Ch.9: Object-oriented programming

Joakim Sundnes[1,2]

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

Oct 23, 2023

0.1 Plan for Week 43

Tuesday Oct 24:

- Exercise 7.3, 7.10, 7.11, 7.12
- Introduction to object-oriented programming (OOP)

Thursday Oct 26:

- Exercise 7.25
- Exercises 9.1, 9.3 (Line/Parabola class)
- More about OOP:
  - Classes for numerical differentiation
  - (Classes for numerical integration)

0.2 The title *Object-oriented programming* (OOP) may mean two different things

1. Programming with classes and objects (better: object-*based* programming)
2. Programming with class hierarchies (class families)

0.3 New concept: collect classes in families (hierarchies)

What is a class hierarchy?

- A family of closely related classes
- A key concept is *inheritance*: child classes can inherit attributes and methods from parent class(es) - this saves much typing and code duplication

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.

0.4 Object-oriented programming

- Everything in Python is an object, so all Python-programming is *object-based*
- Object-oriented programming (OOP) takes the ideas of classes and programming a step further
- We exploit a very useful property of classes; that they can be combined and reused as building blocks.
- If we define a class `class A`, we can define a second class `class B(A)`.
- Class `B` *inherits* all attributes and methods from `A`
- Class becomes an *extension* of class `A`
- We say that `A` is a superclass or base class, and `B` is a subclass of `A`

0.5 OOP fundamentals - inheritance

```
class A:
    def __init__(self,v0,v1)
        self.v0 = v0
        self.v1 = v1

    def f(self, x):
            return x**2

class B(A):
    def g(self, x):
        return x**4

class C(B):
    def h(self, x):
        return x**6
```

We have now defined three classes

- A: two attributes (`v0, v1`) and two methods (`__init__`, `f`)
- B: two attributes (`v0, v1`) and three methods (`__init__`, `f`, `g`)
- C: two attributes (`v0, v1`) and four methods (`__init__`, `f`, `g`, `h`)

0.6 OOP is even more important in other languages

Languages like Java and C++ have static typing. A function is declared to take input arguments of a certain type:

```
void my_func(A my_obj)
{
    ...
}
```

OOP gives extra flexibility, since this function will also accept arguments of classes `B` and `C`.

OOP is still very useful in Python, to avoid code duplication and produce structured and readable code!

0.7 OOP fundamentals - overriding methods

A subclass can override methods in the superclass. Say we want to add some extra attributes in a subclass:

```python
class A:
    def __init__(self,v0,v1)
        self.v0 = v0
        self.v1 = v1

    def f(self, x):
            return x**2

class B(A):
    def __init__(self,v0,v1,v2):
        self.v0 = v0
        self.v1 = v1
        self.v2 = v2

    def g(self, x):
        return x**4
```

Usage:

```python
a = A(v0=1, v1=2)                #calling A.__init__
b = B(v0=1, v1=2, v3=3)                #calling B.__init__
```

0.8 The overridden method can still be called

A more elegant implementation looks like:

```
class A:
    def __init__(self,v0,v1)
        self.v0 = v0
        self.v1 = v1

    def f(self, x):
            return x**2

class B(A):
    def __init__(self,v0,v1,v2):
        super().__init__(v0,v1) #or A.__init__(self,v0,v1)
        self.v2 = v2

    def g(self, x):
        return x**4
```

0.9 Example: a class for straight lines

Problem: Make a class for evaluating lines $y = c_0 + c_1 x$.

Code:

```
impot numpy as np

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 np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

0.10 A class for parabolas

Problem: Make a class for evaluating parabolas $y = c_0 + c_1 x + c_2 x^2$.

Code:

```
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 np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

Observation: This is almost the same code as class `Line`, except for the things with `c2`

0.11 Class Parabola as a subclass of Line; principles

- `Parabola` code = `Line` code + a little extra with the c_2 term
- Can we utilize class `Line` code in class `Parabola`?
- This is what inheritance is about!

