Classes

It is often natural to combine data and methods

A dog has properties (data):

- Name
- Breed
- Size

A dog can do things (methods):

- Eat
- Sleep
- Learn tricks

A collection of data + methods is called an object

Dog object

Data

Name: Fido
Breed: Mutt
Weight: 30 lbs

Methods

Eat
Sleep
Do tricks

We need to distinguish between type (class) and instance (object)

Dog class (generic)

Data

Name
Breed
Weight

Methods

Eat
Sleep
Do tricks

Dog instance (dog "Fido")

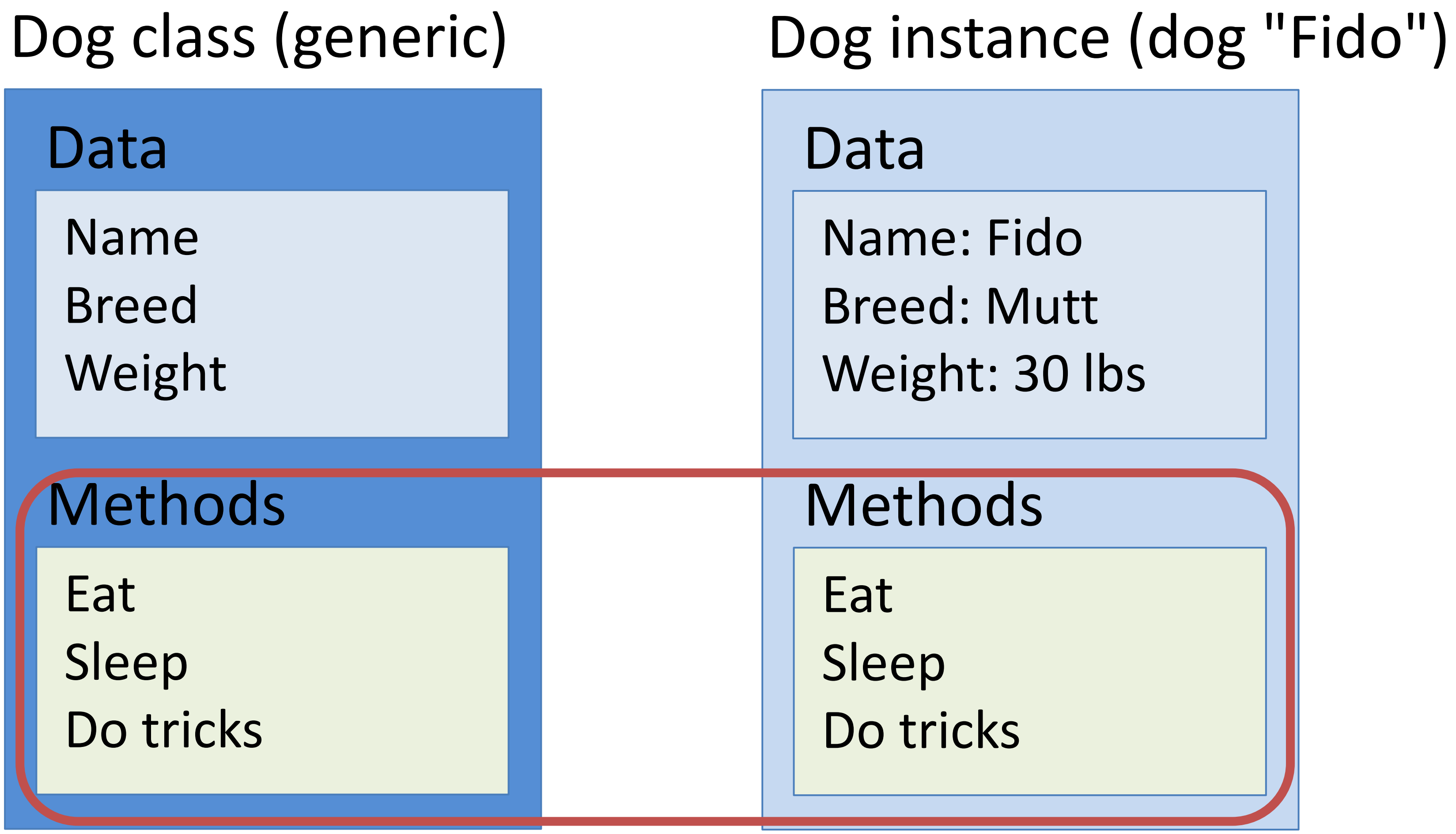
Data

Name: Fido
Breed: Mutt
Weight: 30 lbs

Methods

Eat
Sleep
Do tricks

We need to distinguish between type (class) and instance (object)



The methods are the same in both cases

We have one generic class and many instances (one for each dog)

Dog class (generic)

Data

Name
Breed
Weight

Methods

Eat
Sleep
Do tricks

Dog instances

Data

Name: Buddy
Breed: Poodle
Weight: 50 lbs

Methods

Eat
Sleep
Do tricks

In Python, both data and methods are accessed via a period

```python
dog.name         # name of the dog
dog.breed        # breed of the dog
dog.sleep()      # make the dog sleep
```

You have seen this already with lists and dictionaries

```
In [1]: mylist = [1, 2, 3]
        # call method `append` on list object `mylist`:
        mylist.append(4)
        # mylist is now [1, 2, 3, 4]
        mylist
Out[1]: [1, 2, 3, 4]
```

You have seen this already with lists and dictionaries

```
In [1]: mydict = {'A':1, 'B':2, 'C':3}
        # call method `keys` on dict object `mydict`:
        mydict.keys()
Out[1]: ['A', 'C', 'B']
```

Strings are objects as well

```
In [1]: "hello".upper() # make upper-case version
Out[1]: 'HELLO'
```

The original string remains unchanged.

```
In [2]: "-".join(['A', 'B', 'C']) # join list of strings
Out[2]: 'A-B-C'
```

The `join` function is a method of the string object, and it takes a list of strings to be joined as argument.

Some methods modify an object, others don't

Examples of methods that modify their object:

- `list.append()` # add element to end of list
