```

When writing `a` in an interactive session, Python looks for a special method `__repr__` in `a`. This method is similar to `__str__` in that it turns the instance into a string. But there is a convention that `__str__` is a pretty print of the instance contents while `__repr__` is a complete representation of the contents of the instance. For a lot of Python classes, including `int`, `float`, `complex`, `list`, and `tuple`, these two methods give identical output. In our class `MyClass`, the `__repr__` method is missing. And we need to add it if we want

```
>>> a
```

is actually a `repr(a)` call and

```
>>> print(a)
```

is actually a `print(str(a))` statement. A simple remedy in class `MyClass` is to define

```
class MyClass:
      ⋮
    def __repr__(self):
        return self.__str__()
```

However, as we explain below, it is better to define the `__repr__` in a different way.

Recall that the Python function `eval(e)` evaluates a valid Python expression contained in the string `e`. By convention, `__repr__` returns a string such that `eval` applied to the string recreates the instance. For example, in case of our previous class `Y`, `__repr__` should return `"Y(10)"` if the `u` variable has the value `10`. Then `eval("Y(10)")` will be the same as if we had coded `Y(10)` directly in the program or in an interactive session.

Below are the examples of `__repr__` methods in classes `Y`, `Polynomial`, and `MyClass`:

```
class Y:
```

```
        ⋮
    def __repr__(self):
        return "Y(u={0:s})".format(str(self.u))

class Polynomial:
        ⋮
    def __repr__(self):
        return "Polynomial(coefficients={0:s})".format(str(self.coeff))

class MyClass:
        ⋮
    def __repr__(self):
        return "MyClass()"
```

With the above definitions, `eval(repr(x))` recreates the object `x` if it is of one of the three types above.

To summarize what we have learnt in this section, let us consider how to implement two-dimensional vectors as a Python class such that these vectors act as objects we can add, subtract, form scalar products with, and do other mathematical operations on. Vectors in the plane are described by a pair of real numbers (a, b). The mathematical rules for adding and subtracting vectors, multiplying two vectors (i. e. the dot product), the length of a vector, and multiplication by a scalar are:

$$
\begin{aligned}
(a, b) + (c, d) &= (a + c, b + d) \\
(a, b) - (c, d) &= (a - c, b - d) \\
(a, b) \cdot (c, d) &= ac + bd \\
\|(a, b)\| &= \sqrt{(a, b) \cdot (a, b)}
\end{aligned}
$$

Moreover, two vectors (a, b) and (c, d) are equal if $a = c$ and $b = d$.

We create a Python class `Vec2D` for plane vectors in which the above mathematical operations are implemented by special methods. The class must contain two data attributes, one for each component of the vector called `x` and `y`. We include special methods for addition, subtraction, the dot product (multiplication), the absolute value (length), comparison of two vectors (== and `!=`), as well as a method for printing out a vector. The complete code of this class is listed below.

```python
# vec2d.py
# This program defines the class Vec2D representing plane vectors.
# Last Update on 19 Nov 2020 by F K Chow

import math

class Vec2D:
    """ A class representing plane vectors """

    def __init__(self, x, y):
        """ Initialize the vector with x and y components """
        self.x = x
        self.y = y

    def __add__(self, other):
        """ Return the sum of two vectors as a Vec2D object """
        return Vec2D(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        """ Return the difference of two vectors as a Vec2D object """
        return Vec2D(self.x - other.x, self.y - other.y)

    def __mul__(self, other):
        """ Return the dot product of two vectors """
        return self.x*other.x + self.y*other.y

    def __eq__(self, other):
        """ Check whether two vectors are equal """
        return self.x == other.x and self.y == other.y

    def __str__(self):
        """ Format the output for printing out the vector """
        return "({:g}, {:g})".format(self.x, self.y)
```

```
    def __abs__(self):
        """ Return the length of the vector """
        return math.sqrt(self.x**2 + self.y**2)

    def __ne__(self, other):
        """ Check whether two vectors are unequal """
        return not self.__eq__(other)  # Reuse __eq__
```

The `__add__`, `__sub__`, `__mul__`, `__abs__`, and `__eq__` methods are quite easy to understand from the previous mathematical definitions of these operations. Notice that in the method `__ne__`, we simply reuse the equality operator `__eq__`, but precede it with a **`not`**. We can also implement this method as

