Object-Oriented Programming

9

This chapter introduces the basic ideas of object-oriented programming. Different people put different meanings into the term object-oriented programming: some use the term for programming with objects in general, while others use the term for programming with class hierarchies. The author applies the second meaning, which is the most widely accepted one in computer science. The first meaning is better named *object-based* programming. Since everything in Python is an object, we do object-based programming all the time, yet one usually reserves this term for the case when classes different from Python's basic types (`int`, `float`, `str`, `list`, `tuple`, `dict`) are involved.

Necessary background for the present chapter includes basic knowledge about classes in Python, at least concepts such as attributes (method attributes, data attributes), methods, constructors, the `self` object, and the `__call__` special method. Suitable material for this background is Sects. 7.1, 7.2, and 7.3.1. For Sects. 9.2 and 9.3 one must know the most basic methods for numerical differentiation and integration, for example from Appendix B. During an initial reading of the chapter, it can be beneficial to skip the more advanced material in Sects. 9.2.4–9.2.7.

All the programs associated with this chapter are found in the folder `src/oo`[1].

9.1 Inheritance and Class Hierarchies

Most of this chapter tells you how to put related classes together in families such that the family can be viewed as one unit. This idea helps to hide details in a program, and makes it easier to modify or extend the program.

A family of classes is known as a *class hierarchy*. As in a biological family, there are parent classes and child classes. Child classes can *inherit* data and methods from parent classes, they can modify these data and methods, and they can add their own data and methods. This means that if we have a class with some functionality, we can extend this class by creating a child class and simply add the functionality we need. The original class is still available and the separate child class is small, since it does not need to repeat the code in the parent class.

---

[1] http://tinyurl.com/pwyasaa/oo

H.P. Langtangen, *A Primer on Scientific Programming with Python*,
Texts in Computational Science and Engineering 6, DOI 10.1007/978-3-662-49887-3_9

The magic of object-oriented programming is that other parts of the code do not need to distinguish whether an object is the parent or the child – all generations in a family tree can be treated as a unified object. In other words, one piece of code can work with all members in a class family or hierarchy. This principle has revolutionized the development of large computer systems. As an illustration, two of the most widely used computer languages today are Java and `C#`, and both of them force programs to be written in an object-oriented style.

The concepts of classes and object-oriented programming first appeared in the Simula programming language in the 1960s. Simula was invented by the Norwegian computer scientists Ole-Johan Dahl and Kristen Nygaard, and the impact of the language is particularly evident in C++, Java, and C#, three of the most dominating programming languages in the world today. The invention of object-oriented programming was a remarkable achievement, and the professors Dahl and Nygaard received two very prestigious prizes: the von Neumann medal and the Turing prize (popularly known as the Nobel prize of computer science).

A parent class is usually called *base class* or *superclass*, while the child class is known as a *subclass* or *derived class*. We shall use the terms superclass and subclass from now on.

9.1.1 A Class for Straight Lines

Assume that we have written a class for straight lines, $y = c_0 + c_1x$:

```
class Line(object):
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ’’
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += ’%12g %12g\n’ % (x, y)
        return s
```

The constructor `__init__` initializes the coefficients c_0 and c_1 in the expression for the straight line: $y = c_0 + c_1x$. The call operator `__call__` evaluates the function $c_1x + c_0$, while the `table` method samples the function at `n` points and creates a table of x and y values.

9.1.2 A First Try on a Class for Parabolas

A parabola $y = c_0 + c_1 x + c_2 x^2$ contains a straight line as a special case ($c_2 = 0$). A class for parabolas will therefore be similar to a class for straight lines. All we have do to is to add the new term $c_2 x^2$ in the function evaluation and store c_2 in the constructor:

```
class Parabola(object):
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

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

9.1.3 A Class for Parabolas Using Inheritance

Python and other languages that support object-oriented programming have a special construct, so that class `Parabola` does not need to repeat the code that we have already written in class `Line`. We can specify that class `Parabola` *inherits* all code from class `Line` by adding `(Line)` in the class headline:

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

Class `Parabola` now automatically gets all the code from class `Line`. Exercise 9.1 asks you to explicitly demonstrate the validity of this assertion. We say that class `Parabola` is *derived* from class `Line`, or equivalently, that class `Parabola` is a subclass of its superclass `Line`.

Now, class `Parabola` should not be identical to class `Line`: it needs to add data in the constructor (for the new term) and to modify the call operator (because of the new term), but the `table` method can be inherited as it is. If we implement the constructor and the call operator in class `Parabola`, these will *override* the inherited versions from class `Line`. If we do not implement a `table` method, the one inherited from class `Line` is available as if it were coded visibly in class `Parabola`.

Class `Parabola` must first have the statements from the class `Line` methods `__call__` and `__init__`, and then add extra code in these methods. An important

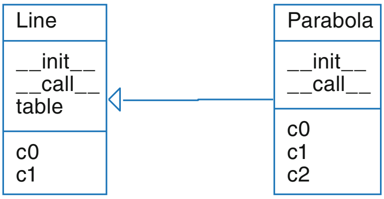


Fig. 9.1 UML diagram for the class hierarchy with superclass `Line` and subclass `Parabola`

principle in computer programming is to avoid repeating code. We should therefore call up functionality in class `Line` instead of copying statements from class `Line` methods to `Parabola` methods. Any method in the superclass `Line` can be called using the syntax

```
Line.methodname(self, arg1, arg2, ...)
# or
super(Parabola, self).methodname(arg1, arg2, ...)
```

The latter construction only works if the super class is derived from Python's general super class `object` (i.e., class `Line` must be a new-style class).

Let us now show how to write class `Parabola` as a subclass of class `Line`, and implement just the new additional code that we need and that is not already written in the superclass:

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)  # let Line store c0 and c1
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

This short implementation of class `Parabola` provides exactly the same functionality as the first version of class `Parabola` that we showed in Sect. 9.1.2 and that did not inherit from class `Line`. Figure 9.1 shows the class hierarchy in UML fashion. The arrow from one class to another indicates inheritance.

A quick demo of the `Parabola` class in a main program,

```
p = Parabola(1, -2, 2)
p1 = p(x=2.5)
print p1
print p.table(0, 1, 3)
```

gives this output:

```
8.5
           0           1
         0.5         0.5
           1           1
```

Program flow The program flow can be somewhat complicated when we work with class hierarchies. Consider the code segment

```
p = Parabola(1, -1, 2)
p1 = p(x=2.5)
```

Let us explain the program flow in detail for these two statements. As always, you can monitor the program flow in a debugger as explained in Sect. F.1 or you can invoke the very illustrative Online Python Tutor[2].

Calling `Parabola(1, -1, 2)` leads to a call to the constructor method `__init__`, where the arguments `c0`, `c1`, and `c2` in this case are `int` objects with values 1, -1, and 2. The `self` argument in the constructor is the object that will be returned and referred to by the variable `p`. Inside the constructor in class `Parabola` we call the constructor in class `Line`. In this latter method, we create two data attributes in the `self` object. Printing out `dir(self)` will explicitly demonstrate what `self` contains so far in the construction process. Back in class `Parabola`'s constructor, we add a third attribute `c2` to the same `self` object. Then the `self` object is invisibly returned and referred to by `p`.

The other statement, `p1 = p(x=2.5)`, has a similar program flow. First we enter the `p.__call__` method with `self` as `p` and `x` as a `float` object with value 2.5. The program flow jumps to the `__call__` method in class `Line` for evaluating the linear part $c_1x + c_0$ of the expression for the parabola, and then the flow jumps back to the `__call__` method in class `Parabola` where we add the new quadratic term.

9.1.4 Checking the Class Type

Python has the function `isinstance(i,t)` for checking if an instance `i` is of class type `t`:

```
>>> l = Line(-1, 1)
>>> isinstance(l, Line)
True
>>> isinstance(l, Parabola)
False
```

A `Line` is not a `Parabola`, but is a `Parabola` a `Line`?

```
>>> p = Parabola(-1, 0, 10)
>>> isinstance(p, Parabola)
True
>>> isinstance(p, Line)
True
```

Yes, from a class hierarchy perspective, a `Parabola` instance is regarded as a `Line` instance too, since it contains everything that a `Line` instance contains.

---

[2] http://www.pythontutor.com/

Every instance has an attribute `__class__` that holds the type of class:

```
>>> p.__class__
<class __main__.Parabola at 0xb68f108c>
>>> p.__class__ == Parabola
True
>>> p.__class__.__name__   # string version of the class name
'Parabola'
```

Note that `p.__class__` is a `class` object (or class definition one may say), while `p.__class__.__name__` is a string. These two variables can be used as an alternative test for the class type:

```
if p.__class__.__name__ == 'Parabola':
    ...
# or
if p.__class__ == Parabola:
    ...
```

However, `isinstance(p, Parabola)` is the recommended programming style for checking the type of an object.

A function `issubclass(c1, c2)` tests if class `c1` is a subclass of class `c2`, e.g.,

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

The superclasses of a class are stored as a tuple in the `__bases__` attribute of the class object:

```
>>> p.__class__.__bases__
(<class __main__.Line at 0xb7c5d2fc>,)
>>> p.__class__.__bases__[0].__name__  # extract name as string
'Line'
```

9.1.5 Attribute vs Inheritance: has-a vs is-a Relationship

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

```
class Parabola(object):
    def __init__(self, c0, c1, c2):
        self.line = Line(c0, c1)  # let Line store c0 and c1
        self.c2 = c2

    def __call__(self, x):
        return self.line(x) + self.c2*x**2
```

Whether to use inheritance or an attribute depends on the problem being solved.

If it is natural to say that class `Parabola` *is* a `Line` object, we say that `Parabola` has an *is-a relationship* with class `Line`. Alternatively, if it is natural to think that class `Parabola` *has a* `Line` object, we speak about a *has-a relationship* with class `Line`. In the present example, we may argue that technically the expression for the parabola *is a* straight line plus another term and hence claim an is-a relationship, but we can also view a parabola as a quantity that *has a* line plus an extra term, which makes the *has-a* relationship relevant.

From a mathematical point of view, many will say that a parabola *is not* a line, but that a line is a special case of a parabola. Adopting this reasoning reverses the dependency of the classes: now it is more natural to let `Line` is a subclass of `Parabola` (`Line` *is a* `Parabola`). This easy, and all we have to do is

```
class Parabola(object):
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c2, c2

    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2

    def table(self, L, R, n):  # implemented as shown above

class Line(Parabola):
    def __init__(self, c0, c1):
        Parabola.__init__(self, c0, c1, 0)
```

The inherited `__call__` method from class `Parabola` will work since the `c2` coefficient is zero. Exercises 9.4 suggests deriving `Parabola` from a general class `Polynomial` and asks you to discuss the alternative class designs.

Extension and restriction of a superclass

In the example where `Parabola` as a subclass of `Line`, we used inheritance to *extend* the functionality of the superclass. The case where `Line` is a subclass of `Parabola` is an example on *restricting* the superclass functionality in a subclass.

How classes depend on each other is influenced by two factors: sharing of code and logical relations. From a sharing of code perspective, many will say that class `Parabola` is naturally a subclass of `Line`, the former adds code to the latter. On the other hand, `Line` is naturally a subclass of `Parabola` from the logical relations in mathematics. Computational efficiency is a third perspective when we implement mathematics. When `Line` is a subclass of `Parabola` we always evaluate the c_2x^2 term in the parabola although this term is zero. Nevertheless, when `Parabola` is a subclass of `Line`, we call `Line.__call__` to evaluate the linear part of the second-degree polynomial, and this call is costly in Python. From a pure efficiency point of view, we would reprogram the linear part in `Parabola.__call__` (which is against the programming habit we have been arguing for!). This little discussion here highlights the many different considerations that come into play when establishing class relations.

9.1.6 Superclass for Defining an Interface

As another example of class hierarchies, we now want to represent functions by classes, as described in Sect. 7.1.2, but in addition to the `__call__` method, we also want to provide methods for the first and second derivative. The class can be sketched as

```
class SomeFunc(object):
    def __init__(self, parameter1, parameter2, ...)
        # Store parameters
    def __call__(self, x):
        # Evaluate function
    def df(self, x):
        # Evaluate the first derivative
    def ddf(self, x):
        # Evaluate the second derivative
```

For a given function, the analytical expressions for first and second derivative must be manually coded. However, we could think of inheriting general functions for computing these derivatives numerically, such that the only thing we must always implement is the function itself. To realize this idea, we create a superclass

```
class FuncWithDerivatives(object):
    def __init__(self, h=1.0E-5):
        self.h = h  # spacing for numerical derivatives

    def __call__(self, x):
        raise NotImplementedError\
        ('___call__ missing in class %s' % self.__class__.__name__)

    def df(self, x):
        """Return the 1st derivative of self.f."""
        # Compute first derivative by a finite difference
        h = self.h
        return (self(x+h) - self(x-h))/(2.0*h)

    def ddf(self, x):
        """Return the 2nd derivative of self.f."""
        # Compute second derivative by a finite difference:
        h = self.h
        return (self(x+h) - 2*self(x) + self(x-h))/(float(h)**2)
```

This class is only meant as a superclass of other classes. For a particular function, say $f(x) = \cos(ax) + x^3$, we represent it by a subclass:

```
class MyFunc(FuncWithDerivatives):
    def __init__(self, a):
        self.a = a

    def __call__(self, x):
        return cos(self.a*x) + x**3
```

```
    def df(self, x):
        a = self.a
        return -a*sin(a*x) + 3*x**2

    def ddf(self, x):
        a = self.a
        return -a*a*cos(a*x) + 6*x
```

The superclass constructor is never called, hence `h` is never initialized, and there are no possibilities for using numerical approximations via the superclass methods `df` and `ddf`. Instead, we override all the inherited methods and implement our own versions.

Tip

Many think it is a good programming style to always call the superclass constructor in a subclass constructor, even in simple classes where we do not need the functionality of the superclass constructor.

For a more complicated function, e.g., $f(x) = \ln|p\tanh(qx\cos rx)|$, we may skip the analytical derivation of the derivatives, and just code $f(x)$ and rely on the difference approximations inherited from the superclass to compute the derivatives:

```
class MyComplicatedFunc(FuncWithDerivatives):
    def __init__(self, p, q, r, h=1.0E-5):
        FuncWithDerivatives.__init__(self, h)
        self.p, self.q, self.r = p, q, r

    def __call__(self, x):
        return log(abs(self.p*tanh(self.q*x*cos(self.r*x))))
```

That's it! We are now ready to use this class:

```
>>> f = MyComplicatedFunc(1, 1, 1)
>>> x = pi/2
>>> f(x)
-36.880306514638988
>>> f.df(x)
-60.593693618216086
>>> f.ddf(x)
3.3217246931444789e+19
```

Class `MyComplicatedFunc` inherits the `df` and `ddf` methods from the superclass `FuncWithDerivatives`. These methods compute the first and second derivatives approximately, provided that we have defined a `__call__` method. If we fail to define this method, we will inherit `__call__` from the superclass, which just raises an exception, saying that the method is not properly implemented in class `MyComplicatedFunc`.

The important message in this subsection is that we introduced a super class to mainly define an *interface*, i.e., the operations (in terms of methods) that one can do with a class in this class hierarchy. The superclass itself is of no direct use, since it does not implement any function evaluation in the `__call__` method. However, it stores a variable common to all subclasses (`h`), and it implements general methods `df` and `ddf` that any subclass can make use of. A specific mathematical function must be represented as a subclass, where the programmer can decide whether analytical derivatives are to be used, or if the more lazy approach of inheriting general functionality (`df` and `ddf`) for computing numerical derivatives is satisfactory.

In object-oriented programming, the superclass very often defines an interface, and instances of the superclass have no applications on their own – only instances of subclasses can do anything useful.

To digest the present material on inheritance, we recommend doing Exercises 9.1–9.4 before reading the next section.

9.2 Class Hierarchy for Numerical Differentiation

Section 7.3.2 presents a class `Derivative` that (approximately) differentiate any mathematical function represented by a callable Python object. The class employs the simplest possible numerical derivative. There are a lot of other numerical formulas for computing approximations to $f'(x)$:

$$f'(x) = \frac{f(x+h)-f(x)}{h} + \mathcal{O}(h), \quad \text{(1st-order forward diff.)} \tag{9.1}$$

$$f'(x) = \frac{f(x)-f(x-h)}{h} + \mathcal{O}(h), \quad \text{(1st-order backward diff.)} \tag{9.2}$$

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + \mathcal{O}(h^2), \quad \text{(2nd-order central diff.)} \tag{9.3}$$

$$f'(x) = \frac{4}{3}\frac{f(x+h)-f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h)-f(x-2h)}{4h} + \mathcal{O}(h^4), \quad \text{(4th-order central diff.)} \tag{9.4}$$

$$f'(x) = \frac{3}{2}\frac{f(x+h)-f(x-h)}{2h} - \frac{3}{5}\frac{f(x+2h)-f(x-2h)}{4h} + \frac{1}{10}\frac{f(x+3h)-f(x-3h)}{6h} + \mathcal{O}(h^6), \quad \text{(6th-order central diff.)} \tag{9.5}$$

$$f'(x) = \frac{1}{h}\left(-\frac{1}{6}f(x+2h) + f(x+h) - \frac{1}{2}f(x) - \frac{1}{3}f(x-h)\right) + \mathcal{O}(h^3), \quad \text{(3rd-order forward diff.)} \tag{9.6}$$

The key ideas about the implementation of such a family of formulas are explained in Sect. 9.2.1. For the interested reader, Sects. 9.2.4–9.2.7 contains more advanced additional material that can well be skipped in a first reading. However, the additional material puts the basic solution in Sect. 9.2.1 into a wider perspective, which may increase the understanding of object orientation.

9.2.1 Classes for Differentiation

It is argued in Sect. 7.3.2 that it is wise to implement a numerical differentiation formula as a class where $f(x)$ and h are data attributes and a `__call__` method makes class instances behave as ordinary Python functions. Hence, when we have a collection of different numerical differentiation formulas, like (9.1)–(9.6), it makes sense to implement each one of them as a class.

Doing this implementation (see Exercise 7.16), we realize that the constructors are identical because their task in the present case to store f and h. Object-orientation is now a natural next step: we can avoid duplicating the constructors by letting all the classes inherit the common constructor code. To this end, we introduce a superclass `Diff` and implement the different numerical differentiation rules in subclasses of `Diff`. Since the subclasses inherit their constructor, all they have to do is to provide a `__call__` method that implements the relevant differentiation formula.

Let us show what the superclass `Diff` looks like and how three subclasses implement the formulas (9.1)–(9.3):

```
class Diff(object):
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)

class Forward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

class Backward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h

class Central2(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)
```

These small classes demonstrates an important feature of object-orientation: code common to many different classes are placed in a superclass, and the subclasses add just the code that differs among the classes.

We can easily implement the formulas (9.4)–(9.6) by following the same method:

```
class Central4(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (4./3)*(f(x+h)   - f(x-h))  /(2*h) - \
               (1./3)*(f(x+2*h) - f(x-2*h))/(4*h)
```

```
class Central6(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (3./2) *(f(x+h)   - f(x-h))  /(2*h) - \
               (3./5) *(f(x+2*h) - f(x-2*h))/(4*h) + \
               (1./10)*(f(x+3*h) - f(x-3*h))/(6*h)

class Forward3(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (-(1./6)*f(x+2*h) + f(x+h) - 0.5*f(x) - \
                (1./3)*f(x-h))/h
```

We have placed all the classes in a module file `Diff.py`. Here is a short interactive example using the module to numerically differentiate the sine function:

```
>>> from Diff import *
>>> from math import sin
>>> mycos = Central4(sin)
>>> mycos(pi)                 # compute sin'(pi)
-1.000000082740371
```

Instead of a plain Python function we may use an object with a `__call__` method, here exemplified through the function $f(t;a,b,c) = at^2 + bt + c$:

```
class Poly2(object):
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
    def __call__(self, t):
        return self.a*t**2 + self.b*t + self.c

f = Poly2(1, 0, 1)
dfdt = Central4(f)
t = 2
print "f'(%g)=%g" % (t, dfdt(t))
```

Let us examine the program flow. When Python encounters `dfdt = Central4(f)`, it looks for the constructor in class `Central4`, but there is no constructor in that class. Python then examines the superclasses of `Central4`, listed in `Central4.__bases__`. The superclass `Diff` contains a constructor, and this method is called. When Python meets the `dfdt(t)` call, it looks for `__call__` in class `Central4` and finds it, so there is no need to examine the superclass. This process of looking up methods of a class is called *dynamic binding*.

Computer science remark Dynamic binding means that a name is bound to a function while the program is running. Normally, in computer languages, a function name is static in the sense that it is hardcoded as part of the function body and will not change during the execution of the program. This principle is known as static binding of function/method names. Object orientation offers the technical means to associate different functions with the same name, which yields a kind of magic for increased flexibility in programs. The particular function that the name refers to can be set at run-time, i.e., when the program is running, and therefore known as dynamic binding.

In Python, dynamic binding is a natural feature since names (variables) can refer to functions and therefore be dynamically bound during execution, just as any ordinary variable. To illustrate this point, let `func1` and `func2` be two Python functions of one argument, and consider the code

```
if input == 'func1':
    f = func1
elif input == 'func2':
    f = func2
y = f(x)
```

Here, the name `f` is bound to one of the `func1` and `func2` function objects while the program is running. This is a result of two features: (i) dynamic typing (so the contents of `f` can change), and (ii) functions being ordinary objects. The bottom line is that dynamic binding comes natural in Python, while it appears more like convenient magic in languages like C++, Java, and C#.

9.2.2 Verification

We have several alternative numerical methods for differentiation implemented in the `Diff` hierarchy, and the `Diff` module should contain one or more test functions for verifying the implementations. The fundamental problem is that even if we know the exact derivative of a function, we do not know what the numerical error in one of the subclass methods is. This fact prevents us from comparing the numerical and the exact derivative.

Fortunately, numerical differentiation formulas of the type we have encountered above are able to differentiate lower order polynomials exactly. All of them are capable of computing $f'(x) = a$, where $f(x) = ax + b$, without approximation errors for any h. We can use this knowledge to construct a test function:

```
def test_Central2():
    def f(x):
        return a*x + b

    def df_exact(x):
        return a

    a = 0.2; b = -4
    df = Central2(f, h=0.55)
    x = 6.2
    msg = 'method Central2 failed: df/dx=%g != %g' % \
          (df(x), df_exact(x))
    tol = 1E-14
    assert abs(df_exact(x) - df(x)) < tol
```

It will be boring to write such a test function for each class in the hierarchy. Therefore, we parameterize the class name and rewrite `test_Central` such that it can be reused for any class in the `Diff` hierarchy:

```
def _test_one_method(method):
    """Test method in string ‘method‘ on a linear function."""
    f = lambda x: a*x + b
    df_exact = lambda x: a
    a = 0.2; b = -4
    df = eval(method)(f, h=0.55)
    x = 6.2
    msg = ’method %s failed: df/dx=%g != %g’ % \
          (method, df(x), df_exact(x))
    tol = 1E-14
    assert abs(df_exact(x) - df(x)) < tol
```

Some comments are needed to explain this function:

- All our test functions are intended for the pytest and nose testing frameworks. (See Sect. H.9 for more information on such test functions.) The function name must then start with `test_` and no arguments are allowed. For the helper function `_test_one_method` with an argument, the function name cannot start with `test`, and that is why an underscore is added.
- Lambda functions (see Sect. 3.1.14) are used to save code in the definitions of `f` and `df_exact`.
- The subclass to be tested is given as a string `method`. Calling the constructor must then be done by `eval(method)(f)`.

It remains to make a loop over all the implemented subclasses and call `_test_one_method` for each of them. As always, we try to find a way to automate boring work, which here consists of listing all the subclasses (and remembering to update the list when new subclasses are added). All global variables in a file is available from the dictionary returned by `globals()`. The key is a variable name and the value is the corresponding object. For example, `print globals()` reveals that all the defined classes are in `globals()`, e.g.,

```
’Central2’: <class Diff.Central2 at 0x1a87c80>,
’Central4’: <class Diff.Central4 at 0x1a87f58>,
’Diff’: <class Diff.Diff at 0x1a870b8>,
```

To find all the relevant classes to test, we grab all names from the `globals()` dictionary, look for names that starts with upper case, and find the names that correspond to a subclass of `Diff` (drop `Diff` itself as this class cannot compute anything and therefore cannot be tested). Translating this algorithm to code gives us a test function that can test all subclasses in the `Diff` hierarchy:

```
def test_all_methods():
    """Call _test_one_method for all subclasses of Diff."""
    print globals()
    names = list(globals().keys())  # all names in this module
    for name in names:
        if name[0].isupper():
            if issubclass(eval(name), Diff):
                if name != ’Diff’:
                    _test_one_method(name)
```

9.2.3 A flexible Main Program

As a demonstration of the power of Python programming, we shall now write a main program for our `Diff` module that accepts a function on the command-line, together with information about the difference type (centered, backward, or forward), the order of the approximation, and a value of the independent variable. The corresponding output is the derivative of the given function. An example of the usage of the program goes like this:

```
Terminal
Diff.py 'exp(sin(x))' Central 2 3.1
-1.04155573055
```

Here, we asked the program to differentiate $f(x) = e^{\sin x}$ at $x = 3.1$ with a central scheme of order 2 (using the `Central2` class in the `Diff` hierarchy).

We can provide any expression with `x` as input and request any scheme from the `Diff` hierarchy, and the derivative will be (approximately) computed. One great thing with Python is that the code is very short:

```
from math import *  # make all math functions available to main

def main():
    from scitools.StringFunction import StringFunction
    import sys

    try:
        formula = sys.argv[1]
        difftype = sys.argv[2]
        difforder = sys.argv[3]
        x = float(sys.argv[4])
    except IndexError:
        print 'Usage:   Diff.py formula difftype difforder x'
        print 'Example: Diff.py "sin(x)*exp(-x)" Central 4 3.14'
        sys.exit(1)

    classname = difftype + difforder
    f = StringFunction(formula)
    df = eval(classname)(f)
    print df(x)

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

Read the code line by line, and convince yourself that you understand what is going on. You may need to review Sects. 4.3.1 and 4.3.3.

One disadvantage is that the code above is limited to `x` as the name of the independent variable. If we allow a 5th command-line argument with the name of the independent variable, we can pass this name on to the `StringFunction` constructor, and suddenly our program works with any name for the independent variable!

```
varname = sys.argv[5]
f = StringFunction(formula, independent_variables=varname)
```

Of course, the program crashes if we do not provide five command-line arguments, and the program does not work properly if we are not careful with ordering of the command-line arguments. There is some way to go before the program is really user friendly, but that is beyond the scope of this chapter.

Many other popular programming languages (C++, Java, C#) cannot perform the `eval` operation while the program is running. The result is that one needs `if` tests to turn the information in `difftype` and `difforder` into creation of subclass instances. Such type of code would look like this in Python:

```
if classname == 'Forward1':
    df = Forward1(f)
elif classname == 'Backward1':
    df = Backward1(f)
...
```

and so forth. This piece of code is very common in object-oriented systems and often put in a function that is referred to as a *factory function*. Thanks to `eval` in Python, factory functions are usually only a matter of applying `eval` to a string.

9.2.4 Extensions

The great advantage of sharing code via inheritance becomes obvious when we want to extend the functionality of a class hierarchy. It is possible to do this by adding more code to the superclass only. Suppose we want to be able to assess the accuracy of the numerical approximation to the derivative by comparing with the exact derivative, if available. All we need to do is to allow an extra argument in the constructor and provide an additional superclass method that computes the error in the numerical derivative. We may add this code to class `Diff`, or we may add it in a subclass `Diff2` and let the other classes for various numerical differentiation formulas inherit from class `Diff2`. We follow the latter approach:

```
class Diff2(Diff):
    def __init__(self, f, h=1E-5, dfdx_exact=None):
        Diff.__init__(self, f, h)
        self.exact = dfdx_exact

    def error(self, x):
        if self.exact is not None:
            df_numerical = self(x)
            df_exact = self.exact(x)
            return df_exact - df_numerical

class Forward1(Diff2):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

The other subclasses, `Backward1`, `Central2`, and so on, must also be derived from `Diff2` to equip all subclasses with new functionality for perfectly assessing

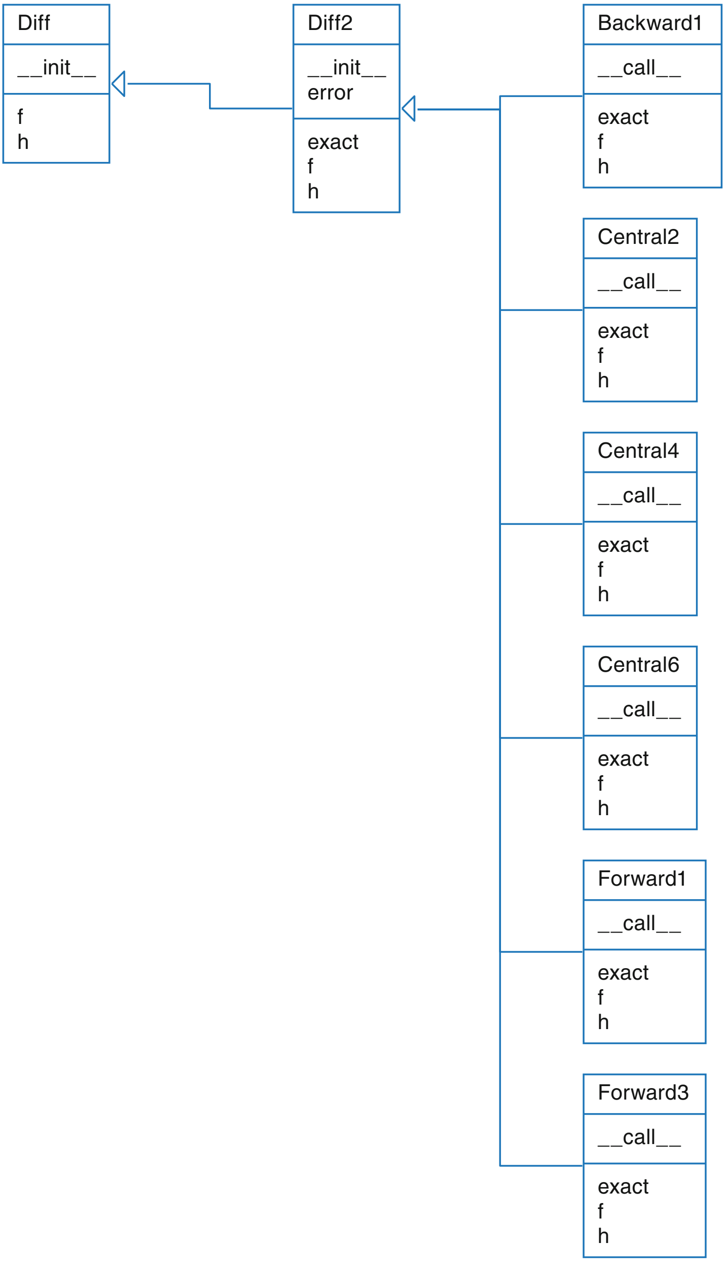


Fig. 9.2 UML diagram of the `Diff` hierarchy for a series of differentiation formulas (`Backward1`, `Central2`, etc.)

the accuracy of the approximation. No other modifications are necessary in this example, since all the subclasses can inherit the superclass constructor and the `error` method. Figure 9.2 shows a UML diagram of the new `Diff` class hierarchy.

Here is an example of usage:

```
mycos = Forward1(sin, dfdx_exact=cos)
print 'Error in derivative is', mycos.error(x=pi)
```

The program flow of the `mycos.error(x=pi)` call can be interesting to follow. We first enter the `error` method in class `Diff2`, which then calls `self(x)`, i.e., the `__call__` method in class `Forward1`, which jumps out to the `self.f` function, i.e., the `sin` function in the `math` module in the present case. After returning to the `error` method, the next call is to `self.exact`, which is the `cos` function (from `math`) in our case.

Application We can apply the methods in the `Diff2` hierarchy to get some insight into the accuracy of various difference formulas. Let us write out a table where the rows correspond to different h values, and the columns correspond to different approximation methods (except the first column, which reflects the h value). The values in the table can be the numerically computed $f'(x)$ or the error in this approximation if the exact derivative is known. The following function writes such a table:

```
def table(f, x, h_values, methods, dfdx=None):
    # Print headline (h and class names for the methods)
    print '      h       ',
    for method in methods:
        print '%-15s' % method.__name__,
    print  # newline
    # Print table
    for h in h_values:
        print '%10.2E' % h,
        for method in methods:
            if dfdx is not None:       # write error
                d = method(f, h, dfdx)
                output = d.error(x)
            else:                      # write value
                d = method(f, h)
                output = d(x)
            print '%15.8E' % output,
        print  # newline
```

The next lines tries three approximation methods on $f(x) = e^{-10x}$ for $x = 0$ and with $h = 1, 1/2, 1/4, 1/16, \ldots, 1/512$:

```
from Diff2 import *
from math import exp

def f1(x):
    return exp(-10*x)

def df1dx(x):
    return -10*exp(-10*x)

table(f1, 0, [2**(-k) for k in range(10)],
      [Forward1, Central2, Central4], df1dx)
```

Note how convenient it is to make a list of class names – class names can be used as ordinary variables, and to print the class name as a string we just use the `__name__` attribute. The output of the main program above becomes

```
      h          Forward1         Central2         Central4
   1.00E+00 -9.00004540E+00  1.10032329E+04 -4.04157586E+07
   5.00E-01 -8.01347589E+00  1.38406421E+02 -3.48320240E+03
   2.50E-01 -6.32833999E+00  1.42008179E+01 -2.72010498E+01
   1.25E-01 -4.29203837E+00  2.81535264E+00 -9.79802452E-01
   6.25E-02 -2.56418286E+00  6.63876231E-01 -5.32825724E-02
   3.12E-02 -1.41170013E+00  1.63556996E-01 -3.21608292E-03
   1.56E-02 -7.42100948E-01  4.07398036E-02 -1.99260429E-04
   7.81E-03 -3.80648092E-01  1.01756309E-02 -1.24266603E-05
   3.91E-03 -1.92794011E-01  2.54332554E-03 -7.76243120E-07
   1.95E-03 -9.70235594E-02  6.35795004E-04 -4.85085874E-08
```

From one row to the next, h is halved, and from about the 5th row and onwards, the `Forward1` errors are also halved, which is consistent with the error $\mathcal{O}(h)$ of this method. Looking at the 2nd column, we see that the errors are reduced to 1/4 when going from one row to the next, at least after the 5th row. This is also according to the theory since the error is proportional to h^2. For the last row with a 4th-order scheme, the error is reduced by 1/16, which again is what we expect when the error term is $\mathcal{O}(h^4)$. What is also interesting to observe, is the benefit of using a higher-order scheme like `Central4`: with, for example, $h = 1/128$ the `Forward1` scheme gives an error of -0.7, `Central2` improves this to 0.04, while `Central4` has an error of -0.0002. More accurate formulas definitely give better results. (Strictly speaking, it is the fraction of the work and the accuracy that counts: `Central4` needs four function evaluations, while `Central2` and `Forward1` only needs two.) The test example shown here is found in the file `Diff2_examples.py`.

9.2.5 Alternative Implementation via Functions

Could we implement the functionality offered by the `Diff` hierarchy of objects by using plain functions and no object orientation? The answer is "yes, almost". What we have to pay for a pure function-based solution is a less friendly user interface to the differentiation functionality: more arguments must be supplied in function calls, because each difference formula, now coded as a straight Python function, must get $f(x)$, x, and h as arguments. In the class version we first store f and h as data attributes in the constructor, and every time we want to compute the derivative, we just supply x as argument.

A Python function for implementing numerical differentiation reads

```
def central2_func(f, x, h=1.0E-5):
    return (f(x+h) - f(x-h))/(2*h)
```

The usage demonstrates the difference from the class solution:

```
mycos = central2_func(sin, pi, 1E-6)
# Compute sin'(pi):
print "g'(%g)=%g (exact value is %g)" % (pi, mycos, cos(pi))
```

Now, `mycos` is a number, not a callable object. The nice thing with the class solution is that `mycos` appeared to be a standard Python function whose mathematical values equal the derivative of the Python function `sin(x)`. But does it matter whether `mycos` is a function or a number? Yes, it matters if we want to apply the difference formula twice to compute the second-order derivative. When `mycos` is a callable object of type `Central2`, we just write

```
mysin = Central2(mycos)
# or
mysin = Central2(Central2(sin))

# Compute g''(pi):
print "g''(%g)=%g" % (pi, mysin(pi))
```

With the `central2_func` function, this composition will not work. Moreover, when the derivative is an object, we can send this object to any algorithm that expects a mathematical function, and such algorithms include numerical integration, differentiation, interpolation, ordinary differential equation solvers, and finding zeros of equations, so the applications are many.

9.2.6 Alternative Implementation via Functional Programming

As a conclusion of the previous section, the great benefit of the object-oriented solution in Sect. 9.2.1 is that one can have some subclass instance `d` from the `Diff` (or `Diff2`) hierarchy and write `d(x)` to evaluate the derivative at a point `x`. The `d(x)` call behaves as if `d` were a standard Python function containing a manually coded expression for the derivative.

The `d(x)` interface to the derivative can also be obtained by other and perhaps more direct means than object-oriented programming. In programming languages where functions are ordinary objects that can be referred to by variables, as in Python, one can make a function that returns the right `d(x)` function according to the chosen numerical derivation rule. The code looks as this (see `Diff_functional.py` for the complete code):

```
def differentiate(f, method, h=1.0E-5):
    h = float(h)  # avoid integer division

    if method == 'Forward1':
        def Forward1(x):
            return (f(x+h) - f(x))/h
    return Forward1

    elif method == 'Backward1':
        def Backward1(x):
            return (f(x) - f(x-h))/h
    return Backward1
    ...
```

And the usage goes like

```
mycos = differentiate(sin,   'Forward1')
mysin = differentiate(mycos, 'Forward1')
x = pi
print mycos(x), cos(x), mysin, -sin(x)
```

The surprising thing is that when we call `mycos(x)` we provide only `x`, while the function itself looks like

