A\_Primer\_Scientific\_Programming\_Python\_5E\_Hans\_c07

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 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 treated in Chap. 9.

The folder src/class1 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 that have a set of parameters in addition to one or more independent variables. Section 7.1.1 explains why such mathematical functions pose difficulties for programmers, and Sect. 7.1.2 shows how the class idea meets these difficulties. Sections 7.1.4 presents another example where a class represents a mathematical function. More advanced materia about classes, which for some readers may clarify the ideas, but which can also be skipped in a first reading, appears in Sects. 7.1.5 and Sect. 7.1.6.

**7.1.1 Challenge: Functions with Parameters**

To motivate for the class concept, we will look at functions with parameters. One example is y.t/ D v0 t 1 2 gt 2. Conceptually, in physics, the y quantity is viewed as a function of t, but y also depends on two other parameters, v0 and g, although it is not natural to view y as a function of these parameters. We may write y.tI v0 ; g/ to indicate that t is the independent variable, while v0 and g are parameters. Strictly speaking, g is a fixed parameter (as long as we are on the surface of the earth and can view g as constant), so only v0 and t can be arbitrarily chosen in the formula. It would then be better to write y.tI v0 /.

In the general case, we may have a function of x that has n parameters p1 ; : : : ; pn: f .xI p1 ; : : : ; pn /. One example could be g.xI A; a/ D Aeax :

g.xI A; a/ D Aeax :

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

f 0.x/ f .x C h/ f .x/

h (7.1)

coded as

def diff(f, x, h=1E-5):

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-6)

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/ D 0, finding extrema, etc. All these libraries will try to call the mathematical function we provide with only one argument. When our function has more arguments, the code inside the library aborts in the call to our function, and such errors may not always be easy to track down.

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.tI v0 /, one with v0 D 1 and one with v0 D 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 v0 parameters could be to introduce two y functions, each with a distinct v0 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 v0 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 Sect. 7.1.6). 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.tI v0 / D v0 t 1 2 gt 2. We may say that v0 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.tI v0 / 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 aid of the 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.

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 line in the class implementation means, we start by showing how the class can be used to compute values of the mathematical function y.tI v0 /.

A class creates a new data type, here of name Y, so when we use the class to make objects, those objects are of type Y. (Actually, all the standard Python objects, such as lists, tuples, strings, floating-point numbers, integers, etc., are built-in Python classes, with names list, tuple, str, float, int, etc.) An object of a userdefined 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. 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 D 0:1I v0 D 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 data attributes. Methods are also known as method attributes. From now on we will use this terminology. In our sample class Y we have two methods or method attributes, \_\_init\_\_ and value, two data attributes, v0 and g, and four attributes in total (\_\_init\_\_, value, v0, and g). The names of 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 common convention to use the constructor for initializing the variables in the class.

Extension of the class We can have as many attributes 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 v0. 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

Be careful with indentation in class programming

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 data attributes (Sect. 7.6) can appear in the indented block under the class headline. Ordinary data 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 v0:

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 Sect. 7.1.1:

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

New-style classes versus classic classes When use Python version 2 and write a class like

class V:

...

we get what is known as an old-style or classic class. A revised implementation of classes in Python came in version 2.2 with new-style classes. The specification of a new-style class requires (object) after the class name:

class V(object):

...

New-style classes have more functionality, and it is in general recommended to work with new-style classes. We shall therefore from now write V(object) rather than just V. In Python 3, all classes are new-style whether we write V or V(object).

Doc strings A function may have a doc string right after the function definition, see Sect. 3.1.11. 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(object):

"""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(object):

"""

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.

