Introduction to Classes

7

A class packs a set of data (variables) together with a set of functions operating on the data. The goal is to achieve more modular code by grouping data and functions into manageable (often small) units. Most of the mathematical computations in this book can easily be coded without using classes, but in many problems, classes enable either more elegant solutions or code that is easier to extend at a later stage. In the non-mathematical world, where there are no mathematical concepts and associated algorithms to help structure the problem solving, software development can be very challenging. Classes may then improve the understanding of the problem and contribute to simplify the modeling of data and actions in programs. As a consequence, almost all large software systems being developed in the world today are heavily based on classes.

Programming with classes is offered by most modern programming languages, also Python. In fact, Python employs classes to a very large extent, but one can – as we have seen in previous chapters – use the language for lots of purposes without knowing what a class is. However, one will frequently encounter the class concept when searching books or the World Wide Web for Python programming information. And more important, classes often provide better solutions to programming problems. This chapter therefore gives an introduction to the class concept with emphasis on applications to numerical computing. More advanced use of classes, including inheritance and object orientation, is the subject of Chapter 9.

The folder `src/class` contains all the program examples from the present chapter.

7.1 Simple Function Classes

Classes can be used for many things in scientific computations, but one of the most frequent programming tasks is to represent mathematical functions which have a set of parameters in addition to one or more independent variables. Chapter 7.1.1 explains why such mathematical functions pose difficulties for programmers, and Chapter 7.1.2 shows how the class idea meets these difficulties. Chapters 7.1.3 presents another example where a class represents a mathematical function. More advanced material about classes, which for some readers may clarify the ideas, but which can also be skipped in a first reading, appears in Chapters 7.1.4 and Chapter 7.1.5.

7.1.1 Problem: Functions with Parameters

To motivate for the class concept, we will look at functions with parameters. The $y(t) = v_0 t - \frac{1}{2}gt^2$ function on page 1 is such a function. Conceptually, in physics, y is a function of t, but y also depends on two other parameters, v_0 and g, although it is not natural to view y as a function of these parameters. We may write $y(t; v_0, g)$ to indicate that t is the independent variable, while v_0 and g are parameters. Strictly speaking, g is a fixed parameter[1], so only v_0 and t can be arbitrarily chosen in the formula. It would then be better to write $y(t; v_0)$.

In the general case, we may have a function of x that has n parameters $p_1, \ldots, p_n$: $f(x; p_1, \ldots, p_n)$. One example could be

$$g(x; A, a) = Ae^{-ax}\,.$$

How should we implement such functions? One obvious way is to have the independent variable and the parameters as arguments:

```
def y(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

def g(x, a, A):
    return A*exp(-a*x)
```

Problem. There is one major problem with this solution. Many software tools we can use for mathematical operations on functions assume that a function of one variable has only one argument in the computer representation of the function. For example, we may have a tool for differentiating a function $f(x)$ at a point x, using the approximation

[1] As long as we are on the surface of the earth, g can be considered fixed, but in general g depends on the distance to the center of the earth.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{7.1}$$

coded as

```
def diff(f, x, h=1E-10):
    return (f(x+h) - f(x))/h
```

The `diff` function works with any function `f` that takes one argument:

```
def h(t):
    return t**4 + 4*t

dh = diff(h, 0.1)

from math import sin, pi
x = 2*pi
dsin = diff(sin, x, h=1E-9)
```

Unfortunately, `diff` will not work with our `y(t, v0)` function. Calling `diff(y, t)` leads to an error inside the `diff` function, because it tries to call our `y` function with only one argument while the `y` function requires two.

Writing an alternative `diff` function for `f` functions having two arguments is a bad remedy as it restricts the set of admissible `f` functions to the very special case of a function with one independent variable and one parameter. A fundamental principle in computer programming is to strive for software that is as general and widely applicable as possible. In the present case, it means that the `diff` function should be applicable to all functions `f` of one variable, and letting `f` take one argument is then the natural decision to make.

The mismatch of function arguments, as outlined above, is a major problem because a lot of software libraries are available for operations on mathematical functions of one variable: integration, differentiation, solving $f(x) = 0$, finding extrema, etc. (see for instance Chapters 3.6.2 and 5.1.9„ and Appendices A and B). All these libraries will try to call the mathematical function we provide with only one argument.

A Bad Solution: Global Variables. The requirement is thus to define Python implementations of mathematical functions of one variable with one argument, the independent variable. The two examples above must then be implemented as

```
def y(t):
    g = 9.81
    return v0*t - 0.5*g*t**2

def g(t):
    return A*exp(-a*x)
```

These functions work only if `v0`, `A`, and `a` are global variables, initialized before one attempts to call the functions. Here are two sample calls where `diff` differentiates `y` and `g`:

```
v0 = 3
dy = diff(y, 1)

A = 1; a = 0.1
dg = diff(g, 1.5)
```

The use of global variables is in general considered bad programming. Why global variables are problematic in the present case can be illustrated when there is need to work with several versions of a function. Suppose we want to work with two versions of $y(t; v_0)$, one with $v_0 = 1$ and one with $v_0 = 5$. Every time we call `y` we must remember which version of the function we work with, and set `v0` accordingly prior to the call:

```
v0 = 1; r1 = y(t)
v0 = 5; r2 = y(t)
```

Another problem is that variables with simple names like `v0`, `a`, and `A` may easily be used as global variables in other parts of the program. These parts may change our `v0` in a context different from the `y` function, but the change affects the correctness of the `y` function. In such a case, we say that changing `v0` has *side effects*, i.e., the change affects other parts of the program in an unintentional way. This is one reason why a golden rule of programming tells us to limit the use of global variables as much as possible.

Another solution to the problem of needing two v_0 parameters could be to introduce two `y` functions, each with a distinct v_0 parameter:

```
def y1(t):
    g = 9.81
    return v0_1*t - 0.5*g*t**2

def y2(t):
    g = 9.81
    return v0_2*t - 0.5*g*t**2
```

Now we need to initialize `v0_1` and `v0_2` once, and then we can work with `y1` and `y2`. However, if we need 100 v_0 parameters, we need 100 functions. This is tedious to code, error prone, difficult to administer, and simply a really bad solution to a programming problem.

So, is there a good remedy? The answer is yes: The class concept solves all the problems described above!

7.1.2 Representing a Function as a Class

A class contains a set of variables (data) and a set of functions, held together as one unit. The variables are visible in all the functions in the class. That is, we can view the variables as "global" in these functions. These characteristics also apply to modules, and modules can be used to obtain many of the same advantages as classes offer (see comments

in Chapter 7.1.5). However, classes are technically very different from modules. You can also make many copies of a class, while there can be only one copy of a module. When you master both modules and classes, you will clearly see the similarities and differences. Now we continue with a specific example of a class.

Consider the function $y(t; v_0) = v_0 t - \frac{1}{2} g t^2$. We may say that v_0 and g, represented by the variables `v0` and `g`, constitute the data. A Python function, say `value(t)`, is needed to compute the value of $y(t; v_0)$ and this function must have access to the data `v0` and `g`, while `t` is an argument.

A programmer experienced with classes will then suggest to collect the data `v0` and `g`, and the function `value(t)`, together as a class. In addition, a class usually has another function, called *constructor* for initializing the data. The constructor is always named `__init__`. Every class must have a name, often starting with a capital, so we choose `Y` as the name since the class represents a mathematical function with name y. Figure 7.1 sketches the contents of class `Y` as a so-called UML diagram, here created with Lumpy (from Appendix E.3) with aid of the little program `class_Y_v1_UML.py`. The UML diagram has two "boxes", one where the functions are listed, and one where the variables are listed. Our next step is to implement this class in Python.

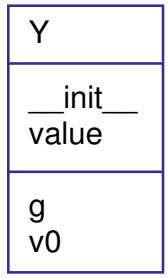

Fig. 7.1 UML diagram with function and data in the simple class `Y` for representing a mathematical function $y(t; v_0)$.

Implementation. The complete code for our class `Y` looks as follows in Python:

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

A puzzlement for newcomers to Python classes is the `self` parameter, which may take some efforts and time to fully understand.

Usage and Dissection. Before we dig into what each in the class implementation means, we start by showing how the class can be used to compute values of the mathematical function $y(t; v_0)$.

A class creates a new data type, so now we have a data type `Y` of which we can make objects of[2]. An object of a user-defined class (like `Y`) is usually called an *instance*. We need such an instance in order to use the data in the class and call the `value` function. An object of a user-defined class (like `Y`) is usually called an *instance*. The following statement constructs an instance bound to the variable name `y`:

```
y = Y(3)
```

Seemingly, we call the class `Y` as if it were a function. Actually, `Y(3)` is automatically translated by Python to a call to the constructor `__init__` in class `Y`. The arguments in the call, here only the number `3`, are always passed on as arguments to `__init__` *after* the `self` argument. That is, `v0` gets the value `3` and `self` is just dropped in the call. This may be confusing, but it is a rule that the `self` argument is never used in calls to functions in classes.

With the instance `y`, we can compute the value $y(t = 0.1; v_0 = 3)$ by the statement

```
v = y.value(0.1)
```

Here also, the `self` argument is dropped in the call to `value`. To access functions and variables in a class, we must prefix the function and variable names by the name of the instance and a dot: the `value` function is reached as `y.value`, and the variables are reached as `y.v0` and `y.g`. We can, for example, print the value of `v0` in the instance `y` by writing

```
print y.v0
```

The output will in this case be `3`.

We have already introduced the term "instance" for the object of a class. Functions in classes are commonly called *methods*, and variables (data) in classes are called *attributes*. From now on we will use this terminology. In our sample class `Y` we have two methods, `__init__` and `value`, and two attributes, `v0` and `g`. The names of methods and attributes can be chosen freely, just as names of ordinary Python functions and variables. However, the constructor must have the name `__init__`, otherwise it is not automatically called when we create new instances.

You can do whatever you want in whatever method, but it is a convention to use the constructor for initializing the variables in the class such that the class is "ready for use".

[2] All familiar Python objects, like lists, tuples, strings, floating-point numbers, integers, etc., are in fact built-in Python classes, with names `list`, `tuple`, `str`, `float`, `int`, etc.

The `self` *Variable.* Now we will provide some more explanation of the `self` parameter and how the class methods work. Inside the constructor `__init__`, the argument `self` is a variable holding the new instance to be constructed. When we write

```
    self.v0 = v0
    self.g = 9.81
```

we define two new attributes in this instance. The `self` parameter is invisibly returned to the calling code. We can imagine that Python translates `y = Y(3)` to

```
Y.__init__(y, 3)
```

so when we do a `self.v0 = v0` in the constructor, we actually initialize `y.v0`. The prefix with `Y.` is necessary to reach a class method (just like prefixing a function in a module with the module name, e.g., `math.exp`). If we prefix with `Y.`, we need to explicitly feed in an instance for the `self` argument, like `y` in the code line above, but if we prefix with `y.` (the instance name) the `self` argument is dropped. It is the latter "instance name prefix" which we shall use when computing with classes.

Let us look at a call to the `value` method to see a similar use of the `self` argument. When we write

```
value = y.value(0.1)
```

Python translates this to a call

```
value = Y.value(y, 0.1)
```

such that the `self` argument in the `value` method becomes the `y` instance. In the expression inside the `value` method,

```
self.v0*t - 0.5*self.g*t**2
```

`self` is `y` so this is the same as

```
y.v0*t - 0.5*y.g*t**2
```

The rules regarding "`self`" are listed below:

- Any class method must have `self` as first argument[3].
- `self` represents an (arbitrary) instance of the class.
- To access another class method or a class attribute, inside class methods, we must prefix with `self`, as in `self.name`, where `name` is the name of the attribute or the other method.
- `self` is dropped as argument in calls to class methods.

[3] The name can be any valid variable name, but the name `self` is a widely established convention in Python.

It takes some time to understand the `self` variable, but more examples and hands-on experience with class programming will help, so just be patient and continue reading.

Extension of the Class. We can have as many attributes and methods as we like in a class, so let us add a new method to class `Y`. This method is called `formula` and prints a string containing the formula of the mathematical function y. After this formula, we provide the value of v_0. The string can then be constructed as

```
'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

where `self` is an instance of class `Y`. A call of `formula` does not need any arguments:

```
print y.formula()
```

should be enough to create, return, and print the string. However, even if the `formula` method does not need any arguments, it must have a `self` argument, which is left out in the call but needed inside the method to access the attributes. The implementation of the method is therefore

```
    def formula(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

For completeness, the whole class now reads

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2

    def formula(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

Example on use may be

```
y = Y(5)
t = 0.2
v = y.value(t)
print 'y(t=%g; v0=%g) = %g' % (t, y.v0, v)
print y.formula()
```

with the output

```
y(t=0.2; v0=5) = 0.8038
v0*t - 0.5*g*t**2; v0=5
```

Remark. A common mistake done by newcomers to the class construction is to place the code that applies the class at the same indentation as the class methods. This is illegal. Only method definitions and assignments to so-called static attributes (Chapter 7.7) can appear in

the indented block under the `class` headline. Ordinary attribute assignment must be done inside methods. The main program using the class must appear with the same indent as the `class` headline.