```
def Forward1(x):
    return (f(x+h) - f(x))/h
return Forward1
```

How do the parameters `f` and `h` get their values when we call `mycos(x)`? There is some magic attached to the `Forward1` function, or literally, there are some variables attached to `Forward1`: this function remembers the values of `f` and `h` that existed as local variables in the `differentiate` function when the `Forward1` function was defined.

In computer science terms, `Forward1` always has access to variables in the *scope* in which the function was defined. The `Forward1` function is call a *closure* and explained in Sect. 7.1.7. Closures are much used in a programming style called *functional programming*. Two key features of functional programming is operations on lists (like list comprehensions) and returning functions from functions. Python supports functional programming, but we will not consider this programming style further in this book.

9.2.7 Alternative Implementation via a Single Class

Instead of making many classes or functions for the many different differentiation schemes, the basic information about the schemes can be stored in one table. With a single method in one single class can use the table information, and for a given scheme, compute the derivative. To do this, we need to reformulate the mathematical problem (actually by using ideas from Sect. 9.3.1).

A family of numerical differentiation schemes can be written

$$f'(x) \approx h^{-1} \sum_{i=-r}^{r} w_i f(x_i), \tag{9.7}$$

where w_i are weights and x_i are points. The $2r + 1$ points are symmetric around some point x:

$$x_i = x + ih, \quad i = -r, \ldots, r\,.$$

The weights depend on the differentiation scheme. For example, the Midpoint scheme (9.3) has

$$w_{-1} = -1, \quad w_0 = 0, \quad w_1 = 1\,.$$

The table below lists the values of w_i for different difference formulas. The type of difference is abbreviated with c for central, f for forward, and b for backward. The number after the nature of a scheme denotes the order of the schemes (for example, "c 2" is a central difference of 2nd order). We have set $r = 4$, which is sufficient for the schemes written up in this book.

	$x-4h$	$x-3h$	$x-2h$	$x-h$	x	$x+h$	$x+2h$	$x+3h$	$x+4h$
c 2	0	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0
c 4	0	0	$\frac{1}{12}$	$-\frac{2}{3}$	0	$\frac{2}{3}$	$-\frac{1}{12}$	0	0
c 6	0	$-\frac{1}{60}$	$\frac{3}{20}$	$-\frac{3}{4}$	0	$\frac{3}{4}$	$-\frac{3}{20}$	$\frac{1}{60}$	0
c 8	$\frac{1}{280}$	$-\frac{4}{105}$	$\frac{12}{60}$	$-\frac{4}{5}$	0	$\frac{4}{5}$	$-\frac{12}{60}$	$\frac{4}{105}$	$-\frac{1}{280}$
f 1	0	0	0	0	1	1	0	0	0
f 3	0	0	0	$-\frac{2}{6}$	$-\frac{1}{2}$	1	$-\frac{1}{6}$	0	0
b 1	0	0	0	-1	1	0	0	0	0

Given a table of the w_i values, we can use (9.7) to compute the derivative. A faster, vectorized computation can have the x_i, w_i, and $f(x_i)$ values as stored in three vectors. Then $h^{-1}\sum_i w_i f(x_i)$ can be interpreted as a dot product between the two vectors with components w_i and $f(x_i)$, respectively.

A class with the table of weights as a static variable, a constructor, and a `__call__` method for evaluating the derivative via $h^{-1}\sum_i w_i f(x_i)$ looks as follows:

```
class Diff3(object):
    table = {
    (’forward’, 1):
    [0, 0, 0, 0, 1, 1, 0, 0, 0],
    (’central’, 2):
    [0, 0, 0, -1./2, 0, 1./2, 0, 0, 0],
    (’central’, 4):
    [ 0, 0, 1./12, -2./3, 0, 2./3, -1./12, 0, 0],
    ...
    }
    def __init__(self, f, h=1.0E-5, type=’central’, order=2):
        self.f, self.h, self.type, self.order = f, h, type, order
        self.weights = np.array(Diff2.table[(type, order)])

    def __call__(self, x):
        f_values = np.array([f(self.x+i*self.h) \
                             for i in range(-4,5)])
        return np.dot(self.weights, f_values)/self.h
```

Here we used numpy's `dot(x, y)` function for computing the inner or dot product between two arrays `x` and `y`.

Class `Diff3` can be found in the file `Diff3.py`. Using class `Diff3` to differentiate the sine function goes like this:

```
import Diff3
mycos = Diff3.Diff3(sin, type=’central’, order=4)
print "sin’(pi):", mycos(pi)
```

Remark The downside of class `Diff3`, compared with the other implementation techniques, is that the sum $h^{-1}\sum_i w_i f(x_i)$ contains many multiplications by zero for lower-order schemes. These multiplications are known to yield zero in advance so we waste computer resources on trivial calculations. Once upon a time, programmers would have been extremely careful to avoid wasting multiplications this way, but today arithmetic operations are quite cheap, especially compared to fetching data from the computer's memory. Lots of other factors also influence the computational efficiency of a program, but this is beyond the scope of this book.

9.3 Class Hierarchy for Numerical Integration

There are many different numerical methods for integrating a mathematical function, just as there are many different methods for differentiating a function. It is thus obvious that the idea of object-oriented programming and class hierarchies can be applied to numerical integration formulas in the same manner as we did in Sect. 9.2.

9.3.1 Numerical Integration Methods

First, we list some different methods for integrating $\int_a^b f(x)dx$ using n evaluation points. All the methods can be written as

$$\int\limits_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i), \tag{9.8}$$

where w_i are weights and x_i are evaluation points, $i = 0, \ldots, n-1$. The Midpoint method has

$$x_i = a + \frac{h}{2} + ih, \quad w_i = h, \quad h = \frac{b-a}{n}, \quad i = 0, \ldots, n-1\,. \tag{9.9}$$

The Trapezoidal method has the points

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}, \quad i = 0, \ldots, n-1, \tag{9.10}$$

and the weights

$$w_0 = w_{n-1} = \frac{h}{2}, \; w_i = h, \quad i = 1, \ldots, n-2\,. \tag{9.11}$$

Simpson's rule has the same evaluation points as the Trapezoidal rule, but

$$h = 2\frac{b-a}{n-1}, \quad w_0 = w_{n-1} = \frac{h}{6}, \tag{9.12}$$

$$w_i = \frac{h}{3} \quad \text{for } i = 2, 4, \ldots, n-3, \tag{9.13}$$

$$w_i = \frac{2h}{3} \quad \text{for } i = 1, 3, 5, \ldots, n-2\,. \tag{9.14}$$

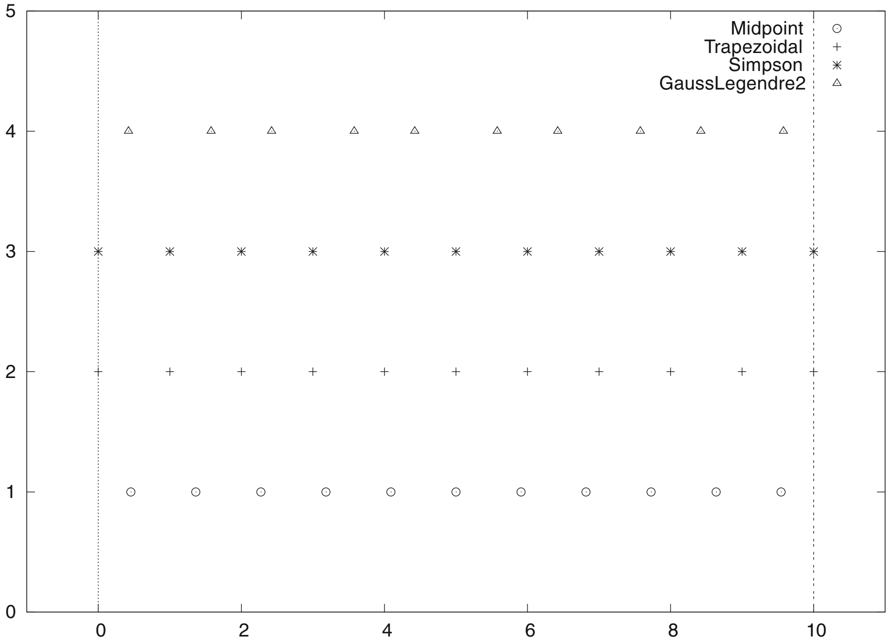


Fig. 9.3 Illustration of the distribution of points for various numerical integration methods. The Gauss-Legendre method has 10 points, while the other methods have 11 points in [0, 10]

Note that n must be odd in Simpson's rule. A Two-Point Gauss-Legendre method takes the form

$$x_i = a + \left(i + \frac{1}{2}\right)h - \frac{1}{\sqrt{3}}\frac{h}{2} \quad \text{for } i = 0, 2, 4, \ldots, n-2, \tag{9.15}$$

$$x_i = a + \left(i + \frac{1}{2}\right)h + \frac{1}{\sqrt{3}}\frac{h}{2} \quad \text{for } i = 1, 3, 5, \ldots, n-1, \tag{9.16}$$

with $h = 2(b-a)/n$. Here n must be even. All the weights have the same value: $w_i = h/2$, $i = 0, \ldots, n-1$. Figure 9.3 illustrates how the points in various integration rules are distributed over a few intervals.

9.3.2 Classes for Integration

We may store x_i and w_i in two NumPy arrays and compute the integral as $\sum_{i=0}^{n-1} w_i f(x_i)$. This operation can also be vectorized as a dot (inner) product between the w_i vector and the $f(x_i)$ vector, provided $f(x)$ is implemented in a vectorizable form.

We argued in Sect. 7.3.3 that it pays off to implement a numerical integration formula as a class. If we do so with the different methods from the previous section, a typical class looks like this:

```
class SomeIntegrationMethod(object):
    def __init__(self, a, b, n):
        # Compute self.points and self.weights

    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
```

Making such classes for many different integration methods soon reveals that all the classes contain common code, namely the `integrate` method for computing $\sum_{i=0}^{n-1} w_i f(x_i)$. Therefore, this common code can be placed in a superclass, and subclasses can just add the code that is specific to a certain numerical integration formula, namely the definition of the weights w_i and the points x_i.

Let us start with the superclass:

```
class Integrator(object):
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.construct_method()

    def construct_method(self):
        raise NotImplementedError('no rule in class %s' %
                                  self.__class__.__name__)

    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
```

As we have seen, we store the a, b, and n data about the integration method in the constructor. Moreover, we compute arrays or lists `self.points` for the x_i points and `self.weights` for the w_i weights. All this code can now be inherited by all subclasses.

The initialization of points and weights is put in a separate method, `construct_method`, which is supposed to be implemented in each subclass, but the superclass provides a default implementation, which tells the user that the method is not implemented. What happens is that when subclasses redefine a method, that method overrides the method inherited from the superclass. Hence, if we forget to redefine `construct_method` in a subclass, we will inherit the one from the superclass, and this method issues an error message. The construction of this error message is quite clever in the sense that it will tell in which class the `construct_method` method is missing (`self` will be the subclass instance and its `__class__.__name__` is a string with the corresponding subclass name).

In computer science one usually speaks about *overloading* a method in a subclass, but the words redefining and overriding are also used. A method that is overloaded is said to be *polymorphic*. A related term, *polymorphism*, refers to coding with polymorphic methods. Very often, a superclass provides some default implementation of a method, and a subclass overloads the method with the purpose of tailoring the method to a particular application.

The `integrate` method is common for all integration rules, i.e., for all subclasses, so it can be inherited as it is. A vectorized version can also be added in the superclass to make it automatically available also in all subclasses:

```
    def vectorized_integrate(self, f):
        return np.dot(self.weights, f(self.points))
```

Let us then implement a subclass. Only the `construct_method` method needs to be written. For the Midpoint rule, this is a matter of translating the formulas in (9.9) to Python:

```
class Midpoint(Integrator):
    def construct_method(self):
        a, b, n = self.a, self.b, self.n  # quick forms
        h = (b-a)/float(n)
        x = np.linspace(a + 0.5*h, b - 0.5*h, n)
        w = np.zeros(len(x)) + h
        return x, w
```

Observe that we implemented directly a vectorized code. We could also have used (slow) loops and explicit indexing:

```
        x = np.zeros(n)
        w = np.zeros(n)
        for i in range(n):
            x[i] = a + 0.5*h + i*h
            w[i] = h
```

Before we continue with other subclasses for other numerical integration formulas, we will have a look at the program flow when we use class `Midpoint`. Suppose we want to integrate $\int_0^2 x^2 dx$ using 101 points:

```
def f(x): return x*x
m = Midpoint(0, 2, 101)
print m.integrate(f)
```

How is the program flow? The assignment to `m` invokes the constructor in class `Midpoint`. Since this class has no constructor, we invoke the inherited one from the superclass `Integrator`. Here data attributes are stored, and then the `construct_method` method is called. Since `self` is a `Midpoint` instance, it is the `construct_method` in the `Midpoint` class that is invoked, even if there is a method with the same name in the superclass. Class `Midpoint` overloads `construct_method` in the superclass. In a way, we "jump down" from the constructor in class `Integrator` to the `construct_method` in the `Midpoint` class. The next statement, `m.integrate(f)`, just calls the inherited `integral` method that is common to all subclasses.

The points and weights for a Trapezoidal rule can be implemented in a vectorized way in another subclass with name `Trapezoidal`:

```
class Trapezoidal(Integrator):
    def construct_method(self):
        x = np.linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)
        w = np.zeros(len(x)) + h
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

Observe how we divide the first and last weight by 2, using index 0 (the first) and -1 (the last) and the /= operator (`a /= b` is equivalent to `a = a/b`). We could also have implemented a scalar version with loops. The relevant code is in function `trapezoidal` in Sect. 7.3.3.

Class Simpson has a slightly more demanding rule, at least if we want to vectorize the expression, since the weights are of two types.

```
class Simpson(Integrator):
    def construct_method(self):
        if self.n % 2 != 1:
            print 'n=%d must be odd, 1 is added' % self.n
            self.n += 1
        x = np.linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)*2
        w = np.zeros(len(x))
        w[0:self.n:2] = h*1.0/3
        w[1:self.n-1:2] = h*2.0/3
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

We first control that we have an odd number of points, by checking that the remainder of `self.n` divided by two is 1. If not, an exception could be raised, but for smooth operation of the class, we simply increase n so it becomes odd. Such automatic adjustments of input is not a rule to be followed in general. Wrong input is best notified explicitly. However, sometimes it is user friendly to make small adjustments of the input, as we do here, to achieve a smooth and successful operation. (In cases like this, a user might become uncertain whether the answer can be trusted if she (later) understands that the input should not yield a correct result. Therefore, do the adjusted computation, and provide a notification to the user about what has taken place.)

The computation of the weights `w` in class `Simpson` applies slices with stride (jump/step) 2 such that the operation is vectorized for speed. Recall that the upper limit of a slice is not included in the set, so `self.n-1` is the largest index in the first case, and `self.n-2` is the largest index in the second case. Instead of the vectorized operation of slices for computing `w`, we could use (slower) straight loops:

```
        for i in range(0, self.n, 2):
            w[i] = h*1.0/3
        for i in range(1, self.n-1, 2):
            w[i] = h*2.0/3
```

The points in the Two-Point Gauss-Legendre rule are slightly more complicated to calculate, so here we apply straight loops to make a safe first implementation:

```
class GaussLegendre2(Integrator):
    def construct_method(self):
        if self.n % 2 != 0:
            print ’n=%d must be even, 1 is subtracted’ % self.n
            self.n -= 1
        nintervals = int(self.n/2.0)
        h = (self.b - self.a)/float(nintervals)
        x = np.zeros(self.n)
        sqrt3 = 1.0/math.sqrt(3)
    for i in range(nintervals):
            x[2*i]   = self.a + (i+0.5)*h - 0.5*sqrt3*h
            x[2*i+1] = self.a + (i+0.5)*h + 0.5*sqrt3*h
        w = np.zeros(len(x)) + h/2.0
        return x, w
```

A vectorized calculation of `x` is possible by observing that the `(i+0.5)*h` expression can be computed by `np.linspace`, and then we can add the remaining two terms:

```
m = np.linspace(0.5*h, (nintervals-1+0.5)*h, nintervals)
x[0:self.n-1:2] = m + self.a - 0.5*sqrt3*h
x[1:self.n:2]   = m + self.a + 0.5*sqrt3*h
```

The array on the right-hand side has half the length of `x` ($n/2$), but the length matches exactly the slice with stride 2 on the left-hand side.

The code snippets above are found in the module file `integrate.py`.

9.3.3 Verification

To verify the implementation we use the fact that all the subclasses implement methods that can integrate a linear function exactly. A suitable test function is therefore

```
def test_Integrate():
    """Check that linear functions are integrated exactly."""
    def f(x):
        return x + 2

    def F(x):
        """Integral of f."""
        return 0.5*x**2 + 2*x

    a = 2; b = 3; n = 4     # test data
    I_exact = F(b) - F(a)
    tol = 1E-15

```

```
    methods = [Midpoint, Trapezoidal, Simpson, GaussLegendre2,
               GaussLegendre2_vec]
    for method in methods:
        integrator = method(a, b, n)

        I = integrator.integrate(f)
        assert abs(I_exact - I) < tol

        I_vec = integrator.vectorized_integrate(f)
        assert abs(I_exact - I_vec) < tol
```

A stronger method of verification is to compute how the error varies with n. Exercise 9.15 explains the details.

9.3.4 Using the Class Hierarchy

To verify the implementation, we first try to integrate a linear function. All methods should compute the correct integral value regardless of the number of evaluation points:

```
def f(x):
    return x + 2

a = 2; b = 3; n = 4
for Method in Midpoint, Trapezoidal, Simpson, GaussLegendre2:
    m = Method(a, b, n)
    print m.__class__.__name__, m.integrate(f)
```

Observe how we simply list the class names as a tuple (comma-separated objects), and `Method` will in the `for` loop attain the values `Midpoint`, `Trapezoidal`, and so forth. For example, in the first pass of the loop, `Method(a, b, n)` is identical to `Midpoint(a, b, n)`.

The output of the test above becomes

```
Midpoint 4.5
Trapezoidal 4.5
n=4 must be odd, 1 is added
Simpson 4.5
GaussLegendre2 4.5
```

Since $\int_2^3 (x+2)dx = \frac{9}{2} = 4.5$, all methods passed this simple test.

A more challenging integral, from a numerical point of view, is

$$\int\limits_0^1 \left(1 + \frac{1}{m}\right) t^{\frac{1}{m}} dt = 1\,.$$

To use any subclass in the `Integrator` hierarchy, the integrand must be a function of one variable only. For the present integrand, which depends on t and m, we use a class to represent it:

```
class F(object):
    def __init__(self, m):
        self.m = float(m)

    def __call__(self, t):
        m = self.m
        return (1 + 1/m)*t**(1/m)
```

We now ask the question: how much is the error in the integral reduced as we increase the number of integration points (n)? It appears that the error decreases exponentially with n, so if we want to plot the errors versus n, it is best to plot the logarithm of the error versus $\ln n$. We expect this graph to be a straight line, and the steeper the line is, the faster the error goes to zero as n increases. A common conception is to regard one numerical method as better than another if the error goes faster to zero as we increase the computational work (here n).

For a given m and method, the following function computes two lists containing the logarithm of the n values, and the logarithm of the corresponding errors in a series of experiments:

```
def error_vs_n(f, exact, n_values, Method, a, b):
    log_n = []  # log of actual n values (Method may adjust n)
    log_e = []  # log of corresponding errors
    for n_value in n_values:
        method = Method(a, b, n_value)
        error = abs(exact - method.integrate(f))
        log_n.append(log(method.n))
        log_e.append(log(error))
    return log_n, log_e
```

We can plot the error versus n for several methods in the same plot and make one plot for each m value. The loop over m below makes such plots:

```
n_values = [10, 20, 40, 80, 160, 320, 640]
for m in 1./4, 1./8., 2, 4, 16:
    f = F(m)
    figure()
    for Method in Midpoint, Trapezoidal, \
            Simpson, GaussLegendre2:
        n, e = error_vs_n(f, 1, n_values, Method, 0, 1)
        plot(n, e); legend(Method.__name__); hold('on')
    title('m=%g' % m); xlabel('ln(n)'); ylabel('ln(error)')
```

The code snippets above are collected in a function `test` in the `integrate.py` file.

The plots for $m > 1$ look very similar. The plots for $0 < m < 1$ are also similar, but different from the $m > 1$ cases. Let us have a look at the results for $m = 1/4$ and $m = 2$. The first, $m = 1/4$, corresponds to $\int_0^1 5x^4 dx$. Figure 9.4 shows that the error curves for the Trapezoidal and Midpoint methods converge more slowly compared to the error curves for Simpson's rule and the Gauss-Legendre method. This is the usual situation for these methods, and mathematical analysis of the methods can confirm the results in Fig. 9.4.

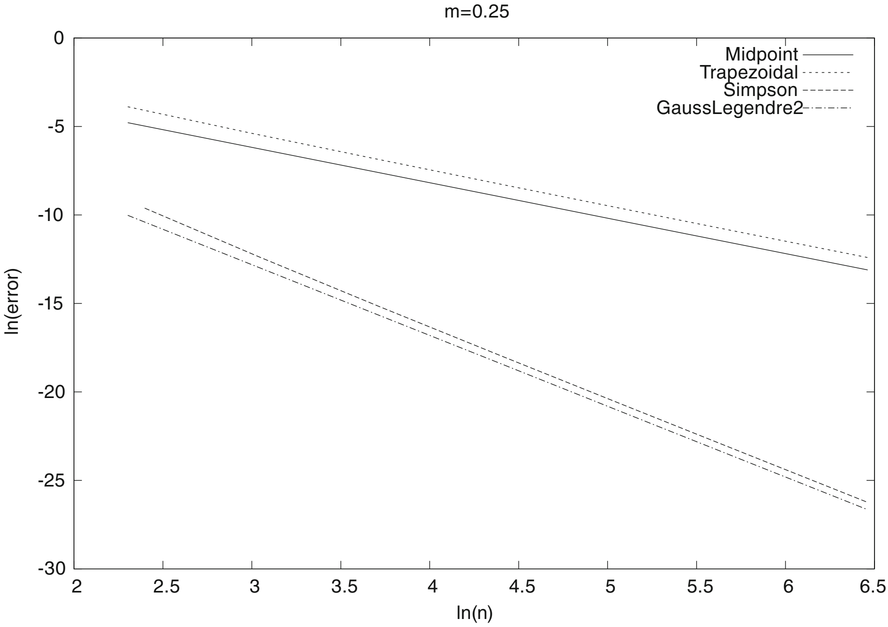


Fig. 9.4 The logarithm of the error versus the logarithm of integration points for integral $5x^4$ computed by the Trapezoidal and Midpoint methods (*upper two lines*), and Simpson's rule and the Gauss-Legendre methods (*lower two lines*)

However, when we consider the integral $\int_0^1 \frac{3}{2}\sqrt{x}dx$, $(m = 2)$ and $m > 1$ in general, all the methods converge with the same speed, as shown in Fig. 9.5. Our integral is difficult to compute numerically when $m > 1$, and the theoretically better methods (Simpson's rule and the Gauss-Legendre method) do not converge faster than the simpler methods. The difficulty is due to the infinite slope (derivative) of the integrand at $x = 0$.

9.3.5 About Object-Oriented Programming

From an implementational point of view, the advantage of class hierarchies in Python is that we can save coding by inheriting functionality from a superclass. In programming languages where each variable must be specified with a fixed type, class hierarchies are particularly useful because a function argument with a special type also works with all subclasses of that type. Suppose we have a function where we need to integrate:

```
def do_math(arg1, arg2, integrator):
    ...
    I = integrator.integrate(myfunc)
    ...
```

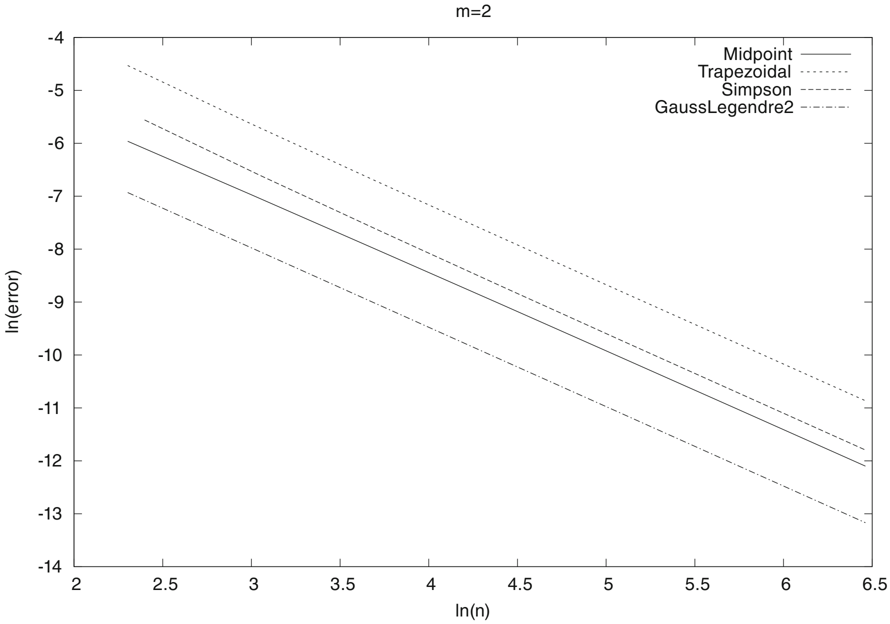


Fig. 9.5 The logarithm of the error versus the logarithm of integration points for integral $\frac{3}{2}\sqrt{x}$ computed by the Trapezoidal method and Simpson's rule (*upper two lines*), and Midpoint and Gauss-Legendre methods (*lower two lines*)

That is, `integrator` must be an instance of some class, or a module, such that the syntax `integrator.integrate(myfunc)` corresponds to a function call, but nothing more (like having a particular type) is demanded.

This Python code will run as long as `integrator` has a method `integrate` taking one argument. In other languages, the function arguments are specified with a type, say in Java we would write

```
void do_math(double arg1, int arg2, Simpson integrator)
```

A compiler will examine all calls to `do_math` and control that the arguments are of the right type. Instead of specifying the integration method to be of type `Simpson`, one can in Java and other object-oriented languages specify `integrator` to be of the superclass type `Integrator`:

```
void do_math(double arg1, int arg2, Integrator integrator)
```

Now it is allowed to pass an object of any subclass type of `Integrator` as the third argument. That is, this method works with `integrator` of type `Midpoint`, `Trapezoidal`, `Simpson`, etc., not just one of them. Class hierarchies and object-oriented programming are therefore important means for parameterizing away types in languages like Java, C++, and C#. We do not need to parameterize types in Python, since arguments are not declared with a fixed type. Object-oriented programming is hence not so technically important in Python as in other languages for providing increased flexibility in programs.

Is there then any use for object-oriented programming beyond inheritance? The answer is yes! For many code developers object-oriented programming is not just a technical way of sharing code, but it is more a way of modeling the world, and understanding the problem that the program is supposed to solve. In mathematical applications we already have objects, defined by the mathematics, and standard programming concepts such as functions, arrays, lists, and loops are often sufficient for solving simpler problems. In the non-mathematical world the concept of objects is very useful because it helps to structure the problem to be solved. As an example, think of the phone book and message list software in a mobile phone. Class `Person` can be introduced to hold the data about one person in the phone book, while class `Message` can hold data related to an SMS message. Clearly, we need to know who sent a message so a `Message` object will have an associated `Person` object, or just a phone number if the number is not registered in the phone book. Classes help to structure both the problem and the program. The impact of classes and object-oriented programming on modern software development can hardly be exaggerated.

A good, real-world, pedagogical example on inheritance is the class hierarchy for numerical methods for ordinary differential equations described in Sect. E.2.

9.4 Class Hierarchy for Making Drawings

Implementing a drawing program provides a very good example on the usefulness of object-oriented programming. In the following we shall develop the simpler parts of a relatively small and compact drawing program for making sketches of the type shown in Fig. 9.6. This is a typical *principal sketch* of a physics problem, here involving a rolling wheel on an inclined plane. The sketch is made up many individual elements: a rectangle filled with a pattern (the inclined plane), a hollow circle with color (the wheel), arrows with labels (the N and Mg forces, and the x axis), an angle with symbol θ, and a dashed line indicating the starting location of the wheel.

Drawing software and plotting programs can produce such figures quite easily in principle, but the amount of details the user needs to control with the mouse can be substantial. Software more tailored to producing sketches of this type would work with more convenient abstractions, such as circle, wall, angle, force arrow, axis, and so forth. And as soon we start *programming* to construct the figure we get a range of other powerful tools at disposal. For example, we can easily translate and rotate parts of the figure and make an animation that illustrates the physics of the problem. Programming as a superior alternative to interactive drawing is the mantra of this section.

Classes are very suitable for implementing the various components that build up a sketch. In particular, we shall demonstrate that as soon as some classes are established, more are easily added. Enhanced functionality for all the classes is also easy to implement in common, generic code that can immediately be shared by all present and future classes.

The fundamental data structure involved in this case study is a hierarchical tree, and much of the material on implementation issues targets how to traverse tree

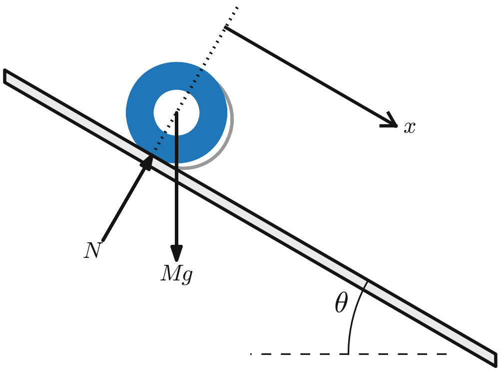


Fig. 9.6 Sketch of a physics problem

structures with recursive function calls. This topic is of key relevance in a wide range of other applications as well.

9.4.1 Using the Object Collection

We start by demonstrating a convenient user interface for making sketches of the type in Fig. 9.6. However, it is more appropriate to start with a significantly simpler example as depicted in Fig. 9.7. This toy sketch consists of several elements: two circles, two rectangles, and a “ground” element.

Basic drawing A typical program creating these five elements is shown next. After importing the `pysketcher` package, the first task is always to define a coordinate system:

```
from pysketcher import *

drawing_tool.set_coordinate_system(
    xmin=0, xmax=10, ymin=-1, ymax=8)
```

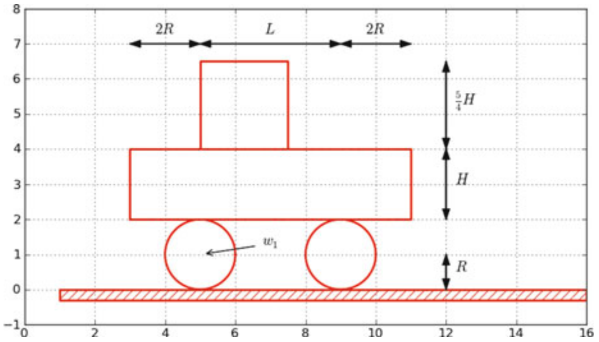


Fig. 9.7 Sketch of a simple figure

Instead of working with lengths expressed by specific numbers it is highly recommended to use variables to parameterize lengths as this makes it easier to change dimensions later. Here we introduce some key lengths for the radius of the wheels, distance between the wheels, etc.:

```
R = 1    # radius of wheel
L = 4    # distance between wheels
H = 2    # height of vehicle body
w_1 = 5  # position of front wheel
drawing_tool.set_coordinate_system(xmin=0, xmax=w_1 + 2*L + 3*R,
                                   ymin=-1, ymax=2*R + 3*H)
```

With the drawing area in place we can make the first `Circle` object in an intuitive fashion:

```
wheel1 = Circle(center=(w_1, R), radius=R)
```

to change dimensions later.

To translate the geometric information about the `wheel1` object to instructions for the plotting engine (in this case Matplotlib), one calls the `wheel1.draw()`. To display all drawn objects, one issues `drawing_tool.display()`. The typical steps are hence:

```
wheel1 = Circle(center=(w_1, R), radius=R)
wheel1.draw()

# Define other objects and call their draw() methods
drawing_tool.display()
drawing_tool.savefig('tmp.png')  # store picture
```

The next wheel can be made by taking a copy of `wheel1` and translating the object to the right according to a displacement vector $(L, 0)$:

```
wheel2 = wheel1.copy()
wheel2.translate((L,0))
```

The two rectangles are also made in an intuitive way:

```
under = Rectangle(lower_left_corner=(w_1-2*R, 2*R),
                  width=2*R + L + 2*R, height=H)
over  = Rectangle(lower_left_corner=(w_1, 2*R + H),
                  width=2.5*R, height=1.25*H)
```

Groups of objects Instead of calling the `draw` method of every object, we can group objects and call `draw`, or perform other operations, for the whole group. For example, we may collect the two wheels in a `wheels` group and the `over` and `under` rectangles in a `body` group. The whole vehicle is a composition of its `wheels` and `body` groups. The code goes like

```
wheels  = Composition({'wheel1': wheel1, 'wheel2': wheel2})
body    = Composition({'under': under, 'over': over})

vehicle = Composition({'wheels': wheels, 'body': body})
```

The ground is illustrated by an object of type `Wall`, mostly used to indicate walls in sketches of mechanical systems. A `Wall` takes the `x` and `y` coordinates of some curve, and a `thickness` parameter, and creates a thick curve filled with a simple pattern. In this case the curve is just a flat line so the construction is made of two points on the ground line ($(w_1 - L, 0)$ and $(w_1 + 3L, 0)$):

```
ground = Wall(x=[w_1 - L, w_1 + 3*L], y=[0, 0], thickness=-0.3*R)
```

The negative thickness makes the pattern-filled rectangle appear below the defined line, otherwise it appears above.

We may now collect all the objects in a "top" object that contains the whole figure:

```
fig = Composition({'vehicle': vehicle, 'ground': ground})
fig.draw()  # send all figures to plotting backend
drawing_tool.display()
drawing_tool.savefig('tmp.png')
```

The `fig.draw()` call will visit all subgroups, their subgroups, and so forth in the hierarchical tree structure of figure elements, and call `draw` for every object.

Changing line styles and colors Controlling the line style, line color, and line width is fundamental when designing figures. The `pysketcher` package allows the user to control such properties in single objects, but also set global properties that are used if the object has no particular specification of the properties. Setting the global properties are done like

```
drawing_tool.set_linestyle('dashed')
drawing_tool.set_linecolor('black')
drawing_tool.set_linewidth(4)
```

At the object level the properties are specified in a similar way:

```
wheels.set_linestyle('solid')
wheels.set_linecolor('red')
```

and so on.

Geometric figures can be specified as *filled*, either with a color or with a special visual pattern:

```
# Set filling of all curves
drawing_tool.set_filled_curves(color='blue', pattern='/')

# Turn off filling of all curves
drawing_tool.set_filled_curves(False)
```

```
# Fill the wheel with red color
wheel1.set_filled_curves('red')
```

The figure composition as an object hierarchy The composition of objects making up the figure is hierarchical, similar to a family, where each object has a parent and a number of children. Do a `print fig` to display the relations:

```
ground
    wall
vehicle
    body
        over
            rectangle
        under
            rectangle
    wheels
        wheel1
            arc
        wheel2
            arc
```

The indentation reflects how deep down in the hierarchy (family) we are. This output is to be interpreted as follows:

- `fig` contains two objects, `ground` and `vehicle`
- `ground` contains an object `wall`
- `vehicle` contains two objects, `body` and `wheels`
- `body` contains two objects, `over` and `under`
- `wheels` contains two objects, `wheel1` and `wheel2`

In this listing there are also objects not defined by the programmer: `rectangle` and `arc`. These are of type `Curve` and automatically generated by the classes `Rectangle` and `Circle`.

More detailed information can be printed by

```
print fig.show_hierarchy('std')
```

yielding the output

```
ground (Wall):
    wall (Curve): 4 coords fillcolor='white' fillpattern='/'
vehicle (Composition):
    body (Composition):
        over (Rectangle):
            rectangle (Curve): 5 coords
        under (Rectangle):
            rectangle (Curve): 5 coords
    wheels (Composition):
        wheel1 (Circle):
            arc (Curve): 181 coords
        wheel2 (Circle):
            arc (Curve): 181 coords
```

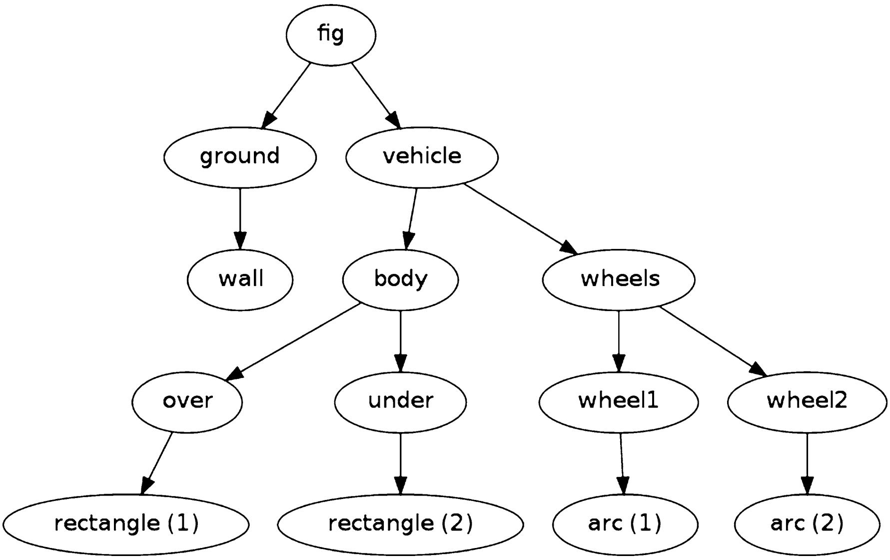


Fig. 9.8 Hierarchical relation between figure objects

Here we can see the class type for each figure object, how many coordinates that are involved in basic figures (`Curve` objects), and special settings of the basic figure (fillcolor, line types, etc.). For example, `wheel2` is a `Circle` object consisting of an `arc`, which is a `Curve` object consisting of 181 coordinates (the points needed to draw a smooth circle). The `Curve` objects are the only objects that really holds specific coordinates to be drawn. The other object types are just compositions used to group parts of the complete figure.

One can also get a graphical overview of the hierarchy of figure objects that build up a particular figure `fig`. Just call `fig.graphviz_dot('fig')` to produce a file `fig.dot` in the *dot format*. This file contains relations between parent and child objects in the figure and can be turned into an image, as in Fig. 9.8, by running the `dot` program:

Terminal

```
Terminal> dot -Tpng -o fig.png fig.dot
```

The call `fig.graphviz_dot('fig', classname=True)` makes a `fig.dot` file where the class type of each object is also visible, see Fig. 9.9. The ability to write out the object hierarchy or view it graphically can be of great help when working with complex figures that involve layers of subfigures.

Any of the objects can in the program be reached through their names, e.g.,