Writing

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

makes `Parabola` inherit all methods and attributes from `Line`, so `Parabola` has attributes `c0` and `c1` and three methods

- `Line` is a *superclass*, `Parabola` is a *subclass* (parent class, base class; child class, derived class)
- Class `Parabola` must add code to `Line`'s constructor (an extra `c2` attribute), `__call__` (an extra term), but `table` can be used unaltered
- The idea is to reuse as much code in `Line` as possible and avoid duplicating code

0.12 Class Parabola as a subclass of Line; code

A subclass method can call a superclass method in this way:

```
superclass_name.method(self, arg1, arg2, ...)
```

Class `Parabola` as a subclass of `Line`:

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

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

What is gained?

- Class `Parabola` just adds code to the already existing code in class `Line` - no duplication of storing `c0` and `c1`, and computing $c_0 + c_1x$
- Class `Parabola` also has a `table` method - it is inherited
- `__init__` and `__call__` are *overridden* or *redefined* in the subclass

0.13 We can check class type and class relations with `isinstance(obj, type)` and `issubclass(subclassname, superclassname)`

```python
>>> 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'
```

0.14 Line as a subclass of Parabola

- Subclasses are often special cases of a superclass
- A line $c_0 + c_1x$ is a special case of a parabola $c_0 + c_1x + c_2x^2$
- Can `Line` be a subclass of `Parabola`?

- No problem - this is up to the programmer's choice
- Many will prefer this relation between a line and a parabola

0.15 Code when Line is a subclass of Parabola

```
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 np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s

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

Note: `__call__` and `table` can be reused in class `Line`!

0.16 Recall the class for numerical differentiation from Ch. 8

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

```
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

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

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

0.17 There are numerous formulas for numerical differentiation

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

0.18 How can we make a module that offers all these formulas?

It's easy:

```
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__
```

0.19 What is the problem with this type of code?

All the constructors are identical so we duplicate a lot of code.

- A general OO idea: place code common to many classes in a superclass and inherit that code
- Here: inhert constructor from superclass,
  let subclasses for different differentiation formulas implement their version of `__call__`

0.20 Class hierarchy for numerical differentiation

Superclass:

```
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
```

Subclass for 4-th order central formula:

```
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)
```

0.21 Use of the differentiation classes

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
```

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

0.22 Formulas for numerical integration

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

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

w_i: weights, x_i: points (specific to a certain formula)

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

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

0.23 More formulas

Simpson's rule has

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}$$

$$w_0 = w_{n-1} = \frac{h}{6}$$

$$w_i = \frac{h}{3} \text{ for } i \text{ even}, \quad w_i = \frac{2h}{3} \text{ for } i \text{ odd}$$

Other rules have more complicated formulas for w_i and x_i

0.24 Why should these formulas be implemented in a class hierarchy?

- A numerical integration formula can be implemented as a class: a, b and n are attributes and an `integrate` method evaluates the formula
- All such classes are quite similar: the evaluation of $\sum_j w_j f(x_j)$ is the same, only the definition of the points and weights differ among the classes
- Recall: code duplication is a bad thing!
- The general OO idea: place code common to many classes in a superclass and inherit that code
- Here we put $\sum_j w_j f(x_j)$ in a superclass (method `integrate`)

- Subclasses extend the superclass with code specific to a math formula, i.e., w_i and x_i in a class method `construct_rule`

0.25 The superclass for integration

```
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
              (f'no rule in class {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))
```

0.26 A subclass: the Trapezoidal rule

```
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
```

0.27 Another subclass: Simpson's rule

- Simpson's rule is more tricky to implement because of different formulas for odd and even points
- Don't bother with the details of w_i and x_i in Simpson's rule now - focus on the class design!

```
class Simpson(Integrator):

    def construct_method(self):
        if self.n % 2 != 1:
            print(f'n={self.n} must be odd, 1 is added')
            self.n += 1

        <code for computing x and w>
        return x, w
```

0.28 About the program flow

Let us integrate $\int_0^2 x^2 dx$ using 101 points:

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

method = Simpson(0, 2, 101)
print method.integrate(f)
```

Important:

- `method = Simpson(...)`: this invokes the superclass constructor, which calls `construct_method` in class `Simpson`
- `method.integrate(f)` invokes the inherited `integrate` method, defined in class `Integrator`

0.29 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
- It often takes time to master *when* and *how* to use OOP
- Typical exercise in OOP; when creating a subclass, examine the superclass to identify the parts that can be reused, and what needs to be added. Often, the subclass definition can be quite short!