Using Methods as Ordinary Functions. We may create several y functions with different values of v_0:

```
y1 = Y(1)
y2 = Y(1.5)
y3 = Y(-3)
```

We can treat `y1.value`, `y2.value`, and `y3.value` as ordinary Python functions of `t`, and then pass them on to any Python function that expects a function of one variable. In particular, we can send the functions to the `diff(f, x)` function from page 339:

```
dy1dt = diff(y1.value, 0.1)
dy2dt = diff(y2.value, 0.1)
dy3dt = diff(y3.value, 0.2)
```

Inside the `diff(f, x)` function, the argument `f` now behaves as a function of one variable that automatically carries with it two variables `v0` and `g`. When `f` refers to (e.g.) `y3.value`, Python actually knows that `f(x)` means `y3.value(x)`, and inside the `y3.value` method `self` is `y3`, and we have access to `y3.v0` and `y3.g`.

Doc Strings. A function may have a doc string right after the function definition, see Chapter 2.2.7. The aim of the doc string is to explain the purpose of the function and, for instance, what the arguments and return values are. A class can also have a doc string, it is just the first string that appears right after the `class` headline. The convention is to enclose the doc string in triple double quotes `"""`:

```
class Y:
    """The vertical motion of a ball."""

    def __init__(self, v0):
        ...
```

More comprehensive information can include the methods and how the class is used in an interactive session:

```
class Y:
    """
    Mathematical function for the vertical motion of a ball.

    Methods:
       constructor(v0): set initial velocity v0.
       value(t): compute the height as function of t.
       formula(): print out the formula for the height.

    Attributes:
       v0: the initial velocity of the ball (time 0).
       g: acceleration of gravity (fixed).
```

```
    Usage:
    >>> y = Y(3)
    >>> position1 = y.value(0.1)
    >>> position2 = y.value(0.3)
    >>> print y.formula()
    v0*t - 0.5*g*t**2; v0=3
    """
```

7.1.3 Another Function Class Example

Let us apply the ideas from the `Y` class to the $v(r)$ function specified in (4.20) on page 230. We may write this function as $v(r;\beta,\mu_0,n,R)$ to indicate that there is one primary independent variable (r) and four physical parameters (β, μ_0, n, and R). The class typically holds the physical parameters as variables and provides an `value(r)` method for computing the v function:

```
class VelocityProfile:
    def __init__(self, beta, mu0, n, R):
        self.beta, self.mu0, self.n, self.R = beta, mu0, n, R

    def value(self, r):
        beta, mu0, n, R = self.beta, self.mu0, self.n, self.R
        n = float(n)  # ensure float divisions
        v = (beta/(2.0*mu0))**(1/n)*(n/(n+1))*\
            (R**(1+1/n) - r**(1+1/n))
        return v
```

There is seemingly one new thing here in that we initialize several variables on the same line[4]:

```
        self.beta, self.mu0, self.n, self.R = beta, mu0, n, R
```

This is perfectly valid Python code and equivalent to the multi-line code

```
        self.beta = beta
        self.mu0 = mu0
        self.n = n
        self.R = R
```

In the `value` method it is convenient to avoid the `self.` prefix in the mathematical formulas and instead introduce the local short names `beta`, `mu0`, `n`, and `R`. This is in general a good idea, because it makes it easier to read the implementation of the formula and check its correctness.

Here is one possible application of class `VelocityProfile`:

[4] The comma-separated list of variables on the right-hand side forms a tuple so this assignment is just the usual construction where a set of variables on the left-hand side is set equal to a list or tuple on the right-hand side, element by element. See page 58.

```
v1 = VelocityProfile(R=1, beta=0.06, mu0=0.02, n=0.1)
# plot v1 versus r:
from scitools.std import *
r = linspace(0, 1, 50)
v = v1.value(r)
plot(r, v, label=('r', 'v'), title='Velocity profile')
```

Remark. Another solution to the problem of sending functions with parameters to a general library function such as `diff` is provided in Appendix E.5. The remedy there is to transfer the parameters as arguments "through" the `diff` function. This can be done in a general way as explained in that appendix.

7.1.4 Alternative Function Class Implementations

To illustrate class programming further, we will now realize class `Y` from Chapter 7.1.2 in a different way. You may consider this section as advanced and skip it, but for some readers the material might improve the understanding of class `Y` and give some insight into class programming in general.

It is a good habit always to have a constructor in a class and to initialize class attributes here, but this is not a requirement. Let us drop the constructor and make `v0` an optional argument to the `value` method. If the user does not provide `v0` in the call to `value`, we use a `v0` value that must have been provided in an earlier call and stored as an attribute `self.v0`. We can recognize if the user provides `v0` as argument or not by using `None` as default value for the keyword argument and then test if `v0 is None`.

Our alternative implementation of class `Y`, named `Y2`, now reads

```
class Y2:
    def value(self, t, v0=None):
        if v0 is not None:
            self.v0 = v0
        g = 9.81
        return self.v0*t - 0.5*g*t**2
```

This time the class has only one method and one attribute as we skipped the constructor and let `g` be a local variable in the `value` method.

But if there is no constructor, how is an instance created? Python fortunately creates an empty constructor. This allows us to write

```
y = Y2()
```

to make an instance `y`. Since nothing happens in the automatically generated empty constructor, `y` has no attributes at this stage. Writing

```
print y.v0
```

therefore leads to the exception

```
AttributeError: Y2 instance has no attribute 'v0'
```

By calling

```
v = y.value(0.1, 5)
```

we create an attribute `self.v0` insde the `value` method. In general, we can create any attribute `name` in any method by just assigning a value to `self.name`. Now trying a

```
print y.v0
```

will print `5`. In a new call,

```
v = y.value(0.2)
```

the previous `v0` value (`5`) is used inside `value` as `self.v0` unless a `v0` argument is specified in the call.

The previous implementation is not foolproof if we fail to initialize `v0`. For example, the code

```
y = Y2()
v = y.value(0.1)
```

will terminate in the `value` method with the exception

```
AttributeError: Y2 instance has no attribute 'v0'
```

As usual, it is better to notify the user with a more informative message. To check if we have an attribute `v0`, we can use the Python function `hasattr`. Calling `hasattr(self, 'v0')` returns `True` only if the instance `self` has an attribute with name `'v0'`. An improved `value` method now reads

```
    def value(self, t, v0=None):
        if v0 is not None:
            self.v0 = v0
        if not hasattr(self, 'v0'):
            print 'You cannot call value(t) without first '\
                  'calling value(t,v0) to set v0'
            return None
        g = 9.81
        return self.v0*t - 0.5*g*t**2
```

Alternatively, we can try to access `self.v0` in a `try-except` block, and perhaps raise an exception `TypeError` (which is what Python raises if there are not enough arguments to a function or method):

```
    def value(self, t, v0=None):
        if v0 is not None:
            self.v0 = v0
        g = 9.81
        try:
            value = self.v0*t - 0.5*g*t**2
        except AttributeError:
            msg = 'You cannot call value(t) without first '
                  'calling value(t,v0) to set v0'
            raise TypeError(msg)
        return value
```

Note that Python detects an `AttributeError`, but from a user's point of view, not enough parameters were supplied in the call so a `TypeError` is more appropriate to communicate back to the calling code.

We think class `Y` is a better implementation than class `Y2`, because the former is simpler. As already mentioned, it is a good habit to include a constructor and set data here rather than "recording data on the fly" as we try to in class `Y2`. The whole purpose of class `Y2` is just to show that Python provides great flexibility with respect to defining attributes, and that there are no requirements to what a class *must* contain.

7.1.5 Making Classes Without the Class Construct

Newcomers to the class conecpt often have a hard time understanding what this concept is about. The present section tries to explain in more detail how we can introduce classes without having the class construct in the computer language. This information may or may not increase your understanding of classes. If not, programming with classes will definitely increase your understanding with time, so there is no reason to worry. In fact, you may safely jump to Chapter 7.3 as there are no important concepts in this section that later sections build upon.

A class contains a collection of variables (data) and a collection of methods (functions). The collection of variables is unique to each instance of the class. That is, if we make ten instances, each of them has its own set of variables. These variables can be thought of as a dictionary with keys equal to the variable names. Each instance then has its own dictionary, and we may roughly view the instance as this dictionary[5].

On the other hand, the methods are shared among the instances. We may think of a method in a class as a standard global function that takes an instance in the form of a dictionary as first argument. The method has then access to the variables in the instance (dictionary) provided in the call. For the `Y` class from Chapter 7.1.2 and an instance

[5] The instance can also contain static class attributes (Chapter 7.7), but these are to be viewed as global variables in the present context.

y, the methods are ordinary functions with the following names and arguments:

```
Y.value(y, t)
Y.formula(y)
```

The class acts as a *namespace*, meaning that all functions must be prefixed by the namespace name, here `Y`. Two different classes, say `C1` and `C2`, may have functions with the same name, say `value`, but when the `value` functions belong to different namespaces, their names `C1.value` and `C2.value` become distinct. Modules are also namespaces for the functions and variables in them (think of `math.sin`, `cmath.sin`, `numpy.sin`).

The only peculiar thing with the class construct in Python is that it allows us to use an alternative syntax for method calls:

```
y.value(t)
y.formula()
```

This syntax coincides with the traditional syntax of calling class methods and providing arguments, as found in other computer languages, such as Java, C#, C++, Simula, and Smalltalk. The dot notation is also used to access variables in an instance such that we inside a method can write `self.v0` instead of `self['v0']` (`self` refers to `y` through the function call).

We could easily implement a simple version of the class concept without having a class construction in the language. All we need is a dictionary type and ordinary functions. The dictionary acts as the instance, and methods are functions that take this dictionary as the first argument such that the function has access to all the variables in the instance. Our `Y` class could now be implemented as

```
def value(self, t):
    return self['v0']*t - 0.5*self['g']*t**2

def formula(self):
    print 'v0*t - 0.5*g*t**2; v0=%g' % self['v0']
```

The two functions are placed in a module called `Y`. The usage goes as follows:

```
import Y
y = {'v0': 4, 'g': 9.81}   # make an "instance"
y1 = Y.value(y, t)
```

We have no constructor since the initialization of the variables is done when declaring the dictionary `y`, but we could well include some initialization function in the `Y` module

```
def init(v0):
    return {'v0': v0, 'g': 9.81}
```

The usage is now slightly different:

```
import Y
y = Y.init(4)       # make an "instance"
y1 = Y.value(y, t)
```

This way of implementing classes with the aid of a dictionary and a set of ordinary functions actually forms the basis for class implementations in many languages. Python and Perl even have a syntax that demonstrates this type of implementation. In fact, every class instance in Python has a dictionary `__dict__` as attribute, which holds all the variables in the instance. Here is a demo that proves the existence of this dictionary in class `Y`:

```
>>> y = Y(1.2)
>>> print y.__dict__
{'v0': 1.2, 'g': 9.8100000000000005}
```

To summarize: A Python class can be thought of as some variables collected in a dictionary, and a set of functions where this dictionary is automatically provided as first argument such that functions always have full access to the class variables.

First Remark. We have in this section provided a view of classes *from a technical point of view*. Others may view a class as a way of modeling the world in terms of data and operations on data. However, in sciences that employ the language of mathematics, the modeling of the world is usually done by mathematics, and the mathematical structures provide understanding of the problem and structure of programs. When appropriate, mathematical structures can conveniently be mapped on to classes in programs to make the software simpler and more flexible.

Second Remark. The view of classes in this section neglects very important topics such as inheritance and dynamic binding, which we treat in Chapter 9. For more completeness of the present section, we briefly describe how our combination of dictionaries and global functions can deal with inheritance and dynamic binding (but this will not make sense unless you know what inheritance is).

Data inheritance can be obtained by letting a subclass dictionary do an `update` call with the superclass dictionary as argument. In this way all data in the superclass are also available in the subclass dictionary. Dynamic binding of methods is more complicated, but one can think of checking if the method is in the subclass module (using `hasattr`), and if not, one proceeds with checking super class modules until a version of the method is found.

7.2 More Examples on Classes

The use of classes to solve problems from mathematical and physical sciences may not be so obvious. On the other hand, in many administrative programs for managing interactions between objects in the real world the objects themselves are natural candidates for being modeled by classes. Below we give some examples on what classes can be used to model.

7.2.1 Bank Accounts

The concept of a bank account in a program is a good candidate for a class. The account has some data, typically the name of the account holder, the account number, and the current balance. Three things we can do with an account is withdraw money, put money into the account, and print out the data of the account. These actions are modeled by methods. With a class we can pack the data and actions together into a new data type so that one account corresponds to one variable in a program.

Class `Account` can be implemented as follows:

```
class Account:
    def __init__(self, name, account_number, initial_amount):
        self.name = name
        self.no = account_number
        self.balance = initial_amount

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def dump(self):
        s = '%s, %s, balance: %s' % \
            (self.name, self.no, self.balance)
        print s
```

Here is a simple test of how class `Account` can ue used:

```
>>> from classes import Account
>>> a1 = Account('John Olsson', '19371554951', 20000)
>>> a2 = Account('Liz Olsson',  '19371564761', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a2.withdraw(10500)
>>> a1.withdraw(3500)
>>> print "a1's balance:", a1.balance
a1's balance: 13500
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> a2.dump()
Liz Olsson, 19371564761, balance: 9500
```

The author of this class does not want users of the class to operate on the attributes directly and thereby change the name, the account

number, or the balance. The intention is that users of the class should only call the constructor, the `deposit`, `withdraw`, and `dump` methods, and (if desired) inspect the `balance` attribute, but never change it. Other languages with class support usually have special keywords that can restrict access to class attributes and methods, but Python does not. Either the author of a Python class has to rely on correct usage, or a special convention can be used: Any name starting with an underscore represents an attribute that should never be touched or a method that should never be called. One refers to names starting with an underscore as *protected* names. These can be freely used inside methods in the class, but not outside.

In class `Account`, it is natural to protect access to the `name`, `no`, and `balance` attributes by prefixing these names by an underscore. For *reading* only of the `balance` attribute, we provide a new method `get_balance`. The user of the class should now only call the methods in the class and not access any attributes.

The new "protected" version of class `Account`, called `AccountP`, reads

```
class AccountP:
    def __init__(self, name, account_number, initial_amount):
        self._name = name
        self._no = account_number
        self._balance = initial_amount

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def get_balance(self):
        return self._balance

    def dump(self):
        s = '%s, %s, balance: %s' % \
            (self._name, self._no, self._balance)
        print s
```

We can technically access the attributes, but we then break the convention that names starting with an underscore should never be touched outside the class. Here is class `AccountP` in action:

```
>>> a1 = AccountP('John Olsson', '19371554951', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a1.withdraw(3500)
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> print a1._balance       # it works, but a convention is broken
13500
print a1.get_balance()      # correct way of viewing the balance
13500
>>> a1._no = '19371554955'  # this is a "serious crime"
```

Python has a special construct, called *properties*, that can be used to protect attributes from being changed. This is very useful, but the

author considers properties a bit too complicated for this introductory book.

7.2.2 Phone Book

You are probably familiar with the phone book on your mobile phone. The phone book contains a list of persons. For each person you can record the name, telephone numbers, email adress, and perhaps other relevant data. A natural way of representing such personal data in a program is to create a class, say class `Person`. The attributes of the class holds data like the name, mobile phone number, office phone number, private phone number, and email address. The constructor may initialize some of the data about a person. Additional data can be specified later by calling methods in the class. One method can print the data. Other methods can register additional telephone numbers and an email address. In addition we initialize some of the attributes in a constructor method. The attributes that are not initialized when constructing a `Person` instance can be added later by calling appropriate methods. For example, adding an office number is done by calling `add_office_number`.

Class `Person` may look as

```
class Person:
    def __init__(self, name,
                 mobile_phone=None, office_phone=None,
                 private_phone=None, email=None):
        self.name = name
        self.mobile = mobile_phone
        self.office = office_phone
        self.private = private_phone
        self.email = email

    def add_mobile_phone(self, number):
        self.mobile = number

    def add_office_phone(self, number):
        self.office = number

    def add_private_phone(self, number):
        self.private = number

    def add_email(self, address):
        self.email = address
```

Note the use of `None` as default value for various attributes: the object `None` is commonly used to indicate that a variable or attribute is defined, but yet not with a sensible value.

A quick demo session of class `Person` may go as follows:

```
>>> p1 = Person('Hans Hanson',
...             office_phone='767828283', email='h@hanshanson.com')
>>> p2 = Person('Ole Olsen', office_phone='767828292')
>>> p2.add_email('olsen@somemail.net')
>>> phone_book = [p1, p2]
```

It can be handy to add a method for printing the contents of a `Person` instance in a nice fashion:

```
    def dump(self):
        s = self.name + '\n'
        if self.mobile is not None:
            s += 'mobile phone:   %s\n' % self.mobile
        if self.office is not None:
            s += 'office phone:   %s\n' % self.office
        if self.private is not None:
            s += 'private phone:  %s\n' % self.private
        if self.email is not None:
            s += 'email address:  %s\n' % self.email
        print s
```

With this method we can easily print the phone book:

```
>>> for person in phone_book:
...     person.dump()
...
Hans Hanson
office phone:   767828283
email address:  h@hanshanson.com

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

A phone book can be a list of `Person` instances, as indicated in the examples above. However, if we quickly want to look up the phone numbers or email address for a given name, it would be more convenient to store the `Person` instances in a dictionary with the name as key:

```
>>> phone_book = {'Hanson': p1, 'Olsen': p2}
>>> for person in sorted(phone_book):  # alphabetic order
...     phone_book[person].dump()
```

The current example of `Person` objects is extended in Chapter 7.3.5.

7.2.3 A Circle

Geometric figures, such as a circle, are other candidates for classes in a program. A circle is uniquely defined by its center point (x_0, y_0) and its radius R. We can collect these three numbers as attributes in a class. The values of x_0, y_0, and R are naturally initialized in the constructor. Other methods can be `area` and `circumference` for calculating the area πR^2 and the circumference $2\pi R$:

```
class Circle:
    def __init__(self, x0, y0, R):
        self.x0, self.y0, self.R = x0, y0, R

    def area(self):
        return pi*self.R**2

    def circumference(self):
        return 2*pi*self.R
```

An example of using class `Circle` goes as follows:

```
>>> c = Circle(2, -1, 5)
>>> print 'A circle with radius %g at (%g, %g) has area %g' % \
...       (c.R, c.x0, c.y0, c.area())
A circle with radius 5 at (2, -1) has area 78.5398
```

The ideas of class `Circle` can be applied to other geometric objects as well: rectangles, triangles, ellipses, boxes, spheres, etc. Exercise 7.4 tests if you are able to adapt class `Circle` to a rectangle and a triangle.

Remark. There are usually many solutions to a programming problem. Representing a circle is no exception. Instead of using a class, we could collect x_0, y_0, and R in a list and create global functions `area` and `circumference` that take such a list as argument:

```
x0, y0, R = 2, -1, 5
circle = [x0, y0, R]

def area(c):
    R = c[2]
    return pi*R**2

def circumference(c):
    R = c[2]
    return 2*pi*R
```

Alternatively, the circle could be represented by a dictionary with keys `'center'` and `'radius'`:

```
circle = {'center': (2, -1), 'radius': 5}

def area(c):
    R = c['radius']
    return pi*R**2

def circumference(c):
    R = c['radius']
    return 2*pi*R
```

7.3 Special Methods

Some class methods 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 one example. This method is automatically called when an instance is created (by calling the class as a function), but we do not need to explicitly write `__init__`. Other special methods make it possible to perform arithmetic operations with instances, to compare instances with `>`, `>=`, `!=`, etc., to call instances as we call ordinary functions, and to test if an instance is true or false, to mention some possibilities.

7.3.1 The Call Special Method

Computing the value of the mathematical function represented by class `Y` on page 341, with `y` as the name of the instance, is performed by writing `y.value(t)`. If we could write just `y(t)`, the `y` instance would look as 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 class `Y` has the method `__call__` defined. We may easily add this special method:

```
class Y:
    ...
    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

The previous `value` method is now redundant. 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`, `callable(a)` returns `True` if `a` is either a Python function or an instance with a `__call__` method.

In particular, an instance of class `Y` can be passed as the `f` argument to the `diff` function on page 339:

```
y = Y(v0=5)
dydt = diff(y, 0.1)
```

Inside `diff`, we can test that `f` is not a function but an instance of class `Y`. However, we only use `f` in calls, like `f(x)`, and for this purpose an instance with a `__call__` method works as a plain function. This feature is very convenient.

The next section demonstrates a neat application of the call operator `__call__` in a numerical algorithm.

7.3.2 Example: Automagic Differentiation

Problem. Given a Python implementation `f(x)` of a mathematical function $f(x)$, we want to create an object that behaves as a Python function for computing the derivative $f'(x)$. For example, if this object is of type `Derivative`, we should be able to write something like

```
>>> def f(x):
        return x**3
...
>>> dfdx = Derivative(f)
>>> x = 2
>>> dfdx(x)
12.000000992884452
```

That is, `dfdx` behaves as a straight Python function for implementing the derivative $3x^2$ of x^3 (well, the answer is only approximate, with an error in the 7th decimal, but the approximation can easily be improved).

Maple, Mathematica, and many other software packages can do exact symbolic mathematics, including differentiation and integration. A Python package SymPy for symbolic mathematics is free and simple to use, and could easily be applied to calculate the exact derivative of a large class of functions $f(x)$. However, functions that are defined in an algorithmic way (e.g., solution of another mathematical problem), or functions with branches, random numbers, etc., pose fundamental problems to symbolic differentiation, and then numerical differentiation is required. Therefore we base the computation of derivatives in `Derivative` instances on finite difference formulas. This strategy also leads to much simpler code compared to exact symbolic differentiation.

Solution. The most basic, but not the best formula for a numerical derivative is (7.1), which we reuse here for simplicity. The reader can easily switch from this formula to a better one if desired. The idea now is that we make a class to hold the function to be differentiated, call it `f`, and a stepsize `h` to be used in the numerical approximation. These variables can be set in the constructor. The `__call__` operator computes the derivative with aid of the general formula (7.1). All this can be coded as

```
class Derivative:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h      # make short forms
        return (f(x+h) - f(x))/h
```

Note that we turn `h` into a `float` to avoid potential integer division.

Below follows an application of the class to differentiate two functions $f(x) = \sin x$ and $g(t) = t^3$:

```
>>> from math import sin, cos, pi
>>> df = Derivative(sin)
>>> x = pi
>>> df(x)
-1.000000082740371
>>> cos(x)  # exact
-1.0
```

```
>>> def g(t):
...     return t**3
...
>>> dg = Derivative(g)
>>> t = 1
>>> dg(t)  # compare with 3 (exact)
3.000000248221113
```

The expressions `df(x)` and `dg(t)` look as ordinary Python functions that evaluate the derivative of the functions `sin(x)` and `g(t)`. Class `Derivative` works for (almost) any function $f(x)$.

Application. In which situations will it be convenient to automatically produce a Python function `df(x)` which is the derivative of another Python function `f(x)`? One example arises when solving nonlinear algebraic equations $f(x) = 0$ with Newton's method and we, because of laziness, lack of time, or lack of training do not manage to derive $f'(x)$ by hand. Consider the `Newton` function from page 248 for solving $f(x) = 0$. Suppose we want to solve

$$f(x) = 10^5(x - 0.9)^2(x - 1.1)^3 = 0$$

by Newton's method. The function $f(x)$ is plotted in Figure 7.2. The

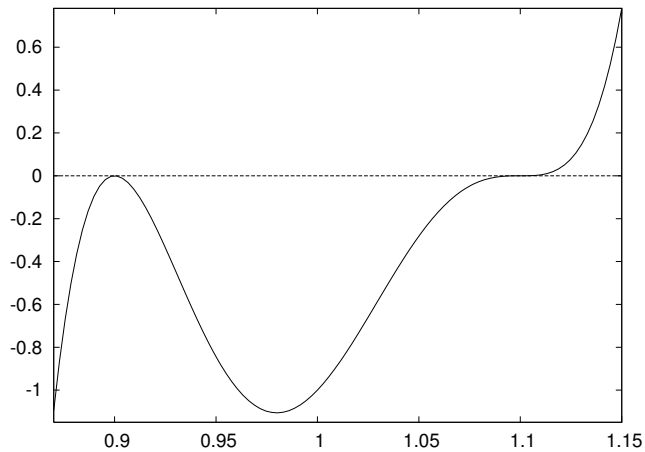

Fig. 7.2 Plot of $y = 10^5(x - 0.9)^2(x - 1.1)^3$.

following session employs the `Derivative` class to quickly make a derivative so we can call Newton's method:

```
>>> from classes import Derivative
>>> from Newton import Newton
>>> def f(x):
...     return 100000*(x - 0.9)**2 * (x - 1.1)**3
...
>>> df = Derivative(f)
>>> Newton(f, 1.01, df, epsilon=1E-5)
(1.0987610068093443, 8, -7.5139644257961411e-06)
```

The output 3-tuple holds the approximation to a root, the number of iterations, and the value of f at the approximate root (a measure of the error in the equation).

The exact root is 1.1, and the convergence toward this value is very slow[6] (for example, an `epsilon` tolerance of 10^{-10} requires 18 iterations with an error of 10^{-3}). Using an exact derivative gives almost the same result:

```
>>> def df_exact(x):
...     return 100000*(2*(x-0.9)*(x-1.1)**3 + \
...                    (x-0.9)**2*3*(x-1.1)**2)
...
>>> Newton(f, 1.01, df_exact, epsilon=1E-5)
(1.0987610065618421, 8, -7.5139689100699629e-06)
```

This example indicates that there are hardly any drawbacks in using a "smart" inexact general differentiation approach as in the `Derivative` class. The advantages are many – most notably, `Derivative` avoids potential errors from possibly incorrect manual coding of possibly lengthy expressions of possibly wrong hand-calculations. The errors in the involved approximations can be made smaller, usually much smaller than other errors, like the tolerance in Newton's method in this example or the uncertainty in physical parameters in real-life problems.

