Fachhochschule Kiel
University of Applied Sciences

Tools and Programming for Data Science
Object-oriented programming

Study Program Data Science
Prof. Dr. Tillmann Schwörer

Our course agenda

- **Introduction and overview**
- **NumPy**: Basic data handling with Numpy arrays
- **Pandas**
  - Exploratory data analysis
  - Data consolidation
  - Data cleaning
- **Data visualization using Matplotlib and Seaborn**
- **Interacting with APIs**
- **Interacting with SQL databases**
- **Version Control with Git and GitHub**
- **Advanced Python**

Python foundations

Data types

Operators

Functions

Control flow and iterators

Programming concepts & paradigms

See also Precourse Programming

Tooling

Installation

Visual Studio Code

Jupyter Notebooks

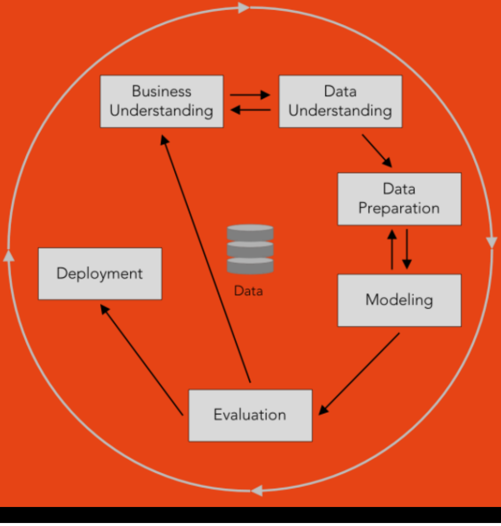
Packages

Virtual Environments

Git and Github

Python

Data Science Workflow



NumPy

pandas

matplotlib

Procedural Programming

Source: Dall-E

Programming Paradigm used so far: Procedural Programming

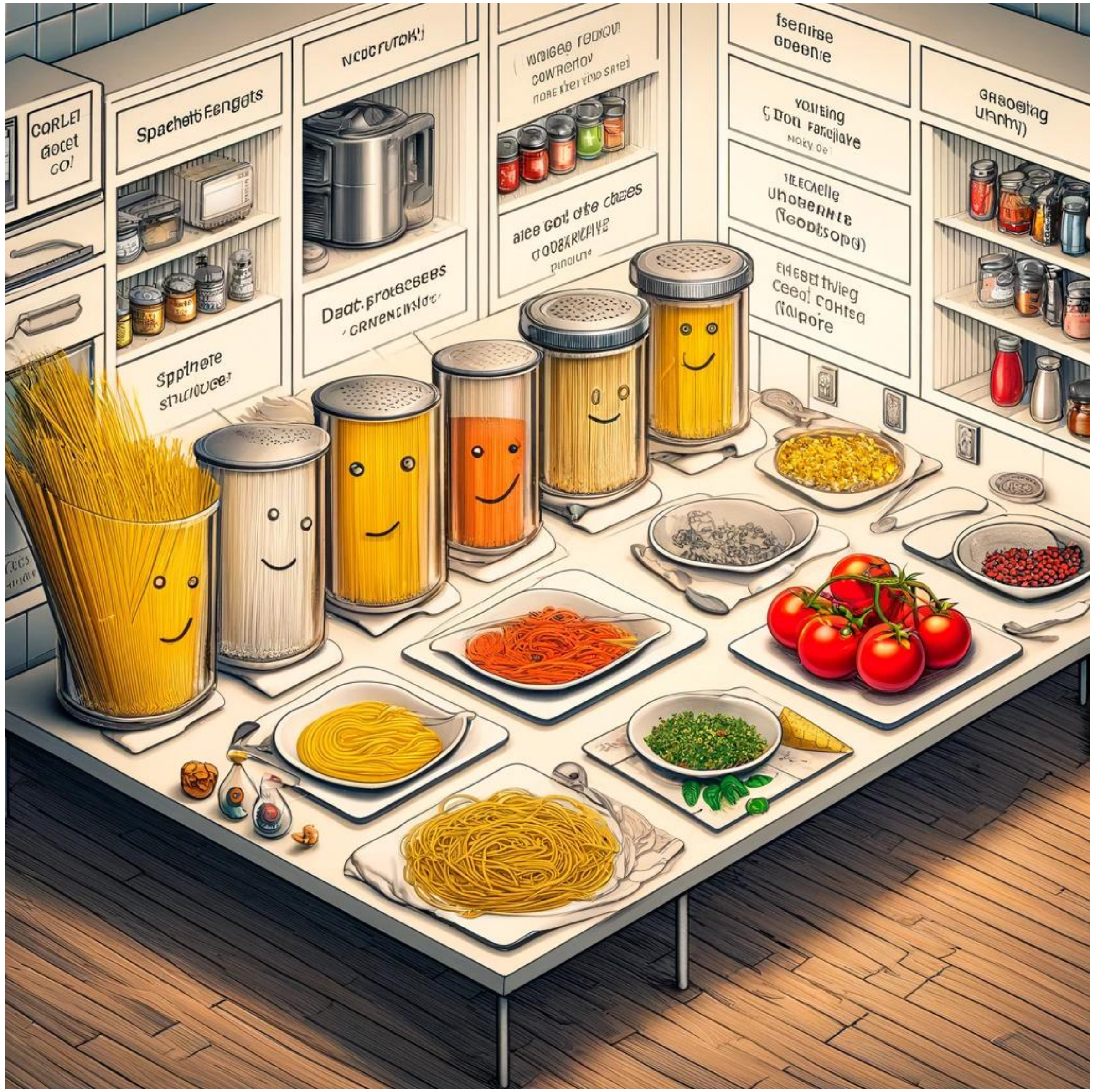
- Sequential processing: reading, cleaning, exploring, and visualizing data
- Great for
  - EDA
  - learning data science techniques
  - smaller projects
- As complexity grows, the code may become hard to understand, extend and reuse → **“Spaghetti Code”**

Why consider OOP for data science?

Source: Dall-E

Object Oriented Programming (OOP)

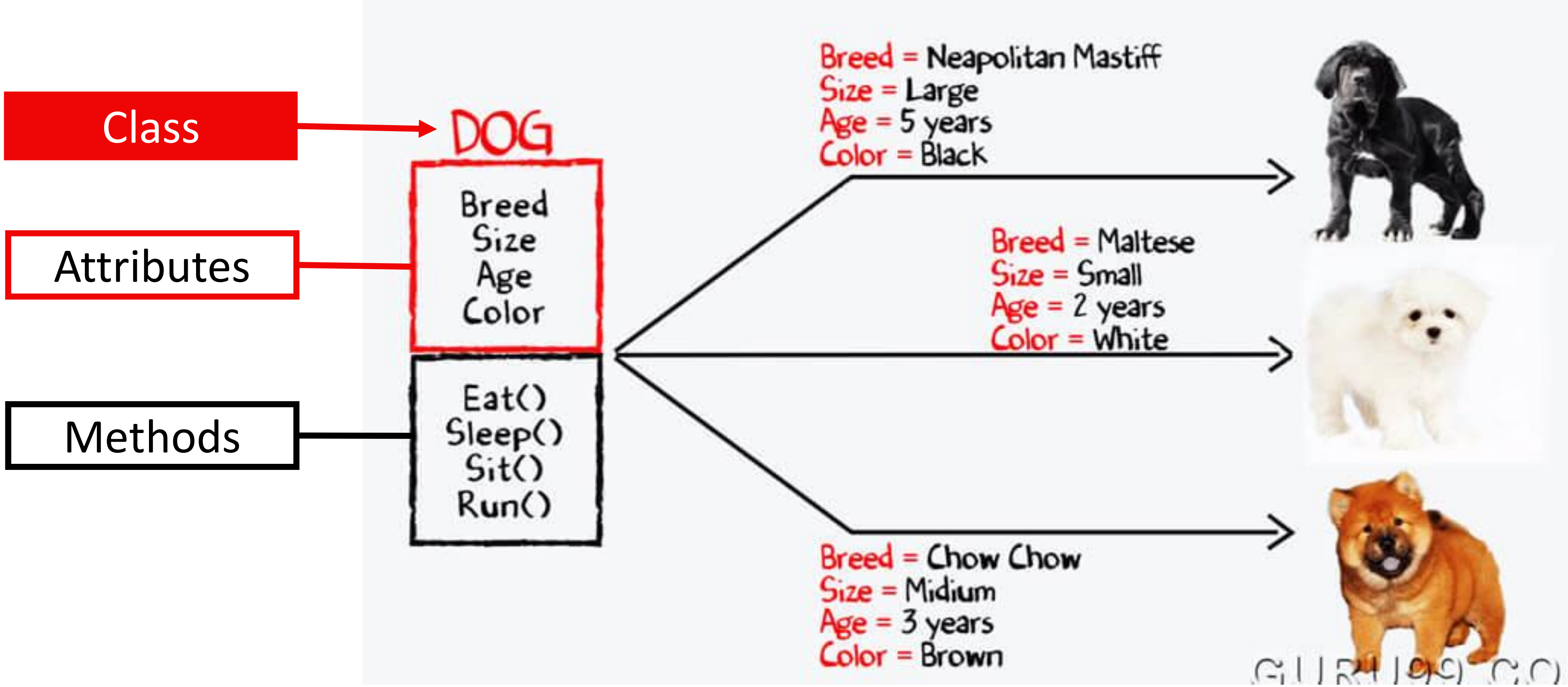
- Code as interaction of different objects
- While more abstract and harder to set up, it may make our code better understandable, extendable, and reusable
- Great for
  - complex projects
  - software development
  - e.g. developing a Python data science package



Overview: Object-Oriented Programming

- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects“
- **Objects** are characterized jointly by
  - data in the form of attributes
  - behaviour in the form of methods
- A **class** is a blueprint for creating objects
  - Defines what sort of data and behaviors are typical for all instances (i.e. all objects of a given class)
- Features of OOP:
  - **Encapsulation**: bundling of attributes and methods
  - **Inheritance:** a class can inherit attributes and methods from another class
  - **Polymorphism:** flexibility of what methods do depending on the type of object

Example 1



Class

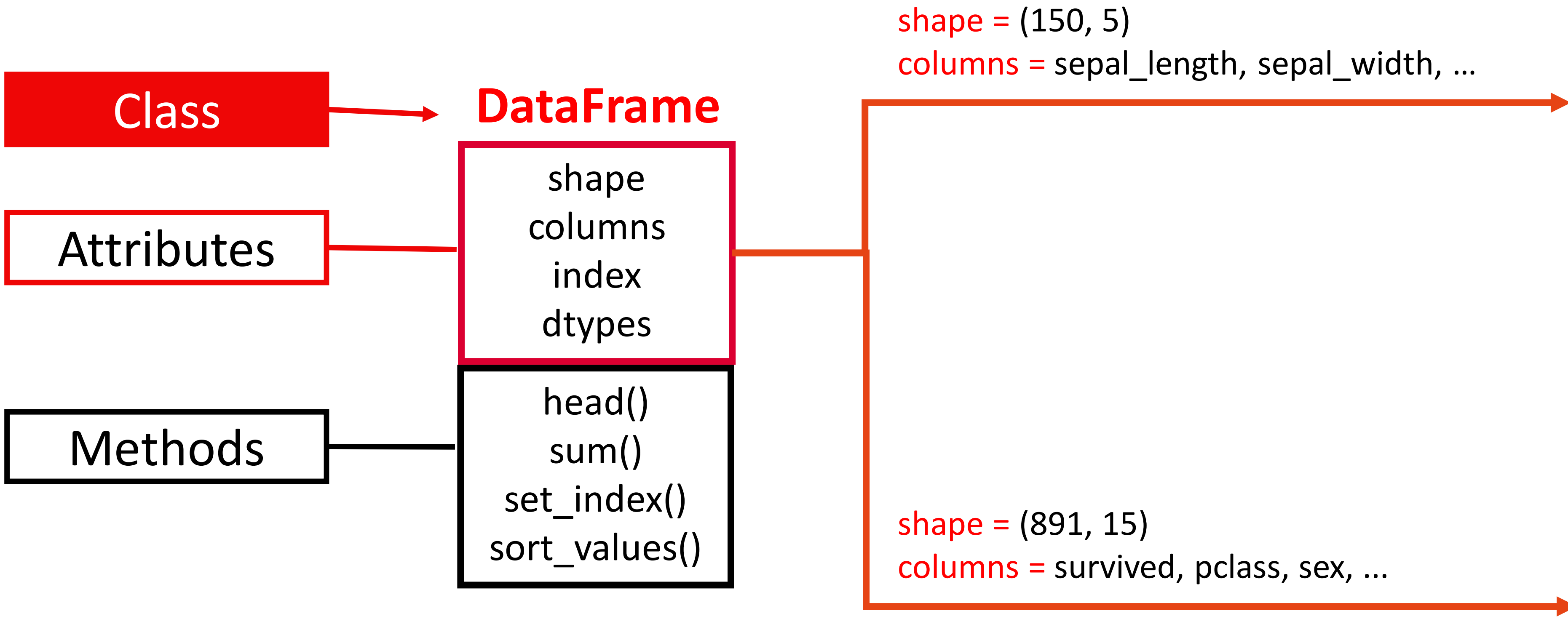
Attributes

Methods

3 Objects =
3 instances of the
Dog class

Source: https://www.guru99.com/difference-between-object-and-class.html

Example 2



Class

DataFrame

Attributes

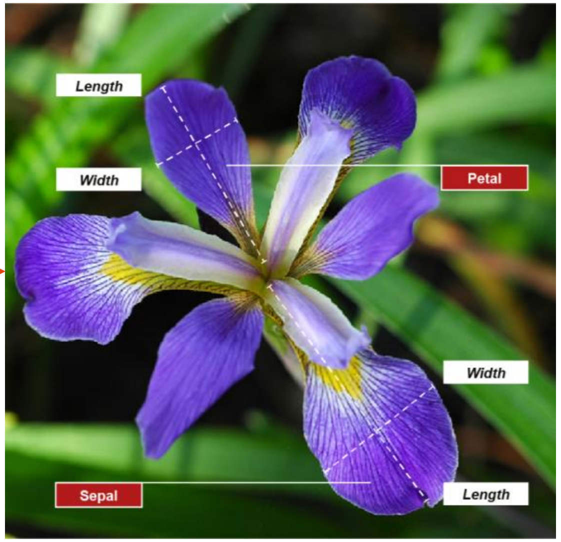
shape
columns
index
dtypes

Methods

head()
sum()
set_index()
sort_values()

shape = (150, 5)
columns = sepal_length, sepal_width, ...

shape = (891, 15)
columns = survived, pclass, sex, ...



