Ch.9: Object-oriented programming

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Oct 19, 2019

When reading the chapter title *Object-oriented programming*, one may wonder why this title is introduced now. We have programmed with objects since Chapter 1, and we started making our own classes and object types in Chapter 7, so what is new in Chapter 9? The answer to this is that the term *object-oriented programming (OOP)* may have two different meanings. The first is simply programming with classes and objects and classes, which we introduced in Chapter 7. This is more commonly referred to as object-*based* programming. The second meaning of OOP is programming with *class hierarchies*, which are families of classes that inherit methods and attributes from eachother. This is the topic of the present chapter. We will learn how to collect classes in families (hierarchies) and let child classes inherit attributes and methods from parent classes.

1 Class hierarchies and inheritance

A class hierarchy is a family of closely related classes, organized in a hierarchical manner. A key concept is *inheritance*, which means that child classes can inherit attributes and methods from parent classes. A typical strategy is to write a general class as a parent class (base class), and then let special cases be represented as child classes (subclasses). This approach can often save much typing and code duplication. As usual, the easiest way to introduce the topic is to look at some examples.

OO is a Norwegian invention by Ole-Johan Dahl and Kristen Nygaard in the 1960s - one of the most important inventions in computer science, because OO is used in all big computer systems today!

1.1 Warning: OO is difficult and takes time to master

- Let ideas mature with time
- Study many examples
- OO is less important in Python than in C++, Java and C#, so the benefits of OO are less obvious in Python
- Our examples here on OO employ numerical methods for $\int_a^b f(x)dx$, $f'(x)$, $u' = f(u,t)$ - make sure you understand the simplest of these numerical methods before you study the combination of OO and numerics
- Our goal: write general, reusable modules with lots of methods for numerical computing of $\int_a^b f(x)dx$, $f'(x)$, $u' = f(u,t)$

Class for lines and parabolas. As a first example, let us create a class for representing and evaluating straight lines; $y = c_0 + c_1 x$. Following the concepts and ideas introduced in the previous chapter, the implementation of the class may look as follows

```
class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, 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 = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

We see that we have equipped the class with a standard constructor, a `__call__` special method for evaluating the linear function, and a method `table` for writing a table of x and y values. Say we now want to write a similar class for evaluating a parabola $y = c_0 + c_1 x + c_2 x^2$. The code could look like:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, 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 = ''
        for x in linspace(L, R, n):
            y = self(x)
```

```
            s += f'{x:12g} {y:12g}\n'
        return s
```

We observe that the two classes are nearly identical, only differing in the parts that involve `c2`. Although we could very quickly just copy all the code from the `Line` class and edit the small parts that are needed, such duplication of code is usually a bad idea. At some point we may need to make some changes to the code, for instance to correct an error or improve the functionality, and having to correct the same error in multiple places is a recipe for wasting time. So, is there a way we can utilize the class `Line` code in `Parabola` without resorting to copy-paste? This is exactly what inheritance is about.

Look at the following class defnition:

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

Here `pass` is just a Python keyword that can be used wherever Python expects to find some code, but we don't really want to define anything. So at first site this `Parabola` class seems to be empty. Notice however the class definition `class Parabola(Line)`, which means that parabola *inherits* all methods and attributes from `Line`. The new `Parabola` class therefore has attributes `c0` and `c1` and three methods `__init__`, `__call__`, and `table`. `Line` is a *base class* (or parent class, superclass) , `Parabola` is a *subclass* (or child class, derived class). So the new `Parabola` class is not as useless as it first seemed, but it is still just a copy of the `Line` class. To make it represent a parabola, we need to add the missing code, i.e. the code that differs between `Line` and `Parabola`. The principle when creating such subclasses is to reuse as much as possible from the base class, and add only what is needed in the subclass. In this way we avoid duplicating code. Inspecting the two original classes above, we see that the `Parabola` class must add code to `Line`'s constructor (an extra `c2` attribute), and an extra term in `__call__`, but `table` can be used unaltered. The full definition of `Parabola` as a subclass of `Line` becomes:

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        super().__init__(c0, c1)  # Line stores c0, c1
        self.c2 = c2

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

To maximize code reuse, we let the `Parabola` class call the methods from `Line`, and then add the parts that are missing. A subclass can always access its base class using the builtin function `super()`, and this is the preferred way to invole methods from the base class. We could, however, also use the class name directly, for instance `Line.__init__(self,c0,c1)`. In general, these two methods for invoking superclass methods look like:

```
SuperClassName.method(self, arg1, arg2, ...)
super().method(arg1, arg2, ...)
```

Notice the difference between the two approaches. When using the class name directly we need to include `self` as argument, while this is handled automatically when using `super()`. The use of `super()` is usually preferred, but in most cases the two approaches are equivalent.