```
fig['vehicle']
fig['vehicle']['wheels']
fig['vehicle']['wheels']['wheel2']
fig['vehicle']['wheels']['wheel2']['arc']
```

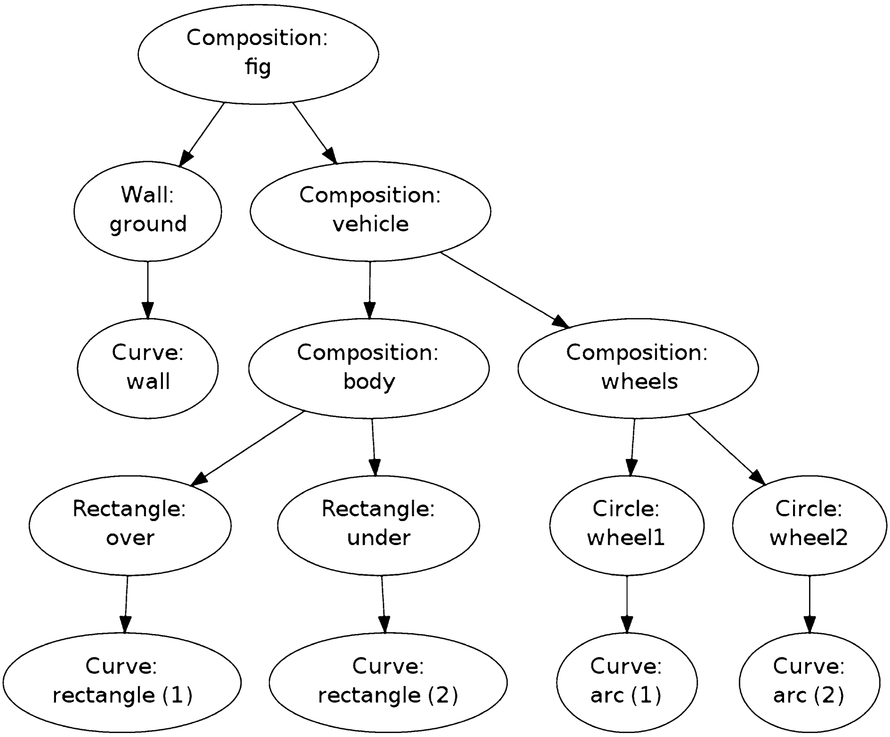


Fig. 9.9 Hierarchical relation between figure objects, including their class names

```
fig['vehicle']['wheels']['wheel2']['arc'].x  # x coords
fig['vehicle']['wheels']['wheel2']['arc'].y  # y coords
fig['vehicle']['wheels']['wheel2']['arc'].linestyle
fig['vehicle']['wheels']['wheel2']['arc'].linetype
```

Grabbing a part of the figure this way is handy for changing properties of that part, for example, colors, line styles (see Fig. 9.10):

```
fig['vehicle']['wheels'].set_filled_curves('blue')
fig['vehicle']['wheels'].set_linewidth(6)
fig['vehicle']['wheels'].set_linecolor('black')

fig['vehicle']['body']['under'].set_filled_curves('red')

fig['vehicle']['body']['over'].set_filled_curves(pattern='/')
fig['vehicle']['body']['over'].set_linewidth(14)
fig['vehicle']['body']['over']['rectangle'].linewidth = 4
```

The last line accesses the `Curve` object directly, while the line above, accesses the `Rectangle` object, which will then set the linewidth of its `Curve` object, and other objects if it had any. The result of the actions above is shown in Fig. 9.10.

We can also change position of parts of the figure and thereby make animations, as shown next.

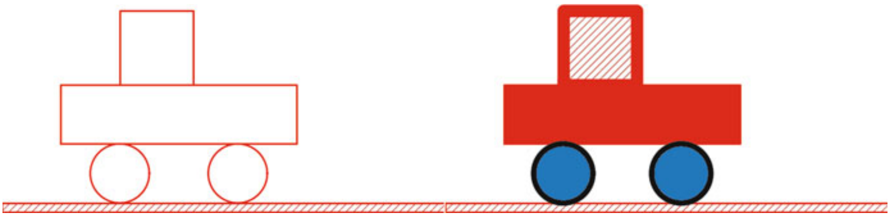


Fig. 9.10 *Left*: Basic line-based drawing. *Right*: Thicker lines and filled parts

Animation: translating the vehicle Can we make our little vehicle roll? A first attempt will be to fake rolling by just displacing all parts of the vehicle. The relevant parts constitute the `fig['vehicle']` object. This part of the figure can be translated, rotated, and scaled. A translation along the ground means a translation in x direction, say a length L to the right:

```
fig['vehicle'].translate((L,0))
```

You need to erase, draw, and display to see the movement:

```
drawing_tool.erase()
fig.draw()
drawing_tool.display()
```

Without erasing, the old drawing of the vehicle will remain in the figure so you get two vehicles. Without `fig.draw()` the new coordinates of the vehicle will not be communicated to the drawing tool, and without calling display the updated drawing will not be visible.

A figure that moves in time is conveniently realized by the function `animate`:

```
animate(fig, tp, action)
```

Here, `fig` is the entire figure, `tp` is an array of time points, and `action` is a user-specified function that changes `fig` at a specific time point. Typically, `action` will move parts of `fig`.

In the present case we can define the movement through a velocity function `v(t)` and displace the figure `v(t)*dt` for small time intervals `dt`. A possible velocity function is

```
def v(t):
    return -8*R*t*(1 - t/(2*R))
```

Our action function for horizontal displacements `v(t)*dt` becomes

```
def move(t, fig):
    x_displacement = dt*v(t)
    fig['vehicle'].translate((x_displacement, 0))
```

Since our velocity is negative for $t \in [0, 2R]$ the displacement is to the left.

The `animate` function will for each time point `t` in `tp` erase the drawing, call `action(t, fig)`, and show the new figure by `fig.draw()` and `drawing_tool.display()`. Here we choose a resolution of the animation corresponding to 25 time points in the time interval $[0, 2R]$:

```
import numpy
tp = numpy.linspace(0, 2*R, 25)
dt = tp[1] - tp[0]  # time step

animate(fig, tp, move, pause_per_frame=0.2)
```

The `pause_per_frame` adds a pause, here 0.2 seconds, between each frame in the animation.

We can also ask `animate` to store each frame in a file:

```
files = animate(fig, tp, move_vehicle, moviefiles=True,
                pause_per_frame=0.2)
```

The `files` variable, here `'tmp_frame_%04d.png'`, is the printf-specification used to generate the individual plot files. We can use this specification to make a video file via `ffmpeg` (or `avconv` on Debian-based Linux systems such as Ubuntu). Videos in the Flash and WebM formats can be created by

```
Terminal> ffmpeg -r 12 -i tmp_frame_%04d.png -vcodec flv mov.flv
Terminal> ffmpeg -r 12 -i tmp_frame_%04d.png -vcodec libvpx mov.webm
```

An animated GIF movie can also be made using the `convert` program from the ImageMagick software suite:

```
Terminal> convert -delay 20 tmp_frame*.png mov.gif
Terminal> animate mov.gif  # play movie
```

The delay between frames, in units of 1/100 s, governs the speed of the movie. To play the animated GIF file in a web page, simply insert `<img src="mov.gif">` in the HTML code.

The individual PNG frames can be directly played in a web browser by running

```
Terminal> scitools movie output_file=mov.html fps=5 tmp_frame*
```

or calling

```
from scitools.std import movie
movie(files, encoder='html', output_file='mov.html')
```

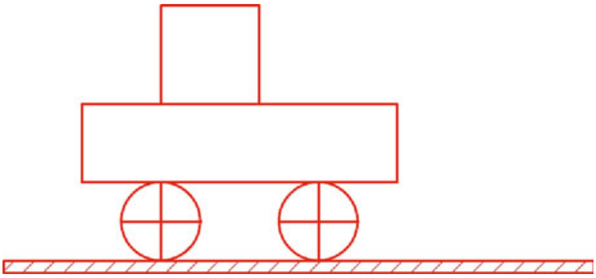


Fig. 9.11 Wheels with spokes to illustrate rolling

in Python. Load the resulting file `mov.html` into a web browser to play the movie.

Try to run `vehicle0.py` and then load `mov.html` into a browser, or play one of the `mov.*` video files. Alternatively, you can view a ready-made movie[3].

Animation: rolling the wheels It is time to show rolling wheels. To this end, we add spokes to the wheels, formed by two crossing lines, see Fig. 9.11. The construction of the wheels will now involve a circle and two lines:

```
wheel1 = Composition({
    'wheel': Circle(center=(w_1, R), radius=R),
    'cross': Composition({'cross1': Line((w_1,0),   (w_1,2*R)),
                          'cross2': Line((w_1-R,R), (w_1+R,R))})})
wheel2 = wheel1.copy()
wheel2.translate((L,0))
```

Observe that `wheel1.copy()` copies all the objects that make up the first wheel, and `wheel2.translate` translates all the copied objects.

The `move` function now needs to displace all the objects in the entire vehicle and also rotate the `cross1` and `cross2` objects in both wheels. The rotation angle follows from the fact that the arc length of a rolling wheel equals the displacement of the center of the wheel, leading to a rotation angle

```
angle = - x_displacement/R
```

With `w_1` tracking the x coordinate of the center of the front wheel, we can rotate that wheel by

```
w1 = fig['vehicle']['wheels']['wheel1']
from math import degrees
w1.rotate(degrees(angle), center=(w_1, R))
```

The `rotate` function takes two parameters: the rotation angle (in degrees) and the center point of the rotation, which is the center of the wheel in this case. The other wheel is rotated by

---

[3] http://tinyurl.com/oou9lp7/mov-tut/vehicle0.html

```
w2 = fig['vehicle']['wheels']['wheel2']
w2.rotate(degrees(angle), center=(w_1 + L, R))
```

That is, the angle is the same, but the rotation point is different. The update of the center point is done by `w_1 += x_displacement`. The complete `move` function with translation of the entire vehicle and rotation of the wheels then becomes

```
w_1 = w_1 + L   # start position

def move(t, fig):
    x_displacement = dt*v(t)
    fig['vehicle'].translate((x_displacement, 0))

    # Rotate wheels
    global w_1
    w_1 += x_displacement
    # R*angle = -x_displacement
    angle = - x_displacement/R
    w1 = fig['vehicle']['wheels']['wheel1']
    w1.rotate(degrees(angle), center=(w_1, R))
    w2 = fig['vehicle']['wheels']['wheel2']
    w2.rotate(degrees(angle), center=(w_1 + L, R))
```

The complete example is found in the file `vehicle1.py`. You may run this file or watch a ready-made movie[4].

The advantages with making figures this way, through programming rather than using interactive drawing programs, are numerous. For example, the objects are parameterized by variables so that various dimensions can easily be changed. Subparts of the figure, possible involving a lot of figure objects, can change color, linetype, filling or other properties through a *single* function call. Subparts of the figure can be rotated, translated, or scaled. Subparts of the figure can also be copied and moved to other parts of the drawing area. However, the single most important feature is probably the ability to make animations governed by mathematical formulas or data coming from physics simulations of the problem, as shown in the example above.

9.4.2 Example of Classes for Geometric Objects

We shall now explain how we can, quite easily, realize software with the capabilities demonstrated in the previous examples. Each object in the figure is represented as a class in a class hierarchy. Using inheritance, classes can inherit properties from parent classes and add new geometric features.

We introduce class `Shape` as superclass for all specialized objects in a figure. This class does not store any data, but provides a series of functions that add functionality to all the subclasses. This will be shown later.

Simple geometric objects One simple subclass is `Rectangle`, specified by the coordinates of the lower left corner and its width and height:

---

[4] http://tinyurl.com/oou9lp7/mov-tut/vehicle1.html

```
class Rectangle(Shape):
    def __init__(self, lower_left_corner, width, height):
        p = lower_left_corner  # short form
        x = [p[0], p[0] + width,
             p[0] + width, p[0], p[0]]
        y = [p[1], p[1], p[1] + height,
             p[1] + height, p[1]]
        self.shapes = {'rectangle': Curve(x,y)}
```

Any subclass of `Shape` will have a constructor that takes geometric information about the shape of the object and creates a dictionary `self.shapes` with the shape built of simpler shapes. The most fundamental shape is `Curve`, which is just a collection of (x, y) coordinates in two arrays `x` and `y`. Drawing the `Curve` object is a matter of plotting `y` versus `x`. For class `Rectangle` the `x` and `y` arrays contain the corner points of the rectangle in counterclockwise direction, starting and ending with in the lower left corner.

Class `Line` is also a simple class:

```
class Line(Shape):
    def __init__(self, start, end):
        x = [start[0], end[0]]
        y = [start[1], end[1]]
        self.shapes = {'line': Curve(x, y)}
```

Here we only need two points, the start and end point on the line. However, we may want to add some useful functionality, e.g., the ability to give an x coordinate and have the class calculate the corresponding y coordinate:

```
    def __call__(self, x):
        """Given x, return y on the line."""
        x, y = self.shapes['line'].x, self.shapes['line'].y
        self.a = (y[1] - y[0])/(x[1] - x[0])
        self.b = y[0] - self.a*x[0]
        return self.a*x + self.b
```

Unfortunately, this is too simplistic because vertical lines cannot be handled (infinite `self.a`). The true source code of `Line` therefore provides a more general solution at the cost of significantly longer code with more tests.

A circle implies a somewhat increased complexity. Again we represent the geometric object by a `Curve` object, but this time the `Curve` object needs to store a large number of points on the curve such that a plotting program produces a visually smooth curve. The points on the circle must be calculated manually in the constructor of class `Circle`. The formulas for points (x, y) on a curve with radius R and center at (x_0, y_0) are given by

$$x = x_0 + R\cos(t),$$
$$y = y_0 + R\sin(t),$$

where $t \in [0, 2\pi]$. A discrete set of t values in this interval gives the corresponding set of (x, y) coordinates on the circle. The user must specify the resolution as the number of t values. The circle's radius and center must of course also be specified.

We can write the `Circle` class as

```
class Circle(Shape):
    def __init__(self, center, radius, resolution=180):
        self.center, self.radius = center, radius
        self.resolution = resolution

        t = linspace(0, 2*pi, resolution+1)
        x0 = center[0];  y0 = center[1]
        R = radius
        x = x0 + R*cos(t)
        y = y0 + R*sin(t)
        self.shapes = {'circle': Curve(x, y)}
```

As in class `Line` we can offer the possibility to give an angle θ (equivalent to t in the formulas above) and then get the corresponding x and y coordinates:

```
    def __call__(self, theta):
        """Return (x, y) point corresponding to angle theta."""
        return self.center[0] + self.radius*cos(theta), \
               self.center[1] + self.radius*sin(theta)
```

There is one flaw with this method: it yields illegal values after a translation, scaling, or rotation of the circle.

A part of a circle, an arc, is a frequent geometric object when drawing mechanical systems. The arc is constructed much like a circle, but t runs in $[\theta_s, \theta_s + \theta_a]$. Giving θ_s and θ_a the slightly more descriptive names `start_angle` and `arc_angle`, the code looks like this:

```
class Arc(Shape):
    def __init__(self, center, radius,
                 start_angle, arc_angle,
                 resolution=180):
        self.start_angle = radians(start_angle)
        self.arc_angle = radians(arc_angle)

        t = linspace(self.start_angle,
                     self.start_angle + self.arc_angle,
                     resolution+1)
        x0 = center[0];  y0 = center[1]
        R = radius
        x = x0 + R*cos(t)
        y = y0 + R*sin(t)
        self.shapes = {'arc': Curve(x, y)}
```

Having the `Arc` class, a `Circle` can alternatively be defined as a subclass specializing the arc to a circle:

```
class Circle(Arc):
    def __init__(self, center, radius, resolution=180):
        Arc.__init__(self, center, radius, 0, 360, resolution)
```

Class curve Class `Curve` sits on the coordinates to be drawn, but how is that done? The constructor of class `Curve` just stores the coordinates, while a method `draw` sends the coordinates to the plotting program to make a graph. Or more precisely, to avoid a lot of (e.g.) Matplotlib-specific plotting commands in class `Curve` we have created a small layer with a simple programming interface to plotting programs. This makes it straightforward to change from Matplotlib to another plotting program. The programming interface is represented by the `drawing_tool` object and has a few functions:

- `plot_curve` for sending a curve in terms of x and y coordinates to the plotting program,
- `set_coordinate_system` for specifying the graphics area,
- `erase` for deleting all elements of the graph,
- `set_grid` for turning on a grid (convenient while constructing the figure),
- `set_instruction_file` for creating a separate file with all plotting commands (Matplotlib commands in our case),
- a series of `set_X` functions where `X` is some property like `linecolor`, `linestyle`, `linewidth`, `filled_curves`.

This is basically all we need to communicate to a plotting program.

Any class in the `Shape` hierarchy inherits `set_X` functions for setting properties of curves. This information is propagated to all other shape objects in the `self.shapes` dictionary. Class `Curve` stores the line properties together with the coordinates of its curve and propagates this information to the plotting program. When saying `vehicle.set_linewidth(10)`, all objects that make up the `vehicle` object will get a `set_linewidth(10)` call, but only the `Curve` object at the end of the chain will actually store the information and send it to the plotting program.

A rough sketch of class `Curve` reads

```
class Curve(Shape):
    """General curve as a sequence of (x,y) coordintes."""
    def __init__(self, x, y):
        self.x = asarray(x, dtype=float)
        self.y = asarray(y, dtype=float)

    def draw(self):
        drawing_tool.plot_curve(
            self.x, self.y,
            self.linestyle, self.linewidth, self.linecolor, ...)

    def set_linewidth(self, width):
        self.linewidth = width

    det set_linestyle(self, style):
        self.linestyle = style
    ...
```

Compound geometric objects The simple classes `Line`, `Arc`, and `Circle` could can the geometric shape through just one `Curve` object. More complicated shapes are built from instances of various subclasses of `Shape`. Classes used for professional drawings soon get quite complex in composition and have a lot of geometric details, so here we prefer to make a very simple composition: the already drawn vehicle from Fig. 9.7. That is, instead of composing the drawing in a Python program as shown above, we make a subclass `Vehicle0` in the `Shape` hierarchy for doing the same thing.

The `Shape` hierarchy is found in the `pysketcher` package, so to use these classes or derive a new one, we need to import `pysketcher`. The constructor of class `Vehicle0` performs approximately the same statements as in the example program we developed for making the drawing in Fig. 9.7.