- `dict.clear()` # empty out dictionary

Examples of methods that don't modify their object:

- `list.copy()` # return a copy of the list
- `dict.keys()` # return a list of all keys in the dict
- `str.upper()` # return upper-case version of string

Some methods modify an object, others don't

- We need to know for each method how it behaves (read the documentation!)
- String methods never modify their object (strings are immutable!)

Implementing a class: A simple example (An object that can count)

Counter class (generic)

Data

count

Methods

increment
decrement
reset

Counter instance

Data

count = 5

Methods

increment
decrement
reset

Implementing a class: A simple example (An object that can count)

```python
class Counter: # start definition of the class `Counter`
    count = 0  # the count, initially set to 0

    def increment(self):   # class method
        self.count += 1
```

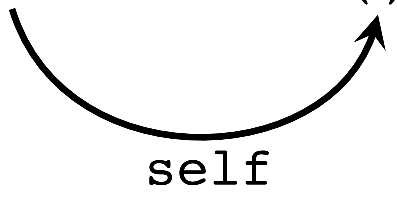
- The method `increment()` takes an argument `self`, which is the instance on which it will act.
- The `self` argument is automatically provided by Python.

Using the counter object

```
In [1]: c = Counter()     # make new Counter object,
                            with count=0
        print(c.count)

        c.increment()     # increase counter by 1
        print(c.count)

Out[1]: 0
        1
```

Compare definition of a member function to how it is used

```
class Counter:
    def increment(self):   # we explicitly list `self`
        self.count += 1

c.increment()        # we don't provide the self argument
                     # Python does this for us
```



Providing a defined initial state: the `__init__()` function

```
class Counter:
    def __init__(self):   # executed every time a new
        self.count = 0    # Counter object is created

c = Counter() # calls __init__() automatically
```

- It is good practice to always define an `__init__()` function for every class
- This function should put each new instance of a class into a defined state (e.g., make sure the counter starts at 0)