Object Oriented Programming with Python

What is Object Oriented Programming (OOP)?

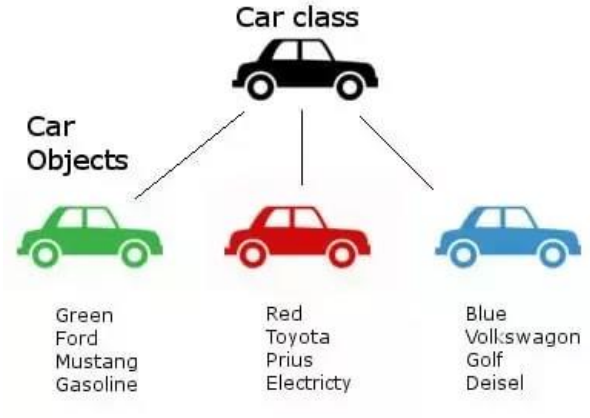
Until now we have being using **Procedural programming** as a way to solve problems and create useful products. **Procedural programming** is powerful, relevant, and popular. It shouldn't be forgotten or discounted.

Another tool we can use in many languages is **Object-Oriented Programming**. This is a type of programming that provides a way to:

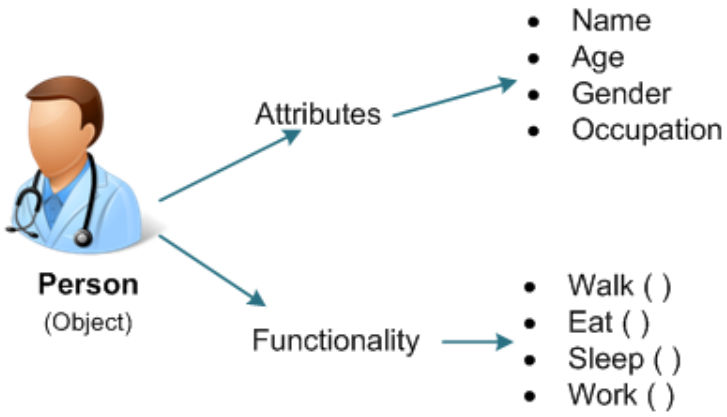
> Structure programs so that **data** and **behaviors** are **bundled together** into individual items called **objects**.

When modelling the real world around us with code it can sometimes be easier to use devices called **objects** to represent real things like a person, car, an email, or website.

For example, an **object** could represent a *car* with **properties** like a color, make, model, fuel type.
and **behaviors** such as driving, signaling, turning, and braking.



Or an **object** could also represent *person* with **properties** like a name, age, and address
and **behaviors** such as walking, talking, breathing, and running.

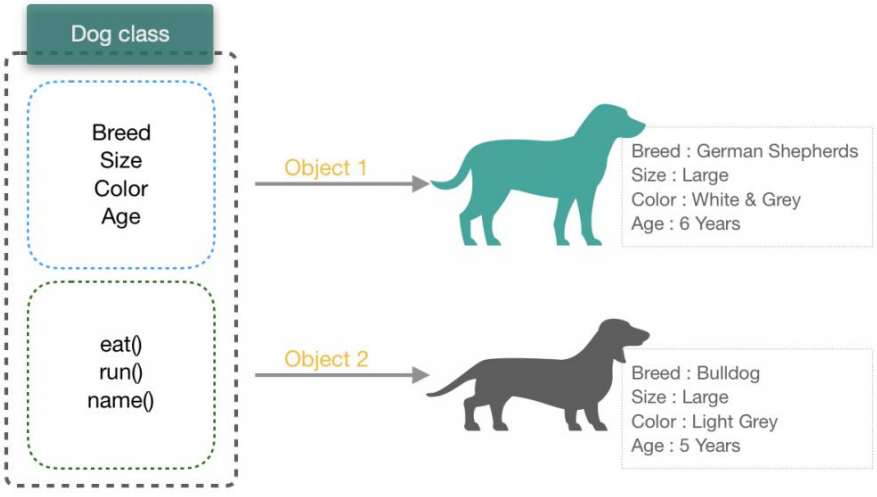


Recap: Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual **objects**.

