Chapter 8
Classes

In this chapter, we introduce classes, which is a fundamental concept in programming. Most modern programming languages support classes or similar concepts, and we have already encountered classes earlier in this book. Recall, for instance, from Chapter 2 how we can check the type of a variable with the `type` function, and the output will be of the form `<class 'int'>`, `<class 'float'>`, and so on. This simply states that the type of an object is defined in the form of a class. Every time we create, for instance, an integer variable in our program, we create an object or *instance* of the `int` class. The class defines how the objects behave and what methods they contain. We have used a large number of different methods bound to objects, such as the `append` method for list objects and `split` for strings. All such methods are part of the definition of the class to which the object belongs. So far, we have only used Python's built-in classes to create objects, but in this chapter we will write our own classes and use them to create objects tailored to our particular needs.

8.1 Basics of Classes

A class packs together data and functions in a single unit. As seen in previous chapters, functions that are bound to a class or an object are usually called methods, and we will stick to this notation in the present chapter. Classes have some similarity with modules, which are also collections of variables and functions that naturally belong together. However, while there can be only a single instance of a module, we can create multiple instances of a class. Different instances of the same class can contain different data, but they all behave in the same way and have the same methods. Think of a basic Python class such as `int`; we can create many integer variables in a program, and they obviously have different values (data), but we know that they all have the same general behavior and the same set of operations defined for them.

J. Sundnes, *Introduction to Scientific Programming with Python*, Simula SpringerBriefs on Computing 6,
https://doi.org/10.1007/978-3-030-50356-7_8

The same goes for more complex Python classes such as lists and strings; different objects contain different data, but they all have the same methods. The classes we create in this chapter behave in exactly the same way.

First example: A class representing a function. To start with a familiar example, we return to the formula calculating atmospheric pressure p as a function of altitude h. The formula we used is a simplification of a more general *barometric formula*, given by:

$$p = p_0 e^{-Mgh/RT}, \tag{8.1}$$

where M is the molar mass of air, g is the gravitational constant, R is the gas constant, T is temperature, and p_0 is the pressure at sea level. We obtain the simpler formula used earlier by defining the scale height as $h_0 = RT/Mg$. It could be interesting to evaluate (8.1) for different temperatures and, for each value of T, to create a table or plot of how the pressure varies with altitude. For each value of T, we need to call the function many times, with different values of h. How should we implement this in a convenient way? One possible solution would be to have both h and T as arguments:

```
from math import exp

def barometric(h, T):
    g = 9.81          #m/(s*s)
    R = 8.314         #J/(K*mol)
    M = 0.02896       #kg/mol
    p0 = 100.0        #kPa

    return p0 * exp(-M*g*h/(R*T))
```

This solution obviously works, but if we want to call the function many times for the same value of `T` then we still need to pass it as an argument every time it is called. However, what if the function is to be passed as an argument to another function that expects it to take a single argument only?[1] In this case, our function with two arguments will not work. A partial solution would be to include a default value for the `T` argument, but we would still have a problem if we want a different value of `T`.

Another solution would be to have h as the only argument, and T as a global variable:

```
T = 245.0

def barometric(h):
    g = 9.81          #m/(s*s)
```

[1] This situation is quite common in Python programs. Consider, for instance, the implementation of Newton's method in Chapter 4, in the functions `Newton` and `Newton2`. These functions expect two functions as arguments (`f` and `dfdx`), and both are expected to take a single argument (`x`). Passing in a function that requires two or more arguments will lead to an error.

```
    R = 8.314         #J/(K*mol)
    M = 0.02896       #kg/mol
    p0 = 100.0        #kPa

    return p0 * exp(-M*g*h/(R*T))
```

We now have a function that takes a single argument, but defining `T` as a global variable is not very convenient if we want to evaluate `y(t)` for different values of `T`. We could also set `T` as a local variable inside the function and define different functions `barometric1(h)`, `barometric2(h)`, etc., for different values of `T`, but this is obviously inconvenient if we want many values of `T`. However, we shall see that programming with classes and objects offers exactly what we need: a convenient solution to create a family of similar functions that all have their own value of `T`.