7.3.3 Example: Automagic Integration

We can apply the ideas from Chapter 7.3.2 to make a class for computing the integral of a function numerically. Given a function $f(x)$, we want to compute

$$F(x;a) = \int_a^x f(t)dt\,.$$

The computational technique consists of using the Trapezoidal rule with n intervals ($n+1$ points):

$$\int_a^x f(t)dt = h\left(\frac{1}{2}f(a) + \sum_{i=1}^{n-1} f(a+ih) + \frac{1}{2}f(x)\right), \tag{7.2}$$

where $h = (x-a)/n$. In an application program, we want to compute $F(x;a)$ by a simple syntax like

```
def f(x):
    return exp(-x**2)*sin(10*x)
```

[6] Newton's method converges very slowly when the derivative of f is zero at the roots of f. Even slower convergence appears when higher-order derivatives also are zero, like in this example. Notice that the error in `x` is much larger than the error in the equation (`epsilon`).

```
a = 0; n = 200
F = Integral(f, a, n)
print F(x)
```

Here, `f(x)` is the Python function to be integrated, and `F(x)` behaves as a Python function that calculates values of $F(x;a)$.

A Simple Implementation. Consider a straightforward implementation of the Trapezoidal rule in a Python function:

```
def trapezoidal(f, a, x, n):
    h = (x-a)/float(n)
    I = 0.5*f(a)
    for i in iseq(1, n-1):
        I += f(a + i*h)
    I += 0.5*f(x)
    I *= h
    return I
```

The `iseq` function is an alternative to `range` where the upper limit is included in the set of numbers (see Chapters 4.3.1 and 4.5.6). We can alternatively use `range(1, n)`, but the correspondence with the indices in the mathematical description of the rule is then not completely direct. The `iseq` function is contained in `scitools.std`, so if you make a "star import" from this module, you have `iseq` available.

Class `Integral` must have some attributes and a `__call__` method. Since the latter method is supposed to take `x` as argument, the other parameters `a`, `f`, and `n` must be class attributes. The implementation then becomes

```
class Integral:
    def __init__(self, f, a, n=100):
        self.f, self.a, self.n = f, a, n

    def __call__(self, x):
        return trapezoidal(self.f, self.a, x, self.n)
```

Observe that we just reuse the `trapezoidal` function to perform the calculation. We could alternatively have copied the body of the `trapezoidal` function into the `__call__` method. However, if we already have this algorithm implemented and tested as a function, it is better to call the function. The class is then known as a *wrapper* of the underlying function. A wrapper allows something to be called with alternative syntax. With the `Integral(x)` wrapper we can supply the upper limit of the integral only – the other parameters are supplied when we create an instance of the `Integral` class.

An application program computing $\int_0^{2\pi} \sin x\, dx$ might look as follows:

```
from math import sin, pi

G = Integral(sin, 0, 200)
value = G(2*pi)
```

An equivalent calculation is

```
value = trapezoidal(sin, 0, 2*pi, 200)
```

Remark. Class `Integral` is inefficient (but probably more than fast enough) for plotting $F(x;a)$ as a function x. Exercise 7.22 suggests to optimize the class for this purpose.

7.3.4 Turning an Instance into a String

Another special method is `__str__`. It is called when a class instance needs to be converted to a string. This happens when we say `print a`, and `a` is an instance. Python will then look into the `a` instance for a `__str__` method, which is supposed to return a string. If such a special method is found, the returned string is printed, otherwise just the name of the class is printed. An example will illustrate the point. First we try to print an `y` instance of class `Y` from Chapter 7.1.2 (where there is no `__str__` method):

```
>>> print y
<__main__.Y instance at 0xb751238c>
```

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

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

```
class Y:
    ...
    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

Typically, `__str__` replaces our previous `formula` method and `__call__` replaces our previous `value` method. Python programmers with the experience that we now have gained will therefore write class `Y` with special methods only:

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2

    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

Let us see the class in action:

```
>>> y = Y(1.5)
>>> y(0.2)
0.1038
>>> print y
v0*t - 0.5*g*t**2; v0=1.5
```

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

7.3.5 Example: Phone Book with Special Methods

Let us reconsider class `Person` from Chapter 7.2.2. The `dump` method in that class is better implemented as a `__str__` special method. This is easy: We just change the method name and replace `print s` by `return s`.

Storing `Person` instances in a dictionary to form a phone book is straightforward. However, we make the dictionary a bit easier to use if we wrap a class around it. That is, we make a class `PhoneBook` which holds the dictionary as an attribute. An `add` method can be used to add a new person:

```
class PhoneBook:
    def __init__(self):
        self.contacts = {}   # dict of Person instances

    def add(self, name, mobile=None, office=None,
            private=None, email=None):
        p = Person(name, mobile, office, private, email)
        self.contacts[name] = p
```

A `__str__` can print the phone book in alphabetic order:

```
    def __str__(self):
        s = ''
        for p in sorted(self.contacts):
            s += str(self.contacts[p])
        return s
```

To retrieve a `Person` instance, we use the `__call__` with the person's name as argument:

```
    def __call__(self, name):
        return self.contacts[name]
```

The only advantage of this method is simpler syntax: For a `PhoneBook` `b` we can get data about `NN` by calling `b('NN')` rather than accessing the internal dictionary `b.contacts['NN']`.

We can make a simple test function for a phone book with three names:

```
b = PhoneBook()
b.add('Ole Olsen', office='767828292',
      email='olsen@somemail.net')
b.add('Hans Hanson',
      office='767828283', mobile='995320221')
b.add('Per Person', mobile='906849781')
print b('Per Person')
print b
```

The output becomes

```
Per Person
mobile phone:   906849781

Hans Hanson
mobile phone:   995320221
office phone:   767828283

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

Per Person
mobile phone:   906849781
```

You are strongly encouraged to work through this last demo program by hand and simulate what the program does. That is, jump around in the code and write down on a piece of paper what various variables contain after each statement. This is an important and good exercise! You enjoy the happiness of mastering classes if you get the same output as above. The complete program with classes `Person` and `PhoneBook` and the test above is found in the file `phone_book.py`. You can run this program, statement by statement, in a debugger (see Appendix D.1) to control that your understanding of the program flow is correct.

Remark. Note that the names are sorted with respect to the first names. The reason is that strings are sorted after the first character, then the second character, and so on. We can supply our own tailored sort function, as explained in Exercise 2.44. One possibility is to split the name into words and use the last word for sorting:

```
def last_name_sort(name1, name2):
    lastname1 = name1.split()[-1]
    lastname2 = name2.split()[-1]
    if lastname1 < lastname2:
        return -1
    elif lastname1 > lastname2:
        return 1
    else: # equality
        return 0

for p in sorted(self.contacts, last_name_sort):
   ...
```

7.3.6 Adding Objects

Let `a` and `b` be instances of some class `C`. Does it make sense to write `a + b`? Yes, this makes sense if class `C` has defined a special method `__add__`:

```
class C:
    ...
    __add__(self, other):
        ...
```

The `__add__` method should add the instances `self` and `other` and return the result as an instance. So when Python encounters `a + b`, it will check if class `C` has an `__add__` method and interpret `a + b` as the call `a.__add__(b)`. The next example will hopefully clarify what this idea can be used for.

7.3.7 Example: Class for Polynomials

Let us create a class `Polynomial` for polynomials. The coefficients in the polynomial can be given to the constructor as a list. Index number i in this list represents the coefficients of the x^i term in the polynomial. That is, writing `Polynomial([1,0,-1,2])` defines a polynomial

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

Polynomials can be added (by just adding the coefficients) so our class may have an `__add__` method. A `__call__` method is natural for evaluating the polynomial, given a value of x. The class is listed below and explained afterwards.

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # start with the longest list and add in the other:
        if len(self.coeff) > len(other.coeff):
            sum_coeff = self.coeff[:]  # copy!
            for i in range(len(other.coeff)):
                sum_coeff[i] += other.coeff[i]
        else:
            sum_coeff = other.coeff[:] # copy!
            for i in range(len(self.coeff)):
                sum_coeff[i] += self.coeff[i]
        return Polynomial(sum_coeff)
```

Implementation. Class `Polynomial` has one attribute: the list of coefficients. To evaluate the polynomial, we just sum up coefficient no. i times x^i for $i = 0$ to the number of coefficients in the list.

The `__add__` method looks more advanced. The idea is to add the two lists of coefficients. However, it may happen that the lists are of unequal length. We therefore start with the longest list and add in the other list, element by element. Observe that `sum_coeff` starts out as a *copy* of `self.coeff`: If not, changes in `sum_coeff` as we compute the sum will be reflected in `self.coeff`. This means that `self` would be the sum of itself and the `other` instance, or in other words, adding two instances, `p1+p2`, changes `p1` – this is not what we want! An alternative implementation of class `Polynomial` is found in Exercise 7.32.

A subtraction method `__sub__` can be implemented along the lines of `__add__`, but is slightly more complicated and left to the reader through Exercise 7.33. A somewhat more complicated operation, from a mathematical point of view, 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 the two polynomials. The product becomes

$$\left(\sum_{i=0}^{M} c_i x^i\right)\left(\sum_{j=0}^{N} d_j x^j\right) = \sum_{i=0}^{M}\sum_{j=0}^{N} c_i d_j x^{i+j}\,.$$

The double sum must be implemented as a double loop, but first the list for the resulting polynomial must be created with length $M+N+1$ (the highest exponent is $M+N$ and then we need a constant term). The implementation of the multiplication operator becomes

```
def __mul__(self, other):
    c = self.coeff
    d = other.coeff
    M = len(c) - 1
    N = len(d) - 1
    result_coeff = zeros(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)
```

We could also include a method for differentiating the polynomial according to the formula

$$\frac{d}{dx}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}\,.$$

If c_i is stored as a list `c`, the list representation of the derivative, say its name is `dc`, fulfills `dc[i-1] = i*c[i]` for `i` running from 1 to the largest index in `c`. Note that `dc` has one element less than `c`.

There are two different ways of implementing the differentiation functionality, either by changing the polynomial coefficients, or by re-

turning a new `Polynomial` instance from the method such that the original polynomial instance is intact. We let `p.differentiate()` be an implementation of the first approach, i.e., this method does not return anything, but the coefficients in the `Polynomial` instance `p` are altered. The other approach is implemented by `p.derivative()`, which returns a new `Polynomial` object with coefficients corresponding to the derivative of `p`.

The complete implementation of the two methods is given below:

```
    def differentiate(self):
        """Differentiate this polynomial in-place."""
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]

    def derivative(self):
        """Copy this polynomial and return its derivative."""
        dpdx = Polynomial(self.coeff[:])  # make a copy
        dpdx.differentiate()
        return dpdx
```

The `Polynomial` class with a `differentiate` method and not a `derivative` method would be mutable (see Chapter 6.2.3) and allow in-place changes of the data, while the `Polynomial` class with `derivative` and not `differentiate` would yield an immutable object where the polynomial initialized in the constructor is never altered[7]. A good rule is to offer only one of these two functions such that a `Polynomial` object is either mutable or immutable (if we leave out `differentiate`, its function body must of course be copied into `derivative` since `derivative` now relies on that code). However, since the main purpose of this class is to illustrate various types of programming techniques, we keep both versions.

Usage. As a demonstration of the functionality of class `Polynomial`, we introduce the two polynomials

$$p_1(x) = 1 - x, \quad p_2(x) = x - 6x^4 - x^5\,.$$

```
>>> p1 = Polynomial([1, -1])
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print p3.coeff
[1, 0, 0, 0, -6, -1]
>>> p4 = p1*p2
>>> print p4.coeff
[0, 1, -1, 0, -6, 5, 1]
>>> p5 = p2.derivative()
```

[7] Technically, it is possible to grab the `coeff` variable in a class instance and alter this list. By starting `coeff` with an underscore, a Python programming convention tells programmers that this variable is for internal use in the class only, and not to be altered by users of the instance, see Chapters 7.2.1 and 7.6.2.

```
>>> print p5.coeff
[1, 0, 0, -24, -5]
```

One verification of the implementation may be to compare `p3` at (e.g.) $x = 1/2$ with $p_1(x) + p_2(x)$:

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

Note that `p1 + p2` is very different from `p1(x) + p2(x)`. In the former case, we add two instances of class `Polynomial`, while in the latter case we add two instances of class `float` (since `p1(x)` and `p2(x)` imply calling `__call__` and that method returns a `float` object).

Pretty Print of Polynomials. The `Polynomial` class can also be equipped with a `__str__` method for printing the polynomial to the screen. A first, rough implementation could simply add up strings of the form `+ self.coeff[i]*x^i`:

```
class Polynomial:
    ...
    def __str__(self):
        s = ''
        for i in range(len(self.coeff)):
            s += ' + %g*x^%d' % (self.coeff[i], i)
        return s
```

However, this implementation leads to ugly output from a mathematical viewpoint. For instance, a polynomial with coefficients `[1,0,0,-1,-6]` gets printed as

```
+ 1*x^0 + 0*x^1 + 0*x^2 + -1*x^3 + -6*x^4
```

A more desired output would be

```
1 - x^3 - 6*x^4
```

That is, terms with a zero coefficient should be dropped; a part `'+ -'` of the output string should be replaced by `'- '`; unit coefficients should be dropped, i.e., `' 1*'` should be replaced by space `' '`; unit power should be dropped by replacing `'x^1 '` by `'x '`; zero power should be dropped and replaced by `1`, initial spaces should be fixed, etc. 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. If you find this type of string manipulation tricky and difficult to understand, you may safely skip further inspection of the improved `__str__` code since the details are not essential for your present learning about the class concept and special methods.

```
class Polynomial:
    ...
    def __str__(self):
        s = ''
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += ' + %g*x^%d' % (self.coeff[i], i)
        # fix layout:
        s = s.replace('+ -', '- ')
        s = s.replace('x^0', '1')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^1 ', 'x ')
        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
```

Programming sometimes turns into coding (what one think is) a general solution followed by a series of special cases to fix caveats in the "general" solution, just as we experienced with the `__str__` method above. This situation often calls for additional future fixes and is often a sign of a suboptimal solution to the programming problem.

