Python 12

Elias Wachmann

2024

Content

What is object oriented programming?

Intro to object oriented programming

How do we classify things in the real world?

We group them by their properties and their behaviour.

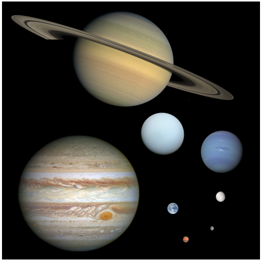
An **object** like a planet is can be described by its properties like mass, radius, position, velocity, etc.

We can also describe its behaviour like how it moves around the sun.

What are classes?

Classes are a way to abstract real world things into categories.

Not all planets are the same, but they can be described by the same properties and behaviour.
We can use a **class** that describes the properties and behaviour of a planet (an **object**).



https://upload.wikimedia.org/wikipedia/commons/c/cf/Planet_collage_to_scale.jpg

Describe objects within a class

How can we distinguish between different planets now?

Earth: $m = 5.97 \times 10^{24}$ kg, $r = 6371$ km, ...

Mars: $m = 6.41 \times 10^{23}$ kg, $r = 3389.5$ km, ...

Jupiter: $m = 1.90 \times 10^{27}$ kg, $r = 69911$ km, ...

Python classes

Let's define a Planet class

In python we can define a class like this:

```
class Planet():
    def __init__(self, name_in, radius_in, mass_in):
        self.name = name_in
        self.radius = radius_in
        self.mass = mass_in
```

The **class**-keyword tells python that we want to define a class.

What is the __init__ function?

The **__init__** function is called when we create a new object of the class.

Objects are therefore concrete **instances** of a class.

In our example the Planet class takes the arguments **mass** and **radius** and stores them as **attributes** of the object.

How can we access these attributes and what is the **self**-keyword?

Pythons self keyword

The **self** keyword is a reference to the object itself.

```
    def __init__(self, name_in, radius_in, mass_in):
        self.name = name_in
        self.radius = radius_in
        self.mass = mass_in
```

This means: If you want to create an attribute of a class, you simply write `self.variable_name` **self** also

has to be passed in as the first argument to every function defined inside the class!

Methods – class functions

Methods are functions inside a class. They can only be accessed by objects of that class and **self** is their first argument.

```
class Planet():
    def __init__(self, name_in, radius_in, mass_in):
        self.name = name_in
        self.radius = radius_in
        self.mass = mass_in

    def getData(self):
        return {"name": self.name, "radius": self.radius, "mass": self.mass}
```

Let’s create a Planet

Finally let’s create a planet object:

```
earth = Planet("Earth", 6371, 5.972e24)
mars = Planet("Mars", 3389, 6.39e23)
```

Just pass in the initial arguments just like you would for functions.

```
earth_data = earth.getData()
print(earth_data["name"])  # Earth
```

We can now call our defined class method on objects.

Modify objects

Certainly a method can modify attributes of an object like in the following example the name.

```
    def changeName(self, newname):
        self.name = newname
```

```
earth.changeName("Earth2.0")
earth_data = earth.getData()
print(earth_data["name"])  # Earth2.0
```

This only changes the name of the ‚earth‘-object while the ‚mars‘-object is not affected.

Dunder-methods

Dunder-methods

Dunder-methods are special methods that are called by python in certain situations.

We have seen the **__init__** method, which is called when a new object is created, before.
Further examples include:

- **__str__** method is called when we try to convert an object to a string.
- **__add__** method is called for + operator.
- **__lt__, __le__, __eq__, __ne__, __gt__, __ge__** methods used for comparisons.

Dunder-methods – Examples

One can now implement dunder methods for our custom classes to make certain functionality (like comparisons between objects) possible.

```python
    def __ge__(self, other):
        if self.radius >= other.radius and self.mass >= other.mass:
            return True
        return False

    def __lt__(self, other):
        return False if self.__ge__(other) else True
```

Dunder-methods – Iterators

__iter__ and **__next__** are special methods that allow us to iterate over objects.

While __**iter**__ returns an iterator object, __**next**__ returns the next element of the iterator.