As mentioned above, the idea of a class is to pack together data and methods (or functions) that naturally operate on the data. We can make a class `Barometric` for the formula at hand, with the variables `R`, `T`,'M', `g`, and `p0` as data, and a method `value(t)` for evaluating the formula. All classes should also have a method named `__init__` to initialize the variables. The following code defines our function class

```
class Barometric:
    def __init__(self, T):
        self.T = T            #K
        self.g = 9.81         #m/(s*s)
        self.R = 8.314        #J/(K*mol)
        self.M = 0.02896      #kg/mol
        self.p0 = 100.0       #kPa


    def value(self, h):
        return self.p0 * exp(-self.M*self.g*h/(self.R*self.T))
```

Having defined this class, we can create *instances* of the class with specific values of the parameter `T`, and then we can call the method `value` with `h` as the only argument:

```
b1 = Barometric(T=245)        # create instance (object)
p1 = b1.value(2469)           # compute function value
b2 = Barometric(T=273)
p2 = b2.value(2469)
```

These code segments introduce a number of new concepts worth dissecting. First, we have a class definition that, in Python, always starts with the word `class`, followed by the name of the class and a colon. The following indented block of code defines the contents of the class. Just as we are used to when we implement functions, the indentation defines what belongs inside the class definition. The first contents of our class, and of most classes, is a method with the special name `__init__`, which is called the *constructor* of the class. This method is automatically called every time we create an instance in the class, as in the line `b1 = Barometric(T=245)` above. Inside the method, we

define all the constants used in the formula – `self.T`, `self.g`, and so on – where the prefix `self` means that these variables become bound to the object created. Such bound variables are called *attributes*. Finally, we define the method `value`, which evaluates the formula using the predefined and object-bound parameters `self.T, self.g, self.R, self.M`, and `self.p0`. After we have defined the class, every time we write a line such as

```
b1 = Barometric(T=245)
```

we create a new variable (instance) `b1` of type `Barometric`. The line looks like a regular function call, but, since `Barometric` is the definition of a class and not a function, `Barometric(T=245)` is instead a call to the class' constructor. The constructor creates and returns an instance of the class with the specified values of the parameters, and we assign this instance to the variable `b`. All the `__init__` functions we encounter in this book will follow exactly the same recipe. Their purpose is to define a number of attributes for the class, and they will typically contain one or more lines of the form `self.A = A`, where A is either an argument passed to the constructor or a value defined inside it.

As always in programming, there are different ways to achieve the same thing, and we could have chosen a different implementation of the class above. Since the only argument to the constructor is `T`, the other attributes never change and they could have been local variables inside the `value` method:

```
class Barometric1:
    def __init__(self, T):
        self.T = T             #K

    def value(self, h):
        g = 9.81; R = 9.314
        M = 0.02896; p0 = 100.0
        return p0 * exp(-M*g*h/(R*self.T))
```

Notice that, inside the `value` method, we only use the `self` prefix for `T`, since this is the only variable that is a class attribute. In this version of the class the other variables are regular local variables defined inside the method. This class does exactly the same thing as the one defined above, and one could argue that this implementation is better, since it is shorter and simpler than the one above. However, defining all the physical constants in one place (in the constructor) can make the code easier to read, and the class easier to extend with more methods. As a third possible implementation, we could move some of the calculations from the `value` method to the constructor:

```
class Barometric2:
    def __init__(self, T):
        g = 9.81          #m/(s*s)
        R = 8.314         #J/(K*mol)
        M = 0.02896       #kg/mol
        self.h0 = R*T/(M*g)
        self.p0 = 100.0       #kPa
```

```
    def value(self, h):
        return self.p0 * exp(-h/self.h0)
```

In this class, we use the definition of the scale height from above and compute and store this value as an attribute inside the constructor. The attribute `self.h0` is then used inside the `value` method. Notice that the constants `g`, `R`, and `M` are, in this case, local variables in the constructor, and neither these nor `T` are stored as attributes. They are only accessible inside the constructor, while `self.p0` and `self.h0` are stored and can be accessed later from within other methods.