How do you create Objects?

To create **objects** in python (and most languages) you need to create a **class**.

A **class** is a **blueprint** for creating **objects.**



Creating a **class** of **objects** in python is easy. You simply write the following:

`class Dog:`

Use the key word "`class`" followed by any name you wish for the class, and a colon.
Capitalization is normally used for class names.
Now we have a `Dog` **class** but we need to add some syntax to make it useful.

Here is how a **class** for dogs might look like in python.

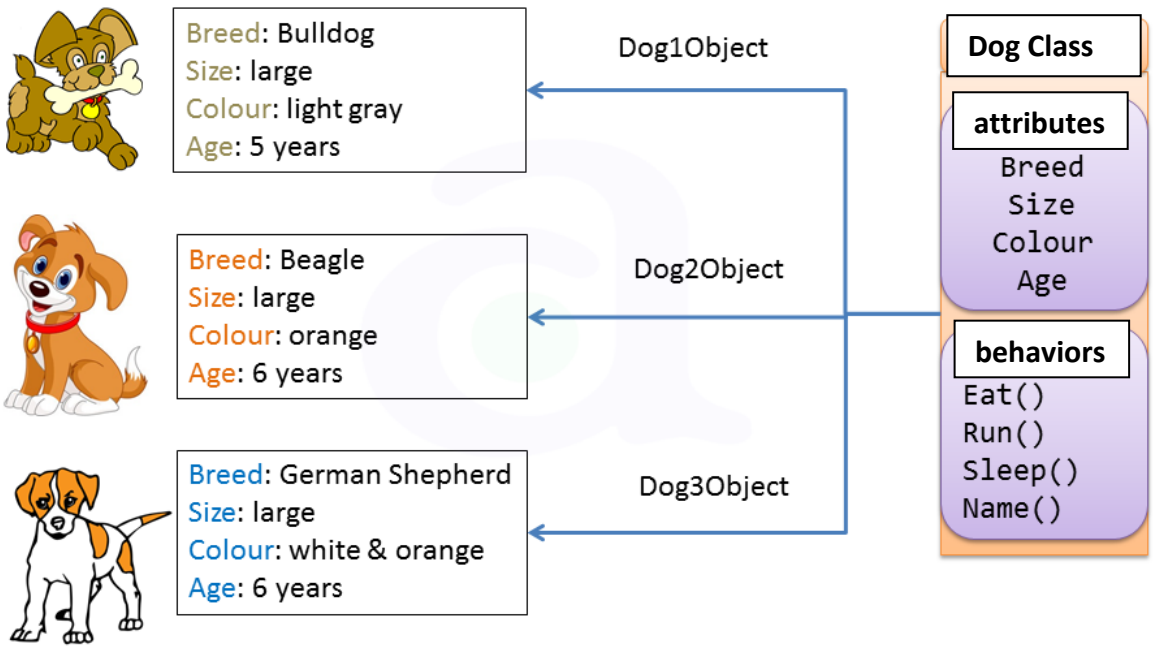
```
class Dog:
    def__init__(self, name, age):
        self.name = name
        self.age = age
```

A couple of key points:

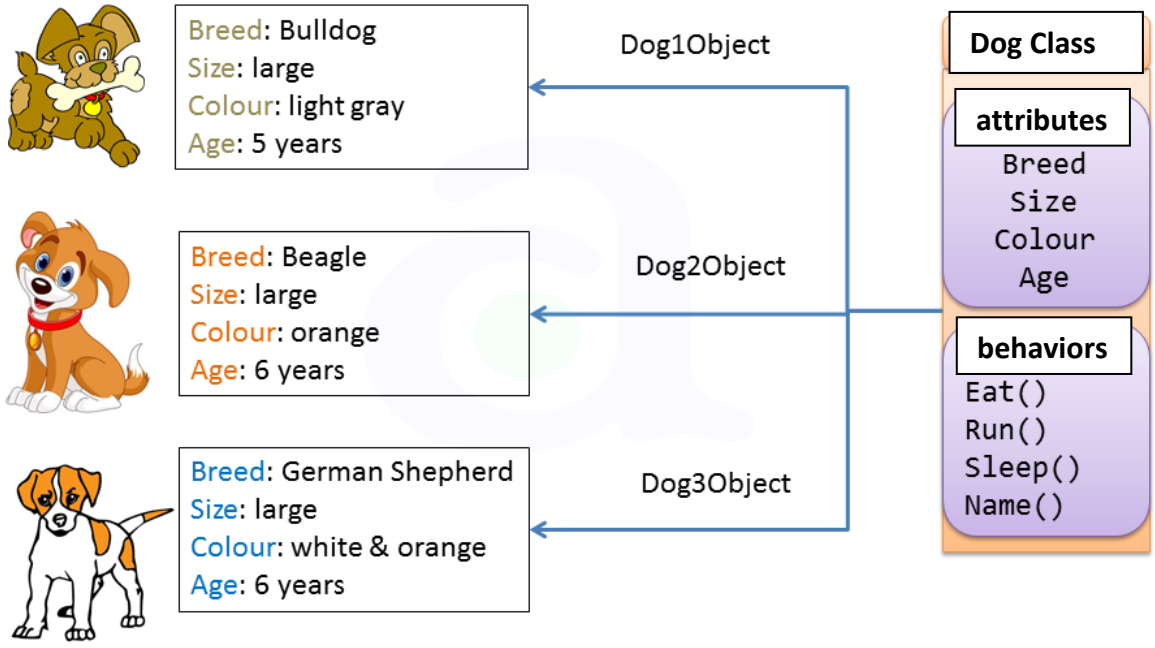
1. `_init_` means initialize this line **initializes** the class.

2. `self` is a necessary parameter that is always used with classes in python. You must include it and use it in the way shown above to create "attributes" or variables in your class. `name` and `age` are called "**attributes**" of the Dog class.

3. We can use the Dog class to create specific dog "**objects**" that have names, ages, or any other **attributes** we wish.



Some important preliminary vocabulary in object oriented programming:



Class – Blueprint for objects

Object – an **instance** or unique item created from a **Class**

Attributes – characteristics of an object

Methods/Behaviors – functions or mini programs that can be run inside a class.

An instance – an object created from a class.

Exercise #1

Create a Dog class by entering the following code into your IDE.
Notice that we have created 4 **attributes** for this **class**: name, age, breed, furcolor
Also note that for classes we need to use **4 spaces to indent**.

```
class Dog:
    def __init__(self, name, age, breed, furcolor):
        self.name = name
        self.age = age
        self.breed = breed
        self.furcolor = furcolor
```

Now the fun begins! Create 3 dog **objects** using the Dog **class** in the following way:

```
d1=Dog("Skippy",3,"German-Shepard","black")

d2=Dog("Scamp",4,"Golden retriever","Golden")

d3=Dog("Ginger",10,"Mix","Brownish-red")
```

Now try the following:

```
print(d1.furcolor)
print(d2.breed)
print(d3.name)
```

You should see that you have now created a Dog **class** that can creates dog **objects** that contain information (**attributes**) about each dog organised in a nice tidy way! Pretty cool!

Class attributes

Class attributes are attributes that have the **same** value for **all** class instances.

Exercise #2

Type the following into your IDE to see if how **class attributes** work

```
class Dog:

      #class attribute:
      species = "Canis familiaris"

    def __init__(self, name, age, breed, furcolor):
        self.name = name
        self.age = age
        self.breed = breed
        self.furcolor = furcolor

print(Dog.species)
print(d1.species)
print(d2.species)
```

You should notice that **ALL objects** or **instances** of the Dog **class** will have a species of Canis familiaris.

Species is a **class attribute**.

A common use of a **Class attribute** is when you need to count the number of objects/instances that are being created. Try **typing in** the code below to see how this might work:

```
class Dog:

      #class attribute:
      species = "Canis familiaris"
      count_instances=0

      def __init__(self,name):
            self.name = name
            Dog.count_instances = Dog.count_instances + 1

d1=Dog("Pickles")
d2=Dog("Rupert")
d3=Dog("Sophie")

print(Dog.count_instances)
```

Object Methods/Behaviors

Method or **Behaviors** are simply functions that are defined inside a class and can only be called from an **object** of that class.

