# 04-classes

November 18, 2022

## 1 Classes

Or “a short and hopefully effective introduction to object-based/oriented programming”.

You can think of classes as a more hierarchical way of organizing your data *and* code.

- A **class** defines the description of an **object**.
- An object of a given class is called an ***instance*** of the class.
- An object contains data (attributes) and the class specify functions that can make use of such data (methods).

### 1.1 Our first class

We will use last week’s example to show how to re-organise our code in classes.

[16]:

```
names = ["NGC 5128", "TXS 0506+056", "NGC 1068", "GB6 J1040+0617", "TXS␣
 ↪2226-184"]
distances = [3.7, 1.75e3, 14.4, 1.51e4, 107.1]  # Mpc
luminosities = [1e40, 3e46, 4.9e38, 6.2e45, 5.5e41] # erg/s
```

[17]:

```
class Source:
    default_distance_unit = 'Mpc'
    default_luminosity_unit = 'erg.s-1'

    def __init__(self, name, distance, luminosity):
        self.name = name
        self.distance = distance
        self.luminosity = luminosity
```

The init method can be referred to as a *constructor*. (Technically in python this is not a constructor, but practically - for our purpose - it is.)

[18]:

```
def load_sources():
    sources = []
    for n, d, l in zip(names, distances, luminosities):
        s = Source(n, d, l)
        sources.append(s)
    return sources
```

```python
sources = load_sources()
```

We have now built a new list. This list does not contain our “raw” information anymore.

[19]:

```python
s = sources[0]
```

[20]:

```python
print(s.name, s.distance, s.luminosity)

# You can also assign them as `s.name = ...`
```

```
NGC 5128 3.7 1e+40
```

We can access the attributes with the dot (.) operator.

Sometimes this is OK.

But the object-base paradigm relies in general on hiding the inner details of how a class works, and exposing to the user a so-called *interface*, i.e. a set of functions (methods) that allows the user to interact with the object.

Direct access to attributes is usually not permitted in object-oriented languages (C++, Java) unless explicitly declared.

[21]:

```python
class Source:
    default_distance_unit = 'Mpc'
    default_luminosity_unit = 'erg.s-1'

    def __init__(self, name, distance, luminosity):
        self.name = name
        self.distance = distance
        self.luminosity = luminosity
        self.detected = False

    # ====================

    def get_name(self):
        return self.name

    def get_distance(self, unit=Source.default_distance_unit):
        if unit == Source.default_distance_unit:
            return self.distance
        elif unit == 'ly':
            return self.distance * 3.26156
        else:
            return None

    def get_luminosity(self):
        return self.luminosity

    # ====================
    def is_detected(self):
        return self.detected

    # ====================

    def set_detected(self, detected):
        self.is_detected = detected

    """
    def set_name(self, name):
        self.name = name

    def set_distance(self, distance):
        self.distance = distance

    def set_luminosity(self, luminosity):
        self.luminosity = luminosity
    """
```

We have defined the so-called *setter methods* and *getter methods*! - Getter methods are useful because allow you to establish a layer of *abstraction* between the inner representation of the class data and the way this information is accessed! - Do you always need a getter method for all attributes? Not necessarily, but it can be a choice that pays off as your code grows more complex. Remember: methods (functions) are much more easily documented than individual attributes! - Setter methods may be useful… or not. Some attributes may need to be modified *after* the object creations, other should be better - There is no way to ensure the immutability of an attribute, but setter methods are a good way to let the user know what should be and what should not be touched!

[22]:
```
sources = load_sources()

s = sources[0]

print(f"{s.name} is {s.get_distance()} Mpc or {s.get_distance(unit='ly')} light␣
  ↪years away")
```

```
NGC 5128 is 3.7 Mpc or 12.067772 light years away
```

### 1.1.1 Class method and static methods!

If a method does not use the instance's attributes but only class attributes, it's better defined as a *class method*.

If a method does not use any attribute, it's better defined as *static method*.

[23]:
```
import math

class Source:
    default_distance_unit = 'Mpc'
    default_luminosity_unit = 'erg.s-1'

    @staticmethod
    def luminosity_to_flux(luminosity, distance):
        """ convert luminosity to flux """
        return luminosity * 4 * math.pi * distance**2

    @classmethod
    def convert_distance(cls, distance, to_unit):
        """ convert a distance from the default unit of the class to another tabulated unit """
        conversion_factors = { cls.default_distance_unit : 1.0, 'ly' : 3.26156 }
        return distance * conversion_factors[to_unit]



    def __init__(self, name, distance, luminosity):
        self.name = name
        self.distance = distance
        self.luminosity = luminosity
        self.detected = False

    # ====================

    def get_name(self):
        return self.name

    def get_distance(self, unit=Source.default_distance_unit):
        return self.convert_distance(self.distance, unit)

    def get_luminosity(self):
        return self.luminosity

    # ====================

    def is_detected(self):
        return self.detected

    # ====================

    def set_detected(self, detected):
        self.is_detected = detected
```