```
    def __ne__(self, other):
        """ Check whether two vectors are unequal. """
        return self.x != other.x or self.y != other.y
```

However, this implementation requires us to write more. It also has the danger of introducing an error in the logics of the boolean expressions. If we know that the `__eq__` method works, then a more reliable approach is to reuse this method and observe that **`not`** == gives us the effect of `!=`.

Besides, we should beware of a problem for our implementation of the equality operator in class `Vec2D`. We have tested for equality by comparing both vector components. However, each component is a floating-point number which may be subject to round-off errors both in the representation on the computer and from previous (inexact) floating-point calculations. Two mathematically equal components may have different inexact representations on the computer. To resolve this problem, we should avoid testing for equality, but instead checking that the difference between the components is sufficiently small. The function `isclose` in the `math` module tells us whether two values are approximately equal or "close" to each other. It is a useful tool for comparing **`float`** objects. With this function, we replace `if a == b` by `if math.isclose(a, b)`. A more reliable equality operator can now be implemented:

```
class Vec2D:
    ⋮
    def __eq__(self, other):
        """ Check whether two vectors are equal """
```

```
        return math.isclose(self.x, other.x) and \
               math.isclose(self.y, other.y)
```

As a rule of thumb, we should never apply the == test to two **float** objects.

In class `Vec2D`, the special method `__len__` could be introduced as a synonym for `__abs__`. That is to say, for a `Vec2D` instance named `v`, `len(v)` is the same as `abs(v)` since the absolute value of a vector is mathematically the same as the length of the vector. However, if we implement

```
    def __len__(self):
        """ Return the length of the vector """
        return self.__abs__() # Reuse __abs__
```

we will run into trouble as we compute `len(v)` whose answer is a **float** object. Python will then complain and tell us that `len(v)` must return an **int**. Thus `__len__` cannot be used as a synonym for the length of the vector in our application. On the other hand, we could let `len(v)` giving the number of components of the vector:

```
    def __len__(self):
        """ Return the number of components of the vector. """
        return 2
```

This is not a very useful function as we already know that all our `Vec2D` vectors have just two components. Nevertheless, the `__len__` method is useful for generalizations of the class to vectors with n components.

Below is an interactive Python session illustrating how to deal with `Vec2D` objects:

```
>>> u = Vec2D(0, 1)
>>> v = Vec2D(1, 0)
>>> w = Vec2D(1, 1)
>>> a = u + v
>>> print(a)
(1, 1)
>>> a == w
True
>>> a = u - v
>>> print(a)
(-1, 1)
>>> print(abs(u))
```

```
1.0
>>> u == v
False
>>> u != v
True
```

When you read through this interactive session, you should check that the calculation is mathematically correct, the resulting object type of a calculation is correct, and how each calculation is performed in the program. The latter topic is investigated by following the program flow through the methods of the class. You need this type of thorough understanding to find and correct any bugs in the class.

For real computations with vectors in the plane, you would probably just use a `NumPy` array of length 2. However, one thing such objects cannot do is evaluating `u*v` as a dot product since the multiplication operator for `NumPy` arrays is not defined as a dot product. Another difference between our `Vec2D` class and `NumPy` arrays is the **`abs`** function, which computes the length of the vector in class `Vec2D`, while it does something completely different with `NumPy` arrays.

3.2.3 Class Variables, Static Methods, and Class Methods

Up to now, we have only worked on instance variables — the data attributes owned by individual instances of a class. Sometimes it is natural to have data attributes that are shared among all instances. For example, we may have a data attribute that counts how many instances of a class have been made so far. Let us illustrate how to do this in a little class for points (x, y, z) in space:

```
class SpacePoint:
    counter = 0
    def __init__(self, x, y, z):
        self.p = (x, y, z)
        SpacePoint.counter += 1
```

The `counter` attribute is initialized at the same indentation level as the methods in the class, and the attribute is not prefixed by `self`. Such attributes declared outside methods of a class are known as **class variables** that are shared among all instances of the class. To access class variables, we use the dot notation `<ClassName>.<cvar>` where `<ClassName>` is the class name and `<cvar>` is the name of the class variable. So we access the `counter` attribute of `SpacePoint` class by using the notation `SpacePoint.counter`. We increase this common counter by 1 in the constructor of this class. That is to say, the counter is updated to keep track of the number of objects created so far each time when a new instance of `SpacePoint` class is made. For example,

```
>>> p1 = SpacePoint(0, 0, 0)
>>> SpacePoint.counter
1
>>> for i in range(100):
...     p = SpacePoint(i, 1, i+1)
...
>>> SpacePoint.counter
101
```

The methods we have seen so far must be called through an instance, which is fed in as the `self` argument in the method. We can also define **class methods** and **static methods** which can be called without having any instances. Class methods are defined with a `self`-like parameter conventionally called `cls` that points to the class when the method is called. They cannot modify the state of individual instances of a class, but can still modify the state of a class that applies to all instances of the class. By contrast, static methods are defined without `self` arguments and they can neither modify the state of a class or the state of individual instances of a class. Just like class variables, we access both class methods and static methods using the dot notation <*ClassName*>.<*methodname*> where <*ClassName*> is the class name and <*methodname*> is the name of the method. To illustrate the syntax for defining these methods, let us make a simple class `Square` with a class method `unit_square` and a static method `area_formula`:

```
>>> class Square:
...     """ A class representing squares """
...
...     def __init__(self, side):
...         """ Initialize the side of the square """
...         self.side = side
...
...     @classmethod
...     def unit_square(cls):
...         """ Factory method creating a unit square """
```

```
...             return cls(1)
...
...         @staticmethod
...         def area_formula():
...             """ Print the area formula of a square """
...             print("Area of a square = side x side")
```

In class `Square`, the class method `unit_square` serves as a factory method which only creates unit squares. And the static method `area_formula` works like a plain Python function contained inside the class. These methods can be used without instantiating the class like this:

```
>>> us = Square.unit_square()
>>> us.side
1
>>> Square.area_formula()
Area of a square = side x side
```

We can also make an instance of class `Square` and call `unit_square` and `area_formula` through that instance:

```
>>> s = Square(2)
>>> s.area_formula()
Area of a square = side x side
>>> us = s.unit_square()
>>> us.side
1
```

Class methods are often used as factory methods that can create specific instances of a class. On the other hand, static methods are used when you want a global function, but find it natural to let the function belong to a class and be prefixed with the class name.

3.3 Class inheritance in Python

Suppose we have written a class for straight line functions $y = c_1 x + c_0$ as follows:

```
class Line:
    """ A class representing straight line functions y = c1*x + c0 """

    def __init__(self, c1, c0):
```

```
        """ Initialize the coefficients in the straight line
            function """
        self.c1 = c1
        self.c0 = c0

    def __call__(self, x):
        """ Evaluate the straight line function for a given value of
            x """
        return self.c1*x + self.c0

    def table(self, L, R, n):
        """ Return a table with n points for L <= x <= R """
        s = ""
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += "{0:10g} {1:10g}\n".format(x, y)
        return s
```

A parabolic function $y = c_2x^2 + c_1x + c_0$ contains a straight line function as a special case with $c_2 = 0$. So the class for parabolic functions will be similar to a class for straight line functions. All we have do to is to add the new term c_2x^2 in the `__call__` method and store c_2 in the constructor `__init__`:

```
class Parabola:
    """ A class representing parabolic functions y = c_2*x*x + c1*x
        + c0 """

    def __init__(self, c2, c1, c0):
        """ Initialize the coefficients in the parabolic function """
        self.c2 = c2
        self.c1 = c1
        self.c0 = c0

    def __call__(self, x):
        """ Evaluate the parabolic function for a given value of x """
```

```
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """ Return a table with n points for L <= x <= R """
        s = ""
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += "{0:10g} {1:10g}\n".format(x, y)
        return s
```

Observe that we can copy the `table` method from class `Line` without any modifications.

Python have a special construct so that class `Parabola` does not need to repeat the code that we have already written in class `Line`. We can specify that class `Parabola` *inherits* all code from class `Line` by adding "`(Line)`" in the class headline:

```
class Parabola(Line):
```

Class `Parabola` now automatically gets all the code from class `Line`. We say that class `Parabola` is *derived* from class `Line`. It means that class `Parabola` is a subclass of its superclass `Line`. In general, a Python subclass may be derived from one or more other superclasses using the syntax:

```
class <Subclass>(<Superclass1>, <Superclass2>, ..., <SuperclassN>)
```

where <*Subclass*> is the name of the subclass and <*Superclass1*>, <*Superclass2*>, ..., <*SuperclassN*> are the names of the superclasses. The subclass can be defined in the same file that defines the superclasses or in a different Python file which imports the superclasses.

Of course, the subclass `Parabola` should not be identical to the superclass `Line`. It needs to add data in the constructor for the new term and to modify the call operator to deal with the new term; but the `table` method can be inherited as it is. If we implement the constructor and the call operator in class `Parabola`, then these methods will *override* the inherited versions from class `Line`. On the other hand, if a `table` method is not implemented in class `Parabola`, then the one inherited from class `Line` would be available as if it were coded visibly in class `Parabola`.

When the methods `__call__` and `__init__` are implemented in subclass `Parabola`, they must first have the statements of the inherited methods from the superclass `Line` and then include the additional code. An important principle in computer programming is to avoid duplicating code. We should thus call up functionality in class `Line` instead of copying statements from class `Line` methods to `Parabola` methods. Indeed, any method in a superclass can be called in its subclass using the syntax:

```
<Superclass>.<methodname>(self, <arg1>, <arg2>, ..., <argN>)
```

or

```
super(<Subclass>, self).<methodname>(<arg1>, <arg2>, ..., <argN>)
```

where `<Superclass>` is the name of the superclass, `<Subclass>` is the name of the subclass, `<methodname>` is the name of the method, and `<arg1>, <arg2>,..., <argN>` are the arguments of the method. Note that the latter construction works only if the superclass is derived from Python's general superclass `object`.

Therefore, we can write class `Parabola` as a subclass of class `Line` by simply implementing the new extra code that we want as shown below:

```
class Parabola(Line):
    """ A class representing parabolic functions y = c_2*x*x + c1*x
        + c0 """

    def __init__(self, c2, c1, c0):
        """ Initialize the coefficients in the parabolic function """
        self.c2 = c2
        Line.__init__(self, c1, c0) # Let class Line store c1 and c0

    def __call__(self, x):
        """ Evaluate the parabolic function for a given value of x. """
        return self.c2*x**2 + Line.__call__(self, x)
```

Such implementation of class `Parabola` provides exactly the same functionality as the previous version of class `Parabola` that did not inherit from class `Line`. Below is a quick demo of the new version of class `Parabola` in an interactive Python session:

```
>>> p = Parabola(1, -4, 3)
>>> p1 = p(2.0)
>>> print(p1)
-1.0
>>> print(p.table(0, 1, 5))
```

```
      0           3
   0.25      2.0625
    0.5        1.25
   0.75      0.5625
      1           0
```

Observe that the statement `p = Parabola(1, -4, 3)` leads to a call to the constructor method in class `Parabola`. Inside the constructor in class `Parabola`, we call the constructor in class `Line`. In this latter method, we create two data attributes `c0` and `c1` in the `self` object. Back in class `Parabola`'s constructor, we add a third data attribute `c2` to the same `self` object. Finally, the `self` object is invisibly returned and referred to by `p`. The other statement `p1 = p(2.0)` has a similar program flow. First, the `__call__` method in class `Parabola` is invoked. Then the program flow jumps to the `__call__` method in class `Line` for evaluating the linear part $c_1x + c_0$ of the parabolic function. Lastly, the flow jumps back to the `__call__` in class `Parabola` where we add the new quadratic term.

Python provides the function `isinstance(i, t)` for checking whether an instance `i` is of type `t`. For example,

```
>>> l = Line(2, -5)
>>> isinstance(l, Line)
True
>>> isinstance(l, Parabola)
False
```

We can see that a `Line` instance is not a `Parabola` instance. But is the reverse true?

