# ELU

Objects

Ricardo Pereira

# Motivation to use Objects

# Objects are a tool

Python is an object-oriented programming language. Object-oriented programming (OOP) focuses on creating reusable patterns of code, in contrast to procedural programming, which focuses on explicit sequenced instructions. When working on complex programs in particular, object-oriented programming lets you reuse code and write code that is more readable, which in turn makes it more maintainable. *reference: DigitalOcean*

- Remember, like any other tool there are times to use the tool and times not to
- The case where we will want to use objects are generally when we need to solve the following two problems:
  - We need to keep state
  - We need to reuse lots of code

# Keeping State

# What does it mean to keep state?

It means: to know the current status of something.

Keeping state = keeping track of it

# Examples of cases where we want to keep state

- There’s a few different types of state that we want to keep
  - how many people are in a room
  - If a switch is on or off
  - the state of a neuron
  - If a car is being used or not
  - The state of your current banking account

# Keeping state solves a different kind of problem

The functions that we've been working with during the week are all about just taking some input, computing something, and spitting out some output.

- In this type of problem there is a time component! Things are changing over time and we need to keep track of it.
- **It's not a one-off problem.**

# Code Reuse

# What does it mean to reuse code?

- Think about a factory making cars. Some parts of the car are the same regardless of what the model is. It would be stupid to “write the code” for each car if we have tools that allow us to share the functionalities.
- Programming is all about automating things
- If you find yourself as the human re-implementing the same thing over and over again, you should probably be looking for a way to reuse your code.

# So if you have a problem

- And the solution to the problem involves being able to keep state
- AND you have a clear need for code-reuse
- then...

# Objects may help you out

# Classes and more generally OOP

- OOP = Object Oriented Programming
- Is a very widely-used method in industry to enable state tracking and code-reuse that follows a conceptual model that is relatively simple

# Objects

The cornerstone of OOP

# What is an Object?

- Let’s consider an robot analogy for a moment to help us understand.
- Say you want to build lots of robots and then send them out into the world.
- These robots will have some functionality in common but will come in a few different flavors.

# What is an Object?

- The first thing that you'll need is some blueprints for the robots. It's what designs the robots and defines what they can do and what characterizes them.
- Then once you have your blueprints, you will need a factory to create individual robots.
- Once you have created a individual robots, you can now send it out into the world to do stuff.
    - Remember that the robots may be different but have many things in common such as being able to walk.

# What is an Object?

- After the two robots have been out in the world for some time, we will notice that they become a bit different.
- The two robots have seen different things and have done different things.
- Each one now has its own state.

# Important vocabulary in **bold**

- In our analogy:
  - The blueprint is the **class**
  - An individual robot is an **instance** of the class

# Let’s let this sink in

- Class - Defines what can be done, acts as the blueprint
- To instantiate - To create an instance of your class
- Instance - An individual and unique entity

# So let’s make a Robot

# The syntax

- Here is a class that does nothing but claims to be a Robot
- Notice the use of the class keyword
- Then the name of the class
- Then a colon
- Then indent 4 spaces

```
class Robot:
    pass
```

Some of this is starting to feel familiar, right?

# Now let’s give the robot the ability to speak

Intended 4 spaces, we have defined a function called speak()

The way that this function is defined “on” the Robot class.

Notice that it’s because of the indentation! If you did not indent 4 spaces, it would not be part of the robot!

```
class Robot:

    def speak(self):
        print('Hello, world')
```

# Now let’s give the robot the ability to speak

The function is pretty much the same except that it has one argument that we’re not using called **self**. Don’t worry about this for now, just accept that it’s required and must be there.

```
class Robot:

    def speak(self):
        print('Hello, world')
```

Now we have a defined a Robot class that does something! Right?

Wrong! We have not! We have only defined the blueprint!

# So let’s fire up the factory and make it real

After you have defined a class, you can instantiate it with the following syntax:

It’s the same as calling a method! Look at that!

Now inside of the variable robot, you have an instance of the class Robot.

```
class Robot:

    def speak(self):
        print('Hello, world')

# Robot is the class
# robot is the instance
robot = Robot()
```

Now let’s make it talk

# Instantiate and use!

Remember, before we can do anything with the Robot, we have to create an instance of it.

The we can call the speak method that is defined “on” the robot.