[24]:

```
sources = load_sources()

s = sources[0]
```

```python
s.get_distance('ly')
```

[24]:

```
12.067772
```

## 1.2 Boring technical details

Important: both **self** and **cls** are conventional but arbitrary names. - Ordinary methods are always passed the object itself as first (implicit) argument - Class methods are always passed the class itself as first (implicit) argument

[25]:

```python
class DummyClass:
    def __init__(self):
        pass

    @staticmethod
    def dummy_static_method(*args):
        print(args)

    @classmethod
    def dummy_class_method(*args):
        print(args)

    def dummy_method(*args):
        print(args)

dummy = DummyClass()

dummy.dummy_static_method("foo")
dummy.dummy_class_method("foo")
dummy.dummy_method("foo")
```

```
('foo',)
(<class '__main__.DummyClass'>, 'foo')
(<__main__.DummyClass object at 0x7f36b0382b60>, 'foo')
```

## 1.3 Composition and inheritance

- Classes can be extended by other classes (inheritance), or contain objects of other classes (composition).
- Sometimes is not clear which one to use: think in terms if “is a” (inheritance) vs “has a” (composition)!
- When in doubt, choose composition over inheritance!

[26]:

```python
class Galaxy(Source):
    def __init__(self, name, distance, luminosity, galaxy_type):
        super().__init__(name, distance, luminosity)
        self.galaxy_type = galaxy_type

class Supernova(Source):
    def __init__(self, name, distance, luminosity, duration, host_galaxy):
        super().__init__(name, distance, luminosity)
        self.duration = duration
        self.host_galaxy = host_galaxy
```

[27]:

```
LMC = Galaxy("LMC", 0.05, None, "satellite")

SN = Supernova("SN1987A", 0.05, 1e42, duration=150, host_galaxy=LMC)
```

## 1.4 More about methods

### 1.4.1 Factory methods

- Sometimes, a single constructor is not enough. You may want to create an object from different type of inputs.
- Unfortunately, python does not allow to specify multiple versions of a method with different arguments (overloading).
- The common paradigm is to use a single `__init__` and define *factory* methods as class methods, that act as interfaces to the constructor.

### 1.4.2 Special methods

- `__repr__` and `__str__` controls how the object will appear when inspected and printed!
- `__call__` allows the object to be used as a function!
- `__eq__` will define how the `==` operator works between two objects of the same class (also possible for all comparison operators, as well as arithmetics (`__add__`)…

See https://docs.python.org/3/reference/datamodel.html for cool stuff!

[28]:

```
class Source:
    default_distance_unit = 'Mpc'
    default_luminosity_unit = 'erg.s-1'

    @staticmethod
    def luminosity_to_flux(luminosity, distance):
        """ convert luminosity to flux """
        return luminosity * 4 * math.pi * distance**2

    @classmethod
    def convert_distance(cls, distance, to_unit):
        """ convert a distance from the default unit of the class to another␣
 ↪tabulated unit """
        conversion_factors = { cls.default_distance_unit : 1.0, 'ly' : 3.26156 }
        return distance * conversion_factors[to_unit]
    @classmethod
    def from_dict(cls, d):
        return cls(d['name'], d['distance'], d['luminosity'])

    def __init__(self, name, distance, luminosity):
        self.name = name
        self.distance = distance
        self.luminosity = luminosity
        self.detected = False

    # ====================

    def get_name(self):
        return self.name

    def get_distance(self, unit=Source.default_distance_unit):
        return self.convert_distance(self.distance, unit)

    def get_luminosity(self):
        return self.luminosity

    # ====================

    def is_detected(self):
        return self.detected

    # ====================

    def set_detected(self, detected):
        self.is_detected = detected

    def __repr__(self):
        return 'Source {0} @ {1} {2}'.format(self.name, self.distance, self.
↪default_distance_unit)

    def __str__(self):
        return "This is a source object with name {0}, distance {1}, luminosity␣
↪{2}".format(self.name, self.distance, self.luminosity)
```

[29]:

```
d = { 'name': 'SN1987A', 'distance' :  0.05, 'luminosity': 1e42}

SN = Source.from_dict(d)

print(SN)
```

```
This is a source object with name SN1987A, distance 0.05, luminosity 1e+42
```

[30]:

```
SN
```

[30]:

```
Source SN1987A @ 0.05 Mpc
```