9. Classes

Creating and Using a Class

In [3]:

```
class Dog():
    """A simple attempt to model a dog."""

    def __init__(self, name, age):
        """Initialize name and age attributes."""
        self.name = name
        self.age = age

    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(self.name.title() + " is now sitting.")

    def roll_over(self):
        """Simulate rolling over in response to a command."""
        print(self.name.title() + " rolled over!")
```

- **Creating a class**
- first we define a class named **'Dog'**
- the parameters are empty because we are creating it from scratch
- we describe the class with the docstring
- the **init()** method:
  - a function that's part of a class is a *method*
  - handle them like functions
  - the **init**() method is for the initialization
  - it runs atomatically
  - the underscores are by convention (default methods)
  - it has 3 parameters *(self, name, age)*
  - this method automatically passes the self parameter, which is a reference for the instance *itself*
  - self will be passed automatically, **we just have to pass name and age later**
  - when two variables are defined they just have the prefix self
  - every variable defined with self is also aviable for every method in the class
  - and we will access those variables through any instance created
    - self.name = name
    - self.age = age
    - those are called **attributes**
  - the two other methods *sit()* and *roll_over()*
  - those two don't need additional information
  - we just define them to have one parameter *self*
  - for now they don't do much
  - in reality this methods would do a specific job from the class
  - if this would be a class for controlling a robot, this methods would be moving the arm or the leg of the robot

In [5]:

```
my_dog = Dog('willie', 6)

print("My dog's name is " + my_dog.name.title() + ".")
print("My dog is " + str(my_dog.age) + " years old.")
```

```
My dog's name is Willie.
My dog is 6 years old.
```

- **Making an Instance from a Class**
- we make a variable *my_dog* and say this is an instance of the class we created earlier
- we give this class two attributes
- the **init()** method will be called
- to **access attributes** use the dot convention
  - my_dog.name
  - the attribute name will be associated with my_dog
  - this is the same attribute reffered to as self.name in the class Dog

In [6]:

```
my_dog.sit()
my_dog.roll_over()
```

```
Willie is now sitting.
Willie rolled over!
```

- **Calling methods**
- we can call any method with the dot notation
- we declared the variable *my_dog* earlier with the class *Dog('willie', 6)*
- now we call the defined methods from the class
- the code in the method will be used when calling the method with this convention

In [8]:

```
my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)

print("My dog's name is " + my_dog.name.title() + ".")
print("My dog is " + str(my_dog.age) + " years old.")
my_dog.sit()

print("\nYour dog's name is " + your_dog.name.title() + ".")
print("Your dog is " + str(your_dog.age) + " years old.")
your_dog.sit()
```

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

- **Creating multiple instances**
- you can create as many instances from a class ass you need
- a second dog **your_dog** is created and assigned to the Class **Dog** with its two parameters

Tasks

- 9-1. Restaurant: Make a class called Restaurant. The **init**() method for Restaurant should store two attributes: a restaurant_name and a cuisine_type. Make a method called describe_restaurant() that prints these two pieces of information, and a method called open_restaurant() that prints a message indicating that the restaurant is open. Make an instance called restaurant from your class. Print the two attributes individually, and then call both methods.
- 9-2. Three Restaurants: Start with your class from Exercise 9-1. Create three different instances from the class, and call describe_restaurant() for each instance.
- 9-3. Users: Make a class called User. Create two attributes called first_name and last_name, and then create several other attributes that are typically stored in a user profile. Make a method called describe_user() that prints a summary of the user's information. Make another method called greet_user() that prints a personalized greeting to the user. Create several instances representing different users, and call both methods for each user.

In [18]:

```
# 9-1

class Restaurant():
    """A class describing a Restaurant."""

    def __init__(self, restaurant_name, cuisine_type):
        """Initialize attributes to describe the restaurant."""
        self.restaurant_name = restaurant_name
        self.cuisine_type = cuisine_type
        self.restaurant_open = True

    def describe_restaurant(self):
        """Print a description of the restaurant."""
        print("The Restaurant " + self.restaurant_name.title() + " has " + self.
cuisine_type.title() + " Cuisine!")

    def open_restaurant(self):
        print("The Restaurant " + self.restaurant_name.title() + " is open!")



my_restaurant = Restaurant('Hells', 'turkish')
print(my_restaurant.restaurant_name)
print(my_restaurant.cuisine_type)


my_restaurant.describe_restaurant()
my_restaurant.open_restaurant()
```

```
Hells
turkish
The Restaurant Hells has Turkish Cuisine!
The Restaurant Hells is open!
```