This allows us to specify what a while/for loop should do when it encounters an object of our class.

Dunder-methods – Iterators – Example

Let’s define a new class:

```
class Animal:
    def __init__(self, name, type, age):
        self.name = name
        self.type = type
        self.age = age

    def __str__(self) -> str:
        return f"{self.name} is {self.age}
    old and a {self.type}!"
```

...

Dunder-methods – Iterators – Example

```
class Animallist:
    def __init__(self):
        self.animals = []
        self.index = 0

    def __add__(self, animal: Animal):
        self.animals.append(animal) if animal not in self.animals else None
```

...

Dunder-methods – Iterators – Example

```
    def __next__(self):
        while self.index < len(self.animals):
            self.index += 1
            return self.animals[self.index-1].__str__()
        raise StopIteration

joe = Animal("Joe", "Dog", 5)
bob = Animal("Bob", "Cat", 3)
my_animals = Animallist()
my_animals + joe #  add animals to list
my_animals + bob
for animal in my_animals:
    print(animal)
```

Inheritance

Inheritance

Inheritance is a way to create a new class from an existing class.

The new class is called **child class** and the existing class is called **parent class**.

The child class inherits all attributes and methods from the parent class and may expand on them or overwrite them.

Inheritance

We can create a child class by passing the parent class as an argument to the child class definition:

```
class Dog(Animal):
```

Here the **child class** Dog is instantiated with the **parent class** Animal as an argument.

```
class Animal():
    def __init__(self, name, type, age):
        self.name = name
        self.type = type
        self.age = age
```

Animal implements a **__str__** method like seen before.

Inheritance – super method

The **super** method allows us to call methods of the parent class.

This way we can call **__init__** in the parent class from the child class:

```
class Cat(Animal):
    def __init__(self, name, age):
```

Inheritance – Example

```
class Cat(Animal):
    def __init__(self, name, age):
        super().__init__(name, "Cat", age)

    def make_sound(self):
        print("Meow!")

class Dog(Animal):
    def __init__(self, name, age):
        super().__init__(name, "Dog", age)

    def make_sound(self):
        print("Woof!")
```

...

Inheritance – Example

```
dogo = Dog("Dogo", 5)
cat = Cat("Cat", 3)
dogo.make_sound()  # Woof!
cat.make_sound()  # Meow!
```

Now the method **make_sound** is implemented in both children classes.

Cats make a **meow** sound while dogs make a **woof** sound.

Inheritance – Overwriting methods

Ok let's add another child class, a frog:

```
class Frog(Animal):
    def __init__(self, name, age):
        super().__init__(name, "Frog", age)
```

We should also add a **make_sound** method to the parent class:

```
    def make_sound(self):
        print("I am an animal!")
```

Inheritance – Overwriting methods

If we now instantiate a frog and call **make_sound** we get:

```
my_frog = Frog("Frog", 1)
my_frog.make_sound()  # I am an animal!
```

Because there is no **make_sound** method in the frog class, the method from the parent class is used.

The Cat and Dog class **overwrite** the method from the parent class and use their own implementation.

Wrap up + What’s next?

Wrap up

By now you should have a good understanding of the basics of python programming for scientific computing.

You have learned how to use python to:

- import data
- modify and analyse the data
- plot/visualize the data
- write your own functions
- define classes and objects . . .

What's next?

There are many more topics to cover in (python) programming . . .

Next up: Computational Physics in 5th semester!

- numerical methods for solving differential equations
- methods for solving linear systems
- solving partial differential equations
- efficient eigenvalues of matrices / compression

What’s next?

Other courses you might be interested in:

- Einführung in die strukturierte Programmierung (C) [INB.04000UF, INB.05000UF]
- Objektorientierte Programmierung 1 (C++) [INF.07002UF, INF.08008UF]
- Data Management (SQL) [INF.01017UF, INF.02018UF]
- Datenstrukturen und Algorithmen 1 [INF.04032UF, INF.03031UF]
- Software Engineering in Physics [PHT.528UF]
- ...