Source: https://www.guru99.com/difference-between-object-and-class.html

Important takeaway

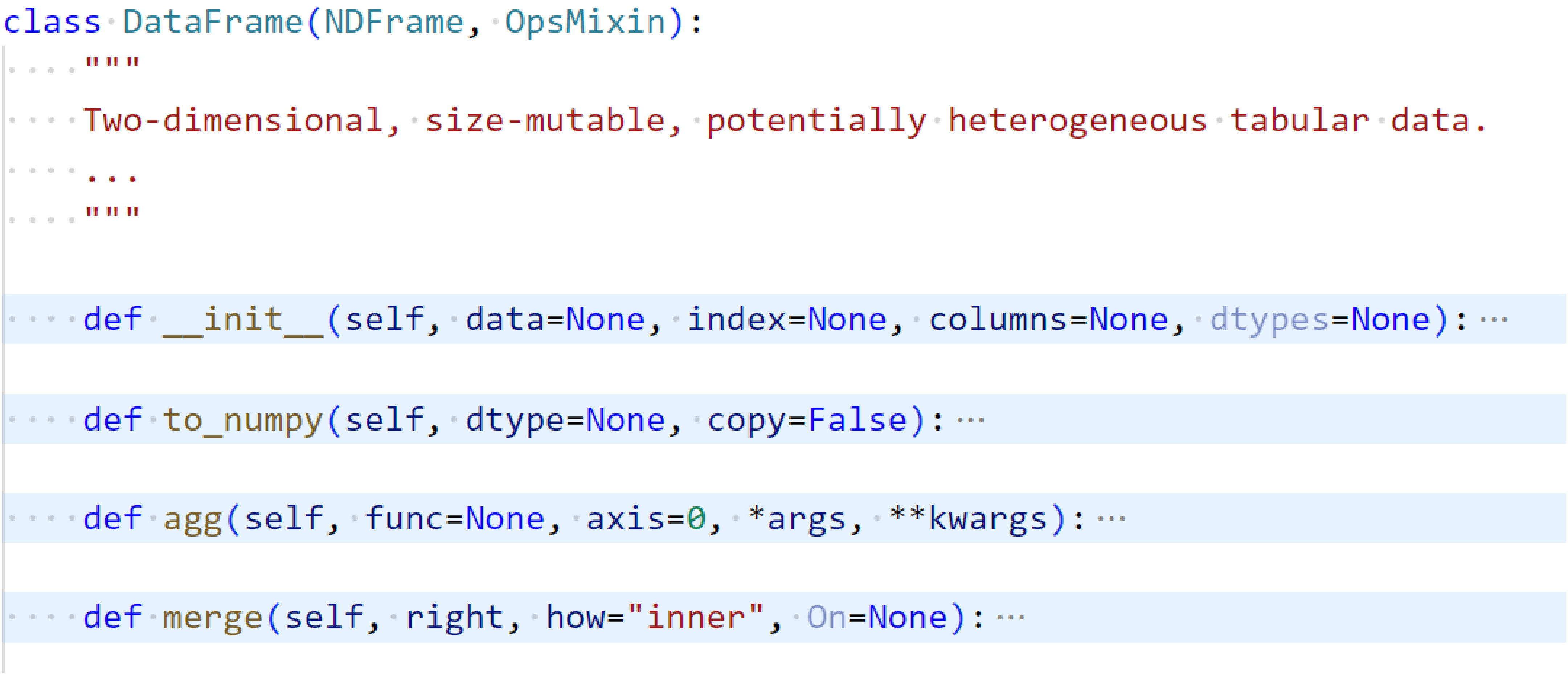
We

- have been working with objects (features of OOP) all the time
- have not asked ourselves how the blueprints for these objects were defined
- do not know yet how to define classes ourselves

DataFrame Class

Inheritance

The DataFrame class inherits certain definitions of attributes and methods from the NDFrame and the OpsMixin class

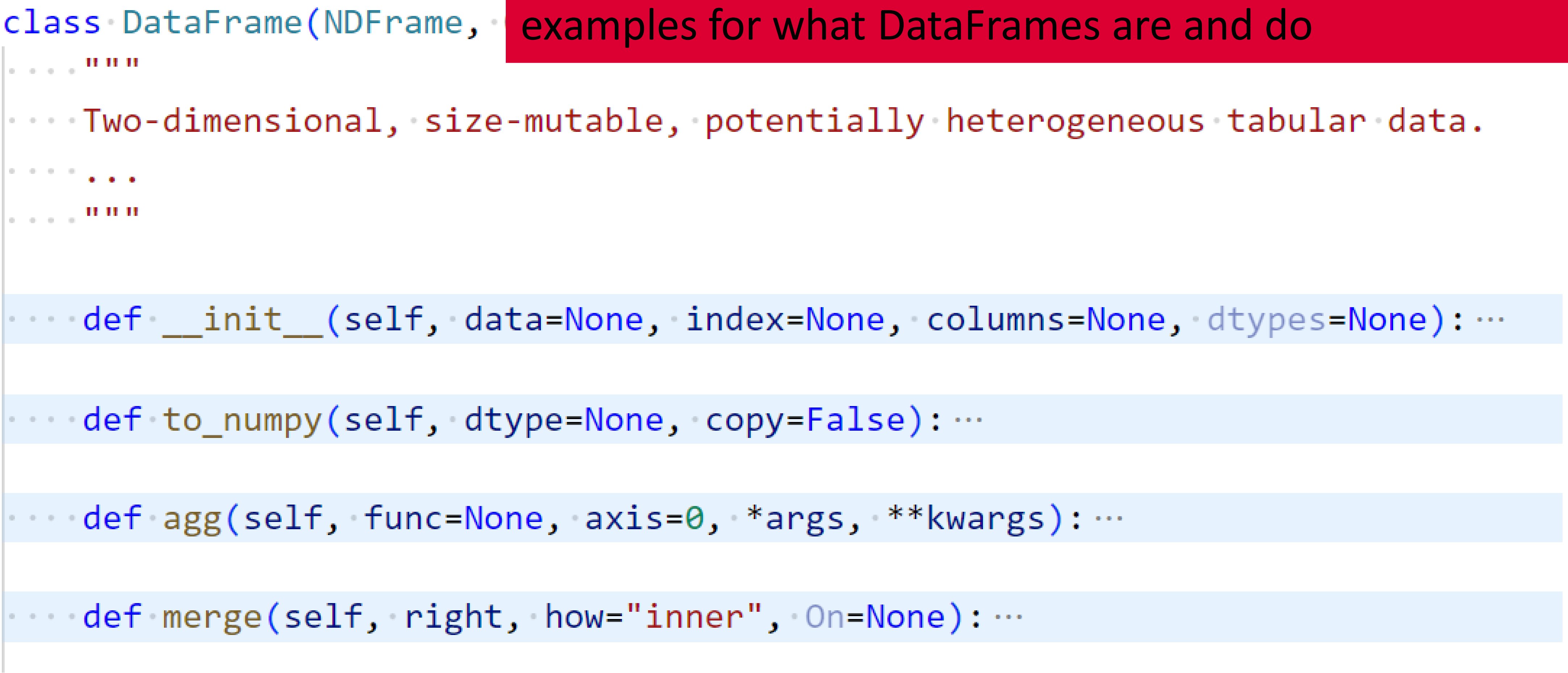


```
class DataFrame(NDFrame, OpsMixin):
    """
    Two-dimensional, size-mutable, potentially heterogeneous tabular data.
    ...
    """

    def __init__(self, data=None, index=None, columns=None, dtypes=None): ···

    def to_numpy(self, dtype=None, copy=False): ···

    def agg(self, func=None, axis=0, *args, **kwargs): ···

    def merge(self, right, how="inner", On=None): ···
```