In [21]:

```
# 9-2

other_restaurant = Restaurant('berlin grill', 'döner')
another_restaurant = Restaurant('pizza hut', 'pizza')
and_another_restaurant = Restaurant('sushi place', 'sushi')

other_restaurant.describe_restaurant()
another_restaurant.describe_restaurant()
and_another_restaurant.describe_restaurant()
```

```
The Restaurant Berlin Grill has Döner Cuisine!
The Restaurant Pizza Hut has Pizza Cuisine!
The Restaurant Sushi Place has Sushi Cuisine!
```

In [51]:

```
# 9-3
class User():
    """A class describing an user."""

    def __init__(self, first_name, last_name, age, sex, membership):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.sex = sex
        self.membership = membership

    def describe_user(self):
        """Prints the user information."""
        print("\n" + self.first_name.title() + " " + self.last_name.title())
        print(" Age: " + str(self.age))
        print(" Sex: " + self.sex)
        print(" Membership: " + self.membership)

    def greet_user(self):
        """Greets the user personalized."""
        msg = "\nHello " + self.first_name.title() + " " + self.last_name.title
() +"!" + "\nHow are you?"
        print(msg)

new_user = User('ugur', 'tigu', 30, 'male', 'gold')
new_user.describe_user()
new_user.greet_user()

another_user = User('jack', 'daniels', 78, 'male', 'premium')
another_user.describe_user()
another_user.greet_user()
```

```
Ugur Tigu
 Age: 30
 Sex: male
 Membership: gold

Hello Ugur Tigu!
How are you?

Jack Daniels
 Age: 78
 Sex: male
 Membership: premium

Hello Jack Daniels!
How are you?
```

Working with Classes and Instances

In [9]:

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
```

```
2016 Audi A4
```

- **creating a new class**
- with the **init**() method we define the self parameter
- additionaly we have 3 more parameters
- the **init**() method will take those parameters and store them in **attributes**
- those attributes will be associated with instances made from the class
- then we define a method called **get_descriptive_name()**
  - this method puts the parameters into a string neatly describing the car
  - again we use the dot convention to work with it
- we make the instance from the Car class and store it in the variable my_new_car
- then we call the method from the class with the dot convention
- note that the **init**() method doesn't need a return, though our get_descriptive_name method needs one

In [10]:

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")


my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

```
2016 Audi A4
This car has 0 miles on it.
```

- **Setting a Default Value for an Attribute**
- every attribute in a class needs an initial value, even if that value is 0
- this is done in the body of the **init**() method, so you don't have to include a paramter for that attribute
- we also have a new method *read_odometer*
- since we have set the default value to 0 we just call the function without any arguments

In [17]:

```
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
This car has 23 miles on it.
```

- **modifying an attribute's value directly**
- through an instance we set the value of the attribute to 23
- first we use dot notation to acces the car's odometer_reading attribute

In [19]:

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = update
        if update:



my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(24)
my_new_car.read_odometer()
```

```
2016 Audi A4
This car has 24 miles on it.
```

- **Modifying an Attribute's Value Through a Method**
- we add a method into the class *update_odometer()*
- this method takes the mileage value and stores it in self.odometer_reading
- we call this method and give it the value 24 later on
- after that we call the read method to see the updated version

In [24]:

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 13

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")


my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(1)
my_new_car.read_odometer()
```

```
2016 Audi A4
You can't roll back an odometer!
This car has 13 miles on it.
```

- we extend the *update_odometer()* method with an if statement which tests if the initialized value is more then the updated value of the mileage
- in cases of a rollback we get an print alert
  - note we changed the default value of the odometer_reading to 13
  - if we update this value with 1 (which is less then the default value / or in other words the default value is less then the updated one)

In [26]:

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

my_used_car = Car('subaru', 'outback', 2013)
print(my_used_car.get_descriptive_name())

my_used_car.update_odometer(23500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

```
2013 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

- **Incrementing an Attribute's Value Through a Method**
- we want to increment an attribute rather than set an entirely new value
- first we create a new instance of the class which is a used car
- then we update the odometer of this car (which has an initial value of 0) but since this car is an used car we make it 23500
- we create a new method *increment_odometer*
  - this method takes the self.odometer_reading and which is updated through the update_odometer method to 23500
  - it increments it with 100 through the increment_odometer method
  - you can also modify this method to reject rollback

- 9-4. Number Served: Start with your program from Exercise 9-1 (page 166). Add an attribute called number_served with a default value of 0. Create an instance called restaurant from this class. Print the number of customers the restaurant has served, and then change this value and print it again. Add a method called set_number_served() that lets you set the number of customers that have been served. Call this method with a new number and print the value again. Add a method called increment_number_served() that lets you increment the number of customers who've been served. Call this method with any number you like that could represent how many customers were served in, say, a day of business.
- 9-5. Login Attempts: Add an attribute called login*attempts to your User class from Exercise 9-3 (page 166). Write a method called increment* login_attempts() that increments the value of login_attempts by 1. Write another method called reset_login*attempts() that resets the value of login* attempts to 0. Make an instance of the User class and call increment_login_attempts() several times. Print the value of login_attempts to make sure it was incremented properly, and then call reset_login_attempts(). Print login_attempts again to make sure it was reset to 0.

