classes

June 22, 2025

1 Classes

In programming, classes, or more general Object Oriented Programing, OOP, is a fundamental paradigm (next to others, mostly functional programming). It is quite powerful when it comes to encapsulation of the logic and states of objects. Some languages, for example Java, are heavily based on it. Python is a multi-paradigm language, meaning there are classes, functions etc.

However, classes play in Python an extremely central role - although it is not needed to explicitely know about it - since actually every object in Python "comes from a class". But we're going ahead of things.

Let's start with a problem.

1.1 Vectors

Let's imagine the following: We want to describe a 3D Vector and want to do some calculations with it and calculate its absolute length.

[1]:
```
# vector 'vector1'
vector1_x = -10
vector1_y = 20
vector1_z = 30


def calc_length_simple(x, y, z):
    return (x ** 2 + y ** 2 + z ** 2) ** 0.5
```

[2]:
```
calc_length_simple(vector1_x, vector1_y, vector1_z)
```

[2]:
```
37.416573867739416
```

1.1.1 First try: using a dict

Alright, but clearly cumbersom and not scalable at all to many vector properties. Better if we could stick it together into one container. Let's use a dict!

[3]:
```
vector1 = {'x': 10,
           'y': 20,
           'z': 30}
```

```
def calc_length(vector):
    """Calculate the absolute length of a 'vector dict'."""
    length_squared = vector['x'] ** 2 + vector['y'] ** 2 + vector['z'] ** 2
    return length_squared ** 0.5
```

[4]:
```
calc_length(vector=vector1)
```

[4]:
```
37.416573867739416
```

1.2 Encapsulation

That looks better! But now, `calc_length` critically depends on the structure of `vector1` if we e.g. want to create new vectors. How can we "communicate" that? (sure, docstrings, but is there a more "formal way"?)

Furthermore: `calc_length` somehow "belongs" to vector1, we want to calculate the absolute length of it. We always use `calc_length` together with a vector dict.

[5]:
```
# trying to attach the function to the "vector"
vector1 = {'x': 10,
           'y': 20,
           'z': 30,
           'calc_length': calc_length}
```

[6]:
```
vector1['calc_length'](vector1)
```

[6]:
```
37.416573867739416
```

1.3 Blueprint for the object

This was cumbersome, but better, we get there! For the communication of the exact layout, what we want is a "template"/blueprint to know how our dict should always look like. So that if we want to create a new vector, we have to make sure to specify x, y and z in the dict (so that it is valid). And then to also add the `calc_length` function.

[7]:
```
def make_vector(x, y, z):
    return {'x': x,
            'y': y,
            'z': z,
            'calc_length': calc_length}
```

[8]:
```
e1 = make_vector(20, 30, 20)
e1['calc_length'](e1)
```

[8]:
```
41.23105625617661
```

The next step will maybe look a bit (as it seems now) overcomplicated, but it will improve the understanding later on. Let's split the above even more (just one last time) into a

- constructor: this “makes” the empty vector with the fields and the methods.
- initializer: adds attributes to the instance, initializes it.

This is a fine difference and the former is in Python usually not needed (as it is implemented and invoked automatically), so in general, the word constructor may also be used for the initializer in Python. The latter serves mostly the same purpose as a constructor would in other languages.

[9]:
```
def make_vector():
    return {'calc_length': calc_length}


def initialize_vector(vector, x, y, z):
    vector['x'] = x
    vector['y'] = y
    vector['z'] = z
    return vector


vector1 = initialize_vector(make_vector(), x=20, y=30, z=20)
```

[10]:
```
# "magic line"
vector1['calc_length'](vector1)
```

[10]:
```
41.23105625617661
```

The call to calculate the mass is still not perfect: We want something that - “feeds itself to the function called”. - is created through a function (“constructor”) - has attributes (better then this [‘...’] accesing would be with the dot)

...more like this: `vector1.calc_length()`

1.4 Welcome to classes

A class is a blueprint of an object

The following code does basically what we did above but solves problems from above. It is a simple implementation of our first class called `SimpleVector`.

Note that the methods, that’s the name of a function “of a class”, are intended into the class body.

[11]:
```
class SimpleVector:
    # what we don't see: before the __init__, there is a (automatic)␣
 ↪make_vector (advanced concept) that we
    # basically never need to care about.

    # the initialiser, basically initialize_vector
    def __init__(self, x, y, z):  # self is the instance, the future object.
        self.x = x
        self.y = y
        self.z = z
```

```
    def calc_length(self):
        # we can just reuse the function from above
        return calc_length_simple(x=self.x, y=self.y, z=self.z)
```

[12]:
```
# where is __init__ called? (magic method again)
# answer: when calling the class
vector1 = SimpleVector(20, 30, z=40)  # not *exactly* equivalent to Vector.
 ↪__init__(), because
# it calls a constructor before (make_vector)
```

1.4.1 self: referencing itself

An object knows about itself by a simple trick. Before we needed to give the vector explicitly to the method in the dict as the first argument. Here we expect the same (we simply called the argument `self` instead of `vector`.

[13]:
```
vector1.calc_length()  # where did self go?
```

[13]:
```
53.85164807134504
```

Furthermore, we can now access attributes instead of using the item access operator `[...]`

[14]:
```
vector1.z
```

[14]:
```
40
```

[15]:
```
vector1.z
```

[15]:
```
40
```

That's what we want!

And we can create more vectors with different x and velocities.

[16]:
```
vector2 = SimpleVector(35, -30, 40)
```

[17]:
```
vector2
```

[17]:
```
<__main__.SimpleVector at 0x790c53e7fb20>
```

[18]:
```
vector2.x
```

[18]:
```
35
```