```
>>> p = Parabola(3, 6, -9)
>>> isinstance(p, Parabola)
True
>>> isinstance(p, Line)
True
```

The answer is yes. From a class hierarchy perspective, an instance of the subclass `Parabola` is regarded as an instance of the superclass `Line` since it contains everything that a `Line` instance contains.

Every instance has an attribute `__class__` that holds the type of class. If `p` is a `Parabola` instance, then

```
>>> p.__class__
```

```
<class 'parabola.Parabola'>
>>> p.__class__ == Parabola
True
>>> p.__class__.__name__
'Parabola'
```

Beware that `p.__class__` is a **class** object while `p.__class__.__name__` is a string of the class name. These two variables can be used as an alternative test for the class type:

```
if p.__class__.__name__ == "Parabola":
    <statements>
```

or

```
if p.__class__ == Parabola:
    <statements>
```

Nevertheless, it is not a recommended programming style to use `isinstance(p, Parabola)` for checking the type of an object.

In addition, Python provides the function `issubclass(c1, c2)` for checking whether a class `c1` is a subclass of class `c2`. For example,

```
>>> issubclass(Parabola, Line)
True
>>> issubclass(Line, Parabola)
False
```

The superclasses of a class are stored as a tuple in the `__bases__` attribute of the **class** object: If `p` is again a `Parabola` instance, then

```
>>> p.__class__.__bases__
(<class 'parabola.Line'>,)
>>> p.__class__.__bases__[0].__name__ # Extract name as string
'Line'
```

Rather than letting class `Parabola` inherit from class `Line`, we may let it *contain* a `Line` instance as a data attribute:

```
class Parabola:
    """ A class representing parabolic functions y = c_2*x*x + c1*x
        + c0 """
```

```
    def __init__(self, c2, c1, c0):
        """ Initialize the coefficients in the parabolic function """
        self.c2 = c2
        # Use a class Line instance store c1 and c0
        self.line = Line(c1, c0)

    def __call__(self, x):
        """ Evaluate the parabolic function for a given value of x """
        return self.c2*x**2 + self.line(x)

    def table(self, L, R, n):
      ⋮
```

Whether to use inheritance or an attribute depends on the problem being solved. If it is natural to say that a class `A` instance *is* a class `B` instance, we say that class `A` has an **is-a relationship** with class `B`. Alternatively, if it is natural to say that a class `A` instance *has* a class `B` instance, we say that class `A` has an **has-a relationship** with class `B`. In the current example, the is-a relationship is more natural since a special case of a parabola is a straight line.

However, from a mathematical viewpoint, many people will think that a line is a special case of a parabola instead of a parabola is a line. Adopting this reasoning reverses the dependency of the classes: now it is more natural to let `Line` be a subclass of `Parabola` (i. e. a `Line` instance is a `Parabola` instance). This is an easy task that can be done by implementing the classes as follows:

```
class Parabola:
    """ A class representing parabolic functions y = c_2*x*x + c1*x
        + c0 """

    def __init__(self, c2, c1, c0):
        """ Initialize the coefficients in the parabolic function """
        self.c2 = c2
        self.c1 = c1
        self.c0 = c0

    def __call__(self, x):
```

```
        """ Evaluate the parabolic function for a given value of x """
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
      ⋮

class Line(Parabola):
    """ A class representing straight line functions y = c1*x + c0 """