In [59]:

```
# 9-4

class Restaurant():
    """A class describing a Restaurant."""

    def __init__(self, restaurant_name, cuisine_type):
        """Initialize attributes to describe the restaurant."""
        self.restaurant_name = restaurant_name
        self.cuisine_type = cuisine_type
        self.number_served = 0

    def describe_restaurant(self):
        """Print a description of the restaurant."""
        print("The Restaurant " + self.restaurant_name.title() + " has " + self.
cuisine_type.title() + " Cuisine!")

    def open_restaurant(self):
        """Print that the restaurant is open."""
        print("The Restaurant " + self.restaurant_name.title() + " is open!")

    def read_number_served(self):
        """Reads how many items are been served."""
        print("This restaurant has " + str(self.number_served) + " items serve
d.")

    def set_number_served(self, served):
        """Sets the number of served items. The initial Value is 0!"""
        self.number_served = served

    def increment_number_served(self, update):
        """Updates the number of served items."""
        self.number_served += update

my_restaurant = Restaurant('Hells', 'turkish')

my_restaurant.describe_restaurant()
my_restaurant.open_restaurant()
print("\n")
my_restaurant.set_number_served(1)
my_restaurant.read_number_served()
my_restaurant.increment_number_served(12)
my_restaurant.read_number_served()
my_restaurant.increment_number_served(-3)
my_restaurant.read_number_served()
```

```
The Restaurant Hells has Turkish Cuisine!
The Restaurant Hells is open!


This restaurant has 1 items served.
This restaurant has 13 items served.
This restaurant has 10 items served.
```

In [72]:

```
#  9-5.

class User():
    """A class describing an user."""

    def __init__(self, first_name, last_name, age, sex, membership):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.sex = sex
        self.membership = membership
        self.login_attempts = 0

    def describe_user(self):
        """Prints the user information."""
        print("\n" + self.first_name.title() + " " + self.last_name.title())
        print(" Age: " + str(self.age))
        print(" Sex: " + self.sex)
        print(" Membership: " + self.membership)

    def greet_user(self):
        """Greets the user personalized."""
        msg = "\nHello " + self.first_name.title() + " " + self.last_name.title
() +"!" + "\nHow are you?"
        print(msg)

    def read_login_attempts(self):
        """Shows the login attempts."""
        print("The User: " + self.first_name.title() + " has " + str(self.login_
attempts) + " login attempts.")

    def increment_login_attempts(self, update):
        """Increments the login attempts. The initial value is 0!"""
        self.login_attempts += update

    def reset_login_attempts(self):
        """Resets the login attempts to 0 again."""
        self.login_attempts = 0


new_user = User('ugur', 'tigu', 30, 'male', 'gold')
new_user.describe_user()
print("\n")
new_user.read_login_attempts()
new_user.increment_login_attempts(12)
new_user.read_login_attempts()
new_user.reset_login_attempts()
new_user.read_login_attempts()
new_user.increment_login_attempts(120)
new_user.read_login_attempts()
new_user.reset_login_attempts()
new_user.read_login_attempts()
```

```
Ugur Tigu
 Age: 30
 Sex: male
 Membership: gold


The User: Ugur has 0 login attempts.
The User: Ugur has 12 login attempts.
The User: Ugur has 0 login attempts.
The User: Ugur has 120 login attempts.
The User: Ugur has 0 login attempts.
```

Inheritance

- you don't always start from scratch wen writing a class
- a specialized version of another class you worte is a child class
- this is done with *inheritance*
- the child class takes all the attributes and methods of the parent

In [27]:

```
class ElectricCar(Car):
    """Represents aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
        super().__init__(make, model, year)

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
```

```
2016 Tesla Model S
```

- **The init() Method for a Child Class**
- the **init()** method of the child needs help from its parent class
- an electric car is just a specific kind of *Car*
- if we want to go into more detail, we don't need to start from scratch, we can take the methods and attributes of the parent
- the parent has to appear before the child
- we define the child class *ElectricCar*
  - the name of the parent must have to be included into the parantheses of the child car *ElectricCar(Car)*
- the *super()* function is a special function that helps to make the connection between the two classes
- the parent is called "*Superclass*" and the child "*Subclass*"
- we make an instance of the new class with the variable my_tesla
- at this point we just want to check the connection of the two classes

In [30]:

```
class ElectricCar(Car):
    """Represents aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car."""
        super().__init__(make, model, year)
        self.battery_size = 70

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")


my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

- **Defining Attributes and Methods for the Child Class**
- we add specific attributes and methods to our child class
  - we have defined an attribute *battery_size* and gave it an initial value of 70
  - we have defined a method which just prints the battery size named *describe_battery*
- then we call our new method
- if the method or the attribute is general (means, that it could be belonging to any car) then define it to the parent class, otherwise if its specific than add it to the child class (electric car) - so everybody who needs the functionality will have it aviable

In [32]:

```
class ElectricCar(Car):
    """Represents aspects of a car, specific to electric vehicles."""

    def fill_gas_tank():
        """Electric cars don't have gas tanks."""
        print("This car doesn't need a gas tank!")
```

- **Overriding Methods from the Parent Class**
- say you have a method in the parent class named *fill_gas_tank()*
- we don't need this method in our child class
- we override this method because it is meaningless
- now if somebody runs the code from the electric car it is overwritten
  - from the parent class you can call this method as before

In [33]:

```
class Battery():
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describinmg the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")


class ElectricCar(Car):
    """Represents aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car."""
        super().__init__(make, model, year)
        self.battery = Battery()



my_tesla = ElectricCar('tesla', 'model s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

- **Instances as Attributes**
- the more detail the classes get, the longer the code gets
- it is useful to build seperate classes for different parts of the Class
- break them into smaller classes and let them work together
- we define a new class Battery
- in the **init**() method of this class we have just one parameter which is the battery_size
- this is an optional parameter, it sets the value to 70, if no other value is given
- the method desribe_battery() is now moved to this new class
- then in our ElectricCar Class, wich has its parent Car() we simply add an attribute called self.battery and assign it to our new class Battery()
- note that this Class has no argument, because the default is already 70
- when we want to describe the battery, we need to work through the car's battery attribute
  - this: my_tesla.describe_battery()
  - went to this: my_tesla.battery.describe_battery()

In [36]:

```
class Battery():
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describinmg the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)

class ElectricCar(Car):
    """Represents aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car."""
        super().__init__(make, model, year)
        self.battery = Battery()



my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

- we add another method to our new class Battery()
- the new method **get_range** tests:
  - the battery size and assigns to it a range
  - prints out the range
- we again have to call it through the battery attribute

Tasks

- 9-6. Ice Cream Stand: An ice cream stand is a specific kind of restaurant. Write a class called IceCreamStand that inherits from the Restaurant class you wrote in Exercise 9-1 (page 166) or Exercise 9-4 (page 171). Either version of the class will work; just pick the one you like better. Add an attribute called flavors that stores a list of ice cream flavors. Write a method that displays these flavors. Create an instance of IceCreamStand, and call this method.
- 9-7. Admin: An administrator is a special kind of user. Write a class called Admin that inherits from the User class you wrote in Exercise 9-3 (page 166) or Exercise 9-5 (page 171). Add an attribute, privileges, that stores a list of strings like "can add post", "can delete post", "can ban user", and so on . Write a method called show_privileges() that lists the administrator's set of privileges. Create an instance of Admin, and call your method.
- 9-8. Privileges: Write a separate Privileges class. The class should have one attribute, privileges, that stores a list of strings as described in Exercise 9-7. Move the show_privileges() method to this class. Make a Privileges instance as an attribute in the Admin class. Create a new instance of Admin and use your method to show its privileges.
- 9-9. Battery Upgrade: Use the final version of electric_car.py from this section. Add a method to the Battery class called upgrade_battery(). This method should check the battery size and set the capacity to 85 if it isn't already. Make an electric car with a default battery size, call get_range() once, and then call get_range() a second time after upgrading the battery. You should see an increase in the car's range.

In [77]:

```
# 9-6

class Restaurant():
    """A class describing a Restaurant."""

    def __init__(self, restaurant_name, cuisine_type):
        """Initialize attributes to describe the restaurant."""
        self.restaurant_name = restaurant_name
        self.cuisine_type = cuisine_type
        self.number_served = 0

    def describe_restaurant(self):
        """Print a description of the restaurant."""
        print("The Restaurant " + self.restaurant_name.title() + " has " + self.
cuisine_type.title() + " Cuisine!")

    def open_restaurant(self):
        """Print that the restaurant is open."""
        print("The Restaurant " + self.restaurant_name.title() + " is open!")

    def read_number_served(self):
        """Reads how many items are been served."""
        print("This restaurant has " + str(self.number_served) + " items serve
d.")