DataFrame Class

Docstrings

We can use “””docstrings””” to precisely describe and give examples for what DataFrames are and do

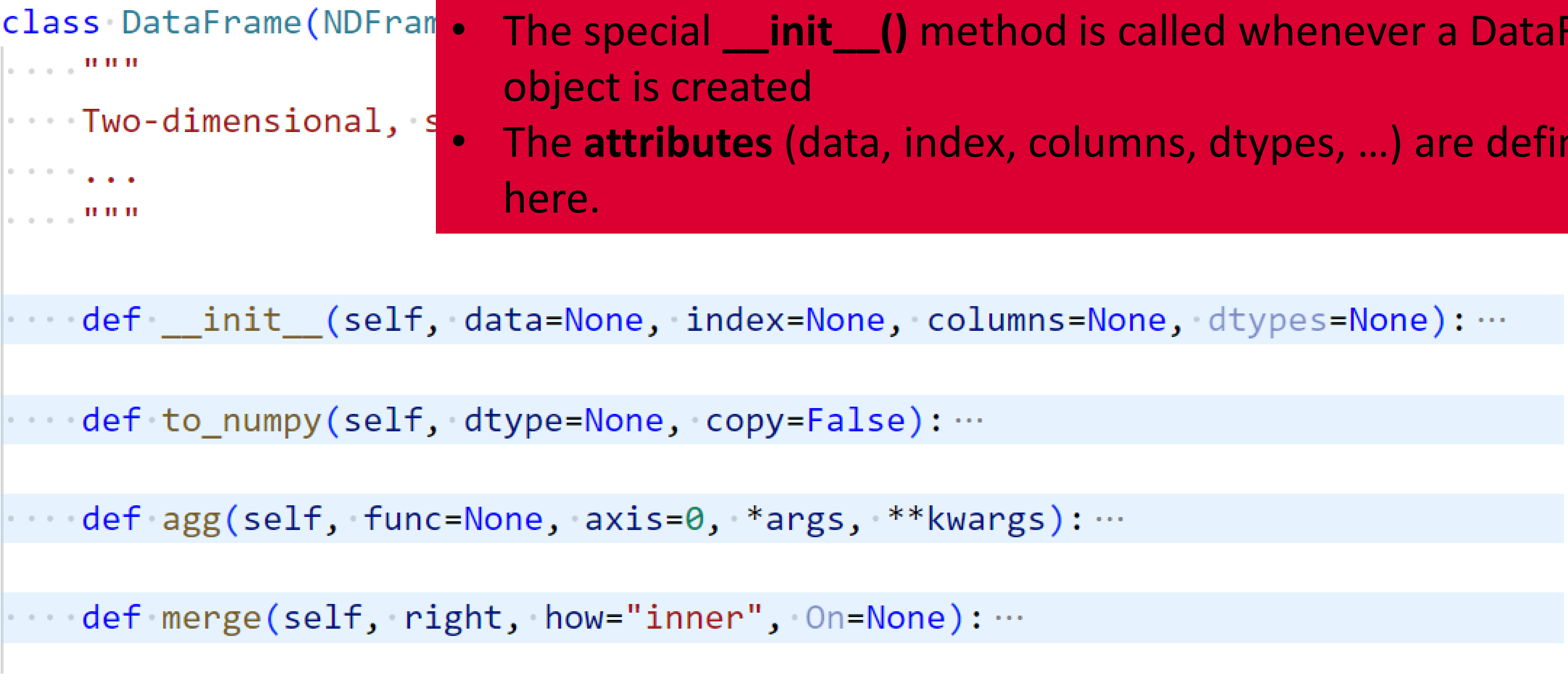


```
class DataFrame(NDFrame, 
    """
    Two-dimensional, size-mutable, potentially heterogeneous tabular data.
    ...
    """

    def __init__(self, data=None, index=None, columns=None, dtypes=None): ...

    def to_numpy(self, dtype=None, copy=False): ...

    def agg(self, func=None, axis=0, *args, **kwargs): ...

    def merge(self, right, how="inner", On=None): ...
```

DataFrame Class

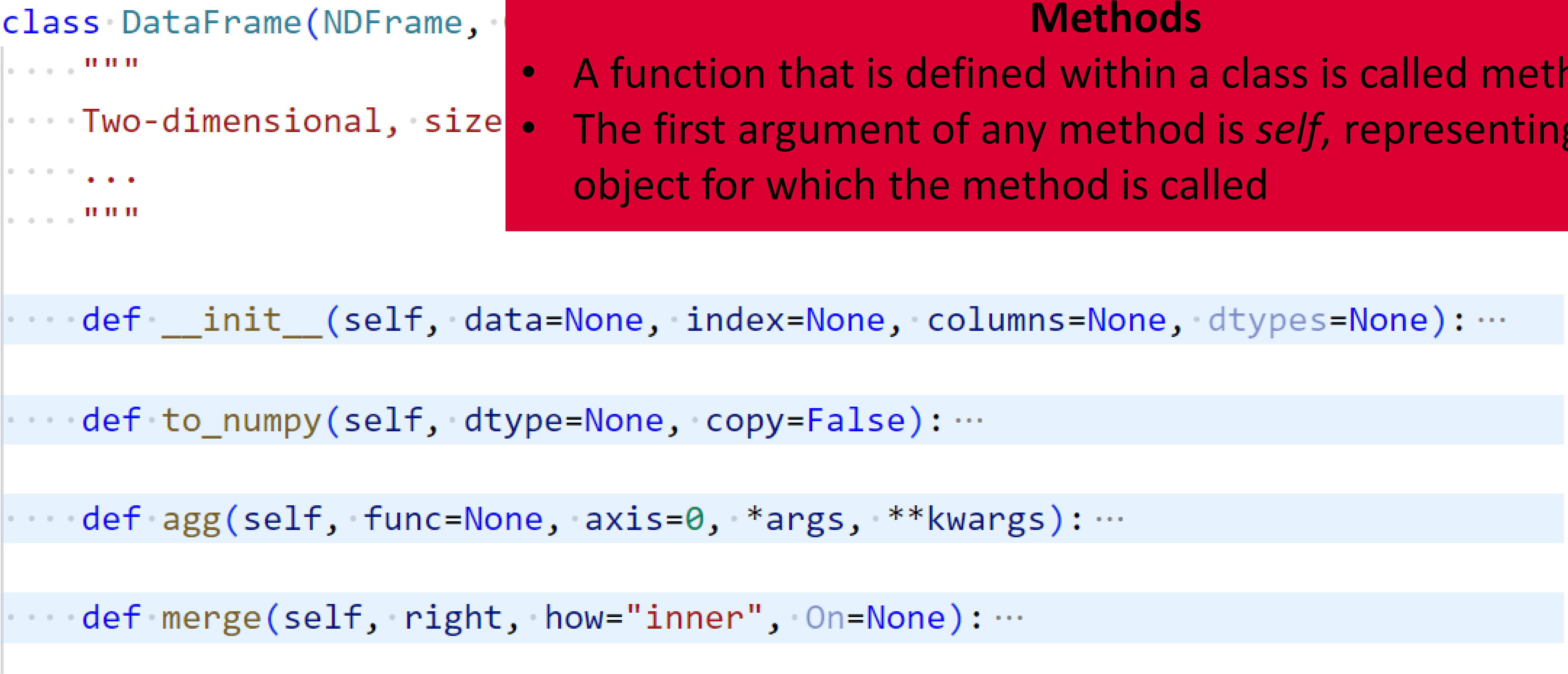
Constructor

- The special **__init__()** method is called whenever a DataFrame object is created
- The **attributes** (data, index, columns, dtypes, …) are defined here.



```
class DataFrame(NDFram
    """
    Two-dimensional, s
    ...
    """

    def __init__(self, data=None, index=None, columns=None, dtypes=None): ⋯

    def to_numpy(self, dtype=None, copy=False): ⋯

    def agg(self, func=None, axis=0, *args, **kwargs): ⋯

    def merge(self, right, how="inner", On=None): ⋯
```