    def __init__(self, c1, c0):
        """ Initialize the coefficients in the straight line
            function """
        # Let class Parabola store c1 and c0
        Parabola.__init__(self, 0, c1, c0)
```

The `__call__` and `table` methods can be inherited in class `Line` as they are defined in class `Parabola`. Notice that the inherited `__call__` method from class `Parabola` will work since the coefficient `c2` is zero.

In the above example, inheritance is used for *restricting* functionality instead of *extending* the functionality of the superclass. More importantly, it becomes clear that there is no unique way of arranging classes in hierarchies. Rather than starting with `Line` and introducing `Parabola`, `Cubic`, and perhaps eventually a general `Polynomial` class, we can start with a general `Polynomial` class and let `Parabola` be a subclass which restricts all coefficients except the first three to be zero. Class `Line` can then be a subclass of `Parabola`, restricting the value of one more coefficient.

How classes depend on each other is influenced by two factors: sharing of code and logical relations. From the sharing of code perspective, many people will say that class `Parabola` is naturally a subclass of class `Line` since the former adds code to the latter. On the other hand, `Line` is naturally a subclass of `Parabola` from the logical relations in mathematics. In fact, we can also consider the class dependency from the perspective of computational efficiency. When `Line` is a subclass of `Parabola`, we always evaluate the c_2x^2 term in the parabolic function although this term is zero. Nevertheless, when `Parabola` is a subclass of `Line`, we call `Line.__call__` to evaluate the linear part of the parabolic function, and this call is costly in Python. From a pure efficiency point of view, we would reprogram the linear part in `Parabola.__call__` (which is against the programming habit we have been arguing for!). We should aware of the many different considerations that come into play when we establish class relationships.

As another example of inheritance, let us consider how to derive `SavingsAccounts` and `CurrentAccount` classes from `BankAccount` class as illustrated in Fig. 3.2. Below are the definitions of these two subclasses in the program `specificaccounts.py`:

```
# specificaccounts.py
# This program defines the classes SavingsAccount and CurrentAccount
# which represent a savings account and a current account,
# respecitvely. These classes are the subclass of BankAccount class.
# Last Update on 5 Jan 2022 by F K Chow

from bankaccount import BankAccount

class SavingsAccount(BankAccount):
    """ A class representing a savings account """

    def __init__(self, name, account_num, interest_rate,
                 opening_balance=0):
        """ Initialize the saving account with a name, an account
            number, an interest rate, and an opening balance whose
            default is 0 """
        self.interest_rate = interest_rate
        BankAccount.__init__(self, name, account_num,
                             opening_balance)

    def add_interest(self):
        """ Add interest to the account at the rate of
            self.interest_rate """
        self.balance *= (1.0 + self.interest_rate/100)

class CurrentAccount(BankAccount):
    """ A class representing a current account """

    def __init__(self, name, account_num, annual_fee,
```

```
                 transaction_limit, opening_balance=0):
        """ Initialize the current account with a name, an account
            number, an annual fee, a single transaction limit, and
            an opening balance whose default is 0 """
        self.annual_fee = annual_fee
        self.transaction_limit = transaction_limit
        BankAccount.__init__(self, name, account_num,
                             opening_balance)

    def apply_annual_fee(self):
        """ Deduct the annual fee from the account balance """
        self.balance = max(0, self.balance - self.annual_fee)

    def withdraw(self, amount):
        """ Withdraw amount from the bank account if there are
            sufficient funds and amount is less than the single
            transaction limit """
        if amount <= 0:
            print("Invalid withdrawal amount:", amount)
        else:
            if amount > self.balance:
                print("Insufficient funds")
            elif amount > self.transaction_limit:
                print("${0:.2f} exceeds the single transaction limit "
                      "of ${1:.2f}.".format(amount,
                                            self.transaction_limit))
            else:
                self.balance -= amount
```

We can observe that both these subclasses override the `__init__` method in the superclass `BankAccount` to handle the new data attributes. Moreover, the `withdraw` method in the derived class `CurrentAccount` overrides that of the same name in the superclass `BankAccount`. As a result, if the `withdraw` is called on a `CurrentAccount` instance, then the one called is that of the `CurrentAccount` class.

Here is an interactive Python session demonstrating how to use these subclasses:

```
>>> savacc = SavingsAccount("Tom Hanks", 415228, 4.5, 1200)
>>> savacc.print_balance()
The balance of account number 415228 is $1200.00.
>>> savacc.add_interest()
>>> savacc.print_balance()
The balance of account number 415228 is $1254.00.
>>> curracc = CurrentAccount("Chris Evans", 783009, 15, 250)
>>> curracc.withdraw(300)
Insufficient funds
>>> curracc.deposit(900)
>>> curracc.print_balance()
The balance of account number 783009 is $900.00.
>>> curracc.withdraw(300)
$300.00 exceeds the single transaction limit of $250.00.
>>> curracc.withdraw(180)
>>> curracc.print_balance()
The balance of account number 783009 is $720.00.
```