Pretty print of `Polynomial` instances can be demonstrated in an interactive session:

```
>>> p1 = Polynomial([1, -1])
>>> print p1
1 - x^1
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p2.differentiate()
>>> print p2
1 - 24*x^3 - 5*x^4
```

7.3.8 Arithmetic Operations and Other Special Methods

Given two instances `a` and `b`, the standard binary arithmetic operations with `a` and `b` are defined by the following special methods:

- `a + b` : `a.__add__(b)`
- `a - b` : `a.__sub__(b)`
- `a*b` : `a.__mul__(b)`
- `a/b` : `a.__div__(b)`
- `a**b` : `a.__pow__(b)`

Some other special methods are also often useful:

- the length of `a`, `len(a)`: `a.__len__()`
- the absolute value of `a`, `abs(a)`: `a.__abs__()`
- `a == b` : `a.__eq__(b)`
- `a > b` : `a.__gt__(b)`
- `a >= b` : `a.__ge__(b)`
- `a < b` : `a.__lt__(b)`

- `a <= b` : `a.__le__(b)`
- `a != b` : `a.__ne__(b)`
- `-a` : `a.__neg__()`
- evaluating `a` as a boolean expression (as in the test `if a:`) implies calling the special method `a.__bool__()`, which must return `True` or `False` – if `__bool__` is not defined, `__len__` is called to see if the length is zero (`False`) or not (`True`)

We can implement such methods in class `Polynomial`, see Exercise 7.33. Chapter 7.5 contains many examples on using the special methods listed above.

7.3.9 More on Special Methods for String Conversion

Look at this class with a `__str__` method:

```
>>> class MyClass:
...     def __init__(self):
...         self.data = 2
...     def __str__(self):
...         return 'In __str__: %s' % str(self.data)
...
>>> a = MyClass()
>>> print a
In __str__: 2
```

Hopefully, you understand well why we get this output (if not, go back to Chapter 7.3.4).

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

```
>>> a
<__main__.MyClass instance at 0xb75125ac>
```

When writing just `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 represention of the contents of the instance. For a lot of Python classes, including `int`, `float`, `complex`, `list`, `tuple`, and `dict`, `__repr__` and `__str__` give identical output. In our class `MyClass` the `__repr__` is missing, and we need to add it if we want

```
>>> a
```

to write the contents like `print a` does.

Given an instance `a`, `str(a)` implies calling `a.__str__()` and `repr(a)` implies calling `a.__repr__()`. This means that

```
>>> 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

```
def __repr__(self):
    return self.__str__()  # or return str(self)
```

However, as we explain below, the `__repr__` is best defined differently.

Recreating Objects from Strings. The Python function `eval(e)` evaluates a valid Python expression contained in the string `e`, see Chapter 3.1.2. It is a convention that `__repr__` returns a string such that `eval` applied to the string recreates the instance. For example, in case of the `Y` class from page 341, `__repr__` should return `'Y(10)'` if the `v0` 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 an interactive session.

Below we show examples of `__repr__` methods in classes `Y` (page 341), `Polynomial` (page 365), and `MyClass` (above):

```
class Y:
    ...
    def __repr__(self):
        return 'Y(v0=%s)' % self.v0

class Polynomial:
    ...
    def __repr__(self):
        return 'Polynomial(coefficients=%s)' % self.coeff

class MyClass:
    ...
    def __repr__(self):
        return 'MyClass()'
```

With these definitions, `eval(repr(x))` recreates the object `x` if it is of one of the three types above. In particular, we can write `x` to file and later recreate the `x` from the file information:

```
# somefile is some file object
somefile.write(repr(x))
somefile.close()
...
data = somefile.readline()
x2 = eval(data)  # recreate object
```

Now, `x2` will be equal to `x` (`x2 == x` evaluates to true).

7.4 Example: Solution of Differential Equations

An ordinary differential equation (ODE), where the unknown is a function $u(t)$, can be written in the generic form

$$u'(t) = f(u(t), t)\,. \tag{7.3}$$

In addition, an initial condition, $u(0) = u_0$, must be associated with this ODE to make the solution of (7.3) unique. The function f reflects an expression with u and/or t. Some important examples of ODEs and their corresponding forms of f are given below.

1. Exponential growth of money or populations:

$$f(u, t) = \alpha u, \tag{7.4}$$

   where α is a given constant expressing the growth rate of u.
2. Logistic growth of a population under limited resources:

$$f(u, t) = \alpha u \left(1 - \frac{u}{R}\right), \tag{7.5}$$

   where α is the initial growth rate and R is the maximum possible value of u.
3. Radioactive decay of a substance:

$$f(u, t) = -au, \tag{7.6}$$

   where a is the rate of decay of u.
4. Body falling in a fluid:

$$f(u, t) = -b|u|u + g, \tag{7.7}$$

   where b models the fluid resistance, g is the acceleration of gravity, and u is the body's velocity (see Exercise 7.25 on page 405).
5. Newton's law of cooling:

$$f(u, t) = -h(u - s), \tag{7.8}$$

   where u is the temperature of a body, h is a heat transfer coefficient between the body and its surroundings, and s is the temperature of the surroundings.

Appendix B gives an introduction to ODEs and their numerical solution, and you should be familiar with that or similar material before reading on.

The purpose of the present section is to design and implement a class for the general ODE $u' = f(u, t)$. You need to have digested the material about classes in Chapters 7.1.2, 7.3.1, and 7.3.2.

7.4.1 A Function for Solving ODEs

A very simple solution method for a general ODE on the form (7.3) is the Forward Euler method:

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)\,. \tag{7.9}$$

Here, u_k denotes the numerical approximation to the exact solution u at time t_k, Δt is a time step, and if all time steps are equal, we have that $t_k = k\Delta t$, $k = 0, \ldots, n$.

First we will implement the method (7.9) in a simple program, tailored to the specific ODE $u' = u$ (i.e., $f(u,t) = u$ in (7.3)). Our goal is to compute $u(t)$ for $t \in [0, T]$. How to perform this computation is explained in detail in Appendix B.2. Here, we follow the same approach, but using lists instead of arrays to store the computed $u_0, \ldots, u_n$ and $t_0, \ldots, t_n$ values. The reason is that lists are more dynamical if we later introduce more sophisticated solution methods where Δt may change during the solution so that the value of n (i.e., length of arrays) is not known on beforehand. Let us start with sketching a "flat program":

```
# Integrate u'=u, u(0)=u0, in steps of dt until t=T
u0 = 1
T = 3
dt = 0.1

u = []; t = []   # u[k] is the solution at time t[k]

u.append(u0)
t.append(0)
n = int(round(T/dt))
for k in range(n):
    unew = u[k] + dt*u[k]

    u.append(unew)
    tnew = t[-1] + dt
    t.append(tnew)
from scitools.std import plot
plot(t, u)
```

The new u_{k+1} and t_{k+1} values, stored in the `unew` and `tnew` variables, are appended to the `u` and `t` lists in each pass in the loop over `k`.

Unfortunately, the code above can only be applied to a specific ODE using a specific numerical method. An obvious improvement is to make a *reusable function* for solving a *general ODE*. An ODE is specified by its right-hand side function $f(u,t)$. We also need the parameters u_0, Δt, and T to perform the time stepping. The function should return the computed $u_0, \ldots, u_n$ and $t_0, \ldots, t_n$ values as two separate arrays to the calling code. These arrays can then be used for plotting or data analysis. An appropriate function may look like

```
def ForwardEuler(f, dt, u0, T):
    """Integrate u'=f(u,t), u(0)=u0, in steps of dt until t=T."""
    u = []; t = []   # u[k] is the solution at time t[k]
    u.append(u0)
```

```
    t.append(0)
    n = int(round(T/dt))
    for k in range(n):
        unew = u[k] + dt*f(u[k], t[k])

        u.append(unew)
        tnew = t[-1] + dt
        t.append(tnew)
    return numpy.array(u), numpy.array(t)
```

Here, `f(u, t)` is a Python implementation of $f(u,t)$ that the user must supply. For example, we may solve $u' = u$ for $t \in (0,3)$, with $u(0) = 1$, and $\Delta t = 0.1$ by the following code utilizing the shown `ForwardEuler` function:

```
def f(u, t):
    return u

u0 = 1
T = 3
dt = 0.1
u, t = ForwardEuler(f, dt, u0, T)

# compare numerical solution and exact solution in a plot:
from scitools.std import plot, exp
u_exact = exp(t)
plot(t, u, 'r-', t, u_exact, 'b-',
     xlabel='t', ylabel='u', legend=('numerical', 'exact'),
     title="Solution of the ODE u'=u, u(0)=1")
```

Observe how easy it is to plot `u` versus `t` and also add the exact solution $u = e^t$ for comparison.

7.4.2 A Class for Solving ODEs

Instead of having the numerical method for solving a general ODE implemented as a function, we now want a class for this purpose. Say the name of the class is `ForwardEuler`. To solve an ODE specified by a Python function `f(u, t)`, from time `t0` to some time `T`, with steps of size `dt`, and initial condition `u0` at time `t0`, it seems convenient to write the following lines of code:

```
method = ForwardEuler(f, dt)
method.set_initial_condition(u0, t0)
u, t = method.solve(T)
```

The constructor of the class stores `f` and the time step `dt`. Then there are two basic steps: setting the initial condition, and calling `solve` to advance the solution to a specified time level. Observe that we do not force the initial condition to appear at $t = 0$, but at some arbitrary time. This makes the code more general, and in particular, we can call the `solve` again to advance the solution further in time, say to $2T$:

```
method.set_initial_condition(u[-1], t[-1])
u2, t2 = method.solve(2*T)
plot(t, u, 'r-', t2, u2, 'r-')
```

The initial condition of this second simulation is the final `u` and `t` values of the first simulation. To plot the complete solution, we just plot the individual simulations.

The task now is to write a class `ForwardEuler` that allow this type of user code. Much of the code from the `ForwardEuler` function above can be reused, but it is reorganized into smaller pieces in a class. Such reorganization is known as refactoring (see also Chapter 3.6.2). An attempt to write the class appears below.

```
class ForwardEuler:
    """
    Class for solving an ODE,

      du/dt = f(u, t)

    by the ForwardEuler method.

    Class attributes:
    t: array of time values
    u: array of solution values (at time points t)
    k: step number of the most recently computed solution
    f: callable object implementing f(u, t)
    dt: time step (assumed constant)
    """
    def __init__(self, f, dt):
        self.f, self.dt = f, dt

    def set_initial_condition(self, u0, t0=0):
        self.u = []    # u[k] is solution at time t[k]
        self.t = []    # time levels in the solution process

        self.u.append(float(u0))
        self.t.append(float(t0))
        self.k = 0  # time level counter

    def solve(self, T):
        """Advance solution in time until t <= T."""
        tnew = 0
        while tnew <= T:
            unew = self.advance()
            self.u.append(unew)
            tnew = self.t[-1] + self.dt
            self.t.append(tnew)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)

    def advance(self):
        """Advance the solution one time step."""
        # load attributes into local variables to
        # obtain a formula that is as close as possible
        # to the mathematical notation:
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]

        unew = u[k] + dt*f(u[k], t)
        return unew
```

We see that we initialize two lists for the u_k and t_k values at the time we set the initial condition. The `solve` method implements the time loop as in the `ForwardEuler` function. However, inside the time loop, a new u_{k+1} value (`unew`) is computed by calling another class method, `self.advance`, which here implements the numerical method (7.9).

Changing the numerical method is just a matter of changing the `advance` function only. We could, of course, put the numerical updating formula explicitly inside the `solve` method, but changing the numerical method would then imply editing internals of a method rather than replacing a complete method. The latter task is considered less error-prone and therefore a better programming strategy.

7.4.3 Verifying the Implementation

We need a problem where the exact solution is known to check the correctness of class `ForwardEuler`. Preferably, we should have a problem where the numerical solution is exact such that we avoid dealing with approximation errors in the Forward Euler method. It turns out that if the solution $u(t)$ is linear in t, then the Forward Euler method will reproduce this solution exactly. Therefore, we choose $u(t) = at + u_0$, with (e.g.) $a = 0.2$ and $u_0 = 3$. The corresponding f is the derivative of u, i.e., $f(u, t) = a$. This is obviously a very simple right-hand side without any u or t. We can make f more complicated by adding something that is zero, e.g., some expression with $u - at - u_0$, say $(u - at - u_0)^4$, so that $f(u, t) = a + (u - at - u_0)^4$.

We implement our special f and the exact solution in two functions `_f1` and `_u_solution_f1`:

```
def _f1(u, t):
    return 0.2 + (u - _u_solution_f1(t))**4

def _u_solution_f1(t):
    return 0.2*t + 3
```

Testing the code is now a matter of performing the steps