Data 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 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 data attributes in this instance. The self parameter is invisibly returned to the calling code. We can imagine that Python translates the syntax y = Y(3) to a call written as

Y.\_\_init\_\_(y, 3)

Now, self becomes the new instance y we want to create, so when we do self.v0 = v0 in the constructor, we actually assign v0 to y.v0. The prefix with Y. illustrates how to reach a class method with a syntax similar to reaching a function in a module (just like 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 in the syntax, and Python will automatically assign the y instance to the self argument. It is the latter “instance name prefix” which we shall use when computing with classes. (Y.\_\_init\_\_(y, 3) will not work since y is undefined and supposed to be an Y object. However, if we first create y = Y(2) and then call Y.\_\_init\_\_(y, 3), the syntax works, and y.v0 is 3 after the call.)

Let us look at a call to the value method to see a similar use of the self argu- ment. 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 use of self may become more apparent when we have multiple class instances. We can make a class that just has one parameter so we can easily identify a class instance by printing the value of this parameter. In addition, every Python object obj has a unique identifier obtained by id(obj) that we can also print to track what self is.

class SelfExplorer(object):

def \_\_init\_\_(self, a):

self.a = a

print ’init: a=%g, id(self)=%d’ % (self.a, id(self))

def value(self, x):

print ’value: a=%g, id(self)=%d’ % (self.a, id(self))

return self.a\*x

Here is an interactive session with this class:

>>> s1 = SelfExplorer(1)

init: a=1, id(self)=38085696

>>> id(s1)

38085696

We clearly see that self inside the constructor is the same object as s1, which we want to create by calling the constructor.

A second object is made by

>>> s2 = SelfExplorer(2)

init: a=2, id(self)=38085192

>>> id(s2)

38085192

Now we can call the value method using the standard syntax s1.value(x) and the “more pedagogical” syntax SelfExplorer.value(s1, x). Using both s1 and s2 illustrates how self take on different values, while we may look at the method SelfExplorer.value as a single function that just operates on different self and x objects:

>>> s1.value(4)

value: a=1, id(self)=38085696

4

>>> SelfExplorer.value(s1, 4)

value: a=1, id(self)=38085696

4

>>> s2.value(5)

value: a=2, id(self)=38085192

10

>>> SelfExplorer.value(s2, 5)

value: a=2, id(self)=38085192

10

Hopefully, these illustrations help to explain that self is just the instance used in the method call prefix, here s1 or s2. If not, patient work with class programming in Python will over time reveal an understanding of what self really is.

**Rules regarding self**

* Any class method must have self as first argument. (The name can be any valid variable name, but the name self is a widely established convention in Python.)
* self represents an (arbitrary) instance of the class.
* To access any class attribute inside class methods, we must prefix with self, as in self.name, where name is the name of the attribute.
* self is dropped as argument in calls to class methods.

**7.1.4 Another Function Class Example**

Let us apply the ideas from the Y class to the function

v.r/ D

ˇ

2 0

1=n n

n C 1

R 1C1=n r 1C1=n ;

where r is the independent variable. We may write this function as v.rI ˇ; 0 ; n; R/ to explicitly indicate that there is one primary independent variable (r) and four physical parameters ˇ, 0, n, and R. Exercise 5.40 describes a physical interpretation of v as the velocity of a fluid. The class typically holds the physical parameters as variables and provides an value(r) method for computing the v function:

class V(object):

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:

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

The comma-separated list of variables on the right-hand side forms a tuple so this assignment is just the a valid 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. An equivalent multi-line code is

self.beta = beta

self.mu0 = mu0

self.n = n

self.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.

Remark Another solution to the problem of sending functions with parameters to a general library function such as diff is provided in Sect. H.7. 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.

**7.1.5 Alternative Function Class Implementations**

To illustrate class programming further, we will now realize class Y from Sect. 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 the data attributes in the class 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 a data 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(object):

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 data 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 data 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 inside 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.

**7.1.6 Making Classes Without the Class Construct**

Newcomers to the class concept 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 Sect. 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. (The instance can also contain static data attributes (Sect. 7.6), but these are to be viewed as global variables in the present context.)

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 Sect. 7.1.2 and an instance 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 (explained in Chap. 9). For more completeness of the present section, we therefore 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.

**7.1.7 Closures**

This section follows up the discussion in Sect. 7.1.6 and presents a more advanced construction that may serve as alternative to class constructions in some cases.