Example:

```
class Dog:

      #class attribute:
      species = "Canis familiaris"

    def __init__(self, name, age, breed, furcolor):
        self.name = name
        self.age = age
        self.breed = breed
        self.furcolor = furcolor
        self.sound = sound

    def protect_master(self, sound):
       return (sound+" ")*5 + "!!!"
```

Above is a **Method** called `protect_master` (in the Dog class) that will make the dog bark when it's master needs protection.

It requires a new variable called sound that must be added when we call the protect_master method.

Exercise #3

Enter the new **method** shown above into your code and then try the following lines:

```
print(d1.protect_master("Bark"))
print(d2.protect_master("Arf"))
```

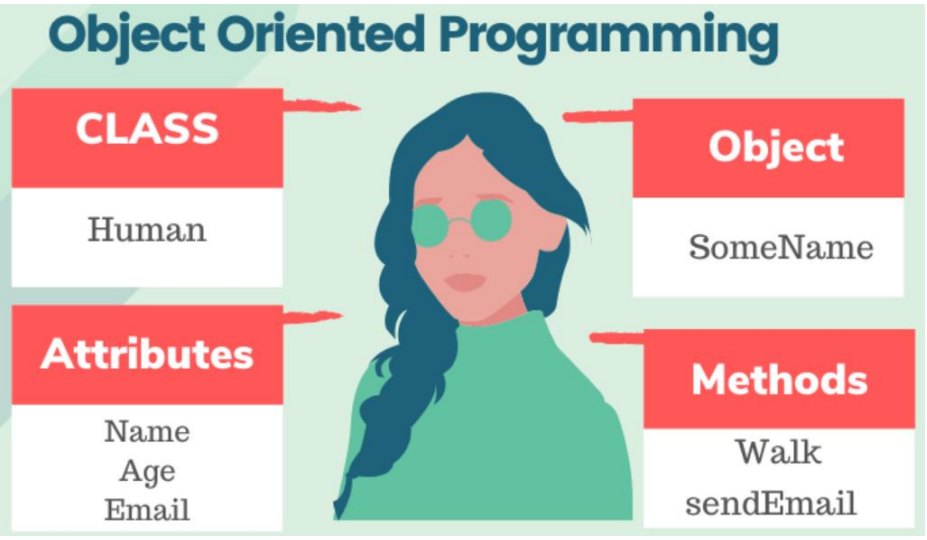
Exercise #4

Now we will create a new method for the Dog class that **will move the dog forward** from it's original y-position.

```
class Dog:
    def __init__(self, name, age, breed, furcolor, yposition=0):
        self.name = name
        self.age = age
        self.breed = breed
        self.furcolor = furcolor
        self.yposition = yposition

    def protect_master(self, sound):
        return (sound+" ")*5 + "!!!"

    def run_forward(self, steps):
        self.yposition=self.yposition+steps

d1=Dog("Skippy",3,"German-Shepard","black")

d2=Dog("Scamp",4,"Golden retriever","Golden")

d3=Dog("Ginger",10,"Mix","Brownish-red")


d1.run_forward(8)
d1.run_forward(3)
print(d1.yposition)
```

Now type the code above into an IDE and run it. Then create 3 more methods in the Dog class that will:

1. move the dog: **backwards** (decrease the y-position).
2. move the dog: **to the right** (increase it's x-position).
3. move the dog: **left** (decrease it's x-position).



Hopefully, now you are getting a sense of how **Object Oriented Programming** can help you complete tasks in a way that (in some cases) may be more organised and logical then **Procedural Programming**.

Before moving on to more complex aspects of using Object Oriented Programming lets solidify our understanding of its structure by practicing creating and playing with some more classes and objects.