At this point, many will be confused by the `self` variable, and the fact that, when we define the methods `__init__` and `value` they take two arguments, but, when calling them, they take only one. The explanation for this behavior is that `self` represents the object itself, and it is automatically passed as the first argument when we call a method bound to the object. When we write

```
p1 = b1.value(2469)
```

it is equivalent to the call

```
p1 = Barometric.value(b1,2469)
```

Here we explicitly call the `value` method that belongs to the `Barometric` class and pass the instance `b1` as the first argument. Inside the method, `b1` then becomes the local variable `self`, as is usual when passing arguments to a function, and we can access its attributes `T`, `g`, and so on. Exactly the same thing happens when we call `b1.value(2469)`, but now the object `b1` is automatically passed as the first argument to the method. It looks as if we are calling the method with a single argument, but in reality it gets two.

The use of the `self` variable in Python classes has been the subject of many discussions. Even experienced programmers find it confusing, and many have questioned why the language was designed this way. There are some obvious advantages to the approach, for instance, it very clearly distinguishes between instance attributes (prefixed with `self`) and local variables defined inside a method. However, if one is struggling to see the reasoning behind the `self` variable, it is sufficient to remember the following two rules: (i) `self` is always the first argument in a method definition, but is never inserted when the method is called, and (ii) to access an attribute inside a method, the attribute needs to be prefixed with `self`.

An advantage of creating a class for our barometric function is that we can now send `b1.value` as an argument to any other function that expects a function argument `f` that takes a single argument. Consider, for instance, the following small example, where the function `make_table` prints a table of the function values for any function passed to it:

```
from math import sin, exp, pi
```

```
from numpy import linspace

def make_table(f, tstop, n):
    for t in linspace(0, tstop, n):
        print(t, f(t))

def g(t):
    return sin(t)*exp(-t)

make_table(g, 2*pi, 11)         # send ordinary function

b1 = Barometric(2469)
make_table(b1.value, 2*pi, 11)   # send class method
```

Because of how `f(t)` is used inside the function, we need to send `make_table` a function that takes a single argument. Our `b1.value` method satisfies this requirement, but we can still use different values of `T` by creating multiple instances.

More general Python classes. Of course, Python classes have far more general applicability than just the representation of mathematical functions. A general Python class definition follows the recipe outlined in the example above, as follows:

```
class MyClass:
    def __init__(self, p1, p2,...):
        self.attr1 = p1
        self.attr2 = p2
    ...

    def method1(self, arg):
          #access attributes with self prefix
        result = self.attr1 + ...
        ...
        #create new attributes if desired
        self.attrx = arg
        ...
        return result

    def method2(self):
        ...
        print(...)
```

We can define as many methods as we want inside the class, with or without arguments. When we create an instance of the class the methods become bound to the instance, and are accessed with the prefix, for instance, `m.method2()` if `m` is an instance of `MyClass`. It is common to have a constructor where attributes are initialized, but this is not a requirement. Attributes can be defined whenever desired, for instance, inside a method, as in the line `self.attrx = arg` in the example above, or even from outside the class:

```
m = MyClass(p1,p2, ...)
m.new_attr = p3
```

The second line here creates a new attribute `new_attr` for the instance `m` of `MyClass`. Such addition of attributes is entirely valid, but it is rarely good programming practice since we can end up with instances of the same class having different attributes. It is a good habit to always equip a class with a constructor and to primarily define attributes inside the constructor.

8.2 Protected Class Attributes

For a more classical computer science example of a Python class, let us look at a class representing a bank account. Natural attributes for such a class will be the name of the owner, the account number, and the balance, and we can include methods for deposits, withdrawals, and printing information about the account. The code for defining such a class could look like this:

```
class BankAccount:
    def __init__(self, first_name, last_name, number, balance):
        self.first_name = first_name
        self.last_name = last_name
        self.number = number
        self.balance = balance


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

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

    def print_info(self):
        first = self.first_name; last = self.last_name
        number = self.number; bal = self.balance
        s = f'{first} {last}, {number}, balance: {balance}'
        print(s)
```

Typical use of the class could be something like the following, where we create two different account instances and call the various methods for deposits, withdrawals, and printing information:

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

```
Liz Olsson, 19371564761, balance: 9500
```

However, there is nothing to prevent a user from changing the attributes of the account directly:

```
>>> a1.first_name = 'Some other name'
>>> a1.balance = 100000
>>> a1.number = '19371564768'
```

Although it can be tempting to adjust a bank account balance when needed, it is not the intended use of the class. Directly manipulating attributes in this way will very often lead to errors in large software systems, and is considered a bad programming style. Instead, attributes should always be changed by calling methods, in this case, `withdraw` and `deposit`. Many programming languages have constructions that can limit the access to attributes from outside the class, so that any attempt to access them will lead to an error message when compiling or running the code. Python has no technical way to limit attribute access, but it is common to mark attributes as *protected* by prefixing the name with an underscore (e.g., `_name`). This convention tells other programmers that a given attribute or method is not supposed to be accessed from outside the class, even though it is still technically possible to do so. An account class with protected attributes can look like the following:

```
class BankAccountP:
    def __init__(self, first_name, last_name, number, balance):
        self._first_name = first_name
        self._last_name = name
        self._number = number
        self._balance = balance

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

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

    def get_balance(self):    # NEW - read balance value
        return self._balance

    def print_info(self):
        first = self.first_name; last = self.last_name
        number = self.number; bal = self.balance
        s = f'{first} {last}, {number}, balance: {balance}'
        print(s)
```

When using this class, it will still be technically possible to access the attributes directly, as in

```
a1 = BankAccountP('John', 'Olsson', '19371554951', 20000)
a1._number = '19371554955'
```

However, all experienced Python programmers will know that the second line is a serious violation of good coding practice and will look for a better

way to solve the task. When using code libraries developed by others, such conventions are risky to break, since internal data structures can change, while the *interface* to the class is more static. The convention of protected variables is how programmers tell users of the class what can change and what is static. Library developers can decide to change the internal data structure of a class, but users of the class might not even notice this change if the methods to access the data remain unchanged. Since the class interface is unchanged, users who followed the convention will be fine, but users who have accessed protected attributes directly could be in for a surprise.

8.3 Special Methods

In the examples above, we define a constructor for each class, identified by its special name `__init__(...)`. This name is recognized by Python, and the method is automatically called every time we create a new instance of the class. The constructor belongs to a family of methods known as *special methods*, which are all recognized by double leading and trailing underscores in the name. The term *special methods* could be a bit misleading, since the methods themselves are not really special. The special thing about them is the name, which ensures that they are automatically called in different situations, such as the `__init__` function being called when class instances are created. There are many more such special methods that we can use to create object types with very useful properties.

Consider, for instance, the first example of this chapter, where the class `Barometric` contained the method `value(h)` to evaluate a mathematical function. After creating an instance named `baro`, we could call the method with `baro.value(t)`. However, it would be even more convenient if we could just write `baro(t)` as if the instance were a regular Python function. This behavior can be obtained by simply changing the name of the `value` method to one of the special method names that Python automatically recognizes. The special method name for making an instance *callable* like a regular Python function is `__call__`:

```
class Barometric:
    def __init__(self, T):
        self.T = T            #K
        self.g = 9.81         #m/(s*s)
        self.R = 8.314        #J/(K*mol)
        self.M = 0.02896      #kg/mol
        self.p0 = 100.0       #kPa


    def __call__(self, h):
        return self.p0 * exp(-self.M*self.g*h/(self.R*self.T))
```

Now we can call an instance of the class `Barometric` just as any other Python function

```
baro = Barometric(245)
p = baro(2346)          #same as p = baro.__call__(2346)
```

The instance `baro` now behaves and looks like a function. The method is exactly the same as the `value` method, but creating a special method by renaming it to `__call__` produces nicer syntax when the class is used.

Special method for printing. We are used to printing an object `a` using `print(a)`, which works fine for Python's built-in object types such as strings and lists. However, if `a` is an instance of a class we defined ourselves, we do not obtain much useful information, since Python does not know what information to show. We can solve this problem by defining a special method named `__str__` in our class. The `__str__` method must return a string object, preferably a string that provides useful information about the object, and it should not take any arguments except `self`. For the function class seen above, a suitable `__str__` method could look like the following:

```
class Barometric:
    ...
    def __call__(self, h):
        return self.p0 * exp(-self.M*self.g*h/(self.R*self.T))

    def __str__(self):
        return f'p0 * exp(-M*g*h/(R*T)); T = {self.T}'
```

If we now call `print` for an instance of the class, the function expression and the value of `T` for that instance will be printed, as follows:

```
>>> b = Barometric(245)
>>> b(2469)
70.86738432067067
>>> print(b)
p0 * exp(-M*g*h/(R*T)); T = 245
```

Special methods for mathematical operations. So far we have seen three special methods, namely, `__init__`, `__call__`, and `__str__`, but there are many more. We will not cover them all in this book, but a few are worth mentioning. For instance, there are special methods for arithmetic operations, such as `__add__`, `__sub__`, `__mul__`, and so forth. Defining these methods inside our class will enable us to perform operations such as `c = a+b`, where `a,b` are instances of the class. The following are relevant arithmetic operations and the corresponding special method that they will call:

```
c = a + b    #  c = a.__add__(b)

c = a - b    #  c = a.__sub__(b)

c = a*b      #  c = a.__mul__(b)
```

```
c = a/b      #  c = a.__div__(b)

c = a**e     #  c = a.__pow__(e)
```

It is natural, in most but not all cases, for these methods to return an object of the same type as the operands. Similarly, there are special methods for comparing objects,as follows:

```
a == b       #  a.__eq__(b)

a != b       #  a.__ne__(b)

a < b        #  a.__lt__(b)

a <= b       #  a.__le__(b)

a > b        #  a.__gt__(b)

a >= b       #  a.__ge__(b)
```

These methods should be implemented to return true or false, to be consistent with the usual behavior of the comparison operators. The actual contents of the special method are in all cases entirely up to the programmer. The only special thing about the methods is their name, which ensures that they are automatically called by various operators. For instance, if you try to multiply two objects with a statement such as `c = a*b`, Python will look for a method named `__mul__` in the instance `a`. If such a method exists, it will be called with the instance `b` as the argument, and whatever the method `__mul__` returns will be the result of our multiplication operation.

The `__repr__` special method. The last special method we will consider here is a method named `__repr__`, which is similar to `__str__` in the sense that it should return a string with information about the object. The difference is that, while `__str__` should provide human-readable information, the `__repr__` string will contain all the information necessary to recreate the object. For an object `a`, the `__repr__` method is called if we call `repr(a)`, where `repr` is a built-in function. The intended function of `repr` is such that `eval(repr(a)) == a`, that is, running the string output by `a.__repr__` should recreate `a`. To illustrate its use, let us add a `__repr__` method to the class `Barometric` from the start of the chapter:

```
class Barometric:
    ...
    def __call__(self, h):
        return self.p0 * exp(-self.M*self.g*h/(self.R*self.T))

    def __str__(self):
        return f'p0 * exp(-M*g*h/(R*T)); T = {self.T}'

    def __repr__(self):
```

```
        """Return code for regenerating this instance."""
        return f'Barometric({self.T})'
```

Again, we can illustrate how it works in an interactive shell:

```
>>> from tmp import *
>>> b = Barometric(271)
>>> print(b)
p0 * exp(-M*g*h/(R*T)); T = 245
>>> repr(b)
'Barometric(271)'
>>>  b2 = eval(repr(b))
>>> print(b2)
p0 * exp(-M*g*h/(R*T)); T = 245
```

The last two lines confirm that the `repr` method works as intended, since running `eval(repr(b)` returns an object identical to `b`. Both `__repr__` and `__str__` return strings with information about an object, the difference being that `__str__` gives information to be read by humans, whereas the output of `__repr__` is intended to be read by Python.

How to know the contents of a class. Sometimes listing the contents of a class can be useful, particularly for debugging. Consider the following dummy class, which does nothing useful except to define a doc string, a constructor, and a single attribute:

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

If we now write `dir(A)` we see that the class actually contains a great deal more than what we put into it, since Python automatically defines certain methods and attributes in all classes. Most of the items listed are default versions of special methods, which do nothing useful except to give the error message `NotImplemented` if they are called. However, if we create an instance of `A`, and use `dir` on that instance, we obtain more useful information:

```
>>> a = A(2)
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'v']
```

We see that the list contains the same (mostly useless) default versions of special methods, but some of the items are more meaningful. If we continue the interactive session to examine some of the items, we obtain

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

```
>>> a.__dict__
{'v': 2}
>>> a.v
2
>>> a.__module__
'__main__'
```

The `__doc__` attribute is the doc string we defined, while `__module__` is the name of the module to which class belongs, which is simply `__main__` in this case, since we defined it in the main program. However, the most useful item is probably `__dict__`, which is a dictionary containing the names and values of all the attributes of the object `a`. Any instance holds its attributes in the `self.__dict__` dictionary, which is automatically created by Python. If we add new attributes to the instance, they are inserted into the `__dict__`:

```
>>> a = A([1,2])
>>> print a.__dict__  # all attributes
{'v': [1, 2]}
>>> a.myvar = 10            # add new attribute (!)
>>> a.__dict__
{'myvar': 10, 'v': [1, 2]}
```

When programming with classes we are not supposed to use the internal data structures such as `__dict__` explicitly, but printing it to check the values of class attributes can be very useful if something goes wrong in our code.

8.4 Example: Automatic Differentiation of Functions

To provide a more relevant and useful example of a `__call__` special method, consider the task of computing the derivative of an arbitrary function. Given some mathematical function in Python, say,

```
def f(x):
    return x**3
```

we want to make a class `Derivative` and write

```
dfdx = Derivative(f)
```

so that `dfdx` behaves as a function that computes the derivative of `f(x)`. When the instance `dfdx` is created, we want to call it like a regular function to evaluate the derivative of `f` in a point `x`:

```
print(dfdx(2))   # computes 3*x**2 for x=2
```

It is tricky to create such a class using analytical differentiation rules, but we can write a generic class by using numerical differentiation:

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

For a small (yet moderate) h, say $h = 10^{-5}$, this estimate will be sufficiently accurate for most applications. The key parts of the implementation are to let the function `f` be an attribute of the `Derivative` class and then implement the numerical differentiation formula in a `__call__` special method:

```
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
```

The following interactive session demonstrates typical use of the class:

```
>>> from math import *
>>> df = Derivative(sin)
>>> x = pi
>>> df(x)
-1.000000082740371
>>> cos(x)  # exact
-1.0
>>> def g(t):
...     return t**3
...
>>> dg = Derivative(g)
>>> t = 1
>>> dg(t)  # compare with 3 (exact)
3.000000248221113
```

For a particularly useful application of the `Derivative` class, consider the solution of a nonlinear equation $f(x) = 0$. In Chapter 4 we implement Newton's method as a general method for solving nonlinear equations, but Newton's method uses the derivative $f'(x)$, which needs to be provided as an argument to the function:

```
def Newton2(f, dfdx, x0, max_it=20, tol= 1e-3):
    ...
    return x0, converged, iter
```

See Chapter 4 for a complete implementation of the function. For many functions $f(x)$, finding $f'(x)$ can require lengthy and boring derivations, and in such cases the `Derivative` class is quite handy:

```
>>> def f(x):
...     return 100000*(x - 0.9)**2 * (x - 1.1)**3
...
>>> dfdx = Derivative(f)
>>> xstart = 1.01
>>> Newton2(f, dfdx, xstart)
```

```
(1.093562409134085, True, 4)
```

8.5 Test Functions for Classes

In Chapter 4 we introduced test functions as a method to verify that our functions were implemented correctly, and the exact same approach can be used to test the implementation of classes. Inside the test function, we define parameters for which we know the expected output, and then call our class methods and compare the results with those expected. The only additional step involved when testing classes is that we will typically create one or more instances of the class inside the test function and then call their. As an example, consider a test function for the `Derivative` class of the previous section. How can we define a test case with known output for this class? Two possible methods are; (i) to compute $(f(x+h)-f(x))/h$ by hand for some f and h, or (ii) utilize the fact that linear functions are differentiated exactly by our numerical formula, regardless of h. A test function based on (ii) could look like the following:

```
def test_Derivative():
    # The formula is exact for linear functions, regardless of h
    f = lambda x: a*x + b
    a = 3.5; b = 8
    dfdx = Derivative(f, h=0.5)
    diff = abs(dfdx(4.5) - a)
    assert diff < 1E-14, 'bug in class Derivative, diff=%s' % diff
```

This function follows the standard recipe for test functions: we construct a problem with a known result, create an instance of the class, call the method, and compare the result with the expected result. However, some of the details inside the test function may be worth commenting on. First, we use a lambda function to define `f(x)`. As you may recall from Chapter 4, a lambda function is simply a compact way of defining a function, with

```
f = lambda x: a*x + b
```

being equivalent to

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

The use of the lambda function inside the test function appears straightforward at first:

```
f = lambda x: a*x + b
a = 3.5; b = 8
dfdx = Derivative(f, h=0.5)
dfdx(4.5)
```

The function `f` is defined to taking one argument `x` and also using two two local variables `a` and `b` that are defined outside the function before it is called. However, looking at this code in more detail can raise questions. Calling `dfdx(4.5)` implies that `Derivative.__call__` is called, but how can this methods know the values of `a` and `b` when it calls our `f(x)` function? These variables are defined inside the test function and are therefore local, whereas the class is defined in the main program. The answer is that a function defined inside another function "remembers," or has access to, *all* the local variables of the function where it is defined. Therefore, all the variables defined inside `test_Derivative` become part of the *namespace* of the function `f`, and `f` can access `a` and `b` in `test_Derivative` even when it is called from the `__call__` method in class `Derivative`. This construction is known as a *closure* in computer science.

8.6 Example: A Polynomial Class

As a summarizing example of classes and special methods, we can consider the representation of polynomials introduced in Chapter 7. A polynomial can be specified by a dictionary or list representing its coefficients and powers. For example, $1-x^2+2x^3$ is

$$1+0\cdot x-1\cdot x^2+2\cdot x^3$$

and the coefficients can be stored as a list `[1, 0, -1, 2]`. We now want to create a class for such a polynomial and equip it with functionality to evaluate and print polynomials and to add two polynomials. Intended use of the class `Polynomial` could look like the following:

```
>>> p1 = Polynomial([1, -1])
>>> print(p1)
1 - x
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print(p3.coeff)
[1, 0, 0, 0, -6, -1]
>>> print(p3)
1 - 6*x^4 - x^5
>>> print(p3(2.0))
-127.0
>>> p4 = p1*p2
>>> p2.differentiate()
>>> print(p2)
1 - 24*x^3 - 5*x^4
```

To make all these operations possible, the class needs the following special methods:

- `__init__`, the constructor, for the line `p1 = Polynomial([1,-1])`
- `__str__`, for doing `print(p1)`
- `__call__`, to enable the call `p3(2.0)`
- `__add__`, to make `p3 = p1 + p2` work
- `__mul__`, to allow `p4 = p1*p2`

In addition, the class needs a method `differentiate` that computes the derivative of a polynomial, and changes it in-place. Starting with the most basic methods, the constructor is fairly straightforward and the call method simply follows the recipe from Chapter 7:

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

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

To enable the addition of two polynomials, we need to implement the `__add__` method, which should take one argument in addition to `self`. The method should return a new `Polynomial` instance, since the sum of two polynomials is a polynomial, and the method needs to implement the rules of polynomial addition. Adding two polynomials means to add terms of equal order, which, in our list representation, means to loop over the `self.coeff` lists and add individual elements, as follows:

```
class Polynomial:
    ...

    def __add__(self, other):
        # return self + other

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

The order of the sum of two polynomials is equal to the highest order of the two, so the length of the returned polynomial must be equal to the length of the longest of the two `coeff` lists. We utilize this knowledge in the code by starting with a copy of the longest list and then looping through the shortest and adding to each element.

The multiplication of two polynomials is slightly more complex than their addition, so it is worth writing down the mathematics before implementing the `__mul__` method. The formula looks like

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

which, in our list representation, means that the coefficient corresponding to the power $i+j$ is $c_i \cdot d_j$. The list `r` of coefficients for the resulting polynomial should have length $N+M+1$, and an element `r[k]` should be the sum of all products `c[i]*d[j]` for which $i+j=k$. The implementation of the method could look like

```
class Polynomial:
    ...
    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1)  # or zeros(M+N+1)
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i]*other.coeff[j]
        return Polynomial(coeff)
```

Just as the `__add__` method, `__mul__` takes one argument in addition to `self`, and returns a new `Polynomial` instance.

Turning now to the `differentiate` method, the rule for differentiating a general polynomial is

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

Therefore, if `c` is the list of coefficients, the derivative has a list of coefficients `dc`, where `dc[i-1] = i*c[i]` for `i` from one to the largest index in `c`. Note that `dc` will have one element less than `c`, since differentiating a polynomial reduces the order by one. The full implementation of the `differentiate` method could look like the following:

```
class Polynomial:
    ...
    def differentiate(self):    # change self
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]

    def derivative(self):       # return new polynomial
        dpdx = Polynomial(self.coeff[:])  # copy
        dpdx.differentiate()
        return dpdx
```

Here, the `differentiate` method will change the polynomial itself, since this is the behavior indicated by the way the function was used above. We have also added a separate function `derivative` that does not change the polynomial but, instead, returns its derivative as a new `Polynomial` object.

Finally, let us implement the `__str__` method for printing the polynomial in human-readable form. This method should return a string representation close to the way we write a polynomial in mathematics, but achieving this can be surprisingly complicated. The following implementation does a reasonably good job:

```
class Polynomial:
    ...
    def __str__(self):
        s = ''
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += f' + {self.coeff[i]:g}*x^{i:g}'
        # fix layout (many special cases):
        s = s.replace('+ -', '- ')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^0', '1')
        s = s.replace('x^1 ', 'x ')
        if s[0:3] == ' + ':  # remove initial +
            s = s[3:]
        if s[0:3] == ' - ':  # fix spaces for initial -
            s = '-' + s[3:]
        return s
```

For all these special methods, as well as special methods in general, it is important to be aware that their contents and behavior are entirely up to the programmer. The only *special* thing about special methods is their name, which ensures that they are automatically called by certain operations. What they actually do and what they return are decided by the programmer writing the class. If we want to write an `__add__` method that returns nothing, or returns something completely different from a sum, we are free to do so. However, it is, of course, a good habit for the `__add__(self, other)` to implement something that seems like a meaningful result of `self + other`.