DS2000
Fall 2021
Handout: Classes and Objects

Python is an *object-oriented* language. In this way, it is a legacy of the C++ programming language. C++ wasn't the first programming language to use the object-oriented paradigm (that was Smalltalk), but it helped launch object-oriented programming into the mainstream. Ever since C++ was introduced in the 1970s, object-oriented programming (OOP) has been popular among both programmers and programming-language designers.

OOP is meant to make programming more intuitive, reflecting the way we see things in real life. A object is a thing that often represents a real, often tangible, thing -- like a person, or a cup of coffee, or a car.

All people are not identical, but we all have overlapping *attributes*. You might say that every person, for example, has a name, a height, an age, etc. Every person also *does* certain things, like breathe or eat. All coffee orders are not identical, but they also have overlapping attributes like size, price, whether it's decaf, etc. All cars are not identical, but they have overlapping attributes like make, model, year, mileage, etc.

You might think of a class as a blueprint, and an object as an actual thing. There's one blueprint for the Eiffel Tower, but they built a building in France and one in Vegas. Same blueprint (I'm sure), but with different physical results.

Making Up Your Own Data Types

A *class* is a data type invented by us, the programmer. It bundles together a bunch of variables and a bunch of functions that operate on the variables (called methods). You can invent your own Car class, for example, by using the keyword *class*. The class's methods are defined within the class structure:

```
class Car:
    # Methods go here
```

A class -- inventing your own data type

A special method called **`__init__`** is used to create an instance of the class. This is also referred to as a *constructor*. Its entire job is to make an object from the class you've defined. When writing this method, you initialize the object's attributes, either with values you choose yourself, or with values provided by the caller.

For all of our methods, we use the keyword **`self`** to denote an instance of the class. Every method we write will take **`self`** as a parameter (it can have other parameters, too, but **`self`** is required). Here's how we would write the Car class's constructor, giving reasonable default values to make, model, and year:

```
class Car:
```

```
    def __init__(self):
        self.make = 'Powell'
        self.model = 'The Homer'
        self.year = 1991
        self.price = 82000.0
```

A class -- writing the construtor. Self is an instance of the class, and self.variable is used for the class's variables. By setting these default values, we're assuming every Car object ever created is a Homer, until the caller changes those values.

Now that we've invented our own data type by defining a *class*, all that code is really just sitting there until we create *objects* out of it. Let's go to our main function, where we might create variables of any data type. Now, we'll create an instance of our Car class:

```
my_car = Car()
```

An object -- creating a car with default values

Once you've defined a class, you can create many objects out of it. Define it once, make as many objects as you like. Each object, like the cars below, will have its own size, price, and name.

```
my_car = Car()
your_car = Car()
another_one = Car()
```

Multiple objects of the same data type.

With each line above, we are calling the Car class's constructor. Each one of the objects will have the same make (`Powell`) , model (`The Homer`), year (`1991`), and price (`$82,000.00`). But now let's customize our car, as one does:

```
my_car.make = 'AMC'
my_car.model = 'Gremlin'
my_car.year = 1978
my_car.price = 1879.0

your_car.make = 'Hillman'
your_car.model = 'Minx'
your_car.year = 1957
your_car.price = 2500.0
```

Assigning values to attributes

The Car class above doesn't have any methods yet except the constructor (we'll add those below). Methods are **defined in the class** and **invoked by an object**. An object can call any of its associated methods. You can't call a method without an object.

Other Methods

All we've written so far is the constructor, but classes can have other methods as well. For example, you might ask for an optional feature when you buy your car, like heated seats. Back in our class, we can define a method to add something on, modifying attributes as we go.

With every method, *self* is always the first parameter. Then we can add more, in this case the add-on for the car.

```
class Car:

    def __init__(self):
        self.make = 'Powell'
        self.model = 'The Homer'
        self.year = 1991
        self.price = 82000.0

    def add_feature(self, item):
        if item == 'heated seats':
            self.price += 200
        elif item == 'power steering':
            self.price += 350
```

A new method -- add a feature to the car and increase the price.

Back in the driver, we can invoke this method directly once we have an instance of the class

```
my_car.add_feature('heated seats')
your_car.add_feature('power steering')
```

Calling the add_feature method from the driver

Constructor with parameters

The constructor we made above takes no parameters and gives reasonable default values to the object's data. Without parameters, the class writer determines the initial values of certain attributes, specified by parameters.

```
class Car:

    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.year = 1991
        self.price = 82000.0

    def add_feature(self, item):
        if item == 'heated seats':
            self.price += 200
        elif item == 'power steering':
            self.price += 350
```

A new method -- add a feature to the car and increase the price.

We can also add parameters to any constructor, and now it's the caller's job to say what the initial values are. Below, we've set it up such that the caller *must* provide initial values for make and model when creating a Car object. The caller *does not* provide year and price when creating a Car object; that has to be done separately.

We want both of the following calls, using the same `__init__` function.

```
# Caller determines the initial values for make and model
your_car = Car('Hillman', 'Minx')
```

Driver calling the __init__ function, with parameters.

Finally, the driver (hah!). Putting these all together, including the import of the class module:

```
from car import Car

if __name__ == "__main__":
    my_car = Car('Powell', 'The Homer')
    my_car.add_feature('heated seats')

    your_car = Car('Hillman', 'Minx')
    your_car.year = 1957
    your_car.price = 2500.0
    your_car.add_feature('power steering')

    print('My car is...', my_car.make, my_car.model)
    print('Your car is...', your_car.make, your_car.model)
    print('You drive yours, I'll drive mine.')

main()
```

Main function. Create Car objects and call methods.

Here's the terminal output after all of this:

```
My car is... Powell The Homer
Your car is... Hillman Minx
You drive yours, I'll drive mine.
```