Exercise #5

Type in the code below to use as a template for a **Bank Account Class** that can create bank accounts for users. **See next page for the rest of the exercise.**

```
class BankAccount:
    #using object oriented programming to create a bank account
    def __init__(self,balance=0):
        self.balance=balance

    def deposit(self,add):
        self.balance=self.balance+add

    def withdraw(self,out):
        self.balance=self.balance-out

jimbank=BankAccount(100)
print (jimbank.balance)
jimbank.deposit(400)
print (jimbank.balance)
```

Exercise #5 *continued*:

Now we will add the following items to your Bank Account class:

1. A **Class Attribute** called "Scotia Bank" that describes the institution where **all** the bank accounts have been created from.
2. At least 6 Objects **attributes** that be different for each object created. They may include things like:
    - owners name
    - account_number,
    - Type (checking, savings, etc),
    - Joint_account (yes/no)
    - Interest_rate

Now create several bank account **objects** from this class with the appropriate attributes: (`b1, b2, b2`,...)

Then call/print the value of specific object attributes to test them out. Examples:

```
print(b1.accout_number)
print(b3.interest_rate)
```

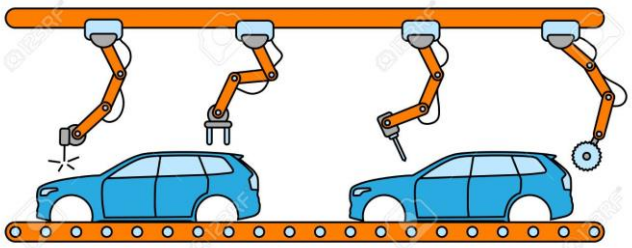
Exercise #5 *continued*:

Add at least 3 new **methods** to this class. They may include things like:

- Calculate_interest_earned
- Pay_a_bill
- Send_emoney_transfer

Make sure you call, run, and test your methods for particular objects.
Bonus if you can include if statements and/or nicely formatted text interface for the user. Save your work and submit.

Exercise #6 Car factory Exercise.



Click on the Car factory Exercise button **on the course page** for this section. You may **copy and paste** the code into Trinket (or any IDE that can output turtle graphics).

Now do the following (make sure you save and submit your work).

1. Run the code to see what it does. Look at all the lines of code after `car1.create()` and make sure you understand each line and what output it matches with.
2. Use the `Add_fuel()` method to fill up the tanks for `car1` and `car2`
3. Create a **class attribute** that keeps track of the amount of cars made.
4. Use the newly created class attribute in #3 to see how many cars have been created.
5. Create a new car object called `car4` with all the correct attributes.
6. Use the `create()`method to create a visual of `car4.`
7. Use the `Add_fuel()`method to fill the tank of `car4.`
8. Use the newly created class attribute in #3 to see how many cars exist now.
9. Create an additional instance **attribute** called `lights_on`. `lights_on` should be a Boolean variable that is equal to `True` or `False` (true means the car lights are on, false means lights are off). You may have a default value of false if you wish.
10. Create a new object **method** that will ask the user if they want the lights on or off. If the user says on make `lights_on=True` if the user says off make `lights_on=False`.
11. Turn the lights of car1 and car2 on and the lights for car3 and car4 off.
12. Create any new method of your choice relates to the actual function of a car and run the method for each vehicle. Examples: `move_forward, oil_level, gas_or_electric`. Have fun and be creative.

Exercise #7

A blast from the past.

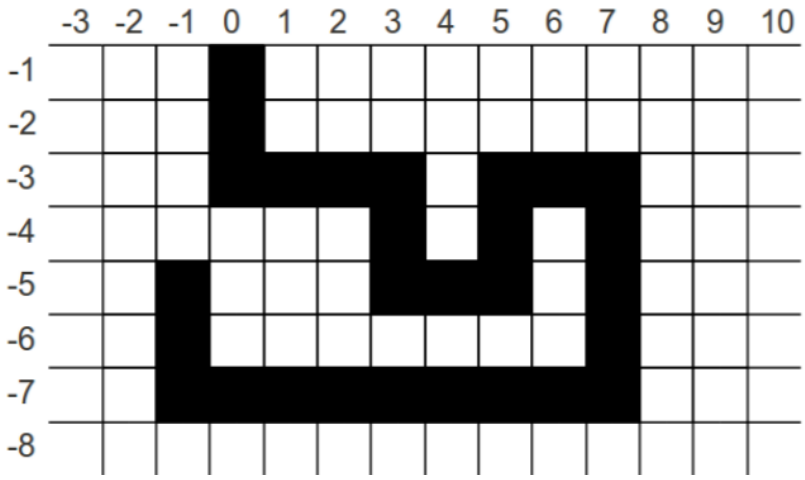
Try the following challenge from the junior CCC using **only object oriented programming**.

Problem Description

Boring is a type of drilling, specifically, the drilling of a tunnel, well, or hole in the earth. With some recent events, such as the Deepwater Horizon oil spill and the rescue of Chilean miners, the public became aware of the sophistication of the current boring technology. Using the technique known as geosteering, drill operators can drill wells vertically, horizontally, or even on a slant angle.

A well plan is prepared before drilling, which specifies a sequence of lines, representing a geometrical shape of the future well. However, as new information becomes available during drilling, the model can be updated and the well plan modified.

Your task is to write a program that verifies validity of a well plan by verifying that the borehole will not intersect itself. A two-dimensional well plan is used to represent a vertical cross-section of the borehole, and this well plan includes some drilling that has occurred starting at $(0, -1)$ and moving to $(-1, -5)$. You will encode in your program the current well plan shown in the figure below:



Input Specification

The input consists of a sequence of drilling command pairs. A drilling command pair begins with one of four direction indicators (`d` for down, `u` for up, `l` for left, and `r` for right) followed by a positive length. There is an additional drilling command indicated by `q` (quit) followed by any integer, which indicates the program should stop execution. You can assume that the input is such that the drill point will not:

- rise above the ground, nor
- be more than 200 units below ground, nor
- be more than 200 units to the left of the original starting point, nor
- be more than 200 units to the right of the original starting point

Output Specification

The program should continue to monitor drilling assuming that the well shown in the figure has already been made. As we can see $(-1, -5)$ is the starting position for your program. After each command, the program must output one line with the coordinates of the new position of the drill, and one of the two comments `safe`, if there has been no intersection with a previous position or `DANGER` if there has been an intersection with a previous borehole location. After detecting and reporting a self-intersection, your program must stop.

Sample Input 1

```
l 2
d 2
r 1
q 0
```

Output for Sample Input 1

```
-3 -5 safe
-3 -7 safe
-2 -7 safe
```

Sample Input 2

```
r 2
d 10
r 4
```

Output for Sample Input 2

```
1 -5 safe
1 -15 DANGER
```

Exercise #8

Click on the **make a deck of cards (OOP)** link on the course page. Follow the video *carefully* and make a deck of cards that you can shuffle and deal using OOP (as per the videos instructions). When you are done:

1. Create a player **class** and two player **objects**
2. Create a game **class** and a game **object** that will allow the players to play a simple card game like "war" or "black jack".

Exercise#9

Use **object oriented programming** to create a **text-based** game that involves a sport, adventure, or battle.

The game should have:

- One or two *user* controlled players.
- The players should be able to move around on a grid minimum size 10 x 10.
- Players can't move outside the boundaries of the grid.
- The players should each have several **attributes**: energy, strength, weapons, abilities etc.
- The players should have several **behaviors** examples: move different directions, jump, shoot, block, pick up, defend, etc.
- The board should have pre-positioned events or objects on the grid that can benefit or harm players examples: opponents, enemies, food, gold, weapons, sports drink, etc.
- A clear goal communicated to the player. Examples: get to a particular spot of the grid, get points, complete a particular task, etc.

Exercise#10 Inheitance

Click on the **sub-classes (inheritance)** link on course page. Watch the video *carefully* and *try running* some of the code the author is using to make sure you fully understand the idea of **inheritance** in OOP.

Now create a *similar* program to the one in the video except instead of using a **company** and **employees** as your subject. Use a **school** and **students**. Make sure in your code you clearly demonstrate you understanding of **inheritance.**