REFERENCE

1. Udemy - The Data Science Course: Complete Data Science Bootcamp 2023
   - Unit 176.Object Oriented Programming
   - link: https://www.udemy.com/course/the-data-science-course-complete-data-science-bootcamp/learn/lecture/10773932#content (https://www.udemy.com/course/the-data-science-course-complete-data-science-bootcamp/learn/lecture/10773932#content)

A. OBJECTS

- Strings, integers, floats and lists can be termed as objects -objects = strings + integers + floats + lists
- OOP programs - java, php, C++
  - object = data + object manipulation
    - Data includes a number or a string
    - object manipulation includes operations allowing us to manipulate the data

B. CLASS, ATTRIBUTES, METHODS

- each **object/instance** belongs to a class (defines the rules for creating that object)

class >>>> object >>>> {attach} attributes

- **Class** defines the rules for creating that object
  - Example: uncle makes bicycles. He belongs to **"Bike-Makers" class
  - the bicycle is the object/instances
- **Attributes/properties** it has include: "color, size, type"
- **Method** - is a consequential logic sequence that can be applied to the object
  - or a method is a **function** that is part of a **class**
  - such as: ".turn_left(), .turn_right(), .slow_down()"

focus should be on the object

Example 1

- Create a **CLASS** create a list named: list_1
- list **OBJECTS**: [15.8, 1.5, -85.9]

list_1 = [15.8, 1.5, -85.9]

- **PROPERTIES/ATTRIBUTES** of the objects: the objects belong to data type called **FLOATS**
- **METHODS** applied to the objects can include:
  - .extend()
  - .index()

In [1]:
```
list_1 = [15.8, 1.5, -85.9]
# to understand the difference introduce a different class
b = (45, 60, 80)
```

In [2]:
```
# identify the class of the objects
type(list_1)
```

Out[2]:
```
list
```

In [3]:
```
# identify the class of the objects
type(b)
```

Out[3]:
```
tuple
```

In [4]:
```
# attributes of the object
# research on how to identify data types in a list
```

In [ ]:
```

```

In [5]:
```
# applying a method
list_1.index(-85.9)
```

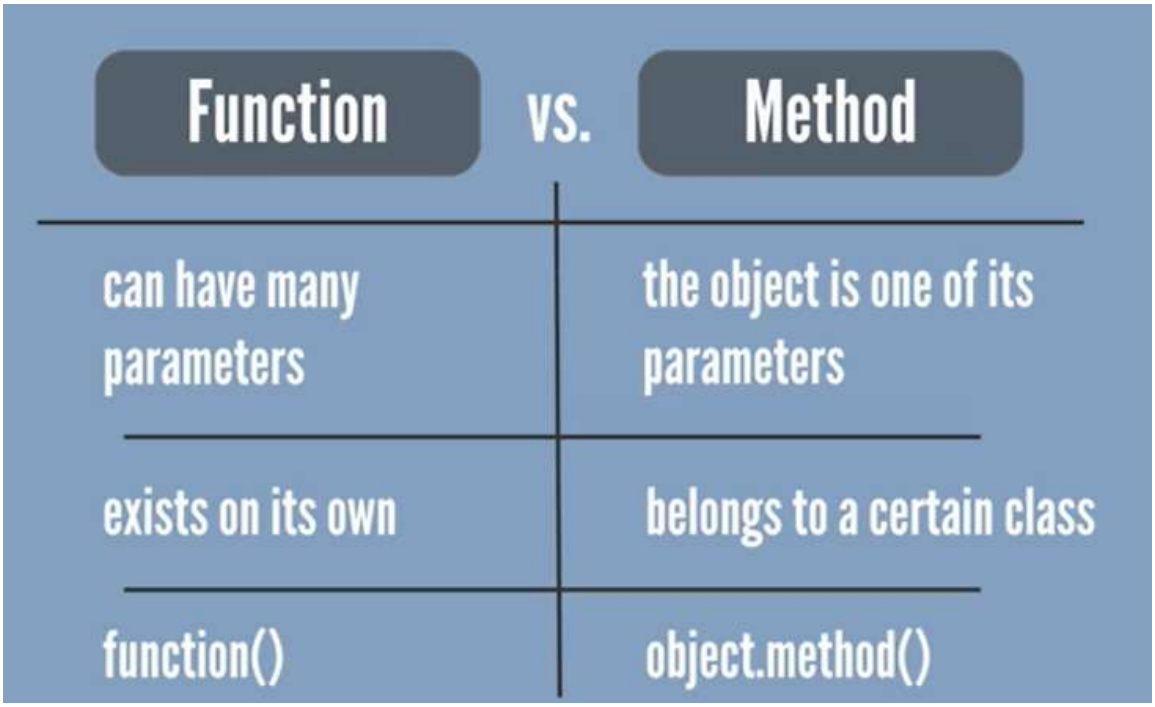
Out[5]:
```
2
```

In [6]:
```
# applying a method
list_2 = [454, 7200.89, 900.65]
list_1.extend(list_2)

list_1
```