Our motivating example is that we want a Python implementation of a mathematical function y.tI v0 / D v0 t 1 2 gt 2 to have t as the only argument, but also have access to the parameter v0. Consider the following function, which returns a function

>>> def generate\_y():

... v0 = 5

... g = 9.81

... def y(t):

... return v0\*t - 0.5\*g\*t\*\*2

... return y

...

>>> y = generate\_y()

>>> y(1)

0.09499999999999975

The remarkable property of the y function is that it remembers the value of v0 and g, although these variables are not local to the parent function generate\_y and not local in y. In particular, we can specify v0 as argument to generate\_y:

>>> def generate\_y(v0):

... g = 9.81

... def y(t):

... return v0\*t - 0.5\*g\*t\*\*2

... return y

...

>>> y1 = generate\_y(v0=1)

>>> y2 = generate\_y(v0=5)

>>> y1(1)

-3.9050000000000002

>>> y2(1)

0.09499999999999975

Here, y1(t) has access to v0=1 while y2(t) has access to v0=5.

The function y(t) we construct and return from generate\_y is called a closure and it remembers the value of the surrounding local variables in the parent function (at the time we create the y function). Closures are very convenient for many purposes in mathematical computing. Examples appear in Sect. 7.3.2. Closures are also central in a programming style called functional programming.

**Generating multiple closures in a function**

As soon as you get the idea of a closure, you will probably use it a lot because it is a convenient way to pack a function with extra data. However, there are some pitfalls. The biggest is illustrated below, but this is considered advanced material!

Let us generate a series of functions v(t) for various values of a parameter v0. Each function just returns a tuple (v0, t) such that we can easily see what the argument and the parameter are. We use lambda to quickly define each function, and we place the functions in a list:

>>> def generate():

... return [lambda t: (v0, t) for v0 in [0, 1, 5, 10]]

...

>>> funcs = generate()

Now, funcs is a list of functions with one argument. Calling each function and printing the return values v0 and t gives

>>> for func in funcs:

... print func(1)

...

(10, 1)

(10, 1)

(10, 1)

(10, 1)

As we see, all functions have v0=10, i.e., they stored the most recent value of v0 before return. This is not what we wanted.

The trick is to let v0 be a keyword argument in each function, because the value of a keyword argument is frozen at the time the function is defined:

>>> def generate():

... return [lambda t, v0=v0: (v0, t)

... for v0 in [0, 1, 5, 10]]

...

>>> funcs = generate()

>>> for func in funcs:

... print func(1)

...

(0, 1)

(1, 1)

(5, 1)

(10, 1)

**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(object):

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 be 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 attributes, 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. 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 data attributes directly

The new “protected” version of class Account, called AccountP, reads

class AccountP(object):

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 data 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 data 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 address, 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 data attributes of the class hold information 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 data 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(object):

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 data 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:

class Person(object):

...

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

430 7 Introduction to Classes

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 Sect. 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 ; y0 / and its radius R. We can collect these three numbers as data attributes in a class. The values of x 0, y0, and R are naturally initialized in the constructor. Other methods can be area and circumference for calculating the area R 2 and the circumference 2R:

class Circle(object):

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.

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.

Verification We should include a test function for checking that the implementation of class Circle is correct:

def test\_Circle():

R = 2.5

c = Circle(7.4, -8.1, R)

from math import pi

expected\_area = pi\*R\*\*2

computed\_area = c.area()

diff = abs(expected\_area - computed\_area)

tol = 1E-14

assert diff < tol, ’bug in Circle.area, diff=%s’ % diff

expected\_circumference = 2\*pi\*R

computed\_circumference = c.circumference()

diff = abs(expected\_circumference - computed\_circumference)

assert diff < tol, ’bug in Circle.circumference, diff=%s’ % diff

The test\_Circle function is written in a way that it can be used in a pytest or nose testing framework (see Sect. H.9, or the brief examples in Sects. 3.3.3, 3.4.2, and 4.9.4). The necessary conventions are that the function name starts with test\_, the function takes no arguments, and all tests are of the form assert success or assert success, msg where success is a boolean condition for the test and msg is an optional message to be written if the test fails (success is False). It is a good habit to write such test functions to verify the implementation of classes.