```
u0 = 3
dt = 0.4
T = 3
method = ForwardEuler(_f1, dt)
method.set_initial_condition(u0, 0)
u, t = method.solve(T)
u_exact = _f1_solution(t)
print 'Numerical:\n', u
print 'Exact:', '\n', u_exact
```

The output becomes

```
Numerical:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56  3.64]
Exact:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56  3.64]
```

showing that the code works as it should in this example.

7.4.4 Example: Logistic Growth

A more exciting application is to solve the logistic equation (B.23),

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right),$$

with the $f(u,t)$ function specified in (7.5).

First we may create a class for holding information about this problem: α, R, the initial condition, and the right-hand side. We may also add a method for printing the equation and initial condition. This problem class can then be expressed as

```
class Logistic:
    """Problem class for a logistic ODE."""
    def __init__(self, alpha, R, u0):
        self.alpha, self.R, self.u0 = alpha, float(R), u0

    def __call__(self, u, t):
        """Return f(u,t) for the logistic ODE."""
        return self.alpha*u*(1 - u/self.R)

    def __str__(self):
        """Return ODE and initial condition."""
        return "u'(t) = %g*u*(1 - u/%g)\nu(0)=%g" % \
               (self.alpha, self.R, self.u0)
```

Running a case with $\alpha = 0.2$, $R = 1$, $u(0) = 0.1$, $\Delta t = 0.1$, and simulating up to time $T = 40$, can be performed in the following function:

```
def logistic():
    problem = Logistic(0.2, 1, 0.1)
    T = 40
    dt = 0.1
    method = ForwardEuler(problem, dt)
    method.set_initial_condition(problem.u0, 0)
    u, t = method.solve(T)

    from scitools.std import plot, hardcopy, xlabel, ylabel, title
    plot(t, u)
    xlabel('t'); ylabel('u')
    title('Logistic growth: alpha=0.2, dt=%g, %d steps' \
          % (dt, len(u)-1))
```

The resulting plot is shown in Figure 7.3. Note one aspect of this function: the "star import", as in `from scitools.std import *`, is not allowed *inside* a function (or class method for that sake), so we need to explicitly list all the functions we need to import. (We could, as in the previous example, just import `plot` and rely on keyword arguments to set the labels, title, and output file.)

The `ForwardEuler` class is further developed in Chapter 9.4, where it is shown how we can easily modify the class to implement other numerical methods. In that chapter we extend the implementation to systems of ODEs as well.

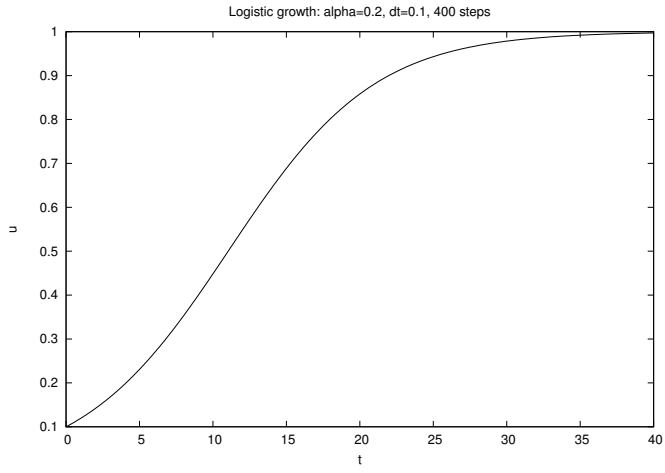

Fig. 7.3 Plot of the solution of the ODE problem $u' = 0.2u(1-u)$, $u(0) = 0.1$.

7.5 Example: Class for Vectors in the Plane

This section explains how to implement two-dimensional vectors in Python such that these vectors act as objects we can add, subtract, form inner products with, and do other mathematical operations on. To understand the forthcoming material, it is necessary to have digested Chapter 7.3, in particular Chapters 7.3.6 and 7.3.8.

7.5.1 Some Mathematical Operations on Vectors

Vectors in the plane are described by a pair of real numbers, (a, b). In Chapter 4.1.2 we presented mathematical rules for adding and subtracting vectors, multiplying two vectors (the inner or dot or scalar product), the length of a vector, and multiplication by a scalar:

$$(a, b) + (c, d) = (a + c, b + d), \tag{7.10}$$

$$(a, b) - (c, d) = (a - c, b - d), \tag{7.11}$$

$$(a, b) \cdot (c, d) = ac + bd, \tag{7.12}$$

$$||(a, b)|| = \sqrt{(a, b) \cdot (a, b)}\,. \tag{7.13}$$

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

7.5.2 Implementation

We may create a class for plane vectors where the above mathematical operations are implemented by special methods. The class must contain two attributes, one for each component of the vector, called `x` and `y` below. We include special methods for addition, subtraction, the scalar

product (multiplication), the absolute value (length), comparison of two vectors (== and !=), as well as a method for printing out a vector.

```
class Vec2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vec2D(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vec2D(self.x - other.x, self.y - other.y)

    def __mul__(self, other):
        return self.x*other.x + self.y*other.y

    def __abs__(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)

    def __ne__(self, other):
        return not self.__eq__(other)  # reuse __eq__
```

The `__add__`, `__sub__`, `__mul__`, `__abs__`, and `__eq__` methods should be quite straightforward to understand from the previous mathematical definitions of these operations. The last method deserves a comment: Here we simply reuse the equality operator `__eq__`, but preceed it with a `not`. We could also have implemented this method as

```
    def __ne__(self, other):
        return self.x != other.x or self.y != other.y
```

Nevertheless, this implementation requires us to write more, and it has the danger of introducing an error in the logics of the boolean expressions. A more reliable approach, when we know that the `__eq__` method works, is to reuse this method and observe that "`not ==`" gives us the effect of "`!=`".

A word of warning is in place regarding our implementation of the equality operator (== via `__eq__`). We test for equality of each component, which is correct from a mathematical point of view. However, each vector component is a floating-point number that 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 be different in their inexact representations on the computer. The remedy for this problem is to avoid testing for equality, but instead check that the difference between the components is sufficiently small. The function `float_eq` found in the module `scitools.numpytutils` (if you do not already have `float_eq` from a `from`

`scitools.std import *`), see also Exercise 2.51, is an easy-to-use tool for comparing `float` objects. With this function we replace

```
if a == b:
```

by

```
if float_eq(a, b):
```

A more reliable equality operator can now be implemented:

```
class Vec2D:
    ...
    def __eq__(self, other):
        return float_eq(self.x, other.x) and \
               float_eq(self.y, other.y)
```

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

The special method `__len__` could be introduced as a synonym for `__abs__`, i.e., for a `Vec2D` instance named `v`, `len(v)` is the same as `abs(v)`, because the absolute value of a vector is mathematically the same as the length of the vector. However, if we implement

```
    def __len__(self):
        # reuse implementation of __abs__
        return abs(self)  # equiv. to self.__abs__()
```

we will run into trouble when we compute `len(v)` and the answer is (as usual) a `float`. Python will then complain and tell us that `len(v)` must return an `int`. Therefore, `__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)` mean the number of components in the vector:

```
    def __len__(self):
        return 2
```

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

7.5.3 Usage

Let us play with some `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)
>>> a = u*v
>>> print a
0
>>> 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, that 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 class methods. As an example, let us consider the expression `u != v`. This is a boolean expression that is true since `u` and `v` are different vectors. The resulting object type should be `bool`, with values `True` or `False`. This is confirmed by the output in the interactive session above. The Python calculation of `u != v` leads to a call to

```
u.__ne__(v)
```

which leads to a call to

```
u.__eq__(v)
```

The result of this last call is `False`, because the special method will evaluate the boolean expression

```
0 == 1 and 1 == 0
```

which is obviously `False`. When going back to the `__ne__` method, we end up with a return of `not False`, which evaluates to `True`. You need this type of thorough understanding to find and correct bugs (and remember that the first versions of your programs will normally contain bugs!).

Comment. For real computations with vectors in the plane, you would probably just use a Numerical Python array of length 2. However, one thing such objects cannot do is evaluating `u*v` as a scalar product. The multiplication operator for Numerical Python arrays is not defined as a scalar product (it is rather defined as $(a, b) \cdot (c, d) = (ac, bd)$). Another difference between our `Vec2D` class and Numerical Python arrays is the `abs` function, which computes the length of the vector in class `Vec2D`, while it does something completely different with Numerical Python arrays.

7.6 Example: Class for Complex Numbers

Imagine that Python did not already have complex numbers. We could then make a class for such numbers and support the standard mathematical operations. This exercise turns out to be a very good pedagogical example of programming with classes and special methods.

The class must contain two attributes: the real and imaginary part of the complex number. In addition, we would like to add, subtract, multiply, and divide complex numbers. We would also like to write out a complex number in some suitable format. A session involving our own complex numbers may take the form

```
>>> u = Complex(2,-1)
>>> v = Complex(1)     # zero imaginary part
>>> w = u + v
>>> print w
(3, -1)
>>> w != u
True
>>> u*v
Complex(2, -1)
>>> u < v
illegal operation "<" for complex numbers
>>> print w + 4
(7, -1)
>>> print 4 - w
(1, 1)
```

We do not manage to use exactly the same syntax with `j` as imaginary unit as in Python's built-in complex numbers so to specify a complex number we must create a `Complex` instance.

7.6.1 Implementation

Here is the complete implementation of our class for complex numbers:

```
class Complex:
    def __init__(self, real, imag=0.0):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real + other.real,
                       self.imag + other.imag)

    def __sub__(self, other):
        return Complex(self.real - other.real,
                       self.imag - other.imag)

    def __mul__(self, other):
        return Complex(self.real*other.real - self.imag*other.imag,
                       self.imag*other.real + self.real*other.imag)

    def __div__(self, other):
        sr, si, or, oi = self.real, self.imag, \
                         other.real, other.imag # short forms
        r = float(or**2 + oi**2)
```

```
        return Complex((sr*or+si*oi)/r, (si*or-sr*oi)/r)

    def __abs__(self):
        return sqrt(self.real**2 + self.imag**2)

    def __neg__(self):   # defines -c (c is Complex)
        return Complex(-self.real, -self.imag)

    def __eq__(self, other):
        return self.real == other.real and self.imag == other.imag

    def __ne__(self, other):
        return not self.__eq__(other)

    def __str__(self):
        return '(%g, %g)' % (self.real, self.imag)

    def __repr__(self):
        return 'Complex' + str(self)

    def __pow__(self, power):
        raise NotImplementedError\
              ('self**power is not yet impl. for Complex')
```

The special methods for addition, subtraction, multiplication, division, and the absolute value follow easily from the mathematical definitions of these operations for complex numbers (see Chapter 1.6). What `-c` means when `c` is of type `Complex`, is also easy to define and implement. The `__eq__` method needs a word of caution: The method is mathematically correct, but as we stated on page 379, comparison of real numbers on a computer should always employ a tolerance. The version of `__eq__` shown above is more about compact code and equivalence to the mathematics than real-world numerical computations.

The final `__pow__` method exemplifies a way to introduce a method in a class, while we postpone its implementation. The simplest way to do this is by inserting an empty function body using the `pass` ("do nothing") statement:

```
    def __pow__(self, power):
        # postpone implementation of self**power
        pass
```

However, the preferred method is to raise a `NotImplementedError` exception so that users writing power expressions are notified that this operation is not available. The simple `pass` will just silently bypass this serious fact!

7.6.2 Illegal Operations

Some mathematical operations, like the comparison operators `>`, `>=`, etc., do not have a meaning for complex numbers. By default, Python allows us to use these comparison operators for our `Complex` instances, but the boolean result will be mathematical nonsense. Therefore, we

should implement the corresponding special methods and give a sensible error message that the operations are not available for complex numbers. Since the messages are quite similar, we make a separate method to gather common operations:

```
    def _illegal(self, op):
        print 'illegal operation "%s" for complex numbers' % op
```

Note the underscore prefix: This is a Python convention telling that the `_illegal` method is local to the class in the sense that it is not supposed to be used outside the class, just by other class methods. In computer science terms, we say that names starting with an underscore are not part of the *application programming interface*, known as the API. Other programming languages, such as Java, C++, and C#, have special keywords, like `private` and `protected` that can be used to technically hide both data and methods from users of the class. Python will never restrict anybody who tries to access data or methods that are considered private to the class, but the leading underscore in the name reminds any user of the class that she now touches parts of the class that are not meant to be used "from the outside".

Various special methods for comparison operators can now call up `_illegal` to issue the error message:

```
    def __gt__(self, other):  self._illegal('>')
    def __ge__(self, other):  self._illegal('>=')
    def __lt__(self, other):  self._illegal('<')
    def __le__(self, other):  self._illegal('<=')
```

7.6.3 Mixing Complex and Real Numbers

The implementation of class `Complex` is far from perfect. Suppose we add a complex number and a real number, which is a mathematically perfectly valid operation:

```
w = u + 4.5
```

This statement leads to an exception,

```
AttributeError: 'float' object has no attribute 'real'
```

In this case, Python sees `u + 4.5` and tries to use `u.__add__(4.5)`, which causes trouble because the `other` argument in the `__add__` method is 4.5, i.e., a `float` object, and `float` objects do not contain an attribute with the name `real` (`other.real` is used in our `__add__` method, and accessing `other.real` is what causes the error).

One idea for a remedy could be to set