And we can see the output as if it was defined on a regular function that we already know and love

```
In [2]: robot = Robot()

In [3]: robot.speak()
Hello, world
```

# Alright everyone, make it happen

Implement it and run that bad boy!

```
class Robot:

    def speak(self):
        print('Hello, world')

# Robot is the class
# robot is the instance
robot = Robot()

# we call methods on the instance
robot.speak()
```

# One more thing

What happens if we run that last line?

```
class Robot:

    def speak(self):
        print('Hello, world')

# Robot is the class
# robot is the instance
robot = Robot()

# we call methods on the instance
robot.speak()

# what happens when we run this?
Robot.speak()
```

# Your class' functions can take arguments

But they always must go after the **self**

Implement it!

```
class Robot:

    def speak(self, who):
        print('Hello,', who)

# Robot is the class
# robot is the instance
robot = Robot()

# we call methods on the instance
robot.speak('sam')
```

Now, about keeping state

# Let’s give the robot some state-keeping abilities

- Let’s say that we now want to give the robot the ability to listen
- And that when he listens to something, he records it
- So the more things he listens to over time, the more he will keep recorded. A.k.a the more state he will have.

# What’s the right data structure?

- Let’s say that the robot can listen to one string at a time
- Remember, he needs to keep adding strings

What kind of data structure do we know of that works well for this?

# So let’s give him an instance variable

All kinds of new stuff here!

- We now have a initializer
- We are using the **self** argument to create a new list
- We have a function that takes a string and appends it to the list
- We have another function that prints the recordings

```
class Robot:

    def __init__(self):
        self.recordings = []

    def listen(self, string):
        self.recordings.append(string)

    def play_recordings(self):
        print(self.recordings)
```

# The initializer

This really messed up looking class function is a special function that behaves a bit different than the others.

This function is called right after the robot leaves the factory in which he is created

Anyone remember what the factory for the robot looks like?

```
class Robot:

    def __init__(self):
        self.recordings = []
```

# When instantiated, the initializer is called

Creating a new instance of the Robot, causes the initializer to be called!

```
class Robot:

    def __init__(self):
        self.recordings = []

r = Robot()
```

# Let’s watch this happen

Change the initializer to print something and then see what happens.

```
class Robot:

    def __init__(self):
        print('fresh out of the factory')

r = Robot()
```

Everybody implement this now!

# Let’s watch this happen

Notice that we didn’t do anything with the robot at all! We just created a new one and yet we can see that this code is being executed!

```
class Robot:

    def __init__(self):
        print('fresh out of the factory')

r = Robot()
```

```
In [10]: robot = Robot()
fresh out of the factory
```

# So going back to the use case

- Coming out of the factory it has the ability to listen but it does not yet have the ability to record because it doesn't have a device to record things onto.
- We can think of the constructor or this robot as installing some storage that it can record to.

```
class Robot:

    def __init__(self):
        self.recordings = []

    def listen(self, string):
        self.recordings.append(string)

    def play_recordings(self):
        print(self.recordings)
```

# Going back to the self

- As we keep state over time, we will need a way to access the hard drive that is unique to a single instance of a robot.
- That is what the self is for. It gives you access to the current instance state.

```
class Robot:

    def __init__(self):
        self.recordings = []

    def listen(self, string):
        self.recordings.append(string)

    def play_recordings(self):
        print(self.recordings)
```

# Exercises

# #1 - 2 robots listening

- Instantiate 2 robots from the last slide
  - Remember that you'll need to store the two different instances in two different variables
- Have each of them listen to some different things
- Play the different recordings from the two instances

- What happens when you play the recordings from each of them?
- Are the recordings different or the same? Can you explain why?

# #2 - Add a new capabilities to your Robot

- Add the capability to your robot to delete all of its recordings in a method called *delete_recordings*.
- Then test it by
  - Instantiating a new robot
  - Have it listen to some things
  - Delete the recordings
  - Then play the recordings to make sure that they are gone
  - Have it listen to some more things
  - Play the recordings to make sure that they are getting the new ones but not the old ones

# #3 - 2 robots deleting things

- Instantiate 2 Robots
- Tell each of them different things
- Delete the recordings from one of them
- Play the recordings of both of them

What happened when you played the recordings? When you delete one robot’s recordings, does the other get deleted as well?

# #4 - From scratch

Now you will write your own class.

- Call it a *LectureRoom*
- Give it one instance variable called *capacity* that is the number of students that the room can hold that is initialized to 40
- Give the ability to increase or decrease the capacity with functions called *increase_capacity* and *decrease_capacity*
    - Each of these functions takes one argument called *amount* which is an integer
    - You then increase or decrease the capacity by the *amount* given
    - Extra restrictions:
        - The room cannot have a capacity more than 100
        - The room cannot have a capacity less than 10
        - If someone tries to set the capacity to an invalid amount, you should print an error telling them that it is not valid