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, y0, 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 evaluates to 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 from Sect. 7.1.2, 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(object):

...

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,

if callable(a):

tests whether a behaves as a callable, i.e., if a is 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 from Sect. 7.1.1:

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 0.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. The Python package sympy for symbolic mathematics (see Sect. 1.7) makes it trivial to calculate the exact derivative of a large class of functions f .x/ and turn the result into an ordinary Python function. However, mathematical 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. Use of exact symbolic differentiation via SymPy is also possible

Solution The most basic (but not the best) formula for a numerical derivative is

f 0.x/ f .x C h/ f .x/

h : (7.2)

The idea is that we make a class to hold the function to be differentiated, call it f, and a step size h to be used in (7.2). These variables can be set in the constructor. The \_\_call\_\_ operator computes the derivative with aid of (7.1). All this can be coded in a few lines:

class Derivative(object):

def \_\_init\_\_(self, f, h=1E-5):

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/ D sin x and g.t/ D 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/.

Verification It is a good programming habit to include a test function for verifying the implementation of a class. We can construct a test based on the fact that the approximate differentiation formula (7.2) is exact for linear functions:

def test\_Derivative():

# The formula is exact for linear functions, regardless of h

f = lambda x: a\*x + b

a = 3.5; b = 8

dfdx = Derivative(f, h=0.5)

diff = abs(dfdx(4.5) - a)

assert diff < 1E-14, ’bug in class Derivative, diff=%s’ % diff

We have here used a lambda function for compactly defining a function f, see Sect. 3.1.14. A special feature of f is that it remembers the variables a and b when f is sent to class Derivative (it is a closure, see Sect. 7.1.7). Note that the test function above follows the conventions for test functions outlined in Sect. 7.2.3.

Application: Newton’s method In what 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/ D 0 with Newton’s method and we, because of laziness, lack of time, or lack of training do not manage to derive f 0.x/ by hand. Consider a function Newton for solving f .x/ D 0: Newton(f, x, dfdx, epsilon=1.0E-7, N=100). Section A.1.10 presents a specific implementation in a module file Newton.py. The arguments are a Python function f for f .x/, a float x for the initial guess (start value) of x, a Python function dfdx for f 0.x/, a float epsilon for the accuracy of the root: the algorithms iterates until jf .x/j < , and an int N for the maximum number of iterations that we allow. All arguments are easy to provide, except dfdx, which requires computing f 0.x/ by hand then implementation of the formula in a Python function. Suppose our target equation reads

f .x/ D 10 5 .x 0:9/2 .x 1:1/3 D 0 :

The function f .x/ is plotted in Fig. 7.2. The 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. (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). For example, an epsilon tolerance of 1010 requires 18 iterations with an error of 103.) 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)

436 7 Introduction to Classes

-1

-0.8

-0.6

-0.4

-0.2

0

0.2

0.4

0.6

0.9 0.95 1 1.05 1.1 1.15

Fig. 7.2 Plot of y D 105 .x 0:9/2 .x 1:1/3

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 handcalculations. 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.

Solution utilizing SymPy Class Derivative is based on numerical differentiation, but it is possible to make an equally short class that can do exact differentiation. In SymPy, one can perform symbolic differentiation of an expression e with respect to a symbolic independent variable x by diff(e, x) (see Sect. 1.7.1). Assuming that the user’s f function can be evaluated for a symbolic independent variable x, we can call f(x) to get the SymPy expression for the formula in f and then use diff to calculate the exact derivative. Thereafter, we turn the symbolic expression of the derivative into an ordinary Python function (via lambdify) and define this function as the \_\_call\_\_ method. The proper Python code is very short:

class Derivative\_sympy(object):

def \_\_init\_\_(self, f):

from sympy import Symbol, diff, lambdify

x = Symbol(’x’)

sympy\_f = f(x) # make sympy expression

sympy\_dfdx = diff(sympy\_f, x)

self.\_\_call\_\_ = lambdify([x], sympy\_dfdx)