    def set_number_served(self, served):
        """Sets the number of served items. The initial Value is 0!"""
        self.number_served = served

    def increment_number_served(self, update):
        """Updates the number of served items."""
        self.number_served += update

class IceCreamStand(Restaurant):
    """A class describing an Ice Cream Stand, which is a child of a Restauran
t."""

    def __init__(self, restaurant_name, cuisine_type):
        """Initialize attributes of the parent class.
        Then initialize attributes specific."""
        super().__init__(restaurant_name, cuisine_type)
        self.flavors = ['vanillia', 'chocolate', 'banana']

    def display_flavors(self):
        """Print the flavors of the ice cream."""
        print("This restaurant serves " + str(self.flavors) + " Ice Cream!")

my_ice_store = IceCreamStand('Luna Ice', 'Ice Cream')
my_ice_store.display_flavors()
```

```
This restaurant serves ['vanillia', 'chocolate', 'banana'] Ice Crea
m!
```

In [96]:

```
# 9-7

class User():
    """A class describing an user."""

    def __init__(self, first_name, last_name, age, sex, membership):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.sex = sex
        self.membership = membership
        self.login_attempts = 0

    def describe_user(self):
        """Prints the user information."""
        print("\n" + self.first_name.title() + " " + self.last_name.title())
        print(" Age: " + str(self.age))
        print(" Sex: " + self.sex)
        print(" Membership: " + self.membership)

    def greet_user(self):
        """Greets the user personalized."""
        msg = "\nHello " + self.first_name.title() + " " + self.last_name.title
() +"!" + "\nHow are you?"
        print(msg)

    def read_login_attempts(self):
        """Shows the login attempts."""
        print("The User: " + self.first_name.title() + " has " + str(self.login_
attempts) + " login attempts.")

    def increment_login_attempts(self, update):
        """Increments the login attempts. The initial value is 0!"""
        self.login_attempts += update

    def reset_login_attempts(self):
        """Resets the login attempts to 0 again."""
        self.login_attempts = 0


class Admin(User):
    """A class describing the Admin, which is also a user!"""

    def __init__(self, first_name, last_name, age, sex, membership):
        """Initialize attributes of the parent class.
        Then initialize attributes specific."""
        super().__init__(first_name, last_name, age, sex, membership)
        self.privileges = ['can add posts', 'can delete posts', 'can edit posts'
]

    def show_privileges(self):
        """Shows the privileges of the admin."""
        for i in self.privileges:
            print("The admin " + i + "!")

my_admin = Admin('ugur', 'tigu', 30, 'male', 'membership')
my_admin.show_privileges()
```

```
The admin can add posts!
The admin can delete posts!
The admin can edit posts!
```

In [108]:

```
# 9-8

class User():
    """A class describing an user."""

    def __init__(self, first_name, last_name, age, sex, membership):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.sex = sex
        self.membership = membership
        self.login_attempts = 0

    def describe_user(self):
        """Prints the user information."""
        print("\n" + self.first_name.title() + " " + self.last_name.title())
        print(" Age: " + str(self.age))
        print(" Sex: " + self.sex)
        print(" Membership: " + self.membership)

    def greet_user(self):
        """Greets the user personalized."""
        msg = "\nHello " + self.first_name.title() + " " + self.last_name.title
() +"!" + "\nHow are you?"
        print(msg)

    def read_login_attempts(self):
        """Shows the login attempts."""
        print("The User: " + self.first_name.title() + " has " + str(self.login_
attempts) + " login attempts.")

    def increment_login_attempts(self, update):
        """Increments the login attempts. The initial value is 0!"""
        self.login_attempts += update

    def reset_login_attempts(self):
        """Resets the login attempts to 0 again."""
        self.login_attempts = 0


class Admin(User):
    """A class describing the Admin, which is also a user!"""

    def __init__(self, first_name, last_name, age, sex, membership):
        """Initialize attributes of the parent class.
        Then initialize attributes specific."""
        super().__init__(first_name, last_name, age, sex, membership)

        self.privileges = Privileges()

class Privileges():
    """A class for privileges."""

    def __init__(self, privileges=[]):
        self.privileges = privileges


    def show_privileges(self):
        print("\nPrivileges:")
```

```
        if self.privileges:
            for privilege in self.privileges:
                print("- " + privilege)
        else:
            print("- No privileges.")

ugur = Admin('ugur', 'tigu', 30, 'male', 'gold')
ugur.describe_user()

ugur.privileges.show_privileges()

print("\nAdding priviliges...")
ugur_privileges = [
    'can add',
    'can delete',
    'can edit',
]

ugur.privileges.privileges = ugur_privileges
ugur.privileges.show_privileges()
```

```
Ugur Tigu
 Age: 30
 Sex: male
 Membership: gold

Privileges:
- No privileges.