```
from pysketcher import *

class Vehicle0(Shape):
    def __init__(self, w_1, R, L, H):
        wheel1 = Circle(center=(w_1, R), radius=R)
        wheel2 = wheel1.copy()
        wheel2.translate((L,0))

        under = Rectangle(lower_left_corner=(w_1-2*R, 2*R),
                          width=2*R + L + 2*R, height=H)
        over  = Rectangle(lower_left_corner=(w_1, 2*R + H),
                          width=2.5*R, height=1.25*H)

        wheels = Composition(
            {'wheel1': wheel1, 'wheel2': wheel2})
        body = Composition(
            {'under': under, 'over': over})

        vehicle = Composition({'wheels': wheels, 'body': body})
        xmax = w_1 + 2*L + 3*R
        ground = Wall(x=[R, xmax], y=[0, 0], thickness=-0.3*R)

        self.shapes = {'vehicle': vehicle, 'ground': ground}
```

Any subclass of `Shape` *must* define the `shapes` attribute, otherwise the inherited `draw` method (and a lot of other methods too) will not work.

The painting of the vehicle, as shown in the right part of Fig. 9.10, could in class `Vehicle0` be offered by a method:

```
    def colorful(self):
        wheels = self.shapes['vehicle']['wheels']
        wheels.set_filled_curves('blue')
        wheels.set_linewidth(6)
        wheels.set_linecolor('black')
        under = self.shapes['vehicle']['body']['under']
        under.set_filled_curves('red')
        over = self.shapes['vehicle']['body']['over']
        over.set_filled_curves(pattern='/')
        over.set_linewidth(14)
```

The usage of the class is simple: after having set up an appropriate coordinate system as previously shown, we can do

```
vehicle = Vehicle0(w_1, R, L, H)
vehicle.draw()
drawing_tool.display()
```

and go on the make a painted version by

```
drawing_tool.erase()
vehicle.colorful()
vehicle.draw()
drawing_tool.display()
```

A complete code defining and using class `Vehicle0` is found in the file `vehicle2.py`.

The `pysketcher` package contains a wide range of classes for various geometrical objects, particularly those that are frequently used in drawings of mechanical systems.

9.4.3 Adding Functionality via Recursion

The really powerful feature of our class hierarchy is that we can add much functionality to the superclass `Shape` and to the "bottom" class `Curve`, and then all other classes for various types of geometrical shapes immediately get the new functionality. To explain the idea we may look at the `draw` method, which all classes in the `Shape` hierarchy must have. The inner workings of the `draw` method explain the secrets of how a series of other useful operations on figures can be implemented.

Basic principles of recursion Note that we work with two types of hierarchies in the present documentation: one Python *class hierarchy*, with `Shape` as superclass, and one *object hierarchy* of figure elements in a specific figure. A subclass of `Shape` stores its figure in the `self.shapes` dictionary. This dictionary represents the object hierarchy of figure elements for that class. We want to make one `draw` call for an instance, say our class `Vehicle0`, and then we want this call to be propagated to *all* objects that are contained in `self.shapes` and all is nested subdictionaries. How is this done?

The natural starting point is to call `draw` for each `Shape` object in the `self.shapes` dictionary:

```
def draw(self):
    for shape in self.shapes:
        self.shapes[shape].draw()
```

This general method can be provided by class `Shape` and inherited in subclasses like `Vehicle0`. Let `v` be a `Vehicle0` instance. Seemingly, a call `v.draw()` just calls

```
v.shapes['vehicle'].draw()
v.shapes['ground'].draw()
```

However, in the former call we call the `draw` method of a `Composition` object whose `self.shapes` attributed has two elements: `wheels` and `body`. Since class `Composition` inherits the same `draw` method, this method will run through `self.shapes` and call `wheels.draw()` and `body.draw()`. Now, the `wheels` object is also a `Composition` with the same `draw` method, which will run through `self.shapes`, now containing the `wheel1` and `wheel2` objects. The `wheel1` object is a `Circle`, so calling `wheel1.draw()` calls the `draw` method in class `Circle`, but this is the same `draw` method as shown above. This method will therefore traverse the circle's `shapes` dictionary, which we have seen consists of one `Curve` element.

The `Curve` object holds the coordinates to be plotted so here `draw` really needs to do something "physical", namely send the coordinates to the plotting program. The `draw` method is outlined in the short listing of class `Curve` shown previously.

We can go to any of the other shape objects that appear in the figure hierarchy and follow their `draw` calls in the similar way. Every time, a `draw` call will invoke a new `draw` call, until we eventually hit a `Curve` object at the "bottom" of the figure hierarchy, and then that part of the figure is really plotted (or more precisely, the coordinates are sent to a plotting program).

When a method calls itself, such as `draw` does, the calls are known as *recursive* and the programming principle is referred to as *recursion*. This technique is very often used to traverse hierarchical structures like the figure structures we work with here. Even though the hierarchy of objects building up a figure are of different types, they all inherit the same `draw` method and therefore exhibit the same behavior with respect to drawing. Only the `Curve` object has a different `draw` method, which does not lead to more recursion.

Explaining recursion Understanding recursion is usually a challenge. To get a better idea of how recursion works, we have equipped class `Shape` with a method `recurse` that just visits all the objects in the `shapes` dictionary and prints out a message for each object. This feature allows us to trace the execution and see exactly where we are in the hierarchy and which objects that are visited.

The `recurse` method is very similar to `draw`:

```
    def recurse(self, name, indent=0):
        # print message where we are (name is where we come from)
        for shape in self.shapes:
            # print message about which object to visit
            self.shapes[shape].recurse(indent+2, shape)
```

The `indent` parameter governs how much the message from this `recurse` method is intended. We increase `indent` by 2 for every level in the hierarchy, i.e., every row of objects in Fig. 9.12. This indentation makes it easy to see on the printout how far down in the hierarchy we are.

A typical message written by `recurse` when `name` is `'body'` and the `shapes` dictionary has the keys `'over'` and `'under'`, will be

```
     Composition: body.shapes has entries 'over', 'under'
     call body.shapes["over"].recurse("over", 6)
```

The number of leading blanks on each line corresponds to the value of `indent`. The code printing out such messages looks like

```
    def recurse(self, name, indent=0):
        space = ’ ’*indent
        print space, ’%s: %s.shapes has entries’ % \
              (self.__class__.__name__, name), \
              str(list(self.shapes.keys()))[1:-1]

        for shape in self.shapes:
            print space,
            print ’call %s.shapes["%s"].recurse("%s", %d)’ % \
                  (name, shape, shape, indent+2)
            self.shapes[shape].recurse(shape, indent+2)
```

Let us follow a `v.recurse(’vehicle’)` call in detail, `v` being a `Vehicle0` instance. Before looking into the output from `recurse`, let us get an overview of the figure hierarchy in the `v` object (as produced by `print v`)

```
ground
    wall
vehicle
    body
        over
            rectangle
        under
            rectangle
    wheels
        wheel1
            arc
        wheel2
            arc
```

The `recurse` method performs the same kind of traversal of the hierarchy, but writes out and explains a lot more.

The data structure represented by `v.shapes` is known as a *tree*. As in physical trees, there is a *root*, here the `v.shapes` dictionary. A graphical illustration of the tree (upside down) is shown in Fig. 9.12. From the root there are one or more branches, here two: `ground` and `vehicle`. Following the `vehicle` branch, it has two new branches, `body` and `wheels`. Relationships as in family trees are often used to describe the relations in object trees too: we say that `vehicle` is the parent of `body` and that `body` is a child of `vehicle`. The term *node* is also often used to describe an element in a tree. A node may have several other nodes as *descendants*.

Recursion is the principal programming technique to traverse tree structures. Any object in the tree can be viewed as a root of a subtree. For example, `wheels` is the root of a subtree that branches into `wheel1` and `wheel2`. So when processing an object in the tree, we imagine we process the root and then recurse into a subtree, but the first object we recurse into can be viewed as the root of the subtree, so the processing procedure of the parent object can be repeated.

A recommended next step is to simulate the `recurse` method by hand and carefully check that what happens in the visits to `recurse` is consistent with the output

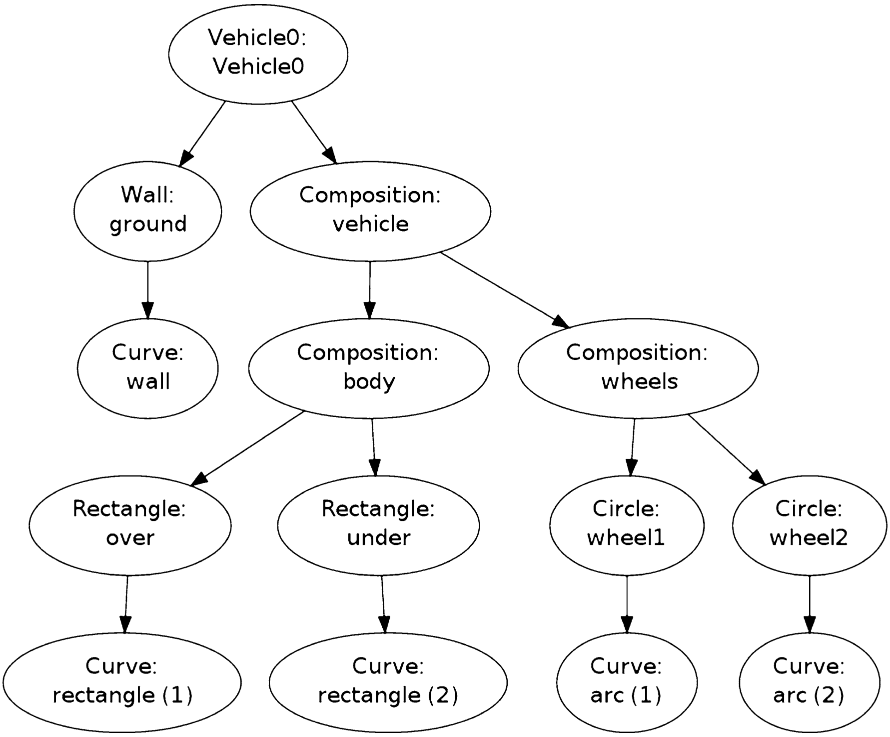


Fig. 9.12 Hierarchy of figure elements in an instance of class `Vehicle0`

listed below. Although tedious, this is a major exercise that guaranteed will help to demystify recursion.

A part of the printout of `v.recurse('vehicle')` looks like

```
 Vehicle0: vehicle.shapes has entries 'ground', 'vehicle'
 call vehicle.shapes["ground"].recurse("ground", 2)
   Wall: ground.shapes has entries 'wall'
   call ground.shapes["wall"].recurse("wall", 4)
     reached "bottom" object Curve
 call vehicle.shapes["vehicle"].recurse("vehicle", 2)
   Composition: vehicle.shapes has entries 'body', 'wheels'
   call vehicle.shapes["body"].recurse("body", 4)
     Composition: body.shapes has entries 'over', 'under'
     call body.shapes["over"].recurse("over", 6)
       Rectangle: over.shapes has entries 'rectangle'
       call over.shapes["rectangle"].recurse("rectangle", 8)
         reached "bottom" object Curve
     call body.shapes["under"].recurse("under", 6)
       Rectangle: under.shapes has entries 'rectangle'
       call under.shapes["rectangle"].recurse("rectangle", 8)
         reached "bottom" object Curve
...
```

This example should clearly demonstrate the principle that we can start at any object in the tree and do a recursive set of calls with that object as root.

9.4.4 Scaling, Translating, and Rotating a Figure

With recursion, as explained in the previous section, we can within minutes equip *all* classes in the `Shape` hierarchy, both present and future ones, with the ability to scale the figure, translate it, or rotate it. This added functionality requires only a few lines of code.

Scaling We start with the simplest of the three geometric transformations, namely scaling. For a `Curve` instance containing a set of n coordinates (x_i, y_i) that make up a curve, scaling by a factor a means that we multiply all the x and y coordinates by a:

$$x_i \leftarrow ax_i, \quad y_i \leftarrow ay_i, \quad i = 0, \ldots, n-1 .$$

Here we apply the arrow as an assignment operator. The corresponding Python implementation in class `Curve` reads

```
class Curve:
    ...
    def scale(self, factor):
        self.x = factor*self.x
        self.y = factor*self.y
```

Note here that `self.x` and `self.y` are Numerical Python arrays, so that multiplication by a scalar number `factor` is a vectorized operation.

An even more efficient implementation is to make use of in-place multiplication in the arrays,

```
class Curve:
    ...
    def scale(self, factor):
        self.x *= factor
        self.y *= factor
```

as this saves the creation of temporary arrays like `factor*self.x`.

In an instance of a subclass of `Shape`, the meaning of a method `scale` is to run through all objects in the dictionary `shapes` and ask each object to scale itself. This is the same delegation of actions to subclass instances as we do in the `draw` (or `recurse`) method. All objects, except `Curve` instances, can share the same implementation of the `scale` method. Therefore, we place the `scale` method in the superclass `Shape` such that all subclasses inherit the method. Since `scale` and `draw` are so similar, we can easily implement the `scale` method in class `Shape` by copying and editing the `draw` method:

```
class Shape:
    ...
    def scale(self, factor):
        for shape in self.shapes:
            self.shapes[shape].scale(factor)
```

This is all we have to do in order to equip all subclasses of `Shape` with scaling functionality! Any piece of the figure will scale itself, in the same manner as it can draw itself.

Translation A set of coordinates (x_i, y_i) can be translated v_0 units in the x direction and v_1 units in the y direction using the formulas

$$x_i \leftarrow x_i + v_0, \quad y_i \leftarrow y_i + v_1, \quad i = 0, \ldots, n-1 .$$

The natural specification of the translation is in terms of the vector $v = (v_0, v_1)$. The corresponding Python implementation in class `Curve` becomes

```
class Curve:
    ...
    def translate(self, v):
        self.x += v[0]
        self.y += v[1]
```

The translation operation for a shape object is very similar to the scaling and drawing operations. This means that we can implement a common method `translate` in the superclass `Shape`. The code is parallel to the `scale` method:

```
class Shape:
    ....
    def translate(self, v):
        for shape in self.shapes:
            self.shapes[shape].translate(v)
```

Rotation Rotating a figure is more complicated than scaling and translating. A counter clockwise rotation of θ degrees for a set of coordinates (x_i, y_i) is given by

$$\begin{aligned}\bar{x}_i &\leftarrow x_i \cos\theta - y_i \sin\theta, \\ \bar{y}_i &\leftarrow x_i \sin\theta + y_i \cos\theta .\end{aligned}$$

This rotation is performed around the origin. If we want the figure to be rotated with respect to a general point (x, y), we need to extend the formulas above:

$$\begin{aligned}\bar{x}_i &\leftarrow x + (x_i - x)\cos\theta - (y_i - y)\sin\theta, \\ \bar{y}_i &\leftarrow y + (x_i - x)\sin\theta + (y_i - y)\cos\theta .\end{aligned}$$

The Python implementation in class `Curve`, assuming that θ is given in degrees and not in radians, becomes

```
    def rotate(self, angle, center):
        angle = radians(angle)
        x, y = center
        c = cos(angle);  s = sin(angle)
        xnew = x + (self.x - x)*c - (self.y - y)*s
        ynew = y + (self.x - x)*s + (self.y - y)*c
        self.x = xnew
        self.y = ynew
```

The `rotate` method in class `Shape` follows the principle of the `draw`, `scale`, and `translate` methods.

We have already seen the `rotate` method in action when animating the rolling wheel at the end of Sect. 9.4.1.

9.5 Classes for DNA Analysis

We shall here exemplify the use of classes for performing DNA analysis as explained in Sects. 3.3.1, 6.5.1, 6.5.2, 6.5.3, 6.5.4, 6.5.5, and 8.3.4. Basically, we create a class `Gene` to represent a DNA sequence (string) and a class `Region` to represent a subsequence (substring), typically an exon or intron.

9.5.1 Class for Regions

The class for representing a region of a DNA string is quite simple:

```
class Region(object):
    def __init__(self, dna, start, end):
        self._region = dna[start:end]

    def get_region(self):
        return self._region

    def __len__(self):
        return len(self._region)

    def __eq__(self, other):
        """Check if two Region instances are equal."""
        return self._region == other._region

    def __add__(self, other):
        """Add Region instances: self + other"""
        return self._region + other._region

    def __iadd__(self, other):
        """Increment Region instance: self += other"""
        self._region += other._region
        return self
```

Besides storing the substring and giving access to it through `get_region`, we have also included the possibility to

- say `len(r)` if `r` is a `Region` instance
- check if two `Region` instances are equal
- write `r1 + r2` for two instances `r1` and `r2` of type `Region`
- perform `r1 += r2`

The latter two operations are convenient for making one large string out of all exon or intron regions.

9.5.2 Class for Genes

The class for gene will be longer and more complex than class `Region`. We already have a bunch of functions performing various types of analysis. The idea of the `Gene` class is that these functions are methods in the class operating on the DNA string and the exon regions stored in the class. Rather than recoding all the functions as methods in the class we shall just let the class "wrap" the functions. That is, the class methods call up the functions we already have. This approach has two advantages: users can either choose the function-based or the class-based interface, and the programmer can reuse all the ready-made functions when implementing the class-based interface.

The selection of functions include

- `generate_string` for generating a random string from some alphabet
- `download` and `read_dnafile` (version `read_dnafile_v1`) for downloading data from the Internet and reading from file
- `read_exon_regions` (version `read_exon_regions_v2`) for reading exon regions from file
- `tofile_with_line_sep` (version `tofile_with_line_sep_v2`) for writing strings to file
- `read_genetic_code` (version `read_genetic_code_v2`) for loading the mapping from triplet codes to 1-letter symbols for amino acids
- `get_base_frequencies` (version `get_base_frequencies_v2`) for finding frequencies of each base
- `format_frequencies` for formatting base frequencies with two decimals
- `create_mRNA` for computing an mRNA string from DNA and exon regions
- `mutate` for mutating a base at a random position
- `create_markov_chain`, `transition`, and `mutate_via_markov_chain` for mutating a base at a random position according to randomly generated transition probabilities
- `create_protein_fixed` for proper creation of a protein sequence (string)

The set of plain functions for DNA analysis is found in the file `dna_functions.py`, while `dna_classes.py` contains the implementations of classes `Gene` and `Region`.

Basic features of class gene Class `Gene` is supposed to hold the DNA sequence and the associated exon regions. A simple constructor expects the exon regions to be specified as a list of (start, end) tuples indicating the start and end of each region:

```
class Gene(object):
    def __init__(self, dna, exon_regions):
        self._dna = dna

        self._exon_regions = exon_regions
        self._exons = []
        for start, end in exon_regions:
            self._exons.append(Region(dna, start, end))
```

```
        # Compute the introns (regions between the exons)
        self._introns = []
        prev_end = 0
        for start, end in exon_regions:
            self._introns.append(Region(dna, prev_end, start))
            prev_end = end
        self._introns.append(Region(dna, end, len(dna)))
```

The methods in class `Gene` are trivial to implement when we already have the functionality in stand-alone functions. Here are a few examples on methods:

```
from dna_functions import *

class Gene(object):
    ...

    def write(self, filename, chars_per_line=70):
        """Write DNA sequence to file with name filename."""
        tofile_with_line_sep(self._dna, filename, chars_per_line)

    def count(self, base):
        """Return no of occurrences of base in DNA."""
        return self._dna.count(base)

    def get_base_frequencies(self):
        """Return dict of base frequencies in DNA."""
        return get_base_frequencies(self._dna)

    def format_base_frequencies(self):
        """Return base frequencies formatted with two decimals."""
        return format_frequencies(self.get_base_frequencies())
```

Flexible constructor The constructor can be made more flexible. First, the exon regions may not be known so we should allow `None` as value and in fact use that as default value. Second, exon regions at the start and/or end of the DNA string will lead to empty intron `Region` objects so a proper test on nonzero length of the introns must be inserted. Third, the data for the DNA string and the exon regions can either be passed as arguments or downloaded and read from file. Two different initializations of `Gene` objects are therefore

```
g1 = Gene(dna, exon_regions)  # user has read data from file
g2 = Gene((urlbase, dna_file), (urlbase, exon_file))  # download
```

One can pass `None` for `urlbase` if the files are already at the computer. The flexible constructor has, not surprisingly, much longer code than the first version. The implementation illustrates well how the concept of overloaded constructors in other languages, like C++ and Java, are dealt with in Python (overloaded constructors take different types of arguments to initialize an instance):

```
class Gene(object):
    def __init__(self, dna, exon_regions):
        """
        dna: string or (urlbase,filename) tuple
        exon_regions: None, list of (start,end) tuples
                      or (urlbase,filename) tuple
        In case of (urlbase,filename) tuple the file
        is downloaded and read.
        """
        if isinstance(dna, (list,tuple)) and \
           len(dna) == 2 and isinstance(dna[0], str) and \
           isinstance(dna[1], str):
            download(urlbase=dna[0], filename=dna[1])
            dna = read_dnafile(dna[1])
        elif isinstance(dna, str):
            pass # ok type (the other possibility)
        else:
            raise TypeError(
                'dna=%s %s is not string or (urlbase,filename) '\
                'tuple' % (dna, type(dna)))

        self._dna = dna

        er = exon_regions
        if er is None:
            self._exons = None
            self._introns = None
        else:
            if isinstance(er, (list,tuple)) and \
                len(er) == 2 and isinstance(er[0], str) and \
                isinstance(er[1], str):
                download(urlbase=er[0], filename=er[1])
                exon_regions = read_exon_regions(er[1])
            elif isinstance(er, (list,tuple)) and \
                isinstance(er[0], (list,tuple)) and \
                isinstance(er[0][0], int) and \
                isinstance(er[0][1], int):
                pass # ok type (the other possibility)
            else:
                raise TypeError(
                    'exon_regions=%s %s is not list of (int,int) '
                    'or (urlbase,filename) tuple' % (er, type(era)))

            self._exon_regions = exon_regions
            self._exons = []
            for start, end in exon_regions:
                self._exons.append(Region(dna, start, end))