```
other = Complex(other)
```

since this construction turns a real number `other` into a `Complex` object. However, when we add two `Complex` instances, `other` is of type `Complex`, and the constructor simply stores this `Complex` instance as `self.real` (look at the method `__init__`). This is not what we want!

A better idea is to test for the type of `other` and perform the right conversion to `Complex`:

```
    def __add__(self, other):
        if isinstance(other, (float,int)):
            other = Complex(other)
        return Complex(self.real + other.real,
                       self.imag + other.imag)
```

We could alternatively drop the conversion of `other` and instead implement two addition rules, depending on the type of `other`:

```
    def __add__(self, other):
        if isinstance(other, (float,int)):
            return Complex(self.real + other, self.imag)
        else:
            return Complex(self.real + other.real,
                           self.imag + other.imag)
```

A third way is to look for what we require from the `other` object, and check that this demand is fulfilled. Mathematically, we require `other` to be a complex or real number, but from a programming point of view, all we demand (in the original `__add__` implementation) is that `other` has `real` and `imag` attributes. To check if an object `a` has an attribute with name stored in the string `attr`, one can use the function

```
hasattr(a, attr)
```

In our context, we need to perform the test

```
if hasattr(other, 'real') and hasattr(other, 'imag'):
```

Our third implementation of the `__add__` method therefore becomes

```
    def __add__(self, other):
        if isinstance(other, (float,int)):
            other = Complex(other)
        elif not (hasattr(other, 'real') and \
                  hasattr(other, 'imag')):
            raise TypeError('other must have real and imag attr.')
        return Complex(self.real + other.real,
                       self.imag + other.imag)
```

The advantage with this third alternative is that we may add instances of class `Complex` and Python's own complex class (`complex`), since all we need is an object with `real` and `imag` attributes.

Computer Science Discussion. The presentations of alternative implementations of the `__add__` actually touch some very important computer science topics. In Python, function arguments can refer to objects of any type, and the type of an argument can change during program execution. This feature is known as *dynamic typing* and supported by languages such as Python, Perl, Ruby, and Tcl. Many other languages, C, C++, Java, and C# for instance, restrict a function argument to be of one type, which must be known when we write the program. Any attempt to call the function with an argument of another type is flagged as an error. One says that the language employs *static typing*, since the type cannot change as in languages having dynamic typing. The code snippet

```
a = 6    # a is integer
a = 'b'  # a is string
```

is valid in a language with dynamic typing, but not in a language with static typing.

Our next point is easiest illustrated through an example. Consider the code

```
a = 6
b = '9'
c = a + b
```

The expression `a + b` adds an integer and a string, which is illegal in Python. However, since `b` is the string `'9'`, it is natural to interpret `a + b` as `6 + 9`. That is, if the string `b` is converted to an integer, we may calculate `a + b`. Languages performing this conversion automatically are said to employ *weak typing*, while languages that require the programmer to explicit perform the conversion, as in

```
c = a + float(b)
```

are known to have *strong typing*. Python, Java, C, and C# are examples of languages with strong typing, while Perl and C++ allow weak typing. However, in our third implementation of the `__add__` method, certain types – `int` and `float` – are automatically converted to the right type `Complex`. The programmer has therefore imposed a kind of weak typing in the behavior of the addition operation for complex numbers.

There is also something called *duck typing* where the language only imposes a requirement of some data or methods in the object. The explanation of the term duck typing is the principle: "if it walks like a duck, and quacks like a duck, it's a duck". An operation `a + b` may be valid if `a` and `b` have certain properties that make it possible to add the objects, regardless of the type of `a` or `b`. To enable `a + b` it is in our third implementation of the `__add__` method sufficient that `b` has `real` and `imag` attributes. That is, objects with `real` and `imag` look

like `Complex` objects. Whether they really are of type `Complex` is not considered important in this context.

There is a continuously ongoing debate in computer science which kind of typing that is preferable: dynamic versus static, and weak versus strong. Static and strong typing, as found in Java and C#, support coding safety and reliability at the expense of long and sometimes repetitive code, while dynamic and weak typing support programming flexibility and short code. Many will argue that short code is more reliable than long code, so there is no simple conclusion.

7.6.4 Special Methods for "Right" Operands

What happens if we add a `float` and a `Complex` in that order?

```
w = 4.5 + u
```

This statement causes the exception

```
TypeError: unsupported operand type(s) for +: 'float' and 'instance'
```

This time Python cannot find any definition of what the plus operation means with a `float` on the left-hand side and a `Complex` object on the right-hand side of the plus sign. The `float` class was created many years ago without any knowledge of our `Complex` objects, and we are not allowed to extend the `__add__` method in the `float` class to handle `Complex` instances. Nevertheless, Python has a special method `__radd__` for the case where the class instance (`self`) is on the right-hand side of the operator and the `other` object is on the left-hand side. That is, we may implement a possible `float` or `int` plus a `Complex` by

```
    def __radd__(self, other):        # defines other + self
        return self.__add__(other)    # other + self = self + other
```

Similar special methods exist for subtraction, multiplication, and division. For the subtraction operator we need to be a little careful because `other - self`, which is the operation assumed to implemented in `__rsub__`, is not the same as `self.__sub__(other)` (i.e., `self - other`). A possible implementation is

```
    def __sub__(self, other):
        print 'in sub, self=%s, other=%s' % (self, other)
        if isinstance(other, (float,int)):
            other = Complex(other)
        return Complex(self.real - other.real,
                       self.imag - other.imag)

    def __rsub__(self, other):
        print 'in rsub, self=%s, other=%s' % (self, other)
        if isinstance(other, (float,int)):
            other = Complex(other)
        return other.__sub__(self)
```

The `print` statements are inserted to better understand how these methods are visited. A quick test demonstrates what happens:

```
>>> w = u - 4.5
in sub, self=(2, -1), other=4.5
>>> print w
(-2.5, -1)
>>> w = 4.5 - u
in rsub, self=(2, -1), other=4.5
in sub, self=(4.5, 0), other=(2, -1)
>>> print w
(2.5, 1)
```

Remark. As you probably realize, there is quite some code to be implemented and lots of considerations to be resolved before we have a class `Complex` for professional use in the real world. Fortunately, Python provides its `complex` class, which offers everything we need for computing with complex numbers. This fact reminds us that it is important to know what others already have implemented, so that we avoid "reinventing the wheel". In a learning process, however, it is a probably a very good idea to look into the details of a class `Complex` as we did above.

7.6.5 Inspecting Instances

The purpose of this section is to explain how we can easily look at the contents of a class instance, i.e., the data attributes and the methods. As usual, we look at an example – this time involving a very simple class:

```
class A:
    """A class for demo purposes."""
    def __init__(self, value):
        self.v = value

    def dump(self):
        print self.__dict__
```

The `self.__dict__` attribute is briefly mentioned in Chapter 7.1.5. Every instance is automatically equipped with this attribute, which is a dictionary that stores all the ordinary attributes of the instance (the variable names are keys, and the object references are values). In class `A` there is only one attribute, so the `self.__dict__` dictionary contains one key, `'v'`:

```
>>> a = A([1,2])
>>> a.dump()
{'v': [1, 2]}
```

Another way of inspecting what an instance `a` contains is to call `dir(a)`. This Python function writes out the names of all methods and variables (and more) of an object:

```
>>> dir(a)
'__doc__', '__init__', '__module__', 'dump', 'v']
```

The `__doc__` variable is a docstring, similar to docstrings in functions (Chapter 2.2.7), i.e., a description of the class appearing as a first string right after the `class` headline:

```
>>> a.__doc__
'A class for demo purposes.'
```

The `__module__` variable holds the name of the module in which the class is defined. If the class is defined in the program itself and not in an imported module, `__module__` equals `'__main__'`.

The rest of the entries in the list returned from `dir(a)` correspond to method and attribute names defined by the programmer of the class, in this example the methods `__init__` and `dump`, and the attribute `v`.

Now, let us try to add new variables to an existing instance[8]:

```
>>> a.myvar = 10
>>> a.dump()
{'myvar': 10, 'v': [1, 2]}
>>> dir(a)
['__doc__', '__init__', '__module__', 'dump', 'myvar', 'v']
```

The output of `a.dump()` and `dir(a)` show that we were successful in adding a new variable to this instance on the fly. If we make a new instance, it contains only the variables and methods that we find in the definition of class `A`:

```
>>> b = A(-1)
>>> b.dump()
{'v': -1}
>>> dir(b)
['__doc__', '__init__', '__module__', 'dump', 'v']
```

We may also add new methods to an instance, but this will not be shown here. The primary message of this subsection is two-fold: (i) a class instance is dynamic and allows attributes to be added or removed while the program is running, and (ii) the contents of an instance can be inspected by the `dir` function, and the data attributes are available through the `__dict__` dictionary.

7.7 Static Methods and Attributes

Up to now, each instance has its own copy of attributes. Sometimes it can be natural to have attributes that are shared among all instances. For example, we may have an attribute that counts how many instances

[8] This may sound scary and highly illegal to C, C++, Java, and C# programmers, but it is natural and legal in many other languages – and sometimes even useful.

that have been made so far. We can exemplify 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 are shared among all instances and called *static attributes*. To access the `counter` attribute, we must prefix by the classname `SpacePoint` instead of `self`: `SpacePoint.counter`. In the constructor we increase this common counter by 1, i.e., every time a new instance is made the counter is updated to keep track of how many objects we have created so far:

```
>>> p1 = SpacePoint(0,0,0)
>>> SpacePoint.counter
1
>>> for i in range(400):
...     p = SpacePoint(i*0.5, i, i+1)
...
>>> SpacePoint.counter
401
```

The methods we have seen so far must be called through an instance, which is fed in as the `self` variable in the method. We can also make class methods that can be called without having an instance. The method is then similar to a plain Python function, except that it is contained inside a class and the method name must be prefixed by the classname. Such methods are known as *static methods*. Let us illustrate the syntax by making a very simple class with just one static method `write`:

```
>>> class A:
...     @staticmethod
...     def write(message):
...         print message
...
>>> A.write('Hello!')
Hello!
```

As demonstrated, we can call `write` without having any instance of class `A`, we just prefix with the class name. Also note that `write` does not take a `self` argument. Since this argument is missing inside the method, we can never access non-static attributes since these always must be prefixed by an instance (i.e., `self`). However, we can access static attributes, prefixed by the classname.

If desired, we can make an instance and call `write` through that instance too:

```
>>> a = A()
>>> a.write('Hello again')
Hello again
```

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 classname.

7.8 Summary

7.8.1 Chapter Topics

Classes. A class contains attributes (variables) and methods. A first rough overview of a class can be to just list the attributes and methods in a UML diagram as we have done in Figure 7.4 on page 393 for some of the key classes in the present chapter.

Below is a sample class with three attributes (`m`, `M`, and `G`) and three methods (a constructor, `force`, and `visualize`). The class represents the gravity force between two masses. This force is computed by the `force` method, while the `visualize` method plots the force as a function of the distance between the masses.

```
class Gravity:
    """Gravity force between two physical objects."""

    def __init__(self, m, M):
        self.m = m           # mass of object 1
        self.M = M           # mass of object 2
        self.G = 6.67428E-11 # gravity constant, m**3/kg/s**2

    def force(self, r):
        G, m, M = self.G, self.m, self.M
        return G*m*M/r**2

    def visualize(self, r_start, r_stop, n=100):
        from scitools.std import plot, linspace
        r = linspace(r_start, r_stop, n)
        g = self.force(r)
        title='Gravity force: m=%g, M=%g' % (self.m, self.M)
        plot(r, g, title=title)
```

Note that to access attributes inside the `force` method, and to call the `force` method inside the `visualize` method, we must prefix with `self`. Also recall that all methods must take `self`, "this" instance, as first argument, but the argument is left out in calls. The assignment of attributes to a local variable (e.g., `G = self.G`) inside methods is not necessary, but here it makes the mathematical formula easier to read and compare with standard mathematical notation.

This class (found in file `Gravity.py`) can be used to find the gravity force between the moon and the earth:

```
mass_moon = 7.35E+22;  mass_earth = 5.97E+24
gravity = Gravity(mass_moon, mass_earth)
r = 3.85E+8  # earth-moon distance in meters
Fg = gravity.force(r)
print 'force:', Fg
```

Special Methods. A collection of special methods, with two leading and trailing underscores in the method names, offers special syntax in Python programs. Table 7.1 on page 392 provides an overview of the most important special methods.

Table 7.1 Summary of some important special methods in classes. `a` and `b` are instances of the class whose name we set to `A`.

`a.__init__(self, args)`	constructor: `a = A(args)`
`a.__call__(self, args)`	call as function: `a(args)`
`a.__str__(self)`	pretty print: `print a`, `str(a)`
`a.__repr__(self)`	representation: `a = eval(repr(a))`
`a.__add__(self, b)`	`a + b`
`a.__sub__(self, b)`	`a - b`
`a.__mul__(self, b)`	`a*b`
`a.__div__(self, b)`	`a/b`
`a.__radd__(self, b)`	`b + a`
`a.__rsub__(self, b)`	`b - a`
`a.__rmul__(self, b)`	`b*a`
`a.__rdiv__(self, b)`	`b/a`
`a.__pow__(self, p)`	`a**p`
`a.__lt__(self, b)`	`a < b`
`a.__gt__(self, b)`	`a > b`
`a.__le__(self, b)`	`a <= b`
`a.__ge__(self, b)`	`a => b`
`a.__eq__(self, b)`	`a == b`
`a.__ne__(self, b)`	`a != b`
`a.__bool__(self)`	boolean expression, as in `if a:`
`a.__len__(self)`	length of `a` (`int`): `len(a)`
`a.__abs__(self)`	`abs(a)`

7.8.2 Summarizing Example: Interval Arithmetics

Input data to mathematical formulas are often subject to uncertainty, usually because physical measurements of many quantities involve measurement errors, or because it is difficult to measure a parameter and one is forced to make a qualified guess of the value instead. In such cases it could be more natural to specify an input parameter by an interval $[a, b]$, which is guaranteed to contain the true value of the parameter. The size of the interval expresses the uncertainty in this parameter. Suppose all input parameters are specified as intervals, what will be the interval, i.e., the uncertainty, of the output data from the formula? This section develops a tool for computing this output uncertainty in

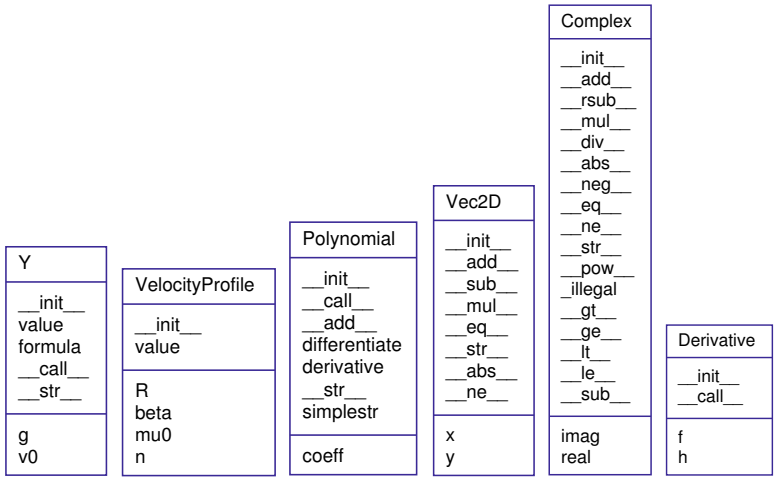

Fig. 7.4 UML diagrams of some classes described in this chapter.

the cases where the overall computation consists of the standard arithmetic operations.

To be specific, consider measuring the acceleration of gravity by dropping a ball and recording the time it takes to reach the ground. Let the ground correspond to $y = 0$ and let the ball be dropped from $y = y_0$. The position of the ball, $y(t)$, is then[9]

$$y(t) = y_0 - \frac{1}{2}gt^2 \,.$$

If T is the time it takes to reach the ground, we have that $y(T) = 0$, which gives the equation $\frac{1}{2}gT^2 = y_0$, with solution

$$g = 2y_0T^{-2} \,.$$

In such experiments we always introduce some measurement error in the start position y_0 and in the time taking (T). Suppose y_0 is known to lie in $[0.99, 1.01]$ m and T in $[0.43, 0.47]$ s, reflecting a 2% measurement error in position and a 10% error from using a stop watch. What is the error in g? With the tool to be developed below, we can find that there is a 22% error in g.

Problem. Assume that two numbers p and q are guaranteed to lie inside intervals,

$$p = [a, b], \quad q = [c, d] \,.$$

The sum $p + q$ is then guaranteed to lie inside an interval $[s, t]$ where $s = a + c$ and $t = b + d$. Below we list the rules of *interval arithmetics*, i.e., the rules for addition, subtraction, multiplication, and division of two intervals:

[9] The formula arises from the solution of Exercise 1.14 when $v_0 = 0$.

1. $p + q = [a + c, b + d]$
2. $p - q = [a - d, b - c]$
3. $pq = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$
4. $p/q = [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$ provided that $[c, d]$ does not contain zero

For doing these calculations in a program, it would be natural to have a new type for quantities specified by intervals. This new type should support the operators `+`, `-`, `*`, and `/` according to the rules above. The task is hence to implement a class for interval arithmetics with special methods for the listed operators. Using the class, we should be able to estimate the uncertainty of two formulas:

1. The acceleration of gravity, $g = 2y_0T^{-2}$, given a 2% uncertainty in y_0: $y_0 = [0.99, 1.01]$, and a 10% uncertainty in T: $T = [T_m \cdot 0.95, T_m \cdot 1.05]$, with $T_m = 0.45$.
2. The volume of a sphere, $V = \frac{4}{3}\pi R^3$, given a 20% uncertainty in R: $R = [R_m \cdot 0.9, R_m \cdot 1.1]$, with $R_m = 6$.

Solution. The new type is naturally realized as a class `IntervalMath` whose data consist of the lower and upper bound of the interval. Special methods are used to implement arithmetic operations and printing of the object. Having understood class `Vec2D` from Chapter 7.5, it should be straightforward to understand the class below:

```
class IntervalMath:
    def __init__(self, lower, upper):
        self.lo = float(lower)
        self.up = float(upper)

    def __add__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        return IntervalMath(a + c, b + d)

    def __sub__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        return IntervalMath(a - d, b - c)

    def __mul__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        return IntervalMath(min(a*c, a*d, b*c, b*d),
                            max(a*c, a*d, b*c, b*d))

    def __div__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        # [c,d] cannot contain zero:
        if c*d <= 0:
            raise ValueError\
                  ('Interval %s cannot be denominator because '\
                  'it contains zero')
        return IntervalMath(min(a/c, a/d, b/c, b/d),
                            max(a/c, a/d, b/c, b/d))

    def __str__(self):
        return '[%g, %g]' % (self.lo, self.up)
```

The code of this class is found in the file `IntervalMath.py`. A quick demo of the class can go as

```
I = IntervalMath
a = I(-3,-2)
b = I(4,5)
expr = 'a+b', 'a-b', 'a*b', 'a/b'
for e in expr:
    print '%s =' % e, eval(e)
```

The output becomes

```
a+b = [1, 3]
a-b = [-8, -6]
a*b = [-15, -8]
a/b = [-0.75, -0.4]
```

This gives the impression that with very short code we can provide a new type that enables computations with interval arithmetics and thereby with uncertain quantities. However, the class above has severe limitations as shown next.

Consider computing the uncertainty of aq if a is expressed as an interval $[4, 5]$ and q is a number (`float`):

```
a = I(4,5)
q = 2
b = a*q
```

This does not work so well:

```
  File "IntervalMath.py", line 15, in __mul__
    a, b, c, d = self.lo, self.up, other.lo, other.up
AttributeError: 'float' object has no attribute 'lo'
```

The problem is that `a*q` is a multiplication between an `IntervalMath` object `a` and a `float` object `q`. The `__mul__` method in class `IntervalMath` is invoked, but the code there tries to extract the `lo` attribute of `q`, which does not exist since `q` is a `float`.

We can extend the `__mul__` method and the other methods for arithmetic operations to allow for a number as operand – we just convert the number to an interval with the same lower and upper bounds:

```
def __mul__(self, other):
    if isinstance(other, (int, float)):
        other = IntervalMath(other, other)
    a, b, c, d = self.lo, self.up, other.lo, other.up
    return IntervalMath(min(a*c, a*d, b*c, b*d),
                        max(a*c, a*d, b*c, b*d))
```

Looking at the formula $g = 2y_0T^{-2}$, we run into a related problem: now we want to multiply 2 (`int`) with y_0, and if y_0 is an interval, this multiplication is not defined among `int` objects. To handle this case, we need to implement an `__rmul__(self, other)` method for doing `other*self`, as explained in Chapter 7.6.4:

```
    def __rmul__(self, other):
        if isinstance(other, (int, float)):
            other = IntervalMath(other, other)
        return other*self
```

Similar methods for addition, subtraction, and division must also be included in the class.

Returning to $g = 2y_0T^{-2}$, we also have a problem with T^{-2} when T is an interval. The expression `T**(-2)` invokes the power operator (at least if we do not rewrite the expression as `1/(T*T)`), which requires a `__pow__` method in class `IntervalMath`. We limit the possibility to have integer powers, since this is easy to compute by repeated multiplications:

```
    def __pow__(self, exponent):
        if isinstance(exponent, int):
            p = 1
            if exponent > 0:
                for i in range(exponent):
                    p = p*self
            elif exponent < 0:
                for i in range(-exponent):
                    p = p*self
                p = 1/p
            else:   # exponent == 0
                p = IntervalMath(1, 1)
            return p
        else:
            raise TypeError('exponent must int')
```

Another natural extension of the class is the possibility to convert an interval to a number by choosing the midpoint of the interval:

```
>>> a = IntervalMath(5,7)
>>> float(a)
6
```

`float(a)` calls `a.__float__()`, which we implement as

```
    def __float__(self):
        return 0.5*(self.lo + self.up)
```

A `__repr__` method returning the right syntax for recreating the present instance is also natural to include in any class:

```
    def __repr__(self):
        return '%s(%g, %g)' % \
               (self.__class__.__name__, self.lo, self.up)
```

We are now in a position to test out the extended class `IntervalMath`.

```
>>> g = 9.81
>>> y_0 = I(0.99, 1.01)      # 2% uncertainty
>>> Tm = 0.45                # mean T
>>> T = I(Tm*0.95, Tm*1.05)  # 10% uncertainty
>>> print T
```

```
[0.4275, 0.4725]
>>> g = 2*y_0*T**(-2)
>>> g
IntervalMath(8.86873, 11.053)
>>> # computing with mean values:
>>> T = float(T)
>>> y = 1
>>> g = 2*y_0*T**(-2)
>>> print '%.2f' % g
9.88
```

Another formula, the volume $V = \frac{4}{3}\pi R^3$ of a sphere, shows great sensitivity to uncertainties in R:

```
>>> Rm = 6
>>> R = I(Rm*0.9, Rm*1.1)   # 20 % error
>>> V = (4./3)*pi*R**3
>>> V
IntervalMath(659.584, 1204.26)
>>> print V
[659.584, 1204.26]
>>> print float(V)
931.922044761
>>> # compute with mean values:
>>> R = float(R)
>>> V = (4./3)*pi*R**3
>>> print V
904.778684234
```

Here, a 20% uncertainty in R gives almost 60% uncertainty in V, and the mean of the V interval is significantly different from computing the volume with the mean of R.

The complete code of class `IntervalMath` is found in `IntervalMath.py`. Compared to the implementations shown above, the real implementation in the file employs some ingenious constructions and help methods to save typing and repeating code in the special methods for arithmetic operations.

7.9 Exercises

Exercise 7.1. *Make a function class.*

Make a class `F` that implements the function

$$f(x; a, w) = e^{-ax}\sin(wx)\,.$$

A `value(x)` method computes values of f, while a and w are class attributes. Test the class with the following main program:

```
from math import *
f = F(a=1.0, w=0.1)
print f.value(x=pi)
f.a = 2
print f.value(pi)
```

Name of program file: `F.py`. ◇

Exercise 7.2. *Make a very simple class.*

Make a class `Simple` with one attribute `i`, one method `double`, which replaces the value of `i` by `i+i`, and a constructor that initializes the attribute. Try out the following code for testing the class:

```
s1 = Simple(4)
for i in range(4):
    s1.double()
print s1.i

s2 = Simple('Hello')
s2.double();  s2.double()
print s2.i
s2.i = 100
print s2.i
```

Before you run this code, convince yourself what the output of the `print` statements will be. Name of program file: `Simple.py`. ◇

Exercise 7.3. *Extend the class from Ch. 7.2.1.*

Add an attribute `transactions` to the `Account` class from Chapter 7.2.1. The new attribute counts the number of transactions done in the `deposit` and `withdraw` methods. The total number of transactions should be printed in the `dump` method. Write a simple test program to demonstrate that `transaction` gets the right value after some calls to `deposit` and `withdraw`. Name of program file: `Account2.py`. ◇

Exercise 7.4. *Make classes for a rectangle and a triangle.*

The purpose of this exercise is to create classes like class `Circle` from Chapter 7.2.3 for representing other geometric figures: a rectangle with width W, height H, and lower left corner (x_0, y_0); and a general triangle specified by its three vertices (x_0, y_0), (x_1, y_1), and (x_2, y_2) as explained in Exercise 2.17. Provide three methods: `__init__` (to initialize the geometric data), `area`, and `circumference`. Name of program file: `geometric_shapes.py`. ◇

Exercise 7.5. *Make a class for straight lines.*

Make a class `Line` whose constructor takes two points `p1` and `p2` (2-tuples or 2-lists) as input. The line goes through these two points (see function `line` in Chapter 2.2.7 for the relevant formula of the line). A `value(x)` method computes a value on the line at the point `x`. Here is a demo in an interactive session:

```
>>> from Line import Line
>>> line = Line((0,-1), (2,4))
>>> print line.value(0.5), line.value(0), line.value(1)
0.25 -1.0 1.5
```

Name of program file: `Line.py`. ◇

Exercise 7.6. *Improve the constructor in Exer. 7.5.*

The constructor in class `Line` in Exercise 7.5 takes two points as arguments. Now we want to have more flexibility in the way we specify