Adding priviliges...

Privileges:
- can add
- can delete
- can edit
```

In [116]:

```
# 9-9.

class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles


class Battery():
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describinmg the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)

    def upgrade_battery(self):
        """This method should check the battery size and set the capacity to 85
 if it isn't already."""
```

```
        if self.battery_size == 70:
            self.battery_size = 85
            print("Upgraded the battery to " + str(self.battery_size) + ".")
        else:
            print("The battery is already upgraded.")


class ElectricCar(Car):
    """Represents aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car."""
        super().__init__(make, model, year)
        self.battery = Battery()



my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
my_tesla.battery.upgrade_battery()
my_tesla.battery.get_range()
```

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
Upgraded the battery to 85.
This car can go approximately 270 miles on a full charge.
```

Importing Classes

In [40]:

```
%%writefile car.py


"""A class that can be used to represent a car."""
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles
```

```
Overwriting car.py
```

In [39]:

```
from car import Car

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
2016 Audi A4
This car has 23 miles on it.
```

- **Importing a single class**
- move the class into a module and import it - make the main program file clean
- first we do a docstring for our class
- then write the file
- then use the *from* and *import* keywoard to import the module
- after that, we make an instance of Car() with the arguments
- we can use this class, we stored in our variable *my_new_car* with the dot convention

In [1]:

```
%%writefile car.py


"""A class that can be used to represent a car."""
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

class Battery():
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describinmg the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)


class ElectricCar(Car):
```

```
    """Represents aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car."""
        super().__init__(make, model, year)
        self.battery = Battery()
```

```
Overwriting car.py
```

In [3]:

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

- **Storing multiple classes in a module**

In [4]:

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
```

```
2016 Volkswagen Beetle
2016 Tesla Model S
```

- **Importing multiple classes from a Module**

In [6]:

```
import car

my_beetle = car.Car('volkswagen', 'beetle', 2016)
print(my_beetle.get_descriptive_name())

my_tesla = car.ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
```

```
2016 Volkswagen Beetle
2016 Tesla Model S
```

- **Importing an Entire Module**
- note that you have to use the *car.** convention

Tasks

- 9-10. Imported Restaurant: Using your latest Restaurant class, store it in a module. Make a separate file that imports Restaurant. Make a Restaurant instance, and call one of Restaurant's methods to show that the import statement is working properly.
- 9-11. Imported Admin: Start with your work from Exercise 9-8 (page 178). Store the classes User, Privileges, and Admin in one module. Create a sepa- rate file, make an Admin instance, and call show_privileges() to show that everything is working correctly.
- 9-12. Multiple Modules: Store the User class in one module, and store the Privileges and Admin classes in a separate module. In a separate file, create an Admin instance and call show_privileges() to show that everything is still working correctly.

In [118]:

```
%%writefile restaurant.py

# 9-10

class Restaurant():
    """A class describing a Restaurant."""

    def __init__(self, restaurant_name, cuisine_type):
        """Initialize attributes to describe the restaurant."""
        self.restaurant_name = restaurant_name
        self.cuisine_type = cuisine_type
        self.number_served = 0

    def describe_restaurant(self):
        """Print a description of the restaurant."""
        print("The Restaurant " + self.restaurant_name.title() + " has " + self.
cuisine_type.title() + " Cuisine!")

    def open_restaurant(self):
        """Print that the restaurant is open."""
        print("The Restaurant " + self.restaurant_name.title() + " is open!")

    def read_number_served(self):
        """Reads how many items are been served."""
        print("This restaurant has " + str(self.number_served) + " items serve
d.")

    def set_number_served(self, served):
        """Sets the number of served items. The initial Value is 0!"""
        self.number_served = served

    def increment_number_served(self, update):
        """Updates the number of served items."""
        self.number_served += update

class IceCreamStand(Restaurant):
    """A class describing an Ice Cream Stand, which is a child of a Restauran
t."""

    def __init__(self, restaurant_name, cuisine_type):
        """Initialize attributes of the parent class.
        Then initialize attributes specific."""
        super().__init__(restaurant_name, cuisine_type)
        self.flavors = ['vanillia', 'chocolate', 'banana']

    def display_flavors(self):
        """Print the flavors of the ice cream."""
        print("This restaurant serves " + str(self.flavors) + " Ice Cream!")

my_ice_store = IceCreamStand('Luna Ice', 'Ice Cream')
my_ice_store.display_flavors()
```

```
Overwriting restaurant.py
```

In [4]:

```
from restaurant import Restaurant, IceCreamStand

my_restaurant = Restaurant('Pomm', 'fries')
my_restaurant.describe_restaurant()
print("\n")
my_ice = IceCreamStand('IceLand', 'ice')
my_ice.display_flavors()
```

```
The Restaurant Pomm has Fries Cuisine!