Out[6]:
```
[15.8, 1.5, -85.9, 454, 7200.89, 900.65]
```

C. function vs object



REFERENCE

1. Python Crash Course, 2nd Edition. A Hands-On, Project-Based Introduction to Programming by Eric Matthes. Page 157. Chapter 9. Classes

1. Creating and using a class

In [7]:
```
class Dog:
    # define a class called DOG
    # By convention capitalized names refer to classes
    #There are no parentheses in the class definition because
    #we're creating this class from scratch"""


    """A simple attempt to model a dog"""
    # triple quotation marks are used to multi-line comment
    # docstring describing what this class does.


    def __init__(self, name, age):

        # OBJECTS: self, name, age
        # METHOD: __init__

        # A **Method** is a function that is part of a class
        # REMEMBER: class >>>> object >>>> attributes >>>> method
        # Make sure to use two underscores on each side of __init__().
        # __init__() method to have three parameters: self, name, and age.

        # The self parameter is required in the method definition,
        # and it must come first before the other parameters.
        # self, is a reference to the instance itself;
        # it gives the individual instance access to the attributes and meth

        """Initialize name and age attributes """

        self.name = name
        self.age = age

        # Any variable prefixed with self is available to every method in th
        # we'll also be able to access these variables through any instance

        # variable = constant.
        # THIS CASE:- variable : 'self.name' & constant : 'name'
        # Variables that are accessible through instances like this are call


        # other methods include: sit(); roll_over()

    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} is now siting.")

    def roll_over(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} rolled over!")

#this line of code generates nothing
#we define each line of the code
```

2. Making an INSTANCE FROM a class

In [25]:
```
# making an instance from a class

my_dog = Dog('Willie', 6)

"""i bet this is calling the function __init__(self, name, age)"""

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")

#Dot notation is used to define the attribute value
```

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

3. calling methods

In [9]:
```
my_dog.sit()
my_dog.roll_over()
```

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

4. we define another instance

In [10]:
```
your_dog = Dog('Lucy', 3)
```

5. call out the prints out

In [11]:
```
print(f"Your dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
your_dog.roll_over()
```

```
Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now siting.
Lucy rolled over!
```

Exercise

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

In [12]:
```
class Restaurant:

    def __init__(self, restaurant_name, cuisine_type):
        self.restaurant_name = restaurant_name
        self.cuisine_type = cuisine_type

    def describe_resautaurant(self):
        #this is a method i.e def describe_resautaurant(self)

        print(f"{self.restaurant_name} is the most lavish restaurant in Nair

    def open_restaurant(self):
        print(f"{self.restaurant_name} now open!\n")
```

In [13]:
```
rest_1 = Restaurant("Stanford flies", "Fast food")
rest_1.describe_resautaurant()
rest_1.open_restaurant()

rest_2 = Restaurant("mojos", "variety fast foods")
rest_2.describe_resautaurant()
rest_2.open_restaurant()
```

```
Stanford flies is the most lavish restaurant in Nairobi, Kenya
registered as a Fast food
Stanford flies now open!

mojos is the most lavish restaurant in Nairobi, Kenya
registered as a variety fast foods
mojos now open!
```

6. Working with classess and instances

In [14]:
```
class Car:

    def __init__(self, make, model, year):

        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        long_name = f"{self.make} {self.model} {self.year}"
        return long_name.title()

my_new_car = Car("BMW", "X1", "2022")
print(my_new_car.get_descriptive_name())
```

```
Bmw X1 2022
```

7. setting a default value for an attribute

In [15]:
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

#7.The default value to be set for an attribute**
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.make} {self.model} {self.year}"
        return long_name.title()

#7. create a new method to read the mileage
    def read_odometer(self):
        """Print a statement showing the car's mileage."""

        print(f"This car has {self.odometer_reading} miles on it.")


#Assign variables to the class Car
my_new_car = Car('BMW', 'X1', 2019)

print(my_new_car.get_descriptive_name())

my_new_car.read_odometer()
```

```
Bmw X1 2019
This car has 0 miles on it.
```

8.modifying an attribute value

In [16]:
```
#8. modifing an attribute value

my_new_car.odometer_reading = 15.754
my_new_car.read_odometer()
```

```
This car has 15.754 miles on it.
```

In [17]:
```
#8. modifing an attribute value

my_new_car.odometer_reading = 15
my_new_car.read_odometer()
```

```
This car has 15 miles on it.
```

10. modfying an attribute's value through a method

- It can be helpful to have methods that update certain attributes for you.
- Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.
- Here's an example showing a method called update_odometer():

In [18]:
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

#8.The default value to be set for an attribute**
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.make} {self.model} {self.year}"
        return long_name.title()

#8.create a new method to read the mileage
    def read_odometer(self):
        """Print a statement showing the car's mileage."""

        print(f"This car has {self.odometer_reading} miles on it.")

#10. define a new method
    def update_odometer(self, mileage):
        #set the odometer_reading to the given value

        self.odometer_reading = mileage


#Assign variables to the class Car
my_new_car = Car('BMW', 'X1', 2019)

print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

