Ch.7: Introduction to classes

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, De

Oct 11, 2019

In this chapter we will introduce classes, which is a fundamental concept in programming. Most modern programming languages support classes or similar concepts, and we have already used classes extensively throughout this book. Recall how we could check the type of a variable with the `type` method, and the output would be on the form `<class 'int'>`, `<class 'float'>`, etc. 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 number of different methods bound to objects, such as the `append` method for list objects, `split` for strings, and many more. All such methods are part of the definition of the class that the object belongs to. So far we have only used Python's builtin 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.

1 Basics of classes

A class packs together data and functions, or methods, in a single unit. As we have seen in previous chapters, functions that are bound to a class or objects 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 is only a single instance of a module, we can create multiple instances of a class. Different instances of the same class may contain different data, but they all behave in the same way. and have the same methods. Think of a basic Python class like `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. The same goes for more complex Python classes like lists and strings; different objects contain different data but they all have the same methods. The classes we will create in this chapter behaves in exactly the same way.

First example; a class representing a function. To start with a familiar example, consider the function introduced earlier that defines the height of an object as a function of t:

$$y(t; v_0) = v_0 t - \frac{1}{2} g t^2$$

We need both v_0, t, and g to compute $y.$, but how should we implement this? It is natural to think of g as constant, but both v_0 and t may vary with every call. However, for many applications of functions of this kind t will vary much more frequently than v_0. Since v_0 can be thought of as a parameter in a model, it is quite common to call such a function many times with the same v_0 but different t-values. How should we implement this in a convenient way? The default solution would be to have both t and v_0 as arguments, possibly with a default value for v_0:

```
def y(t, v0 = 5):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

This solution obviously works, but if we want a different value of v_0 we need to pass the value to the function every time it is called. And what if the function is to be passed as an argument to another function, which expects it to take a single argument only? [1]

Another solution would to have t as argument and v_0 as global variable:

```
def y(t):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

We now have a function which only takes a single argument, but defining `v0` as a global variable is not very convenient if we want to evaluate `y(t)` for different values of `v0`. A third possible solution would be to set `v0` as a local variable inside the function, and define different functions `y1(t), y2(t), y3(t)`, etc. for each value of `v0`. This solution is obviously not very convenient if we want many values of `v0`, but we shall see that programming with classes and objects offers exactly that; a convenient solution to create a family of similar functions.

Representing a function by a class. With a class, `y(t)` can be a function of `t` only, but still have `v0` and `g` as parameters with given values. The class packs together a function (or method) `y(t)` and data (`v0`, `g`). We make a class `Y` for $y(t; v_0)$ with variables `v0` and `g` and a function `value(t)` for computing $y(t; v_0)$. All classes should also have a function named `__init__` for initializing the variables. The following code defines our function class

```
class Y:
    def __init__(self, v0):
```

[1] This situation is fairly common in Python programs. Consider for instance the function implementing Newton's method in Appendix A. This function takes two functions as arguments, and because of how they are used inside the function both need to take a single argument (`x`). If we want to pass a parameterized function as argument to such a function, it needs to be modified.

```
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

Having defined this class, we can create *instances* of the class with specific values of the parameter `v0`, and then we can call the method `value` with `t` as the only argument:

```
y1 = Y(v0=3)             # create instance (object)
v1 = y1.value(0.1)        # compute function value
y2 = Y(v0=5)
v2 = y2.value(0.1)
```

Although this code is short, there are a number of new concepts worth dissecting. A class definition which 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 defininining 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 the *constructor* of the class. This method is automatically called every time we create an instance in the class, as in the line `y = Y(v0=3)` above. Inside the method, we define two variables `self.v0` and `self.g`, 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 pre-defined and object-bound parameters `self.v0` and `self.g`. After we have defined the class, every time we write a line like

```
y = Y(v0=3)
```

we create a new variable (instance) `y` of type `Y`. The line looks like a regular function call, but since `Y` is the definition of a class and not a function, `Y(v0=3)` is instead a call to the class' *constructor*.

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

```
v1 = y1.value(0.1)
```

it is equivalent to the call

```
v1 = Y.value(y1,0.1)
```

Here we explicitly call the `value` method that belongs to the class, and pass the instance `y1` as the first argument. Inside the method `y1` then becomes the local variable `self`, as usual when passing arguments to a function, and we can access its variables `v0` and `g`. Exactly the same thing happens when we call `y1.value(0.1)`, but now the object `y1` is automatically passed as the first

argument to the method. It looks like 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 people question why the language was designed in this way. There are some obvious advantages to the approach, for instance that it gives a very clear distinction between instance attributes (prefixed with `self`) and local variables defined inside a method. However, if one struggles to see the reasoning behind the `self` variable it is sufficient to remember the two rules; (i) `self` is always the first argument in a method definition, but never inserted when the method is called, and (ii) to access an attribute inside a method it needs to be prefixed with `self`.

> *In mathematics you don't understand things. You just get used to them.* John von Neumann, mathematician, 1903-1957.

The advantage of creating a class in this example is that we can now send `y.value` as an ordinary function of `t` to any other function that expects a function `f(t)` of one variable. Consider for instance the following small example, where the function `make_table` will print a table of function values for any function passed to it:

```
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)

table(g, 2*pi, 101)          # send ordinary function

y = Y(6.5)
table(y.value, 2*pi, 101)    # send class method
```

We need to send `make_table` a function that takes a single argument, Because of how `f(t)` is used inside the function. Our our `y.value` method satisfies this requirement and still allows us to use different values of `v0`.

More general Python classes. As a first generalization of the example above, consider a class to represent a function with $n+1$ parameters and one independent variable,

$$f(x; p_0, \ldots, p_n).$$

The natural class representation is a simple extension of the previous class, where $p_0, \ldots, p_n$ are attributes defined in the class constructor, and we define a method, say `value(self, x)`, to evaluate $f(x)$:

```
class MyFunc:
    def __init__(self, p0, p1, p2, ..., pn):
        self.p0 = p0
```

```
        self.p1 = p1
        ...
        self.pn = pn

    def value(self, x):
        return ...
```

Of course, Python classes have far more general applicability than just to represent mathematical functions. A general Python class follows the recipe outlined in the examples above:

```
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 an 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 example above, or even from outside the class:

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

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

A class for a bank account. For a more classical computer science example of a Python class, let us look at a class to represent a bank account. Natural attributes for such a class will be the name of the owner, the account number, and the balance, and we may include methods to deposit, withdraw, and print information about the account. The code for defining such a class may look like this:

```
class Account:
    def __init__(self, name, account_number, initial_amount):
        self.name = name
        self.no = account_number
        self.balance = initial_amount

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

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

    def dump(self):
        s = f'{self.name}, {self.no}, balance: {self.balance}'
        print(s)
```

Typical use of the class may be something like the following, where we create two different account instances and call the various methods to deposit, withdraw, and print:

```
>>> a1 = Account('John Olsson', '19371554951', 20000)
>>> a2 = Account('Liz Olsson',  '19371564761', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a2.withdraw(10500)
>>> a1.withdraw(3500)
>>> print "a1's balance:", a1.balance
a1's balance: 13500
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> a2.dump()
Liz Olsson, 19371564761, balance: 9500
```

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

```
>>> a1.name = 'Some other name'
>>> a1.balance = 100000
>>> a1.no = '19371564768'
```

While it may be tempting to adjust a bank account balance when needed, it is not the intended use of the class. Directly manipulating attributes will very often lead to errors in large software systems, and is considered to be bad programming style. Instead, attributes should always be changed by calling methods, in this case `withdraw` and `deposit`. Many programming languages have constructions that may 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 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, although it is still technically possible to do so. An account class with protected attributes may look like this:

```
class AccountP:
    def __init__(self, name, account_number, initial_amount):
        self._name = name
```

```
        self._no = account_number
        self._balance = initial_amount

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

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

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

    def dump(self):
        print(f'{self._name}, {self._no}, balance: {self._balance}')
```

When using this class, it will still be technically possible to do something like this:

```
a1 = Account('John Olsson', '19371554951', 20000)
a1._no = '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 problem. When using code libraries developed by others, breaking conventions is risky since internal data structures may change, while the interface to the class is more static. The convention of protected variables is how programmers tell users of the class what may change and what is static. For instance, in a library used by many others over a long period of time, the developers may decide to change the internal data structures of a class. However, if the methods to access the data remains unchanged, the users of the class may not even notice such changes, since the class *interface* is not changed. But users who have broken the convention, and accessed protected attributes directly, may be in for a surprise.

2 Special methods

In the examples above we defined a constructur 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* may 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 when class instances are created. There are many more such special methods, which we can use to create object types with very useful properties.

Consider for instance the first example of this chapter, where the class contained a method `value(t)` to evaluate the mathematical function. After creating an instance `y`, we would call the method with `y.value(t)`. Wouldn't it be more convenient if we could just write `y(t)` as if the instance was a regular Python function? This can obtained if we replace the `value` method with a special method named `__call__`:

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

Now we can call ay instance of the class `Y` just as any other Python function

```
y = Y(3)
v = y(0.1) # same as v = y.__call__(0.1) or Y.__call__(y, 0.1)
```

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

Example; automatic differentiation of functions. To provide another example of using the `__call__` special method, consider the task of computing derivatives of an arbitrary function. Given some mathematical function in Python, say

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

can we 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 make 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 solution of nonlinear equations $f(x) = 0$. In Appendix A we implement Newton's method as a general method for this task, but Newton's method uses the derivative $f'(x)$, which needs to be provided as an argument to the function:

```
def Newton(f, xstart, dfdx, epsilon=1E-6):
    ...
    return x, no_of_iterations, f(x)
```

See Appendix A for a complete implementation of the function. For many functions $f(x)$, finding $f'(x)$ may 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
...
>>> df = Derivative(f)
>>> xstart = 1.01
>>> Newton(f, xstart, df, epsilon=1E-5)
(1.0987610068093443, 8, -7.5139644257961411e-06)
```

Testing our class for automatic differentiation. How can we test the `Derivative` class? Two possible methods are; (i) compute $(f(x+h) - f(x))/h$ by hand for some f and h, or (ii) utilize that linear functions are differentiated exactly by our numerical formula, regardless of h. A test function based on (ii) may look as follows:

```
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 where we know the result, create an instance of the class, call the function and compare the result wiht the expected. However, some of the details inside the test function may be worth commenting. First, we use a lambda function to define `f(x)`. As we may recall from Chapter 3, 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 looks straightforward at first:

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

But looking at this code in more detail may give rise to some questions. When we call `dfdx(4.5)` it implies calling `Derivative.__call__` but how can this function know the values of know `a` and `b` when it calls our `f(x)` function? The answer is that a function defined inside another function "remembers", or has access to, *all* local variables in the function it is defined are defined. Therefore all variables defined inside `test_Derivative` become part of the *namespace* of the function `f`. Therefore `f` can access `a` and `b` in `test_Derivative` even when it is called from the `__call__` method in class `Derivative`. The construction is known as a *closure* in computer science.

Special method for printing. We are used to printing an object `a` using `print(a)`, which works fine for Python's builtin object types such as strings, lists, etc. However, if `a` is an instance of a class we defined ourselves we do not get 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 gives some useful information about the object, and should not take any arguments except `self`. For the function class seen above, a suitable `__str__` method may look like this:

```
class Y:
    ...
    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2

    def __str__(self):
        return f'v0*t - 0.5*g*t**2; v0={self.v0}'
```

If we now call print for an instance of the class, it will print the function expression:

```
>>> y = Y(1.5)
>>> y(0.2)
0.1038
>>> print(y)
v0*t - 0.5*g*t**2; v0=1.5
```

2.1 Special methods for arithmetic operations

So far we have seen three special methods; `__init__`, `__call__`, and `__str__`, but there are many more. We will not cover all of them in this book, but a few are worth mentioning. For instance, there are special methods for arithmetic operations, such as `__add__`, `__sub__`, `__mul__`, etc. Defining these methods inside our class will enable us to perform operations like `c = a+b`, where `a,b` are instances of the class. To illustrate this with an example, consider the representation of polynomials introduced in Chapter 6. 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 for evaluating and printing a polynomial and to add two polynomials. Intended use of the class `Polynomial` may look as follows:

```
>>> 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 pretty print, 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`, which computes the derivative of a polynomial, and changes it in-place. Starting with the most basic ones, the constructor is fairly straightforward and the call method simply follows the recipe from Chapter 6:

```
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 adding two polynomials, we need to implement the `__add__` method, which should take one argument in addition to `self`. The methot 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. This is basically to add together terms of equal order, which in our list representation means to loop over the `coeff` lists and add individual elements.

```
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.

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

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

and in our list representation this means that the coefficient corresponding to power $i+j$ is $c_i \cdot d_j$. The list `r` of coefficients of the resulting polynomial becomes

```
`r[i+j] = c[i]*d[j]`
```

where `i` and `j` run from 0 to M and N, respectively. The implementation of the method may 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

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

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

```
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 how the function was used above. We have also added a separate function `derviative`, which does not change the polynomial but instead returns its derivative as a new `Polynomial` object.

Finally, let us implement the `__str__` method for pretty print of polynomials. This method should return a string representation of the polynomial, but achieving this can actually be fairly complicated. The following implementatino 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 += ' + %g*x^%d' % (self.coeff[i], i)
        # fix layout (lots of special cases):
        s = s.replace('+ -', '- ')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^0', '1')
        s = s.replace('x^1 ', 'x ')
```

```
    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, and for special methods in general, it is important to be aware that the contents and behavior of the methods 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, is up to the programmer when 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. But it is of course a good habit for the `__add__(self, other)` to implement something that seems like a meaningful result of `self + other`.

Special methods for mathematical operations. We can equip our Python classes to support more than just addition and multiplication. Here are some 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 cases, but not always, that these methods return an opject of the same type as the operands. Similarly, there are special methods for comparing objects:

```
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 should be implemented to return `True/False` to be consistent with the usual behavior of the comparison operators.

Example; a class for vectors in the plane. Two-dimensional (2D) vectors have a set of well-defined mathematical operations. For two vectors (a, b) and

(c, d), we have

$$
\begin{aligned}
(a, b) + (c, d) &= (a + c, b + d) \\
(a, b) - (c, d) &= (a - c, b - d) \\
(a, b) \cdot (c, d) &= ac + bd \\
(a, b) &= (c, d) \text{ if } a = c \text{ and } b = d
\end{aligned}
$$

We want to implement a class for such 2D vectors, which supports these operations. We may recall that NumPy arrays support all of these operations, but the result of the operation is not always what we want. NumPy defines array operations, which for addition, subtraction, and equality give the same results as the rules defined above. However, multiplying two arrays gives a new array and not a scalar, so to support all the operations we need to implement things differently. We want the class to support the following usage:

```
>>> u = Vec2D(0,1)
>>> v = Vec2D(1,0)
>>> print u + v
(1, 1)
>>> a = u + v
>>> w = Vec2D(1,1)
>>> a == w
True
>>> print u - v
(-1, 1)
>>> print u*v
0
```

The implementation of the `Vec2D` class may look as follows:

```
class Vec2D:
    def __init__(self, x, y):
        self.x = x;  self.y = y

    def __add__(self, other):
        return Vec2D(self.x+other.x, self.y+other.y)

    def __sub__(self, other):
        return Vec2D(self.x-other.x, self.y-other.y)

    def __mul__(self, other):
        return self.x*other.x + self.y*other.y

    def __abs__(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return f'({self.x}, {self.y})'

    def __ne__(self, other):
        return not self.__eq__(other)  # reuse __eq__
```

Here, since we wanted `__mul__` to represent the scalar product, it returns a number rather than a `Vec2D` object.

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 info about the object. The difference is that while `__str__` should provide human readable information, the `__repr__` string shall 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 builtin function. The intended function of `repr` is such that `eval(repr(a)) == a`, i.e., running the string output by `a.__repr__` should recreate `a`. To illustrate its use, let us add a `__repr__` method to the class `Y` from the start of the chapter:

```
class Y:
    """Class for function y(t; v0, g) = v0*t - 0.5*g*t**2."""

    def __init__(self, v0):
        """Store parameters."""
        self.v0 = v0
        self.g = 9.81

    def __call__(self, t):
        """Evaluate function."""
        return self.v0*t - 0.5*self.g*t**2

    def __str__(self):
        """Pretty print."""
        return f'v0*t - 0.5*g*t**2; v0={self.v0}'

    def __repr__(self):
        """Print code for regenerating this instance."""
        return f'Y({self.v0})'
```

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

```
>>> from tmp import *
>>> y = Y(3)
>>> print(y)
v0*t - 0.5*g*t**2; v0=3
>>> repr(y)
'Y(3)'
>>> z = eval(repr(y))
>>> print(z)
v0*t - 0.5*g*t**2; v0=3
```

The last two lines confirm that the `repr` method works as intended, since running `eval(repr(y)` returns an object identical to `y`. The methods `__repr__` and `__str__` are fairly differentquite different

How can we know the contents of a class? Sometimes it is useful to be able to list the contents of a class, in particular for debugging. Consider the following dummy class, which does nothing useful but defines 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 lot 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 giving an error message `NotImplemented` if they are called. However, if we create an instance of `A`, and use `dir` again on that instance, we get more useful information:

```
>>> a = A(2)
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
```

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

```
>>> 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 module that the class belongs to, 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 names and values of all 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 like `__dict__` explicitly, but it may be very useful to print it to check values of variables if something goes wrong in our code.

3 Summary of class programming

Although the class concept itself is quite complex, the Python syntax of class programming is fairly simple. To define a class, simply write the keyword `class` followed by the class name and an indented block of code. The indented block will typically be method definitions, one of the methods being the constructor where the class attributes are defined:

```
class Gravity:
    """Gravity force between two objects."""
    def __init__(self, m, M):
        self.m = m
        self.M = M
        self.G = 6.67428E-11 # gravity constant

    def force(self, r):
        G, m, M = self.G, self.m, self.M
        return G*m*M/r**2

    def visualize(self, r_start, r_stop, n=100):
        from matplotlib.pyplot import plot, show
        from numpy import linspace
        r = linspace(r_start, r_stop, n)
        g = self.force(r)
        title = f'm={self.m}, M={self.M}'
        plot(r, g, title=title)
        show()
```

The least intuitive part of the class definition is probably the use of `self` as the first argument in all methods. As indicated by the name, one should always think of `self` as the instance itself, and stick to the following three rules; (i) always include `self` as the first argument when defining methods in a class, (ii) never include `self` when calling the methods from an instance, and (iii) always prefix attributes with `self` when they are used inside the methods.

To use the class, we first create one or more instances of the class, and then call the methods of interest:

```
mass_moon = 7.35E+22
mass_earth = 5.97E+24

# make instance of class Gravity:
gravity = Gravity(mass_moon, mass_earth)

r = 3.85E+8  # earth-moon distance in meters
Fg = gravity.force(r)   # call class method
```

Finally, we looked at special methods, which are special in the sense that they have very particular names, and they are called automatically when certain operations are performed on instances of the class. The most fundamental one is `__init__`, which is calledwhenever a new instance is created, but there are many more, for instance:

- `c = a + b` implies `c = a.__add__(b)`
- There are special methods for `a+b`, `a-b`, `a*b`, `a/b`, `a**b`, `-a`, `if a:`, `len(a)`, `str(a)` (pretty print), `repr(a)` (recreate `a` with `eval`), etc.
- With special methods we can create new mathematical objects like vectors, polynomials and complex numbers and write "mathematical code" (arithmetics)
- The `__call__` special method is particularly handy: `v = c(5)` means `v = c.__call__(5)`

- Functions with parameters should be represented by a class with the parameters as attributes and with a call special method for evaluating the function