What exactly have we gained by creating a subclass?

- Class `Parabola` just adds code to the already existing code in class `Line` - no duplication of storing `c0` and `c1`, and computing $c_0 + c_1 x$
- Class `Parabola` also has a `table` method - it is inherited and did not need to be written
- `__init__` and `__call__` are *overridden* or *redefined* in the subclass, with no code duplication

We use the `Parabola` class and call its methods just as if they were implemented in the class directly:

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

What does inheritance really mean? From a practical viewpoint, and for the examples in this book, the point of inheritance is to reuse methods and attributes from the base class, and minimize code duplication. On a more theoretical level, inheritance should be thought of as an "is-a"-relationship between the the two classes. By this we mean that if `Parabola` is a subclass of `Line`, an instance of `Parabola` is also a `Line` instance. The `Parabola` class is thought of as a special case of the `Line` class, and therefore every `Parabola` is also a `Line`, but not vice versa. We can check class type and class relations with the builtin functions `isinstance(obj, type)` and `issubclass(subclassname, superclassname)`:

```
>>> from Line_Parabola import Line, Parabola
>>> l = Line(-1, 1)
>>> isinstance(l, Line)
True
>>> isinstance(l, Parabola)
False
>>> p = Parabola(-1, 0, 10)
>>> isinstance(p, Parabola)
True
>>> isinstance(p, Line)
True
>>> issubclass(Parabola, Line)
True
>>> issubclass(Line, Parabola)
False
>>> p.__class__ == Parabola
True
>>> p.__class__.__name__   # string version of the class name
'Parabola'
```

We will not use these methods much in practical applications[1], but they are very useful for getting a feel for class relationships when learning OOP.

Mathematically oriented readers may have noticed a logical fault in the small class hierarchy we have presented so far. We stated that a subclass is usually thought of as a special case of the base class, but a parabola is not really a special case of a straight line. In fact it is the other way around, a line $c_0 + c_1 x$ is a parabola $c_0 + c_1 x + c_2 x^2$ with $c_2 = 0$. Could then `Line` be a subclass of `Parabola`? Certainly, and many will prefer this relation between a line and a parabola, since it follows the usual is-a relationship between a subclass and its base. The code may look like:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, 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 = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s

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

Notice that this version allows even more code reuse than the previous one, since both `__call__` and `table` can be reused without changes.

1.2 Class hierarchies for numerical methods

Common tasks in scientific computing, such as differentiation and integration, can be solved with a large variety of numerical methods. Many such methods are closely related, and can easily be grouped into families of methods that are very suitable for implementing in a class hierarchy.

Classes for numerical differentation. As a first example we consider methods for numerical differentiation. The simplest formula is a one-sided finite difference;

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

which can be implemented in the following class:

[1] In fact, if you have to use `isinstance` in your code to check what kind of object you are working with it is usually a sign that the design of the program is wrong. There are exceptions, but normally `isinstance` and `issubclass` should only be used for learning and debugging.

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

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

To use the `Derivative` class, we simply define the function `f(x)`, create an instance of the class, and call it as if it was a regular function (effectively calling the `__call__` method behind the scenes):

```
def f(x):
    return exp(-x)*cos(tanh(x))

from math import exp, cos, tanh
dfdx = Derivative(f)
print dfdx(2.0)
```

But there are numerous formulas for numerical differentiation;

$$
\begin{aligned}
f'(x) &= \frac{f(x+h)-f(x)}{h} + \mathcal{O}(h), \\
f'(x) &= \frac{f(x)-f(x-h)}{h} + \mathcal{O}(h), \\
f'(x) &= \frac{f(x+h)-f(x-h)}{2h} + \mathcal{O}(h^2), \\
f'(x) &= \frac{4}{3}\frac{f(x+h)-f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h)-f(x-2h)}{4h} + \mathcal{O}(h^4), \\
f'(x) &= \frac{3}{2}\frac{f(x+h)-f(x-h)}{2h} - \frac{3}{5}\frac{f(x+2h)-f(x-2h)}{4h} + \\
&\quad \frac{1}{10}\frac{f(x+3h)-f(x-3h)}{6h} + \mathcal{O}(h^6), \\
f'(x) &= \frac{1}{h}\left(-\frac{1}{6}f(x+2h) + f(x+h) - \frac{1}{2}f(x) - \frac{1}{3}f(x-h)\right) + \mathcal{O}(h^3),
\end{aligned}
$$

and we can easily make a module that offers all of them:

```
class Forward1:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)

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

class Backward1:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
```

```
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h

class Central2:
    # same constructor
    # put relevant formula in __call__
```

