

# Programming fundamentals with Python

Pepe García

2020-04-20

# Programming fundamentals with Python



SCHOOL OF  
SCIENCE &  
TECHNOLOGY

# Plan for today

Understand **Object Oriented Programming**

Learn the difference between **objects** and **classes**

Modeling data with classes

Modelling functionality with **methods**



# Object Oriented Programming

**OOP** is a programming paradigm that uses objects to encapsulate code and data

**OOP** is really useful for us because we can map almost everything we know to an object, we just need to know its attributes and its functionality.

# Object Oriented Programming

**Classes** are templates for objects

**Objects** are instances of classes

**Attributes** describe the characteristics of an object

**Methods** model functionality in an object



**Classes** are **descriptions for objects**, which we can instantiate later.

Classes can describe the **attributes** of an object, the functions it has (called **methods**)



We use the **class** keyword to declare classes:

```
class Car:  
    pass
```



We can create an indefinite number of objects from the class:

```
car1 = Car()
```

```
car2 = Car()
```

```
#...
```

```
car23493 = Car()
```

```
# What will this print?
```

```
print(car1 == car2)
```





When we create classes we are creating a new type with them!

```
car = Car()
```

```
print(type(car))
```



# Constructing objects

The construction of objects from a class is handled by something called the **init** method



# Constructing objects

```
class Car:
    def __init__(self):
        pass
```



# Constructing objects

The `init` method is executed when we instantiate the object, and we can use it to add attributes to our class:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```



# Constructing objects

It is a common thing to add validation logic to the **init** method.

```
class Car:
    def __init__(self, brand, model):
        if brand != "Ford" or brand != "Audi":
            raise ValueError("Don't know how to create a {}".format(brand))

        self.brand = brand
        self.model = model
```



Create a class that represents a clock. It should contain **hours** and **minutes** only.

Validate that the attributes passed to the constructor make sense.



# Constructing objects

```
mondeo = Car("Ford", "Mondeo")  
a3 = Car("Audi", "A3")  
  
print(a3.brand)  
print(mondeo.model)
```



Methods model functionality in our objects.

They are functions that receive a `self` parameter before others:





```
class Car:
    started = False

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start_engine(self):
        self.started = True

    def stop_engine(self):
        self.started = False
```



```
car = Car("Ford", "Mondeo")
```

```
print(car.started)
```

```
car.start_engine()
```

```
print(car.started)
```

```
car.stop_engine()
```

```
print(car.started)
```



**class** != **object**

**attribute** != **method**

**init** is for constructing objects



<https://classroom.github.com/a/uVnKYU94>



Let's model a **Rock band**:

- RockBand
  - add\_member()
  - rehearse()
- Member
  - name
  - instrument
  - play()



- Learn about inheritance
- visibility and encapsulation



With inheritance we can create hierarchies of classes that share **attributes** and **methods**.



# Inheritance



SCHOOL OF  
SCIENCE &  
TECHNOLOGY



When declaring a class that **extends** another class (meaning it **inherits** from it), we put the parent class between parentheses

```
class ClassName(Parent):  
    pass
```



# Inheritance

When we inherit classes, the methods from the parents are inherited too!

```
class Vehicle:
    def start(self):
        print("BRROOOMMMMM!")
```

```
class Car(Vehicle):
    pass
```

```
car = Car()
car.start()
```

Here we declare a normal class, with a simple **start** method

So we can use methods from the parent in the child class



Something else we can do with **Inheritance** is method overriding.

**Method overriding** allows us to change the behavior of methods in a child class, let's see an example.

# Method overriding

```
class Vehicle:
    def start(self):
        print("BRROOOMMMMM!")
```

```
class Car(Vehicle):
    pass
```

```
class Tesla(Car):
    def start(self):
        print("blip!")
```

```
vehicle = Vehicle()
vehicle.start()
```

```
car = Car()
car.start()
```

```
tesla = Tesla()
```



Create a class **Polygon** with a method **calculate\_area()**.

Create two subclasses of it, **Square** and **Circle** that override the **calculate\_area** method.



Now that we're introducing bigger class hierarchies, we need to be aware of the differences between **type()** and **isinstance()**



# type vs isinstance

**type** returns the class we used to instantiate the object

```
type(tesla) == Vehicle  
# returns False
```

```
isinstance(tesla, Vehicle)  
# returns True
```

**isinstance** returns True if the class is in the hierarchy of the object



# type vs isinstance





# Creating our own Exceptions

We can create our own exceptions by creating a class that inherits from an exception:

```
class FormValidationError(ValueError):  
    pass  
  
if not is_valid(email):  
    raise FormValidationError()
```



# Calling methods from the parent class

Inside a class, we can use the **super()** function in order to access the parent class.

**super()** is very useful to extend the functionality of methods in the superclass.



# Calling methods from the parent class

```
class Polygon:
    def __init__(self, name):
        self.name = name

class Triangle():
    def __init__(self, base, height):
        super().__init__("triangle")
        self.base = base
        self.height = height
```



We use encapsulation to hide the internal state of an object from the outside.

Attributes or methods that are hidden from the outside, we call them **private**.

In order to declare a method or attribute as private, we use the prefix **\_\_** (**double underscore**)



# Encapsulation

```
class Person:
    def __init__(self, name):
        self.__name = name
```

```
pepe = Person("Pepe")
```

```
pepe.__name
```

*#AttributeError: 'Person' object has no attribute '\_\_name'*

The class **Person** has a **private attribute** **\*\*\_\_name.\*\***



# Encapsulation

But, what do we do if we want to access a private attribute from the outside?

A common technique is to use a **getter method**

```
class Person:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name
```

```
pepe = Person("Pepe")
pepe.get_name()
# "Pepe"
```



<https://realpython.com/python3-object-oriented-programming>

<https://www.py4e.com/html3/14-objects>