```
Bmw X1 2019
This car has 23 miles on it.
```

10.1 giving the update_odometer more functions

In [19]:
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

#8. The default value to be set for an attribute**
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.make} {self.model} {self.year}"
        return long_name.title()

#8. create a new method to read the mileage
    def read_odometer(self):
        """Print a statement showing the car's mileage."""

        print(f"This car has {self.odometer_reading} miles on it.")

#10. define a new method
    def update_odometer(self, mileage):
        #set the odometer_reading to the given value

        """
        10.1
        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!")


#Assign variables to the class Car
my_new_car = Car('BMW', 'X1', 2019)

print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

```
Bmw X1 2019
This car has 23 miles on it.
```

In [20]:
```
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(120)
my_new_car.read_odometer()
```

```
Bmw X1 2019
This car has 120 miles on it.
```

In [21]:
```
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(70)
my_new_car.read_odometer()
```

```
Bmw X1 2019
You can't roll back an odometer!
This car has 120 miles on it.
```

11. Incrementing an Attribute's value through a method

In [22]:
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

#8. The default value to be set for an attribute**
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.make} {self.model} {self.year}"
        return long_name.title()

#8. create a new method to read the mileage
    def read_odometer(self):
        """Print a statement showing the car's mileage."""

        print(f"This car has {self.odometer_reading} miles on it.")

#10. define a new method
    def update_odometer(self, mileage):
        #set the odometer_reading to the given value

        """
        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!")



#11. define a new method to update an increment in mileage
    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles
        print(f"This car has {self.odometer_reading} miles on it after servi



#Assign variables to the class Car for my_new_car

#1. defining __init__() function
my_new_car = Car('BMW', 'X1', 2023)

#2. calling the defined get_descriptive_name() function
print(my_new_car.get_descriptive_name())

#3. calling the update_odometer() function
my_new_car.update_odometer(127)

#4. calling the read_odometer() function
my_new_car.read_odometer()

#5. calling the increment_odometer() function
my_new_car.increment_odometer(100)
```

```
#6. calling the read_odometer() function
my_new_car.read_odometer()



#Assign variables to the class Car for my_used_car

#1. defining __init__() function
my_used_car = Car('subaru', 'outback', 2015)

#2. calling the defined get_descriptive_name() function
print(my_used_car.get_descriptive_name())

#3. calling the update_odometer() function
my_used_car.update_odometer(23_500)

#4. calling the read_odometer() function
my_used_car.read_odometer()

#5. calling the increment_odometer() function
my_used_car.increment_odometer(100)

#6. calling the read_odometer() function
my_used_car.read_odometer()
```

```
Bmw X1 2023
This car has 127 miles on it.
This car has 227 miles on it after service testing.
This car has 227 miles on it.
Subaru Outback 2015
This car has 23500 miles on it.
This car has 23600 miles on it after service testing.
This car has 23600 miles on it.
```

12. INHERITANCE

- You don't always have to start from scratch when writing a class.
- If the class you're writing is a specialized version of another class you wrote, you can use inheritance.

When one class inherits from another, it takes on the attributes and methods of the first class.

- The original class is called **the parent class**, and the new class is **the child class**.
- The child class can inherit any or all of the attributes and methods of its parent class, but it's also free to define new attributes and methods of its own.

In [23]:
```
#we begin by writing the parent class/superclass

class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.make} {self.model} {self.year}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""

        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        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

#12. Inheritance
# the parent class must be part of the current file and
# must appear before the child class/subclass in the file.

# we define the child class, ElectricCar.
#The name of the parent class must be included in parentheses
#in the definition of a child class.

#The __init__() method at takes in the information
#required to make a Car instance.

#The super() function at is a special function
#that allows you to call a method from the parent class.

#super().__init__(make, model, year)
#This line tells Python to call the __init__() method from Car,
#which gives an ElectricCar instance all the attributes defined
#in that method.



class ElectricCar(Car):
    """Represent 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', 2019)
print(my_tesla.get_descriptive_name())
```

```
print(f"My electric car is a {my_tesla.get_descriptive_name()}\n")

my_new_car = Car('BMW', 'X1', 2023)
print(my_new_car.get_descriptive_name())
print(f"My diesel combustion car is a {my_new_car.get_descriptive_name().upp
```

```
Tesla Model S 2019
My electric car is a Tesla Model S 2019

Bmw X1 2023
My diesel combustion car is a BMW X1 2023
```

13.Defining Attributes and Methods for the Child Class

In [24]:
```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.make} {self.model} {self.year}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage

        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)

        #13. adding another attribute

        self.battery_size = 75

    #13. defining a function to describe the size of the battery

    def describe_battery(self):

        #13. the message to output

        print(f"This car has a {self.battery_size}-kWh battery.")



my_tesla = ElectricCar('tesla', 'model s', 2019)
print(f"My electric car is a {my_tesla.get_descriptive_name().upper()}\n")

my_tesla.describe_battery()
```

```
My electric car is a TESLA MODEL S 2019

This car has a 75-kWh battery.
```

In [ ]: