Classes

- At least the important stuff

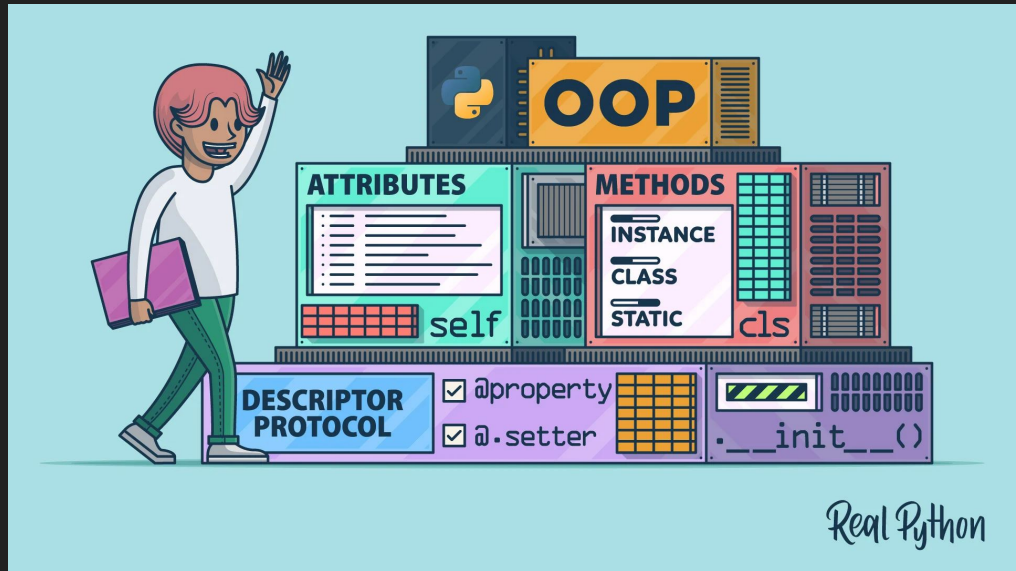
```
class class:
    def __init__(class, class):
        class.class = class

    def class(class):
        print(class.class)
```

Tyson Dial

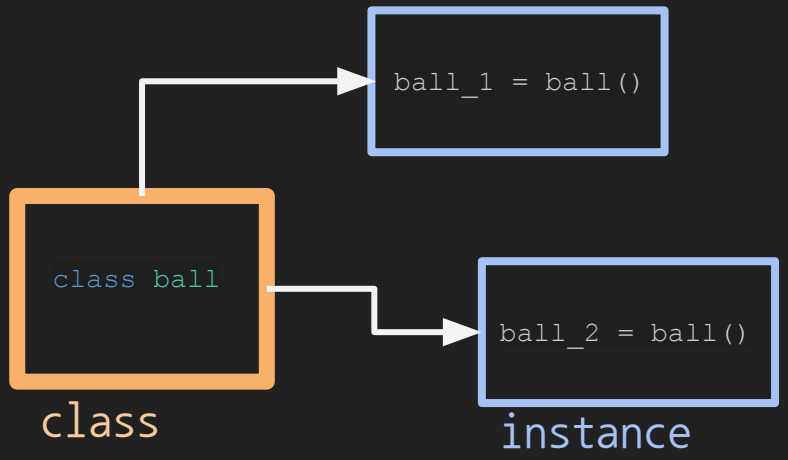
The purpose of a class

- OOP, data container, etc.
- A class is an ‘object’ that contains data and functions
- Useful for isolating code, holding data in a structured way



Making a class

- A class is a blueprint
- Build thing based on blueprint (we can do as many times as we want!)



Making a class

- Class constructor
- Attributes
- Methods
- Class variables

- __init__, code that is run when making an instance

```
# make the class
class ball:

    # class constructor
    def __init__(self, radius):

        self.radius = radius
```

Making a class

- Class constructor
- Attributes
- Methods
- Class variables

- Some data about that class instance

```
# make the class
class ball:

    # class constructor
    def __init__(self, radius):

        self.radius = radius
```

Making a class

- Class constructor
- Attributes
- Methods
- Class variables

- def, functions defined in class always have a first argument ‘self’

- self, current instance

```
# make the class
class ball:

    # class constructor
    def __init__(self, radius):

        self.radius = radius

    # another function
    def get_volume(self):
```

Making a class

- Class constructor
- Attributes
- Methods
- Class variables

- Class variable is an attribute of the base class, and any instance derived from that class will share that variable.

- Will check if the instance has this attribute, else will check the base class.

```
# make the class
class ball:

    density = 0.15

    # class constructor
    def __init__(self, radius):

        self.radius = radius

    # another function
    def get_volume(self):

    def get_mass(self):

        return self.density * self.get_volume()
```

https://wiki.python.org/moin/Decorators

Decorators

- Getters
- Setters

- Run code while retrieving an attribute

- Requires the @property decorator

```
# another function
@property
def volume(self):

    return 4/3 * np.pi * self.radius**3

@property
def mass(self):

    return self.density * self.volume
```

Decorators

- Getters
- Setters

- Run code while setting an attribute

Getter

```
@property
def material(self):

    return self._material
```

```
@material.setter
def material(self, material):

    self._material = material
    self.density = materials[material]
```

Setter

Magic Methods

- Allow for seamless integration with in-built python functions and operations

- Operator overloading!!!

https://docs.python.org/3/reference/datamodel.html#emulating-callable-objects
(see sections 3.3.1 and 3.3.6 + 3.3.7)

Magic Methods

- __str__
- __add__

```
def __str__(self):

    # return str
    pstr = ""
    pstr += f"ball made of…”

    return pstr
```

- Override python print() function.

Magic Methods

- __str__
- __add__

- Override python + operator.

```
def __add__(self, ball2):

    total_mass = self.mass + ball2.mass
    volume = total_mass/self.density
    total_r = (3*volume/(4*np.pi))**(1/3)

    return ball(total_r, self.material)
```

Basically the end

- Didn’t touch inheritance…

- Here is a nice youtube playlist on classes…

https://www.youtube.com/watch?v=ZDa-Z5JzLYM&list=PL-osiE80TeTsqhIuOqKhwlXsIBIdSeYtc