The problem with this code is of course that all the constructors are identical, so we duplicate a lot of code. As mentioned above, a general idea of OOP is to place code common to many classes in a superclass and inherit that code. In this case, we could make a superclass containing the constructor, and let the different subclasses implement their own version of the `__call__` method. The superclass will be very simple, and not really useful on its own:

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

Subclass for simple 1st-order forward formula:

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

Subclasses for 2nd order and 4-th order central formulae:

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

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)
```

Interactive example: $f(x) = \sin x$, compute $f'(x)$ for $x = \pi$

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

Here `Central4(sin)` calls inherited constructor in superclass, while `mycos(pi)` calls `__call__` in the subclass `Central4`

As indicated by the $O(h^n)$ terms in the formulas above, the different methods have different accuracy. We can empirically investigate the accuracy of our family of 6 numerical differentiation formulas, using the class hierarchy created above. The code may look like

- Sample function: $f(x) = \exp(-10x)$

- See the book for a little program that computes the errors:

```
.   h         Forward1          Central2          Central4
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
```

Observations:

- Halving h from row to row reduces the errors by a factor of 2, 4 and 16, i.e, the errors go like h, h^2, and h^4
- `Central4` has really superior accuracy compared with `Forward1`

Class hierarchy for numerical integration. There are numerous formulas for numerical integration, and all of them can be put into a common notation:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i)$$

Based on this general formula, different methods are realized by choosing the integration points x_i and associated weights w_i. For instance, the Trapezoidal rule has $h = (b-a)/(n-1)$ and

$$x_i = a + ih, \quad w_0 = w_{n-1} = \frac{h}{2}, \ w_i = h \ (i \neq 0, n-1),$$

the midpoint rule has $h = (b-a)/n$ and

$$x_i = a + \frac{h}{2} + ih, \quad w_i = h,$$

while Simpson's rule has

$$\begin{aligned} x_i &= a + ih, \quad h = \frac{b-a}{n-1}, \\ w_0 &= w_{n-1} = \frac{h}{6}, \\ w_i &= \frac{h}{3} \text{ for } i \text{ even}, \quad w_i = \frac{2h}{3} \text{ for } i \text{ odd}. \end{aligned}$$

Other methods have more complicated formulas for w_i and x_i

A numerical integration formula can be implemented as a class, with a, b and n as attributes and an `integrate` method to evaluate the formula and compute the integral. As with the family of numerical differentiation methods considered above, all such classes will be quite similar. The evaluation of $\sum_j w_j f(x_j)$ is the

same, and the only difference between the methods is the definition of the points and weights. Following the ideas above, it makes sense to place all common code in a subperclass, and code specific to the different methods in subclasses. Here, we can put $\sum_j w_j f(x_j)$ in a superclass (method `integrate`), and let the subclasses extend this class with code specific to a specific formula, i.e. w_i and x_i. This method-specific code can be placed inside a method, for instance named `construct_rule`. The superclass for the numerical integration hierarchy may look like this:

```
class Integrator:
    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

    def vectorized_integrate(self, f):
        # f must be vectorized for this to work
        return dot(self.weights, f(self.points))
```

Notice the implementation of `construct_method`, which will raise an error if it is called, indicating that the only purpose of the `Integrator` is as a superclass, it should not be used directly. Alternatively, we could of course just not include `construct_method` method in the superclass at all. However, the approach used here makes it even more obvious that the class is just a superclass and that this method has to be implemented in subclasses.

The superclass provides a common framework for implementing the different methods, which can be then be realized as subclasses. The trapezoidal and midpoint methods may be implemented like this:

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

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
```

And the more complex Simpson's rule can be added in the following subclass:

```
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
```

Simpson's rule is more complex because it uses different wiights for odd and even points. We present all the details here for completeness, but it is not necessary to study the detailed implementation of all the formualas. The important parts here are the class design and usage of the class hierarchy.

To demonstrate how the classs can be used, let us compute the integral $\int_0^2 x^2 dx$ using 101 points:

```
def f(x):
    return x*x

simpson = Simpson(0, 2, 101)
print(simpson.integrate(f))
trapez = Trapezoidal(0,2,101)
print(trapez.integrate(f))
```

The program flow in this case may not be entirely obvious. When we construct the instance with `method = Simpson(...)`, it invokes the superclass constructor, but then this method calls `construct_method` in class `Simpson`. The call `method.integrate(f)` invokes the `integrate` method inherited from the class `Integrator`. However, to us as users of the class none of these details really matter. We use the `Simpson` class just as if all the methods were implemented directly in the class, regardless of whether they were actually inherited from another class.

1.3 Summary of object-orientation principles

- A subclass inherits everything from the superclass
- When to use a subclass/superclass?
  - if code common to several classes can be placed in a superclass
  - if the problem has a natural child-parent concept
- The program flow jumps between super- and sub-classes, but users of the classes will not notice this