DataFrame Class

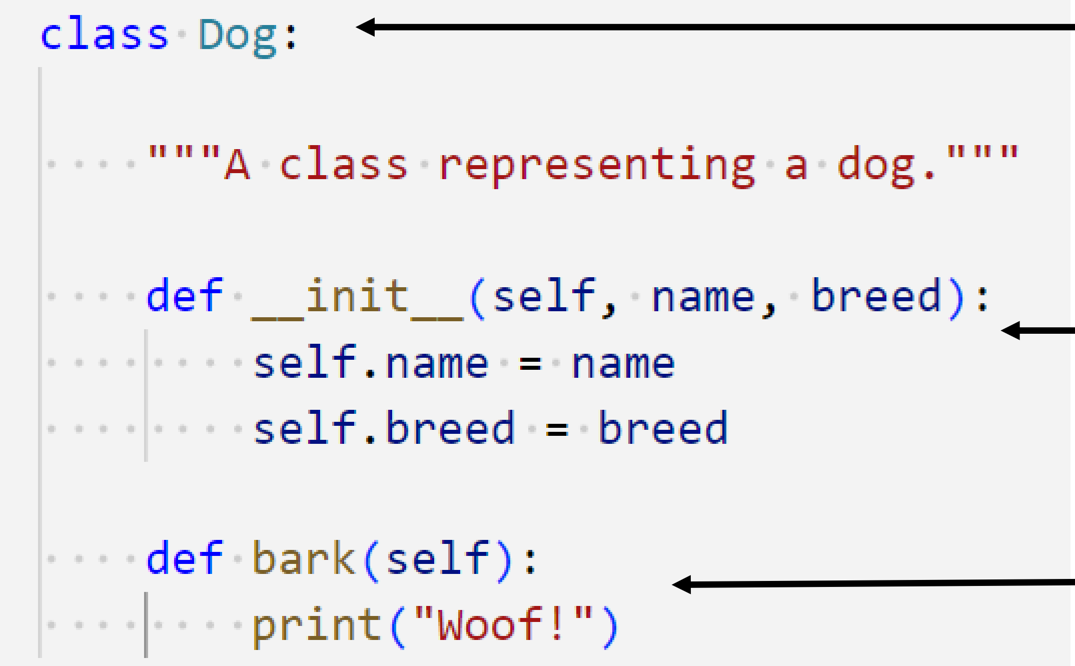


```
class DataFrame(NDFrame, 
    """
    Two-dimensional, size
    ...
    """

    def __init__(self, data=None, index=None, columns=None, dtypes=None): ...

    def to_numpy(self, dtype=None, copy=False): ...

    def agg(self, func=None, axis=0, *args, **kwargs): ...

    def merge(self, right, how="inner", On=None): ...
```

Methods

- A function that is defined within a class is called method
- The first argument of any method is *self*, representing the object for which the method is called

A simple Dog class

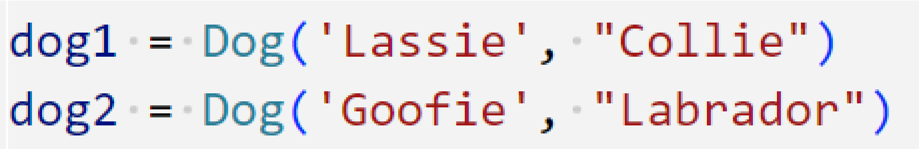


```python
class Dog:

    """A class representing a dog."""

    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

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

CamelCase for class names

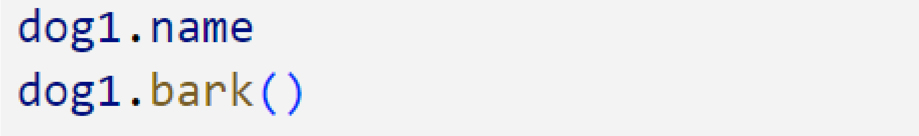
Constructor: __*init*__ method defines that any instance of the Dog class must have a name and a breed **attribute**

Method: the only behaviour defined for the Dog class is how they bark