Note how the \_\_call\_\_ method is defined by assigning a function to it (even though the function returned by lambdify is a function of x only, it works to call obj(x) for an instance obj of type Derivative\_sympy)

Both demonstration of the class and verification of the implementation can be placed in a test function:

def test\_Derivative\_sympy():

def g(t):

return t\*\*3

dg = Derivative\_sympy(g)

t = 2

exact = 3\*t\*\*2

computed = dg(t)

tol = 1E-14

assert abs(exact - computed) < tol

def h(y):

return exp(-y)\*sin(2\*y)

from sympy import exp, sin

dh = Derivative\_sympy(h)

from math import pi, exp, sin, cos

y = pi

exact = -exp(-y)\*sin(2\*y) + exp(-y)\*2\*cos(2\*y)

computed = dh(y)

assert abs(exact - computed) < tol

The example with the g(t) should be straightforward to understand. In the constructor of class Derivative\_sympy, we call g(x), with the symbol x, and g returns the SymPy expression x\*\*3. The \_\_call\_\_ method then becomes a function lambda x: 3\*x\*\*2.

The h(y) function, however, deserves more explanation. When then constructor of class Derivative\_sympy makes the call h(x), with the symbol x, the h function will return the SymPy expression exp(-x)\*sin(2\*x), provided exp and sin are SymPy functions. Since we do from sympy import exp, sin prior to calling the constructor in class Derivative\_sympy, the names exp and sin are defined in the test function, and our local h function will have access to all local variables, as it is a closure as mentioned above and in Sect. 7.1.7 (see also Sect. 9.2.6). This means that h has access to sympy.sin and sympy.cos when the constructor in class Derivative\_sympy calls h. Thereafter, we want to do some numerical computing and need exp, sin, and cos from the math module. If we had tried to do Derivative\_sympy(h) after the import from math, h would then call math.exp and math.sin with a SymPy symbol as argument, and would cause a TypeError since math.exp expects a float, not a Symbol object from SymPy.

Although the Derivative\_sympy class is small and compact, its construction and use as explained here bring up more advanced topics than class Derivative and its plain numerical computations. However, it may be interesting to see that a class for exact differentiation of a Python function can be realized in very few lines.

**7.3.3 Example: Automagic Integration**

We can apply the ideas from Sect. 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 .xI a/ D

xZ

a

f .t/dt :

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

xZ

a

f .t/dt D h 1

2 f .a/ C

n1X

i D1

f .a C ih/ C 1

2 f .x/

!

; (7.3)

where h D .x a/=n. In an application program, we want to compute F .xI a/ by a simple syntax like

def f(x):

return exp(-x\*\*2)\*sin(10\*x)

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 .xI 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 range(1, n):

I += f(a + i\*h)

I += 0.5\*f(x)

I \*= h

return I

Class Integral must have some data 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 data attributes. The implementation then becomes

class Integral(object):

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.

An application program computing R 2 0 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)

Verification via symbolic computing We should always provide a test function for verification of the implementation. To avoid dealing with unknown approximation errors of the Trapezoidal rule, we use the obvious fact that linear functions are integrated exactly by the rule. Although it is really easy to pick a linear function, integrate it, and figure out what an integral is, we can also demonstrate how to automate such a process by SymPy. Essentially, we define an expression in SymPy, ask SymPy to integrate it, and then turn the resulting symbolic integral to a plain Python function for computing:

>>> import sympy as sp

>>> x = sp.Symbol(’x’)

>>> f\_expr = sp.cos(x) + 5\*x

>>> f\_expr

5\*x + cos(x)

>>> F\_expr = sp.integrate(f\_expr, x)

>>> F\_expr

5\*x\*\*2/2 + sin(x)

>>> F = sp.lambdify([x], F\_expr) # turn f\_expr to F(x) func.