This restaurant serves ['vanillia', 'chocolate', 'banana'] Ice Crea
m!
```

In [6]:

```
%%writefile admin.py
# 9-11

class User():
    """A class describing an user."""

    def __init__(self, first_name, last_name, age, sex, membership):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.sex = sex
        self.membership = membership
        self.login_attempts = 0

    def describe_user(self):
        """Prints the user information."""
        print("\n" + self.first_name.title() + " " + self.last_name.title())
        print(" Age: " + str(self.age))
        print(" Sex: " + self.sex)
        print(" Membership: " + self.membership)

    def greet_user(self):
        """Greets the user personalized."""
        msg = "\nHello " + self.first_name.title() + " " + self.last_name.title
() +"!" + "\nHow are you?"
        print(msg)

    def read_login_attempts(self):
        """Shows the login attempts."""
        print("The User: " + self.first_name.title() + " has " + str(self.login_
attempts) + " login attempts.")

    def increment_login_attempts(self, update):
        """Increments the login attempts. The initial value is 0!"""
        self.login_attempts += update

    def reset_login_attempts(self):
        """Resets the login attempts to 0 again."""
        self.login_attempts = 0


class Admin(User):
    """A class describing the Admin, which is also a user!"""

    def __init__(self, first_name, last_name, age, sex, membership):
        """Initialize attributes of the parent class.
        Then initialize attributes specific."""
        super().__init__(first_name, last_name, age, sex, membership)

        self.privileges = Privileges()

class Privileges():
    """A class for privileges."""

    def __init__(self, privileges=[]):
        self.privileges = privileges


    def show_privileges(self):
```

```
        print("\nPrivileges:")
        if self.privileges:
            for privilege in self.privileges:
                print("- " + privilege)
        else:
            print("- No privileges.")

ugur = Admin('ugur', 'tigu', 30, 'male', 'gold')
ugur.describe_user()

ugur.privileges.show_privileges()

print("\nAdding priviliges...")
ugur_privileges = [
    'can add',
    'can delete',
    'can edit',
]

ugur.privileges.privileges = ugur_privileges
ugur.privileges.show_privileges()
```

```
Overwriting admin.py
```

In [13]:

```
from admin import Admin, Privileges

my_admin = Admin('ugur', 'tigu', 30, 'male', 'gold')
my_admin.describe_user()
my_admin.privileges.privileges = [
    'can add',
    'can delete',
    'can edit',
]
my_admin.privileges.show_privileges()
```

```
Ugur Tigu
 Age: 30
 Sex: male
 Membership: gold

Privileges:
- can add
- can delete
- can edit
```

In [24]:

```
%%writefile user_single.py

# 9-12

class User():
    """A class describing an user."""

    def __init__(self, first_name, last_name, age, sex, membership):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.sex = sex
        self.membership = membership
        self.login_attempts = 0

    def describe_user(self):
        """Prints the user information."""
        print("\n" + self.first_name.title() + " " + self.last_name.title())
        print(" Age: " + str(self.age))
        print(" Sex: " + self.sex)
        print(" Membership: " + self.membership)

    def greet_user(self):
        """Greets the user personalized."""
        msg = "\nHello " + self.first_name.title() + " " + self.last_name.title
() +"!" + "\nHow are you?"
        print(msg)

    def read_login_attempts(self):
        """Shows the login attempts."""
        print("The User: " + self.first_name.title() + " has " + str(self.login_
attempts) + " login attempts.")

    def increment_login_attempts(self, update):
        """Increments the login attempts. The initial value is 0!"""
        self.login_attempts += update

    def reset_login_attempts(self):
        """Resets the login attempts to 0 again."""
        self.login_attempts = 0
```

```
Writing user_single.py
```

In [2]:

```
%%writefile admin_priviliges.py

from user import User

class Admin(User):
    """A class describing the Admin, which is also a user!"""

    def __init__(self, first_name, last_name, age, sex, membership):
        """Initialize attributes of the parent class.
        Then initialize attributes specific."""
        super().__init__(first_name, last_name, age, sex, membership)
        self.privileges = Privileges()

class Privileges():
    """A class for privileges."""

    def __init__(self, privileges=[]):
        self.privileges = privileges


    def show_privileges(self):
        print("\nPrivileges:")
        if self.privileges:
            for privilege in self.privileges:
                print("- " + privilege)
        else:
            print("- No privileges.")
```

```
Overwriting admin_priviliges.py
```

In [5]:

```
from user_single import User
from admin_priviliges import Admin, Privileges


my_user = User('ugur', 'tigu', 30, 'male', 'gold')
my_user.describe_user()

my_admin = Admin('admin', 'adolf', 40, 'male', 'admin')
my_admin.describe_user()
my_admin.privileges.privileges = [
    'can add',
    'can delete',
    'can edit',
]
my_admin.privileges.show_privileges()
```

```
Ugur Tigu
 Age: 30
 Sex: male
 Membership: gold

Admin Adolf
 Age: 40
 Sex: male
 Membership: admin

Privileges:
- can add
- can delete
- can edit
```

The Python Standad Library

In [15]:

```
from collections import OrderedDict

favorite_languages = OrderedDict()

favorite_languages['jen'] = 'python'
favorite_languages['sarah'] = 'c'
favorite_languages['edward'] = 'ruby'
favorite_languages['phil'] = 'python'

for name, language in favorite_languages.items():
    print(name.title() + "'s favorite language is " + language.title() + ".")

print("\n")
print(favorite_languages)
```

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.


OrderedDict([('jen', 'python'), ('sarah', 'c'), ('edward', 'ruby'),
('phil', 'python')])
```

- we import the standard library OrderedDict from our collection of standard python modules
- we create an instance of OrderedDict() class
- we then add each value pair in our empty dictionary
- we loop through our items
- we will always get responses back in the order they where added
- it combines the benefits of a list (retaining original order) with the main feature of dictionaries (connecting pieces of information)

Tasks

- 9-14. Dice: The module random contains functions that generate random numbers in a variety of ways. The function randint() returns an integer in the range you provide. The following code returns a number between 1 and 6:

  ```
  from random import randint
  x = randint(1, 6)
  ```

  Make a class Die with one attribute called sides, which has a default value of 6. Write a method called roll_die() that prints a random number between 1 and the number of sides the die has. Make a 6-sided die and roll it 10 times. Make a 10-sided die and a 20-sided die. Roll each die 10 times.
- 9-15. Python Module of the Week: One excellent resource for exploring the Python standard library is a site called Python Module of the Week. Go to http://pymotw.com/ (http://pymotw.com/) and look at the table of contents. Find a module that looks interesting to you and read about it, or explore the documentation of the collections and random modules.

In [35]:

```
# 9-14

from random import randint

class Die():
    """An attempt to roll a dice."""

    def __init__(self,sides=6):
        self.sides = sides

    def roll_die(self):
        return randint(1, self.sides)

# a 6-sided die, 10 rolls.
die6 = Die()
results = []
for roll_num in range(10):
    result = die6.roll_die()
    results.append(result)
print("10 rolls of a 6-sided die")
print(results)

die10 = Die(sides=10)
results = []
for roll_num in range(10):
    result = die10.roll_die()
    results.append(result)
print("\n10 rolls of a 10-sided die")
print(results)

die20 = Die(sides=10)
results = []
for roll_num in range(10):
    result = die20.roll_die()
    results.append(result)
print("\n10 rolls of a 20-sided die")
print(results)
```

```
10 rolls of a 6-sided die
[3, 6, 1, 5, 5, 2, 5, 1, 6, 4]

10 rolls of a 10-sided die
[8, 10, 2, 5, 4, 9, 9, 3, 8, 9]

10 rolls of a 20-sided die
[4, 3, 10, 3, 5, 4, 8, 9, 9, 8]
```

In [36]:

```
%%writefile textwrap_example.py
# 9-15


sample_text = '''
    The textwrap module can be used to format text for output in
    situations where pretty-printing is desired.  It offers
    programmatic functionality similar to the paragraph wrapping
    or filling features found in many text editors.
    '''
```

```
Writing textwrap_example.py
```

In [37]:

```
import textwrap
from textwrap_example import sample_text

print(textwrap.fill(sample_text, width=50))
```

```
     The textwrap module can be used to format
text for output in     situations where pretty-
printing is desired.  It offers     programmatic
functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

In [38]:

```
dedented_text = textwrap.dedent(sample_text)
print('Dedented: ')
print(dedented_text)
```

```
Dedented:

The textwrap module can be used to format text for output in
situations where pretty-printing is desired.  It offers
programmatic functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

In [40]:

```
dedented_text = textwrap.dedent(sample_text).strip()
for width in [45, 60]:
    print('{} Columns:\n' .format(width))
    print(textwrap.fill(dedented_text, width=width))
    print()
```

```
45 Columns:

The textwrap module can be used to format
text for output in situations where pretty-
printing is desired.  It offers programmatic
functionality similar to the paragraph
wrapping or filling features found in many
text editors.

60 Columns:

The textwrap module can be used to format text for output in
situations where pretty-printing is desired.  It offers
programmatic functionality similar to the paragraph wrapping
or filling features found in many text editors.
```