```python
dog1 = Dog('Lassie', "Collie")
dog2 = Dog('Goofie', "Labrador")
```

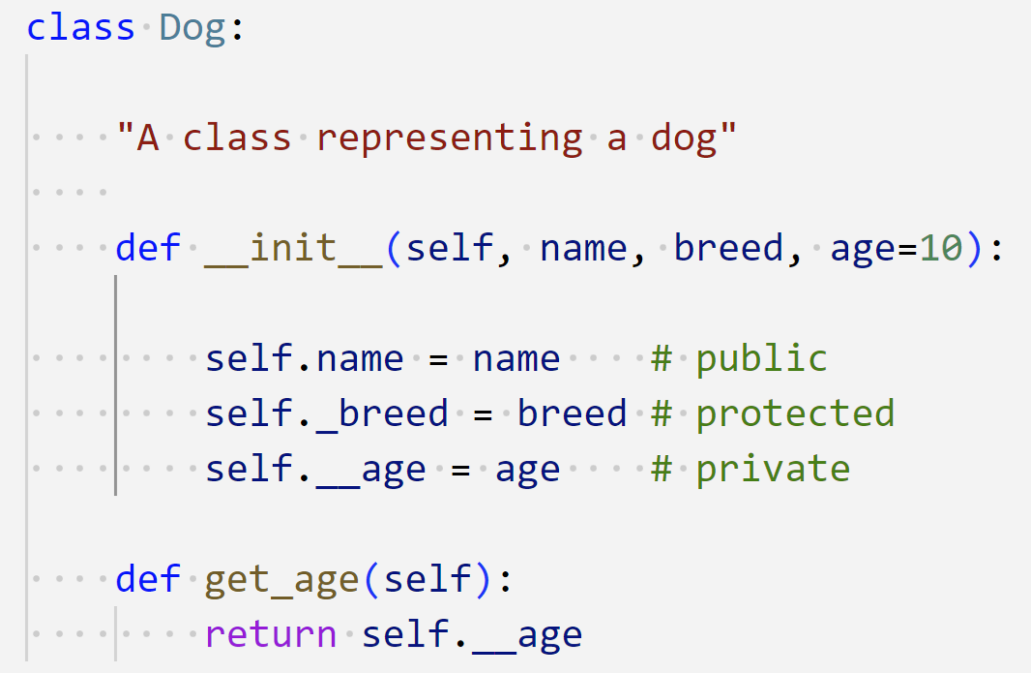
Objects: two instances of the dog class are created



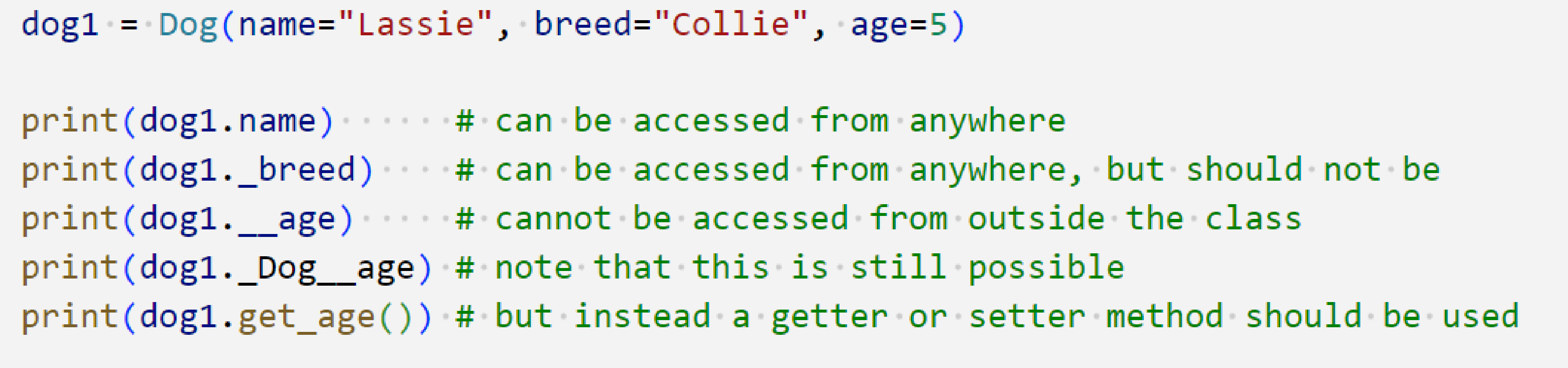
```python
dog1.name
dog1.bark()
```

Attributes and methods: we can get (or set) the attributes of dogs, and perform barking behaviour

Public, protected and private attributes

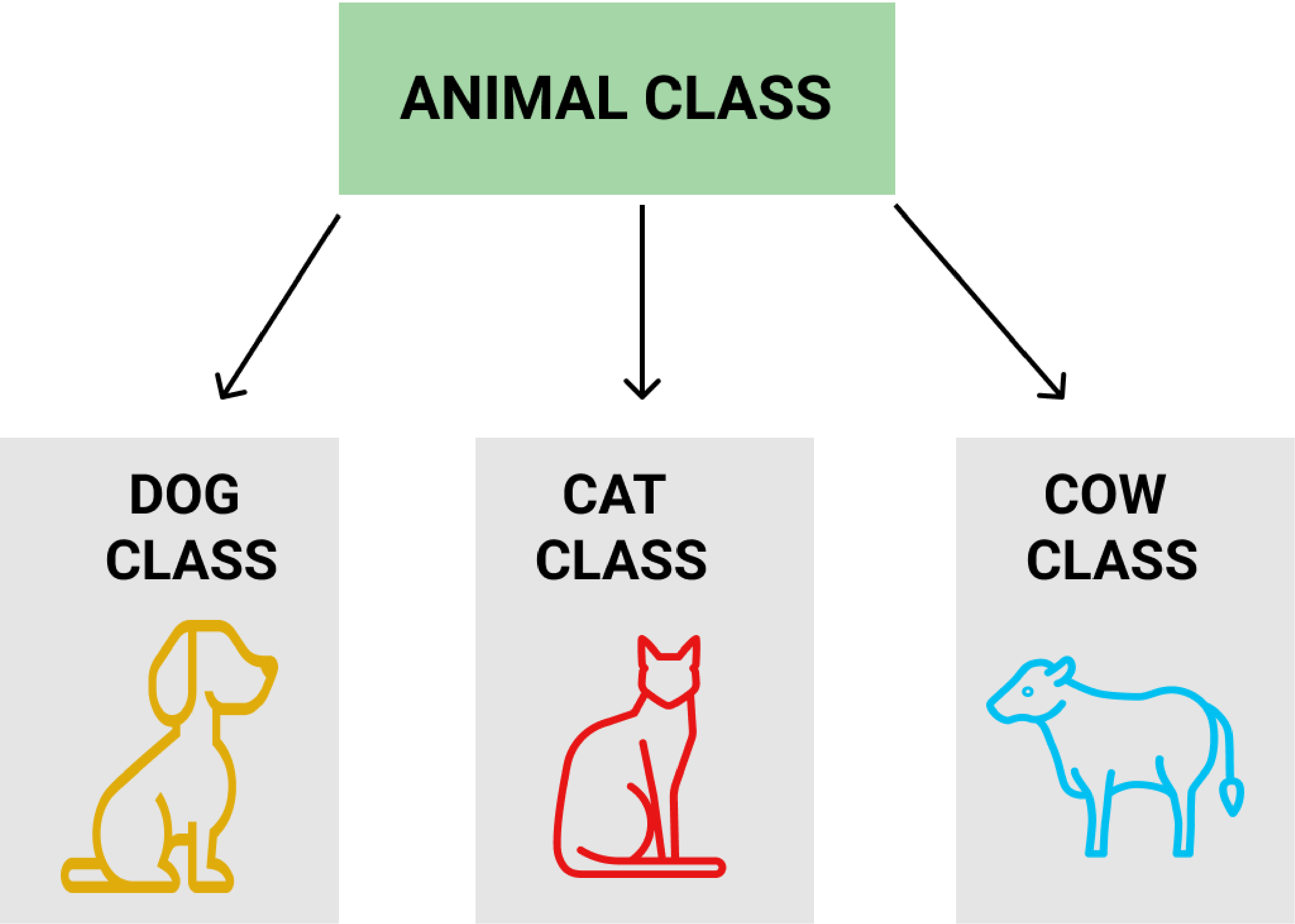


```python
class Dog:

    "A class representing a dog"

    def __init__(self, name, breed, age=10):
        self.name = name    # public
        self._breed = breed # protected
        self.__age = age    # private

    def get_age(self):
        return self.__age
```



```python
dog1 = Dog(name="Lassie", breed="Collie", age=5)