            # Compute the introns (regions between the exons)
            self._introns = []
            prev_end = 0
            for start, end in exon_regions:
                if start - prev_end > 0:
                    self._introns.append(
                        Region(dna, prev_end, start))
                prev_end = end
            if len(dna) - end > 0:
                self._introns.append(Region(dna, end, len(dna)))
```

Note that we perform quite detailed testing of the object type of the data structures supplied as the `dna` and `exon_regions` arguments. This can well be done to ensure safe use also when there is only one allowed type per argument.

Other methods A `create_mRNA` method, returning the mRNA as a string, can be coded as

```
    def create_mRNA(self):
        """Return string for mRNA."""
        if self._exons is not None:
            return create_mRNA(self._dna, self._exon_regions)
        else:
            raise ValueError(
                'Cannot create mRNA for gene with no exon regions')
```

Also here we rely on calling an already implemented function, but include some testing whether asking for mRNA is appropriate.

Methods for creating a mutated gene are also included:

```
    def mutate_pos(self, pos, base):
        """Return Gene with a mutation to base at position pos."""
        dna = self._dna[:pos] + base + self._dna[pos+1:]
        return Gene(dna, self._exon_regions)

    def mutate_random(self, n=1):
        """
        Return Gene with n mutations at a random position.
        All mutations are equally probable.
        """
        mutated_dna = self._dna
        for i in range(n):
            mutated_dna = mutate(mutated_dna)
        return Gene(mutated_dna, self._exon_regions)

    def mutate_via_markov_chain(markov_chain):
        """
        Return Gene with a mutation at a random position.
        Mutation into new base based on transition
        probabilities in the markov_chain dict of dicts.
        """
        mutated_dna = mutate_via_markov_chain(
            self._dna, markov_chain)
        return Gene(mutated_dna, self._exon_regions)
```

Some “get” methods that give access to the fundamental attributes of the class can be included:

```
    def get_dna(self):
        return self._dna

    def get_exons(self):
        return self._exons
```

```
    def get_introns(self):
        return self._introns
```

Alternatively, one could access the attributes directly: `gene._dna`, `gene._exons`, etc. In that case we should remove the leading underscore as this underscore signals that these attributes are considered “protected”, i.e., not to be directly accessed by the user. The “protection” in “get” functions is more mental than actual since we anyway give the data structures in the hands of the user and she can do whatever she wants (even delete them).

Special methods for the length of a gene, adding genes, checking if two genes are identical, and printing of compact gene information are relevant to add:

```
    def __len__(self):
        return len(self._dna)

    def __add__(self, other):
        """self + other: append other to self (DNA string)."""
        if self._exons is None and other._exons is None:
            return Gene(self._dna + other._dna, None)
        else:
            raise ValueError(
                ’cannot do Gene + Gene with exon regions’)

    def __iadd__(self, other):
        """self += other: append other to self (DNA string)."""
        if self._exons is None and other._exons is None:
            self._dna += other._dna
            return self
        else:
            raise ValueError(
                ’cannot do Gene += Gene with exon regions’)

    def __eq__(self, other):
        """Check if two Gene instances are equal."""
        return self._dna == other._dna and \
               self._exons == other._exons

    def __str__(self):
        """Pretty print (condensed info)."""
        s = ’Gene: ’ + self._dna[:6] + ’...’ + self._dna[-6:] + \
            ’, length=%d’ % len(self._dna)
        if self._exons is not None:
            s += ’, %d exon regions’ % len(self._exons)
        return s
```

Here is an interactive session demonstrating how we can work with class `Gene` objects:

```
>>> from dna_classes import Gene
>>> g1 = Gene(’ATCCGTAATTGCGCA’, [(2,4), (6,9)])
>>> print g1
Gene: ATCCGT...TGCGCA, length=15, 2 exon regions
>>> g2 = g1.mutate_random(10)
>>> print g2
```

```
Gene: ATCCGT...TGTGCT, length=15, 2 exon regions
>>> g1 == g2
False
>>> g1 += g2  # expect exception
Traceback (most recent call last):
...
ValueError: cannot do Gene += Gene with exon regions
>>> g1b = Gene(g1.get_dna(), None)
>>> g2b = Gene(g2.get_dna(), None)
>>> print g1b
Gene: ATCCGT...TGCGCA, length=15
>>> g3 = g1b + g2b
>>> g3.format_base_frequencies()
'A: 0.17, C: 0.23, T: 0.33, G: 0.27'
```

9.5.3 Subclasses

There are two fundamental types of genes: the most common type that codes for proteins (indirectly via mRNA) and the type that only codes for RNA (without being further processed to proteins). The product of a gene, mRNA or protein, depends on the type of gene we have. It is then natural to create two subclasses for the two types of gene and have a method `get_product` which returns the product of that type of gene.

The `get_product` method can be declared in class `Gene`:

```
    def get_product(self):
        raise NotImplementedError(
            'Subclass %s must implement get_product' % \
            self.__class__.__name__)
```

The exception here will be triggered by an instance (`self`) of any subclass that just inherits `get_product` from class `Gene` without implementing a subclass version of this method.

The two subclasses of `Gene` may take this simple form:

```
class RNACodingGene(Gene):
    def get_product(self):
        return self.create_mRNA()

class ProteinCodingGene(Gene):
    def __init__(self, dna, exon_positions):
        Gene.__init__(self, dna, exon_positions)
        urlbase = 'http://hplgit.github.com/bioinf-py/data/'
        genetic_code_file = 'genetic_code.tsv'
        download(urlbase, genetic_code_file)
        code = read_genetic_code(genetic_code_file)
        self.genetic_code = code

    def get_product(self):
        return create_protein_fixed(self.create_mRNA(),
                                    self.genetic_code)
```

A demonstration of how to load the lactase gene and create the lactase protein is done with

```
def test_lactase_gene():
    urlbase = 'http://hplgit.github.com/bioinf-py/data/'
    lactase_gene_file = 'lactase_gene.txt'
    lactase_exon_file = 'lactase_exon.tsv'
    lactase_gene = ProteinCodingGene(
        (urlbase, lactase_gene_file),
        (urlbase, lactase_exon_file))

    protein = lactase_gene.get_product()
    tofile_with_line_sep(protein, 'output', 'lactase_protein.txt')
```

Now, envision that the Lactase gene would instead have been an RNA-coding gene. The only necessary changes would have been to exchange `ProteinCodingGene` by `RNACodingGene` in the assignment to `lactase_gene`, and one would get out a final RNA product instead of a protein.

9.6 Summary

9.6.1 Chapter Topics

A subclass inherits everything from its superclass, in particular all data attributes and methods. The subclass can add new data attributes, overload methods, and thereby enrich or restrict functionality of the superclass.

Subclass example Consider class `Gravity` from Sect. 7.7.1 for representing the gravity force GMm/r^2 between two masses m and M being a distance r apart. Suppose we want to make a class for the electric force between two charges q_1 and q_2, being a distance r apart in a medium with permittivity ϵ_0 is Gq_1q_2/r^2, where $G^{-1} = 4\pi\epsilon_0$. We use the approximate value $G = 8.99 \cdot 10^9\,\mathrm{Nm}^2/\mathrm{C}^2$ (C is the Coulomb unit used to measure electric charges such as q_1 and q_2). Since the electric force is similar to the gravity force, we can easily implement the electric force as a subclass of `Gravity`. The implementation just needs to redefine the value of G!

```
class CoulombsLaw(Gravity):
    def __init__(self, q1, q2):
        Gravity.__init__(self, q1, q2)
        self.G = 8.99E9
```

We can now call the inherited `force(r)` method to compute the electric force and the `visualize` method to make a plot of the force:

```
c = CoulombsLaw(1E-6, -2E-6)
print 'Electric force:', c.force(0.1)
c.visualize(0.01, 0.2)
```

However, the `plot` method inherited from class `Gravity` has an inappropriate title referring to “Gravity force” and the masses m and M. An easy fix could be to have the plot title as a data attribute set in the constructor. The subclass can then override the contents of this attribute, as it overrides `self.G`. It is quite common to discover that a class needs adjustments if it is to be used as superclass.

Subclassing in general The typical sketch of creating a subclass goes as follows:

```
class SuperClass(object):
    def __init__(self, p, q):
        self.p, self.q = p, q

    def where(self):
        print 'In superclass', self.__class__.__name__

    def compute(self, x):
        self.where()
        return self.p*x + self.q


class SubClass(SuperClass):
    def __init__(self, p, q, a):
        SuperClass.__init__(self, p, q)
        self.a = a

    def where(self):
        print 'In subclass', self.__class__.__name__

    def compute(self, x):
        self.where()
        return SuperClass.compute(self, x) + self.a*x**2
```

This example shows how a subclass extends a superclass with one data attribute (`a`). The subclass' `compute` method calls the corresponding superclass method, as well as the overloaded method `where`. Let us invoke the `compute` method through superclass and subclass instances:

```
>>> super = SuperClass(1, 2)
>>> sub = SubClass(1, 2, 3)
>>> v1 = super.compute(0)
In superclass SuperClass
>>> v2 = sub.compute(0)
In subclass SubClass
In subclass SubClass
```

Observe that in the subclass `sub`, method `compute` calls `self.where`, which translates to the `where` method in `SubClass`. Then the `compute` method in `SuperClass` is invoked, and this method also makes a `self.where` call, which is a call to `SubClass`' `where` method (think of what `self` is here, it is `sub`, so it is natural that we get `where` in the subclass (`sub.where`) and not `where` in the superclass part of `sub`).

In this example, classes `SuperClass` and `SubClass` constitute a class hierarchy. Class `SubClass` inherits the attributes `p` and `q` from its superclass, and overrides the methods `where` and `compute`.

Terminology The important computer science topics in this chapter are

- superclass
- subclass
- inheritance
- class hierarchies
- tree structures
- recursion

9.6.2 Example: Input Data Reader

The summarizing example of this chapter concerns a class hierarchy for simplifying reading input data into programs. Input data may come from several different sources: the command line, a file, or from a dialog with the user, either of `input` form or in a graphical user interface (GUI). Therefore it makes sense to create a class hierarchy where subclasses are specialized to read from different sources and where the common code is placed in a superclass. The resulting tool will make it easy for you to let your programs read from many different input sources by adding just a few lines.

Problem Let us motivate the problem by a case where we want to write a program for dumping n function values of $f(x)$ to a file for $x \in [a, b]$. The core part of the program typically reads

```
import numpy as np
with open(filename, 'w') as outfile:
    for x in np.linspace(a, b, n):
        outfile.write('%12g  %12g\n' % (x, f(x)))
```

Our purpose is to read data into the variables `a`, `b`, `n`, `filename`, and `f`. For the latter we want to specify a formula and use the `StringFunction` tool (Sect. 4.3.3) to make the function `f`:

```
from scitools.StringFunction import StringFunction
f = StringFunction(formula)
```

How can we read `a`, `b`, `n`, `formula`, and `filename` conveniently into the program?

The basic idea is that we place the input data in a dictionary, and create a tool that can update this dictionary from sources like the command line, a file, a GUI, etc. Our dictionary is then

```
p = dict(formula='x+1', a=0, b=1, n=2, filename='tmp.dat')
```

This dictionary specifies the names of the input parameters to the program and the default values of these parameters.

Using the tool is a matter of feeding `p` into the constructor of a subclass in the tools' class hierarchy and extract the parameters into, for example, distinct variables:

```
inp = Subclassname(p)
a, b, filename, formula, n = inp.get_all()
```

Depending on what we write as `Subclassname`, the five variables can be read from the command line, the terminal window, a file, or a GUI. The task now is to implement a class hierarchy to facilitate the described flexible reading of input data.

Solution We first create a very simple superclass `ReadInput`. Its main purpose is to store the parameter dictionary as a data attribute, provide a method `get` to extract single values, and a method `get_all` to extract all parameters into distinct variables:

```
class ReadInput(object):
    def __init__(self, parameters):
        self.p = parameters

    def get(self, parameter_name):
        return self.p[parameter_name]

    def get_all(self):
        return [self.p[name] for name in sorted(self.p)]

    def __str__(self):
        import pprint
        return pprint.pformat(self.p)
```

Note that we in the `get_all` method must sort the keys in `self.p` such that the list of returned variables is well defined. In the calling program we can then list variables in the same order as the alphabetic order of the parameter names, for example:

```
a, b, filename, formula, n = inp.get_all()
```

The `__str__` method applies the `pprint` module to get a pretty print of all the parameter names and their values.

Class `ReadInput` cannot read from any source – subclasses are supposed to do this. The forthcoming text describes various types of subclasses for various types of reading input.

Prompting the user The perhaps simplest way of getting data into a program is to use `raw_input`. We then prompt the user with a text `Give name:` and get an appropriate object back (recall that strings must be enclosed in quotes). The subclass `PromptUser` for doing this then reads

```
class PromptUser(ReadInput):
    def __init__(self, parameters):
        ReadInput.__init__(self, parameters)
        self._prompt_user()

```

```
    def _prompt_user(self):
        for name in self.p:
            self.p[name] = eval(raw_input("Give " + name + ": "))
```

Note the underscore in `_prompt_user`: the underscore signifies that this is a "private" method in the `PromptUser` class, not intended to be called by users of the class.

There is a major difficulty with using `eval` on the input from the user. When the input is intended to be a string object, such as a filename, say `tmp.inp`, the program will perform the operation `eval(tmp.inp)`, which leads to an exception because `tmp.inp` is treated as a variable `inp` in a module `tmp` and not as the string `'tmp.inp'`. To solve this problem, we use the `str2obj` function from the `scitools.misc` module. This function will return the right Python object also in the case where the argument should result in a string object (see Sect. 4.11.1 for some information about `str2obj`). The bottom line is that `str2obj` acts as a safer `eval(raw_input(...))` call. The key assignment in class `PromptUser` is then changed to

```
self.p[name] = str2obj(raw_input("Give " + name + ": "))
```

Reading from file We can also place `name = value` commands in a file and load this information into the dictionary `self.p`. An example of a file can be

```
formula     = sin(x) + cos(x)
filename    = tmp.dat
a           = 0
b           = 1
```

In this example we have omitted `n`, so we rely on its default value.

A problem is how to give the filename. The easy way out of this problem is to read from standard input, and just redirect standard input from a file when we run the program. For example, if the filename is `tmp.inp`, we run the program as follows in a terminal window

Terminal

```
Terminal> python myprog.py < tmp.inp
```

(The redirection of standard input from a file does not work in IPython so we are in this case forced to run the program in a terminal window.)

To interpret the contents of the file, we read line by line, split each line with respect to `=`, use the left-hand side as the parameter name and the right-hand side as the corresponding value. It is important to strip away unnecessary blanks in the name and value. The complete class now reads

```
class ReadInputFile(ReadInput):
    def __init__(self, parameters):
        ReadInput.__init__(self, parameters)
        self._read_file()
```

```
    def _read_file(self, infile=sys.stdin):
        for line in infile:
            if "=" in line:
                name, value = line.split("=")
                self.p[name.strip()] = str2obj(value.strip())
```

A nice feature with reading from standard input is that if we do not redirect standard input to a file, the program will prompt the user in the terminal window, where the user can give commands of the type `name = value` for setting selected input data. A `Ctrl+d` is needed to terminate the interactive session in the terminal window and continue execution of the program.

Reading from the command line For input from the command line we assume that parameters and values are given as option-value pairs, e.g., as in

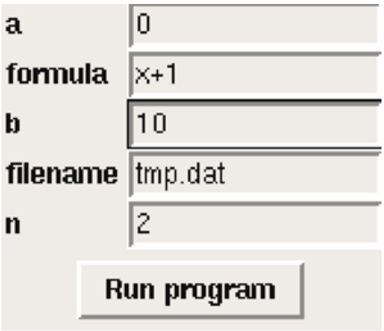
```
--a 1 --b 10 --n 101 --formula "sin(x) + cos(x)"
```

We apply the `argparse` module (see Sect. 4.4) to parse the command-line arguments. The list of legal option names must be constructed from the list of keys in the `self.p` dictionary. The complete class takes the form

```
class ReadCommandLine(ReadInput):
    def __init__(self, parameters):
        self.sys_argv = sys.argv[1:]  # copy
        ReadInput.__init__(self, parameters)
        self._read_command_line()

    def _read_command_line(self):
        parser = argparse.ArgumentParser()
        # Make argparse list of options
        for name in self.p:
            # Default type: str
            parser.add_argument('--'+name, default=self.p[name])

        args = parser.parse_args()
        for name in self.p:
            self.p[name] = str2obj(getattr(args, name))

import Tkinter
try:
```

We could specify the type of a parameter as `type(self.p[name])` or `self.p[name].__class__`, but if a `float` parameter has been given an integer default value, the type will be `int` and `argparse` will not accept a decimal number as input. Our more general strategy is to drop specifying the type, which implies that all parameters in the `args` object become strings. We then use the `str2obj` function to convert to the right type, a technique that is used throughout the `ReadInput` module.

Reading from a gui We can with a little extra effort also make a graphical user interface (GUI) for reading the input data. An example of a user interface is displayed



a: 0
formula: x+1
b: 10
filename: tmp.dat
n: 2

Fig. 9.13 Screen dump of a graphical user interface to read input data into a program (class `GUI` in the `ReadInput` hierarchy)

in Fig. 9.13. Since the technicalities of the implementation is beyond the scope of this book, we do not show the subclass `GUI` that creates the GUI and loads the user input into the `self.p` dictionary.

More flexibility in the superclass Some extra flexibility can easily be added to the `get` method in the superclass. Say we want to extract a variable number of parameters:

```
a, b, n = inp.get(’a’, ’b’, ’n’)  # 3 variables
n = inp.get(’n’)                  # 1 variable
```

The key to this extension is to use a variable number of arguments as explained in Sect. H.7.1:

```
class ReadInput(object):
    ...
    def get(self, *parameter_names):
        if len(parameter_names) == 1:
            return self.p[parameter_names[0]]
        else:
            return [self.p[name] for name in parameter_names]
```

Demonstrating the tool Let us show how we can use the classes in the `ReadInput` hierarchy. We apply the motivating example described earlier. The name of the program is `demo_ReadInput.py`. As first command-line argument it takes the name of the input source, given as the name of a subclass in the `ReadInput` hierarchy. The code for loading input data from any of the sources supported by the `ReadInput` hierarchy goes as follows:

```
p = dict(formula=’x+1’, a=0, b=1, n=2, filename=’tmp.dat’)
from ReadInput import *
input_reader = eval(sys.argv[1])  # PromptUser, ReadInputFile, ...
del sys.argv[1]  # otherwise argparse don’t like our extra option
inp = input_reader(p)
a, b, filename, formula, n = inp.get_all()
print inp
```

Note how convenient `eval` is to automatically create the right subclass for reading input data.

Our first try on running this program applies the `PromptUser` class:

Terminal

```
demo_ReadInput.py PromptUser
Give a: 0
Give formula: sin(x) + cos(x)
Give b: 10
Give filename: function_data
Give n: 101
{'a': 0,
 'b': 10,
 'filename': 'function_data',
 'formula': 'sin(x) + cos(x)',
 'n': 101}
```

The next example reads data from a file `tmp.inp` with the same contents as shown in paragraph above about reading from file.

Terminal