>>> F(0)

0.0

>>> F(1)

3.3414709848078967

Using such functionality to do exact integration, we can write our test function as

def test\_Integral():

# The Trapezoidal rule is exact for linear functions

import sympy as sp

x = sp.Symbol(’x’)

f\_expr = 2\*x + 5

# Turn sympy expression into plain Python function f(x)

f = sp.lambdify([x], f\_expr)

440 7 Introduction to Classes

# Find integral of f\_expr and turn into plain Python function F

F\_expr = sp.integrate(f\_expr, x)

F = sp.lambdify([x], F\_expr)

a = 2

x = 6

exact = F(x) - F(a)

computed = Integral(f, a, n=4)

diff = abs(exact - computed)

tol = 1E-15

assert diff < tol, ’bug in class Integral, diff=%s’ % diff

If you think it is overkill to use SymPy for integrating linear functions, you can equally well do it yourself and define f = lambda x: 2\*x + 5 and F = lambda x: x\*\*2 + 5\*x.

Remark Class Integral is inefficient (but probably more than fast enough) for plotting F .xI 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 useful 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 Sect. 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(object):

...

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(object):

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.

Note that the \_\_str\_\_ method is called whenever we do str(a), and print a is effectively print str(a), i.e., print a.\_\_str\_\_().

**7.3.5 Example: Phone Book with Special Methods**

Let us reconsider class Person from Sect. 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(object):

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

442 7 Introduction to Classes

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]) + ’\n’

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 demo code 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 PhoneBook.py. You can run this program, statement by statement, either in the Online Python Tutor 2 or in a debugger (see Sect. F.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 3.39. 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(object):

...

\_\_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 C 0 x 1 x 2 C 2 x 3 D 1 x 2 C 2x 3

Polynomials can be added (by just adding the coefficients corresponding to the same powers) 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(object):

def \_\_init\_\_(self, coefficients):

self.coeff = coefficients

def \_\_call\_\_(self, x):

"""Evaluate the polynomial."""

s = 0

for i in range(len(self.coeff)):

s += self.coeff[i]\*x\*\*i

return s

def \_\_add\_\_(self, other):

"""Return self + other as Polynomial object."""

# Two cases:

#

# self: X X X X X X X

# other: X X X

#

# or:

#

# self: X X X X X

# other: X X X X X X X X

# Start with the longest list and add in the other

if len(self.coeff) > len(other.coeff):

result\_coeff = self.coeff[:] # copy!

for i in range(len(other.coeff)):

result\_coeff[i] += other.coeff[i]

else:

result\_coeff = other.coeff[:] # copy!

for i in range(len(self.coeff)):

result\_coeff[i] += self.coeff[i]

return Polynomial(result\_coeff)

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

The \_\_add\_\_ method looks more advanced. The goal 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 result\_coeff starts out as a copy of self.coeff: if not, changes in result\_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 7.24.

A subtraction method \_\_sub\_\_ can be implemented along the lines of \_\_add\_\_, but is slightly more complicated and left as Exercise 7.25. You are strongly encouraged to do this exercise as it will help increase the understanding of the interplay between mathematics and programming in class Polynomial.

A more complicated operation on polynomials, from a mathematical point of view, is the multiplication of two polynomials. Let p.x/ D PM i D0 ci x i and q.x/ D PN

j D0 dj x j be the two polynomials. The product becomes

MX

i D0

ci x i

! 0

@

NX

j D0

dj x j

1

A D

MX

i D0

NX

j D0

ci dj x i Cj :

The double sum must be implemented as a double loop, but first the list for the resulting polynomial must be created with length M CN C1 (the highest exponent is M CN 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 = numpy.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

d

dx

nX

i D0

ci x i D

nX

i D1

ici x i 1 :

If ci 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 returning 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:

class Polynomial(object):

...