print(dog1.name)      # can be accessed from anywhere
print(dog1._breed)    # can be accessed from anywhere, but should not be
print(dog1.__age)     # cannot be accessed from outside the class
print(dog1._Dog__age) # note that this is still possible
print(dog1.get_age()) # but instead a getter or setter method should be used
```

Inheritance



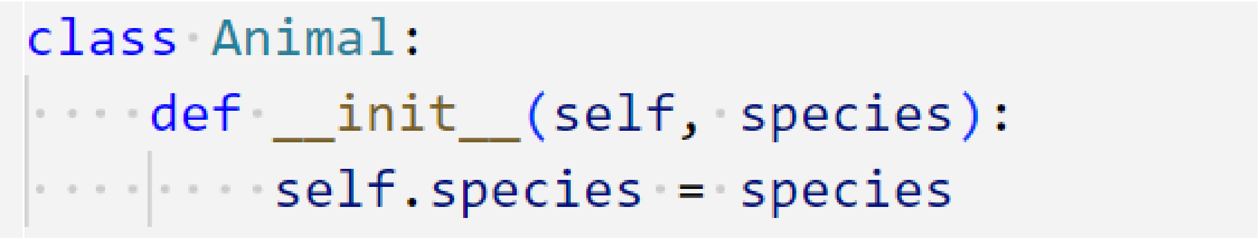
ANIMAL CLASS

DOG CLASS

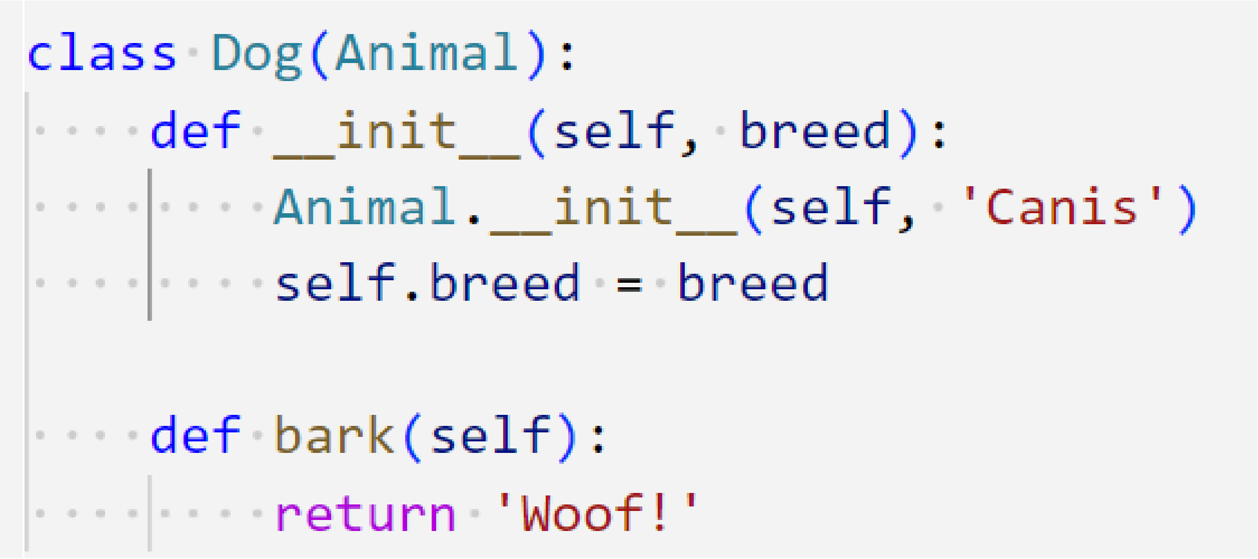
CAT CLASS

COW CLASS

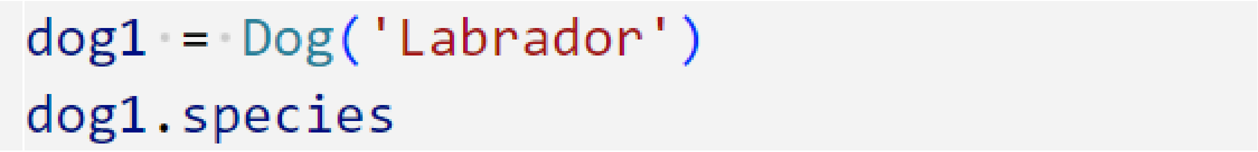
Inheritance



```python
class Animal:
    def __init__(self, species):
        self.species = species
```



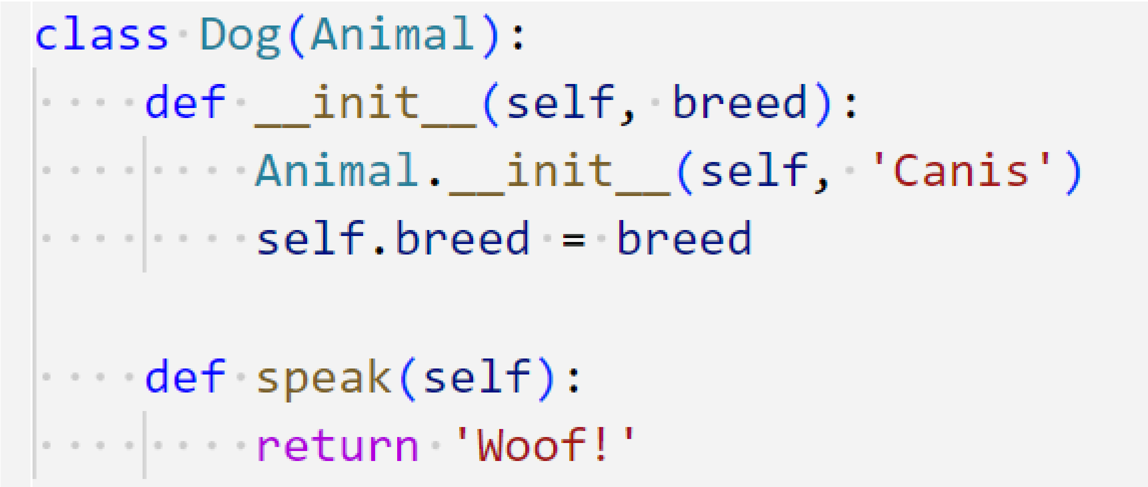
```python
class Dog(Animal):
    def __init__(self, breed):
        Animal.__init__(self, 'Canis')
        self.breed = breed

    def bark(self):
        return 'Woof!'
```



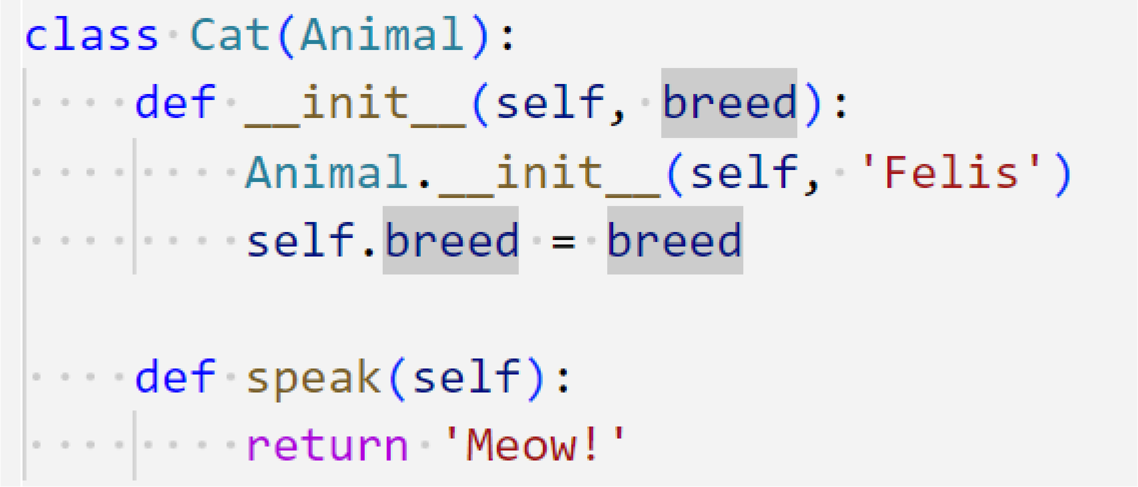
```python
dog1 = Dog('Labrador')
dog1.species
```

- **Inheritance**
  - parent class Animal
  - child class Dog
  - child class inherits attributes and methods of some parent class
- Child classes can
  - define new attributes and methods
  - override or extend attributes and methods of the parent class

Polymorphism



