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**

SCHOOL OF SCIENCE & TECHNOLOGY

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.

SCHOOL OF SCIENCE & TECHNOLOGY

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

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

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

Classes

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

```
class Car:
    pass
```

Classes

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

```
car1 = Car()
car2 = Car()
#...
car23493 = Car()

# What will this print?
print(car1 == car2)
```

SCHOOL OF SCIENCE & TECHNOLOGY

Classes

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
```

SCHOOL OF SCIENCE & TECHNOLOGY

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
```

SCHOOL OF SCIENCE & TECHNOLOGY

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 {}".f

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

SCHOOL OF SCIENCE & TECHNOLOGY

Practice

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)
```

SCHOOL OF SCIENCE & TECHNOLOGY

Methods

Methods model functionality in our objects.

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

Methods

```
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
```

SCHOOL OF SCIENCE & TECHNOLOGY

Methods

```
car = Car("Ford", "Mondeo")
print(car.started)

car.start_engine()
print(car.started)

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

SCHOOL OF SCIENCE & TECHNOLOGY

Recap

class != **object**

attribute != **method**

init is for constructing objects

Homework

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

SCHOOL OF SCIENCE & TECHNOLOGY

Methods

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

- RockBand

- add_member()

- rehearse()

- Member

- name

- instrument

- play()

Session 2

- Learn about inheritance
- visibility and encapsulation

SCHOOL OF SCIENCE & TECHNOLOGY

Inheritance

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

SCHOOL OF SCIENCE & TECHNOLOGY

Inheritance

SCHOOL OF
SCIENCE &
TECHNOLOGY

Inheritance

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

Method overriding

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()
```

SCHOOL OF SCIENCE & TECHNOLOGY

Practice

Create a class **Polygon** with a method **calculate_area()**.

Create two subclasses of it, **Square** and **Circle** that override the **calculate_area** method.

SCHOOL OF SCIENCE & TECHNOLOGY

type vs isinstance

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

SCHOOL OF SCIENCE & TECHNOLOGY

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

SCHOOL OF SCIENCE & TECHNOLOGY

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()
```

SCHOOL OF SCIENCE & TECHNOLOGY

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.

SCHOOL OF SCIENCE & TECHNOLOGY

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
```

SCHOOL OF SCIENCE & TECHNOLOGY

Encapsulation

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)**

SCHOOL OF SCIENCE & TECHNOLOGY

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.**

SCHOOL OF SCIENCE & TECHNOLOGY

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"
```

SCHOOL OF SCIENCE & TECHNOLOGY

Reading materials

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

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