CS: Objects in Python

Mutability

Class and Static Methods

Learning Objectives - Class and Static Methods

- **Define the term class method**
- **Explain how `cls` is used in class methods**
- **Define the term static method**
- **Identify the decorators needed for static and class methods**
- **Explain when to use an instance method, when to use a static method, and when to use a class method**

Class Methods

Defining a Class Method

A class method is a method that has access to the class, but not the object instance. That means a class method does not have access to the instance variables (the attributes in the constructor). It does have access to the class itself. If the class has class attributes, then a class method can access them.

The National Football League (NFL) has a salary cap. That means every team must spend below a certain amount of money. The class below defines an NFL team. Since the same salary cap applies to all teams, it is stored as a class attribute.

```
class NFLteam:
    """Define the NFL team class"""
    salary_cap = 198200000
    def __init__(self, city, name):
        self.city = city
        self.name = name
```

Each year, the salary cap is adjusted based on the league's revenues. So there needs to be a way in which to change the `salary_cap` attribute. This attribute is a class attribute, not an instance attribute. A class method is best suited for this job. Class methods are differentiated from instance methods by the `@classmethod` decorator. The decorator informs Python that the method is a class method and not an instance method. In addition, class methods also take `cls` as the parameter.

▼ **`self` versus `cls`**

An instance method needs access to the instance variables. `self` refers to the instantiated object of the class, which is what gives the method access to the instance variables. A class method needs access to the class variables. `cls` refers to the class as a whole, which is what gives the method access to the class variables.

```
class NFLteam:
    """Define the NFL team class"""
    salary_cap = 198200000
    def __init__(self, city, name):
        self.city = city
        self.name = name

    @classmethod
    def change_salary_cap(cls, new_cap):
        cls.salary_cap = new_cap

team_1 = NFLteam("Cincinnati", "Bengals")
team_2 = NFLteam("New England", "Patriots")
NFLteam.change_salary_cap(199000000)

print(team_1.salary_cap)
print(team_2.salary_cap)
```

Originally the salary cap was `198200000`. Then the class method `change_salary_cap` changed it to `199000000`. Since `team_1` and `team_2` are instances of the `NFLteam` class, their class attribute `salary_cap` was affected.

challenge

Try this variation:

```
class NFLteam:
    """Define the NFL team class"""
    salary_cap = 198200000
    def __init__(self, city, name):
        self.city = city
        self.name = name

    @classmethod
    def change_salary_cap(cls, new_cap):
        cls.salary_cap = new_cap

team_1 = NFLteam("Cincinnati", "Bengals")
team_2 = NFLteam("New England", "Patriots")
team_1.change_salary_cap(199000000)

print(team_1.salary_cap)
print(team_2.salary_cap)
```

▼ **Why was `team_2` affected when the method was called on `team_1`?**
If `change_salary_cap` was called on `team_1`, it seems reasonable that the change in `salary_cap` would only affect `team_1`. However, `salary_cap` is a class attribute, and `change_salary_cap` is a class method. That means **all** instances of the `NFLteam` class will be affected.

Class Methods Cannot Affect Instance Variables

It makes sense that a class method cannot change an instance variable. Class methods take `cls` as a parameter and not `self`. Take a look at the code below. The `change_city` method is a class method (as shown by its decorator), but it takes `self` as a parameter. The method also changes the instance variable `city` from `"Dallas"` to `"Portland"`. Run the code below and see what happens.

```
class NFLteam:
    """Define the NFL team class"""
    salary_cap = 198200000
    def __init__(self, city, name):
        self.city = city
        self.name = name

    @classmethod
    def change_city(self, new_city):
        self.city = new_city

team = NFLteam("Dallas", "Cowboys")
team.change_city("Portland")
print(team.city)
```

The instance variable was not changed. The `@classmethod` decorator keeps `change_city` from altering the instance variables even if `self` is passed to the method.

challenge

What happened to the string “Portland”?

If the instance variable was not changed and there was no error message, what happened? Change the end of your code to look like this:

```
team = NFLteam("Dallas", "Cowboys")
team.change_city("Portland")
print(team.city)
print(NFLteam.city)
```

`change_city` is a class method. So Python looks for the class attribute `city` to assign it the value `"Portland"`. This attribute does not exist. The only class attribute is `salary_cap`. So Python creates `city` **as a class attribute** and assigns it the value `"Portland"`.