```python
class Dog(Animal):
    def __init__(self, breed):
        Animal.__init__(self, 'Canis')
        self.breed = breed

    def speak(self):
        return 'Woof!'
```

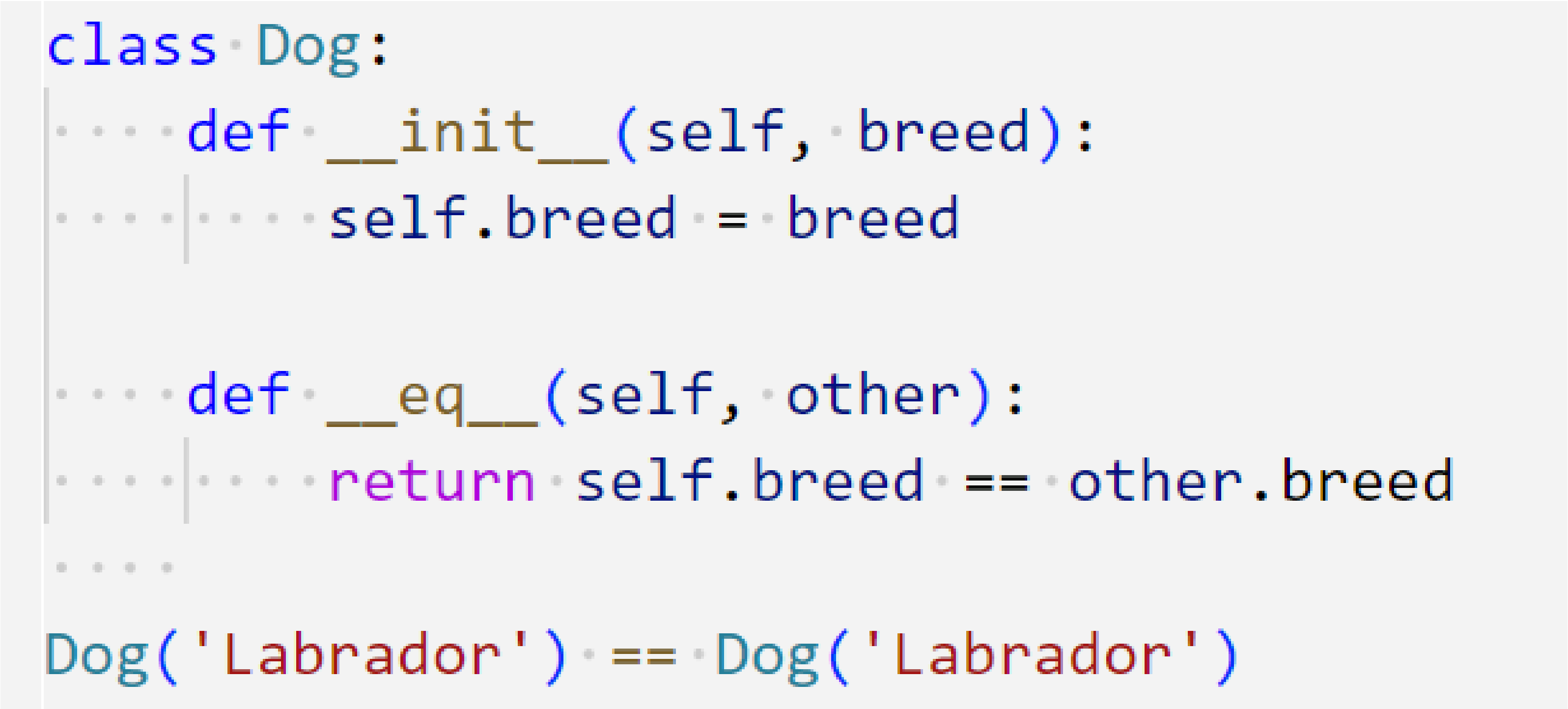


```python
class Cat(Animal):
    def __init__(self, breed):
        Animal.__init__(self, 'Felis')
        self.breed = breed

    def speak(self):
        return 'Meow!'
```

- **Polymorphism**: the same method or operator has different (but related) effects depending on the class they are applied on

Operator overloading

- We often have some notion of equality (==) that we are interested in
- Operator overloading allows us to define for a specific class, what we mean by equality in the context of this class
- For the equality operator, we need to define the __eq__() method



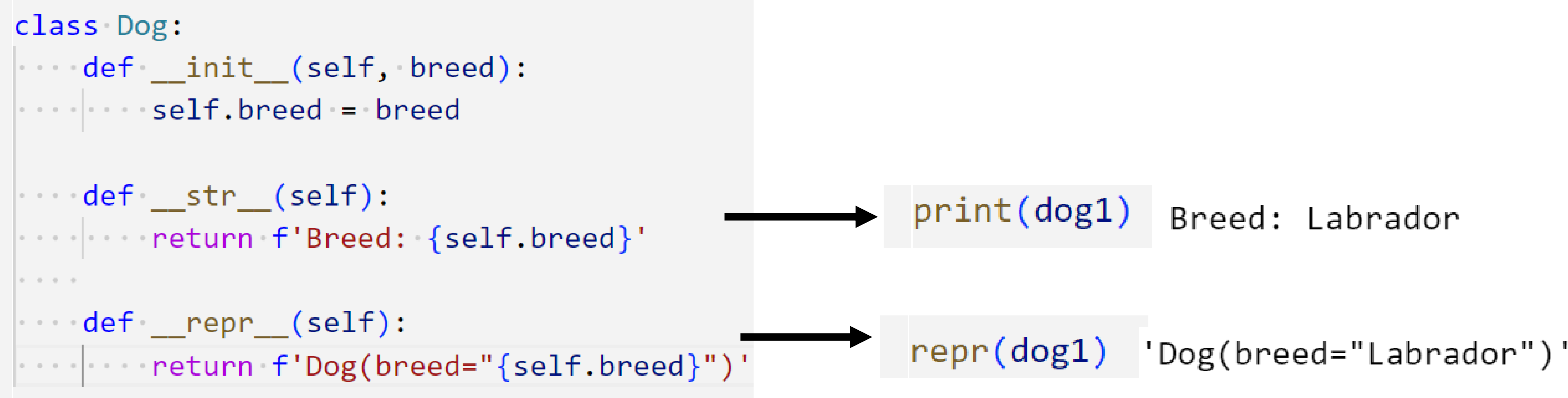
```
class Dog:
    def __init__(self, breed):
        self.breed = breed

    def __eq__(self, other):
        return self.breed == other.breed
    
Dog('Labrador') == Dog('Labrador')
```

Operator overloading

Operator	Dunder Method	Description
+	__add__	Addition
-	__sub__	Subtraction
*	__mul__	Multiplication
/	__truediv__	Division
**	__pow__	Exponentiation
<	__lt__	Less than
>	__gt__	Greater than
>=	__ge__	Greater or equal than
…		

Printing and string representation of objects

- The dunder method
  - __str__() defines how the object is printed
  - __repr__() defines the string representation of the object (useful to re-create the object from code)



```python
class Dog:
    def __init__(self, breed):
        self.breed = breed

    def __str__(self):
        return f'Breed: {self.breed}'
    
    def __repr__(self):
        return f'Dog(breed="{self.breed}")'
```

```
print(dog1)
```

```
Breed: Labrador
```

```
repr(dog1)
```

```
'Dog(breed="Labrador")'
```