```
Terminal> demo_ReadInput.py ReadFileInput < tmp.inp
{'a': 0, 'b': 1, 'filename': 'tmp.dat',
 'formula': 'sin(x) + cos(x)', 'n': 2}
```

We can also drop the redirection of standard input to a file, and instead run an interactive session in IPython or the terminal window:

Terminal

```
demo_ReadInput.py ReadFileInput
n = 101
filename = myfunction_data_file.dat
^D
{'a': 0,
 'b': 1,
 'filename': 'myfunction_data_file.dat',
 'formula': 'x+1',
 'n': 101}
```

Note that `Ctrl+d` is needed to end the interactive session with the user and continue program execution.

Command-line arguments can also be specified:

Terminal

```
demo_ReadInput.py ReadCommandLine \
          --a -1 --b 1 --formula "sin(x) + cos(x)"
{'a': -1, 'b': 1, 'filename': 'tmp.dat',
 'formula': 'sin(x) + cos(x)', 'n': 2}
```

Finally, we can run the program with a GUI,

Terminal

```
demo_ReadInput.py GUI
{'a': -1, 'b': 10, 'filename': 'tmp.dat',
 'formula': 'x+1', 'n': 2}
```

The GUI is shown in Fig. 9.13.

Fortunately, it is now quite obvious how to apply the `ReadInput` hierarchy of classes in your own programs to simplify input. Especially in applications with a large number of parameters one can initially define these in a dictionary and then automatically create quite comprehensive user interfaces where the user can specify only some subset of the parameters (if the default values for the rest of the parameters are suitable).

9.7 Exercises

Exercise 9.1: Demonstrate the magic of inheritance

Consider class `Line` from Sect. 9.1.1 and a subclass `Parabola0` defined as

```python
class Parabola0(Line):
    pass
```

That is, class `Parabola0` does not have any own code, but it inherits from class `Line`. Demonstrate in a program or interactive session, using `dir` and looking at the `__dict__` object, (see Sect. 7.5.6) that an instance of class `Parabola0` contains everything (i.e., all attributes) that an instance of class `Line` contains.
Filename: `dir_subclass`.

Exercise 9.2: Make polynomial subclasses of parabolas

The task in this exercise is to make a class `Cubic` for cubic functions

$$c_3x^3 + c_2x^2 + c_1x + c_0$$

with a call operator and a `table` method as in classes `Line` and `Parabola` from Sect. 9.1. Implement class `Cubic` by inheriting from class `Parabola`, and call up functionality in class `Parabola` in the same way as class `Parabola` calls up functionality in class `Line`.

Make a similar class `Poly4` for 4-th degree polynomials

$$c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

by inheriting from class `Cubic`. Insert `print` statements in all the `__call__` methods such that you can easily watch the program flow and see when `__call__` in the different classes is called.

Evaluate cubic and a 4-th degree polynomial at a point, and observe the printouts from all the superclasses.
Filename: `Cubic_Poly4`.

Remarks This exercise follows the idea from Sect. 9.1 where more complex polynomials are subclasses of simpler ones. Conceptually, a cubic polynomial *is not* a parabola, so many programmers will not accept class `Cubic` as a subclass of `Parabola`; it should be the other way around, and Exercise 9.2 follows that approach. Nevertheless, one can use inheritance solely for sharing code and not for expressing that a subclass `is a` kind of the superclass. For code sharing it is natural to start with the simplest polynomial as superclass and add terms to the inherited data structure as we make subclasses for higher degree polynomials.

Exercise 9.3: Implement a class for a function as a subclass

Implement a class for the function $f(x) = A\sin(wx) + ax^2 + bx + c$. The class should have a call operator for evaluating the function for some argument x, and a constructor that takes the function parameters `A`, `w`, `a`, `b`, and `c` as arguments. Also a `table` method as in classes `Line` and `Parabola` should be present. Implement the class by deriving it from class `Parabola` and call up functionality already implemented in class `Parabola` whenever possible.
Filename: `sin_plus_quadratic`.

Exercise 9.4: Create an alternative class hierarchy for polynomials

Let class `Polynomial` from Sect. 7.3.7 be a superclass and implement class `Parabola` as a subclass. The constructor in class `Parabola` should take the three coefficients in the parabola as separate arguments. Try to reuse as much code as possible from the superclass in the subclass. Implement class `Line` as a subclass specialization of class `Parabola`.

Which class design do you prefer, class `Line` as a subclass of `Parabola` and `Polynomial`, or `Line` as a superclass with extensions in subclasses? (See also remark in Exercise 9.2.)
Filename: `Polynomial_hier`.

Exercise 9.5: Make circle a subclass of an ellipse

Section 7.2.3 presents class `Circle`. Make a similar class `Ellipse` for representing an ellipse. Then create a new class `Circle` that is a subclass of `Ellipse`.
Filename: `Ellipse_Circle`.

Exercise 9.6: Make super- and subclass for a point

A point (x, y) in the plane can be represented by a class:

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

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

We can extend the `Point` class to also contain the representation of the point in polar coordinates. To this end, create a subclass `PolarPoint` whose constructor takes the polar representation of a point, (r, θ), as arguments. Store r and θ as data attributes and call the superclass constructor with the corresponding x and y

values (recall the relations $x = r\cos\theta$ and $y = r\sin\theta$ between Cartesian and polar coordinates). Add a `__str__` method in class `PolarPoint` which prints out r, θ, x, and y. Write a test function that creates two `PolarPoint` instances and compares the four data attributes `x`, `y`, `r`, and `theta` with the expected values.
Filename: `PolarPoint`.

Exercise 9.7: Modify a function class by subclassing

Consider a class `F` implementing the function $f(t;a,b) = e^{-at}\sin(bt)$:

```
class F(object):
    def __init__(self, a, b):
        self.a, self.b = a, b
    def __call__(self, t):
        return exp(-self.a*t)*sin(self.b*t)
```

We now want to study how the function $f(t;a,b)$ varies with the parameter b, given t and a. Mathematically, this means that we want to compute $g(b;t,a) = f(t;a,b)$. Write a subclass `Fb` of `F` with a new `__call__` method for evaluating $g(b;t,a)$. Do not reimplement the formula, but call the `__call__` method in the superclass to evaluate $f(t;a,b)$. The `Fs` should work as follows:

```
f = Fs(t=2, a=4.5)
print f(3)   # b=3
```

Hint Before calling `__call__` in the superclass, the data attribute `b` in the superclass must be set to the right value.
Filename: `Fb`.

Exercise 9.8: Explore the accuracy of difference formulas

The purpose of this exercise is to investigate the accuracy of the `Backward1`, `Forward1`, `Forward3`, `Central2`, `Central4`, `Central6` methods for the function

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}}\,.$$

Compute the errors in the approximations for $x = 0, 0.9$ and $\mu = 1, 0.01$. Illustrate in a plot how the $v(x)$ function looks like for these two μ values.

Hint Modify the `src/oo/Diff2_examples.py` program which produces tables of errors of difference approximations as discussed at the end of Sect. 9.2.4.
Filename: `boundary_layer_derivative`.

Exercise 9.9: Implement a subclass

Make a subclass `Sine1` of class `FuncWithDerivatives` from Sect. 9.1.6 for the $\sin x$ function. Implement the function only, and rely on the inherited `df` and `ddf` methods for computing the derivatives. Make another subclass `Sine2` for $\sin x$ where you also implement the `df` and `ddf` methods using analytical expressions for the derivatives. Compare `Sine1` and `Sine2` for computing the first- and second-order derivatives of $\sin x$ at two x points.
Filename: `Sine12`.

Exercise 9.10: Make classes for numerical differentiation

Carry out Exercise 7.16. Find the common code in the classes `Derivative`, `Backward`, and `Central`. Move this code to a superclass, and let the three mentioned classes be subclasses of this superclass. Compare the resulting code with the hierarchy shown in Sect. 9.2.1.
Filename: `numdiff_classes`.

Exercise 9.11: Implement a new subclass for differentiation

A one-sided, three-point, second-order accurate formula for differentiating a function $f(x)$ has the form

$$f'(x) \approx \frac{f(x-2h) - 4f(x-h) + 3f(x)}{2h}\,. \tag{9.17}$$

Implement this formula in a subclass `Backward2` of class `Diff` from Sect. 9.2. Compare `Backward2` with `Backward1` for $g(t) = e^{-t}$ for $t = 0$ and $h = 2^{-k}$ for $k = 0, 1, \ldots, 14$ (write out the errors in $g'(t)$).
Filename: `Backward2`.

Exercise 9.12: Understand if a class can be used recursively

Suppose you want to compute $f''(x)$ of some mathematical function $f(x)$, and that you apply some class from Sect. 9.2 twice, e.g.,

```
ddf = Central2(Central2(f))
```

Will this work?

Hint Follow the program flow, and find out what the resulting formula will be. Then see if this formula coincides with a formula you know for approximating $f''(x)$ (actually, to recover the well-known formula with an h parameter, you would use $h/2$ in the nested calls to `Central2`).

Exercise 9.13: Represent people by a class hierarchy

Classes are often used to model objects in the real world. We may represent the data about a person in a program by a class `Person`, containing the person's name, address, phone number, date of birth, and nationality. A method `__str__` may print the person's data. Implement such a class `Person`.

A worker is a person with a job. In a program, a worker is naturally represented as class `Worker` derived from class `Person`, because a worker *is* a person, i.e., we have an is-a relationship. Class `Worker` extends class `Person` with additional data, say name of company, company address, and job phone number. The print functionality must be modified accordingly. Implement this `Worker` class.

A scientist is a special kind of a worker. Class `Scientist` may therefore be derived from class `Worker`. Add data about the scientific discipline (physics, chemistry, mathematics, computer science, …). One may also add the type of scientist: theoretical, experimental, or computational. The value of such a type attribute should not be restricted to just one category, since a scientist may be classified as, e.g., both experimental and computational (i.e., you can represent the value as a list or tuple). Implement class `Scientist`.

Researcher, postdoc, and professor are special cases of a scientist. One can either create classes for these job positions, or one may add an attribute (`position`) for this information in class `Scientist`. We adopt the former strategy. When, e.g., a researcher is represented by a class `Researcher`, no extra data or methods are needed. In Python we can create such an empty class by writing `pass` (the empty statement) as the class body:

```
class Researcher(Scientist):
    pass
```

Finally, make a demo program where you create and print instances of classes `Person`, `Worker`, `Scientist`, `Researcher`, `Postdoc`, and `Professor`. Print out the attribute contents of each instance (use the `dir` function).

Remark An alternative design is to introduce a class `Teacher` as a special case of `Worker` and let `Professor` be both a `Teacher` and `Scientist`, which is natural. This implies that class `Professor` has two superclasses, `Teacher` and `Scientist`, or equivalently, class `Professor` inherits from two superclasses. This is known as *multiple inheritance* and technically achieved as follows in Python:

```
class Professor(Teacher, Scientist):
    pass
```

It is a continuous debate in computer science whether multiple inheritance is a good idea or not. One obvious problem in the present example is that class `Professor` inherits two names, one via `Teacher` and one via `Scientist` (both these classes inherit from `Person`).
Filename: `Person`.

Exercise 9.14: Add a new class in a class hierarchy

a) Add the Monte Carlo integration method from Sect. 8.5.2 as a subclass `MCint` in the `Integrator` hierarchy explained in Sect. 9.3. Import the superclass `Integrator` from the `integrate` module in the file with the new integration class.
b) Make a test function for class `MCint` where you fix the seed of the random number generator, use three function evaluations only, and compare the result of this Monte Carlo integration with results calculated by hand using the same three random numbers.
c) Run the Monte Carlo integration class in a case with known analytical solution and see how the error in the integral changes with $n = 10^k$ function evaluations, $k = 3, 4, 5, 6$.

Filename: `MCint_class`.

Exercise 9.15: Compute convergence rates of numerical integration methods

Numerical integration methods can compute "any" integral $\int_a^b f(x)dx$, but the result is not exact. The methods have a parameter n, closely related to the number of evaluations of the function f, that can be increased to achieve more accurate results. In this exercise we want to explore the relation between the error E in the numerical approximation to the integral and n. Different numerical methods have different relations.

The relations are of the form

$$E = Cn^r,$$

where and C and $r < 0$ are constants to be determined. That is, r is the most important of these parameters, because if Simpson's method has a more negative r than the Trapezoidal method, it means that increasing n in Simpson's method reduces the error more effectively than increasing n in the Trapezoidal method.

One can estimate r from numerical experiments. For a chosen $f(x)$, where the exact value of $\int_a^b f(x)dx$ is available, one computes the numerical approximation for $N + 1$ values of n: $n_0 < n_1 < \cdots < n_N$ and finds the corresponding errors $E_0, E_1, \ldots, E_N$ (the difference between the exact value and the value produced by the numerical method).

One way to estimate r goes as follows. For two successive experiments we have

$$E_i = Cn_i^r\,.$$

and

$$E_{i+1} = Cn_{i+1}^r$$

Divide the first equation by the second to eliminate C, and then take the logarithm to solve for r:

$$r = \frac{\ln(E_i/E_{i+1})}{\ln(n_i/n_{i+1})}\,.$$

We can compute r for all pairs of two successive experiments. Say r_i is the r value found from experiment i and $i + 1$,

$$r_i = \frac{\ln(E_i/E_{i+1})}{\ln(n_i/n_{i+1})}, \quad i = 0, 1, \ldots, N-1\,.$$

Usually, the last value, r_{N-1}, is the best approximation to the true r value. Knowing r, we can compute C as $E_i n_i^{-r}$ for any i.

Use the method above to estimate r and C for the Midpoint method, the Trapezoidal method, and Simpson's method. Make your own choice of integral problem: $f(x)$, a, and b. Let the parameter n be the number of function evaluations in each method, and run the experiments with $n = 2^k + 1$ for $k = 2, \ldots, 11$. The `Integrator` hierarchy from Sect. 9.3 has all the requested methods implemented. Filename: `integrators_convergence`.

Exercise 9.16: Add common functionality in a class hierarchy

Suppose you want to use classes in the `Integrator` hierarchy from Sect. 9.3. to calculate integrals of the form

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

Such functions $F(x)$ can be efficiently computed by the method from Exercise 7.22. Implement this computation of $F(x)$ in an additional method in the superclass `Integrator`. Test that the implementation is correct for $f(x) = 2x - 3$ for all the implemented integration methods (the Midpoint, Trapezoidal and Gauss-Legendre methods, as well as Simpson's rule, integrate a linear function exactly).
Filename: `integrate_efficient`.

Exercise 9.17: Make a class hierarchy for root finding

Given a general nonlinear equation $f(x) = 0$, we want to implement classes for solving such an equation, and organize the classes in a class hierarchy. Make classes for three methods: Newton's method (in Sect. A.1.10), the Bisection method (in Sect. 4.11.2), and the Secant method (in Exercise A.10).

It is not obvious how such a hierarchy should be organized. One idea is to let the superclass store the $f(x)$ function and its derivative $f'(x)$ (if provided – if not, use a finite difference approximation for $f'(x)$). A method

```
def solve(start_values=[0], max_iter=100, tolerance=1E-6):
    ...
```

in the superclass can implement a general iteration loop. The `start_values` argument is a list of starting values for the algorithm in question: one point for Newton, two for Secant, and an interval $[a, b]$ containing a root for Bisection. Let `solve` define a list `self.x` holding all the computed approximations. The initial value of `self.x` is simply `start_values`. For the Bisection method, one can use the convention a, b, c = `self.x[-3:]`, where $[a, b]$ represents the most recently computed interval and c is its midpoint. The `solve` method can return an approximate root x, the corresponding $f(x)$ value, a boolean indicator that is `True` if $|f(x)|$ is less than the `tolerance` parameter, and a list of all the approximations and their f values (i.e., a list of $(x, f(x))$ tuples).

Do Exercise A.11 using the new class hierarchy.
Filename: `Rootfinders`.

Exercise 9.18: Make a calculus calculator class

Given a function $f(x)$ defined on a domain $[a, b]$, the purpose of many mathematical exercises is to sketch the function curve $y = f(x)$, compute the derivative $f'(x)$, find local and global extreme points, and compute the integral $\int_a^b f(x)dx$. Make a class `CalculusCalculator` which can perform all these actions for any function $f(x)$ using numerical differentiation and integration, and the method explained in Exercise 7.34. for finding extrema.

Here is an interactive session with the class where we analyze $f(x) = x^2e^{-0.2x}\sin(2\pi x)$ on $[0, 6]$ with a grid (set of x coordinates) of 700 points:

```
>>> from CalculusCalculator import *
>>> def f(x):
...     return x**2*exp(-0.2*x)*sin(2*pi*x)
...
>>> c = CalculusCalculator(f, 0, 6, resolution=700)
>>> c.plot()                # plot f
>>> c.plot_derivative()     # plot f’
>>> c.extreme_points()

All minima: 0.8052, 1.7736, 2.7636, 3.7584, 4.7556, 5.754, 0
All maxima: 0.3624, 1.284, 2.2668, 3.2604, 4.2564, 5.2548, 6
Global minimum: 5.754
Global maximum: 5.2548

>>> c.integral
-1.7353776102348935
>>> c.df(2.51)    # c.df(x) is the derivative of f
-24.056988888465636
>>> c.set_differentiation_method(Central4)
>>> c.df(2.51)
-24.056988832723189
>>> c.set_integration_method(Simpson)  # more accurate integration
>>> c.integral
-1.7353857856973565
```

Design the class such that the above session can be carried out.

Hint Use classes from the `Diff` and `Integrator` hierarchies (Sects. 9.2 and 9.3) for numerical differentiation and integration (with, e.g., `Central2` and `Trapezoidal` as default methods for differentiation and integration). The method `set_differentiation_method` takes a subclass name in the `Diff` hierarchy as argument, and makes a data attribute `df` that holds a subclass instance for computing derivatives. With `set_integration_method` we can similarly set the integration method as a subclass name in the `Integrator` hierarchy, and then compute the integral $\int_a^b f(x)dx$ and store the value in the attribute `integral`. The `extreme_points` method performs a `print` on a `MinMax` instance, which is stored as an attribute in the calculator class.
Filename: `CalculusCalculator`.

Exercise 9.19: Compute inverse functions

Extend class `CalculusCalculator` from Exercise 9.18 to offer computations of inverse functions.

Hint A numerical way of computing inverse functions is explained in Sect. A.1.11. Other, perhaps more attractive methods are described in Exercises E.17–E.20.
Filename: `CalculusCalculator2`.

Exercise 9.20: Make line drawing of a person; program
A very simple sketch of a human being can be made of a circle for the head, two lines for the arms, one vertical line, a triangle, or a rectangle for the torso, and two lines for the legs. Make such a drawing in a program, utilizing appropriate classes in the `Shape` hierarchy.
Filename: `draw_person`.

Exercise 9.21: Make line drawing of a person; class
Use the code from Exercise 9.20 to make a subclass of `Shape` that draws a person. Supply the following arguments to the constructor: the center point of the head and the radius R of the head. Let the arms and the torso be of length $4R$, and the legs of length $6R$. The angle between the legs can be fixed (say 30 degrees), while the angle of the arms relative to the torso can be an argument to the constructor with a suitable default value.
Filename: `Person`.

Exercise 9.22: Animate a person with waving hands
Make a subclass of the class from Exercise 9.21 where the constructor can take an argument describing the angle between the arms and the torso. Use this new class to animate a person who waves her/his hands.
Filename: `waving_person`.