Factory Methods

Alternate Constructor

A common use for class methods is to create an “alternative” constructor. Imagine that you have a `Pizza` class, whose constructor takes a list of toppings (sauce, cheese, vegetables, meats, herbs, etc.). Using the standard constructor, create a Margherita pizza (sauce, mozzarella, and basil).

```
class Pizza:
  """Pizza class with a list of toppings"""
  def __init__(self, toppings):
    self.toppings= toppings

my_pizza = Pizza(["tomato sauce", "mozzarella", "basil"])
print(my_pizza.toppings)
```

Margherita pizza is very popular, and the toppings are very specific. It would be nice to tell Python to make a Margherita pizza instead of having to list out all of the toppings each time. That is where class methods help out. You can write a class method that calls the constructor with predetermined set of parameters. This type of class method is referred to as a factory method.

```
class Pizza:
  """Pizza class with a list of toppings"""
  def __init__(self, toppings):
    self.toppings= toppings

  @classmethod
  def make_margherita(cls):
    return Pizza(["tomato sauce", "mozzarella", "basil"])

my_pizza = Pizza.make_margherita()
print(my_pizza.toppings)
```

▼ **Do not forget `return` in the class method**

It is important to use the `return` statement when using class methods as an alternative constructor. If not, `my_pizza` will have the value of `None` and `my_pizza.toppings` will be undefined. Variables can have a value of `None`, but an undefined attribute will cause an error.

challenge

Try this variation:

- Create a class method that creates a veggie pizza
  ▼ **Possible solution**

```
class Pizza:
  """Pizza class with a list of toppings"""
  def __init__(self, toppings):
    self.toppings= toppings

  @classmethod
  def make_veggie(cls):
    return Pizza(["tomato sauce", "mozzarella", "mushroom", "bell pepper"

my_pizza = Pizza.make_veggie()
print(my_pizza.toppings)
```

- Create a class method that creates a three-cheese pizza
  ▼ **Possible solution**

```
class Pizza:
  """Pizza class with a list of toppings"""
  def __init__(self, toppings):
    self.toppings= toppings

  @classmethod
  def make_three_cheese(cls):
    return Pizza(["tomato sauce", "mozzarella", "fontina", "parmesan"])

my_pizza = Pizza.make_three_cheese()
print(my_pizza.toppings)
```

Static Methods

Static Methods

Imagine a `Rectangle` class in which objects have a length, width, and a method to calculate the area. Create two instances of the `Rectangle` class, and then calculate the combined area of the two rectangles.

```
class Rectangle:
  """Rectangle class"""
  def __init__(self, l, w):
    self.length = l
    self.width= w

  def area(self):
    """Calculate the area of the rectangle"""
    return self.length * self.width

rect_1 = Rectangle(12, 27)
rect_2 = Rectangle(9, 3)
combined_area = rect_1.area() + rect_2.area()
print(combined_area)
```

This works, but the combined area has to be calculated by the user. Since the combined area is related to the `Rectangle` class, a better solution would be to add this functionality to the class. The third type of method in Python is a static method. Static methods are most often used to add functionality (like calculating the combined area) to a class. These methods use the `@staticmethod` decorator, and they do not require a special parameter like `self` or `cls`.

```
class Rectangle:
  """Rectangle class"""
  def __init__(self, l, w):
    self.length = l
    self.width= w

  def area(self):
    """Calculate the area of the rectangle"""
    return self.length * self.width

  @staticmethod
  def combined_area(r1, r2):
    """Find combined area of two rectangles"""
    return r1.area() + r2.area()

rect_1 = Rectangle(12, 27)
rect_2 = Rectangle(9, 3)
combined_area = Rectangle.combined_area(rect_1, rect_2)
print(combined_area)
```

challenge

Try this variation:

Call the `combined_area` method from the instance `rect_1` with `rect_1` and `rect_2` as parameters

```
combined_area = rect_1.combined_area(rect_1, rect_2)
```

▼ **Why does this work?**

Like class methods, static methods can be called from the class itself or from an instance of a class.

▼ **Can static methods be used for factory methods?**

Yes, using the `@staticmethod` decorator for factory methods will not cause an error. However, the Python community decided to use the `@classmethod` decorator when making factory methods. This community-based decision is called a convention. Code that does not follow convention will still run, but it is not considered to be “proper” Python.