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]

446 7 Introduction to Classes

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 (i.e., the object’s content can change) and allow inplace 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. (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 Sects. 7.2.1 and 7.5.2.) 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

p1 .x/ D 1 x; p2 .x/ D 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()

>>> print p5.coeff

[1, 0, 0, -24, -5]

One verification of the implementation may be to compare p3 at (e.g.) x D 1=2 with p1 .x/ C p2

.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 objec

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(object):

...

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(object):

...

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 ’)

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

Verifying the implementation It is always a good habit to include a test function test\_Polynomial() for verifying the functionality in class Polynomial. To this end, we construct some examples of addition, multiplication, and differentiation of polynomials by hand and make tests that class Polynomial reproduces the correct results. Testing the \_\_str\_\_ method is left as Exercise 7.26.

Rounding errors may be an issue in class Polynomial: \_\_add\_\_, derivative, and differentiate will lead to integer coefficients if the polynomials to be added have integer coefficients, while \_\_mul\_\_ always results in a polynomial with the coefficients stored in a numpy array with float elements. Integer coefficients in lists can be compared using == for lists, while coefficients in numpy arrays must be compared with a tolerance. One can either subtract the numpy arrays and use the max method to find the largest deviation and compare this with a tolerance, or one can use numpy.allclose(a, b, rtol=tol) for comparing the arrays a and b with a (relative) tolerance tol.

Let us pick polynomials with integer coefficients as test cases such that \_\_add\_\_, derivative, and differentiate can be verified by testing equality (==) of the

coeff lists. Multiplication in \_\_mul\_\_ must employ numpy.allclose.

We follow the convention that all tests are on the form assert success, where

success is a boolean expression for the test. (The actual version of the test func-

tion in the file Polynomial.py adds an error message msg to the test: assert

success, msg.) Another part of the convention is that the function starts with

test\_ and the function takes no arguments.

Our test function now becomes

def test\_Polynomial():

p1 = Polynomial([1, -1])

p2 = Polynomial([0, 1, 0, 0, -6, -1])

p3 = p1 + p2

p3\_exact = Polynomial([1, 0, 0, 0, -6, -1])

assert p3.coeff == p3\_exact.coeff

p4 = p1\*p2

p4\_exact = Polynomial(numpy.array([0, 1, -1, 0, -6, 5, 1]))

assert numpy.allclose(p4.coeff, p4\_exact.coeff, rtol=1E-14

7.3 Special Methods 449

p5 = p2.derivative()

p5\_exact = Polynomial([1, 0, 0, -24, -5])

assert p5.coeff == p5\_exact.coeff

p6 = Polynomial([0, 1, 0, 0, -6, -1]) # p2

p6.differentiate()

p6\_exact = p5\_exact

assert p6.coeff == p6\_exact.coeff

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 7.25. Sec-

tion 7.4 contains examples on implementing the special methods listed above.

7.3.9 Special Methods for String Conversion

Look at this class with a \_\_str\_\_ method:

>>> class MyClass(object):

... def \_\_init\_\_(self):

... self.data = 2

... def \_\_str\_\_(self):

450 7 Introduction to Classes

... 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

Sect. 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 representation of the contents of the in-

stance. 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 Sect. 4.3.1. 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 Sect. 7.1.2, \_\_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

7.4 Example: Class for Vectors in the Plane 451

Below we show examples of \_\_repr\_\_ methods in classes Y (Sect. 7.1.2),

Polynomial (Sect. 7.3.7), and MyClass (above):

class Y(object):

...

def \_\_repr\_\_(self):

return ’Y(v0=%s)’ % self.v0

class Polynomial(object):

...

def \_\_repr\_\_(self):

return ’Polynomial(coefficients=%s)’ % self.coeff

class MyClass(object):

...

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: 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 Sect. 7.3, in particular Sects. 7.3.6 and 7.3.8.

7.4.1 Some Mathematical Operations on Vectors

Vectors in the plane are described by a pair of real numbers, .a; b/. In Sect. 5.1.2

we present 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 .c; d / D .a C c; b C d /; (7.4)

.a; b/ .c; d / D .a c; b d /; (7.5)