

CSE231 - Lab 11

Classes

What is a class? (from last time)

Classes are programmatic representations of some entity. What this entity represents, and what it can do, is decided by the programmer.

We might want to think of something like a car, where that car contains features/attributes that we can manipulate and enact “functions” on.

- Car
 - Manufacturer: Tesla
 - Model: Model 3
 - Range: 423 km
 - Functionality: `accelerate()`, `decelerate()`, `steer()`, `recharge()`

(I know absolutely nothing about cars, please don't ask me questions regarding cars)

Class Structure Overview

```
class MyCoolClass():  # Declaration

    def __init__(self, a, b):  # Constructor

        self.x = a  # Data Member, Attribute

        self.y = b

    def my_method(self, c):  # Method

        return c + self.x
```

`__init__()`

The constructor is a special, pre-defined function where you can dictate the behaviour of your object when one is created. The constructor is so special, in fact, that we refer to it as a *magic method*.

In the constructor, we typically initialize *data members* (also sometimes referred to as *attributes*), that can store information *about* your class.

You can, of course, give the constructor parameters. The programmer would then have to pass arguments into your class to initialize one.

We've been using constructors a lot

```
my_str = "hello!"
```

```
my_list = list(my_str)      # This is the list constructor!
```

```
print(my_list)             # ['h', 'e', 'l', 'l', 'o', '!']
```

The `list()` function that we've been using this entire time is the *constructor* function for the `list` class!

The `list` class' constructor likely has some logic built-in to iterate through other iterable types to create a `list` instantiation.

Method Functions

We've talked a lot about method functions too, right? When we use something like `.lower()` on a string, we're accessing that string's `.lower()` method function. The function goes into its data member (the string's value), and lower-cases all upper-case characters.

The “method functions” we're talking about here are the same! Just that now, we're the ones *creating* the methods. They can be whatever you want, and are able to access the data members you created.

self

So, you're probably wondering what this “self” keyword being spilled everywhere is. What is it? What does it do?

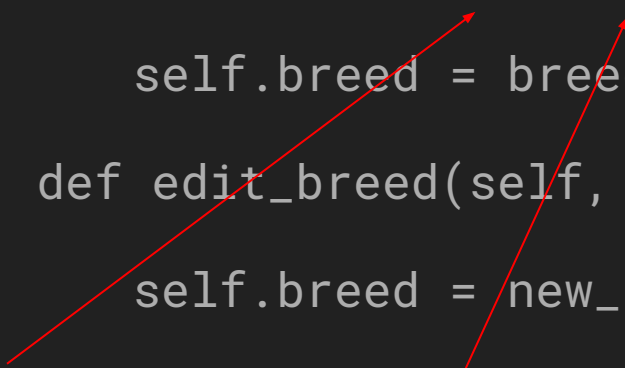
self is a way for the class you're making to *refer* to itself. In a sense, the class doesn't “know” what it contains, and so you have to pass it in as a “parameter”, but not in a traditional way.

When you call `.lower()` on a string, for example, the string you're calling the method function on *is* the “self” in that case.

We'll go through an example.

Constructor Call

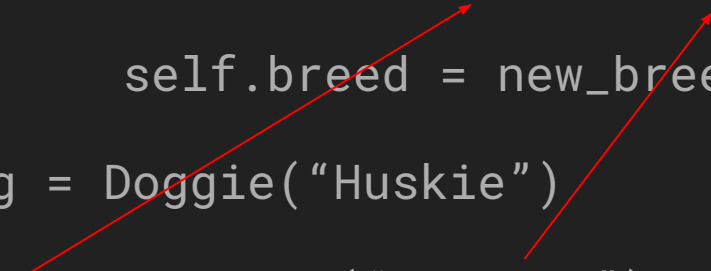
```
class Doggie():  
    def __init__(self, breed):  
        self.breed = breed  
    def edit_breed(self, new_breed):  
        self.breed = new_breed  
dog = Doggie("Huskie")  
dog.edit_breed("Samoyed")
```



The diagram consists of two red arrows. The first arrow originates from the string argument "Huskie" in the constructor call `Doggie("Huskie")` and points to the `breed` parameter in the `__init__` method's signature. The second arrow originates from the string argument "Samoyed" in the method call `dog.edit_breed("Samoyed")` and points to the `new_breed` parameter in the `edit_breed` method's signature.

Method Call


```
class Doggie():  
    def __init__(self, breed):  
        self.breed = breed  
  
    def edit_breed(self, new_breed):  
        self.breed = new_breed  
  
dog = Doggie("Huskie")  
dog.edit_breed("Samoyed")
```



The diagram consists of two red arrows. The first arrow originates from the `dog` variable in the line `dog = Doggie("Huskie")` and points to the `self` parameter in the `__init__` method definition. The second arrow originates from the `dog` attribute in the line `dog.edit_breed("Samoyed")` and points to the `self` parameter in the `edit_breed` method definition.

Data Member Call

```
class Doggie():  
    def __init__(self, breed):  
        self.breed = breed  
    def edit_breed(self, new_breed):  
        self.breed = new_breed  
  
dog = Doggie("Huskie")  
print( dog.breed )      # 'Huskie'
```



The diagram consists of two vertical red lines with arrows at the top. The left line starts from the `dog` variable in the line `dog = Doggie("Huskie")` and points to the `self` parameter in the `__init__` method. The right line starts from the `dog.breed` attribute access in the line `print(dog.breed)` and points to the `self.breed` attribute access in the `__init__` method. This illustrates the lookup path for the `breed` attribute.

`__str__()`, `__repr__()`

In addition to `__init__()`, there's another pre-defined function called `__str__()`. Normally, when you try to print an instance of a class, Python will try to convert the instance into a string. Without this method, you'll get some... unhelpful things. `__str__()` gives you the ability to represent your class as a string if you ever need to.

There's another pre-defined method, `__repr__()`, which does the exact same thing, but leans more on the side of industry use. It's usually good practice to make a `__repr__()` method, even if you just call `__str__()` with it.

There are more of these things

There are *a lot* more magic methods that we won't be covering right now. You'll learn about them in the coming weeks.

```
__add__()
```

```
__sub__()
```

```
__mul__()
```

```
__pow__()
```

```
...
```