# OOPS Concepts in Python

Object-oriented Programming, or *OOP* for short, is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm)which provides a means of structuring programs so that properties and behaviors are bundled into individual *objects*.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an Animal() class to track properties about the Animal like the name and age.

It’s important to note that a class just provides structure—it’s a blueprint for how something should be defined, but it doesn’t actually provide any real content itself. The Animal() class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal’s name or age is.

**Python Objects (Instances)**

While the class is the blueprint, an *instance* is a copy of the class with *actual*values, literally an object belonging to a specific class. It’s not an idea anymore; it’s an actual animal, like a dog named Roger who’s eight years old.

Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains actual information relevant to you.

You can fill out multiple copies to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, we must first specify what is needed by defining a class.

Instance Attributes

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the \_\_init\_\_() method to initialize (e.g., specify) an object’s initial attributes by giving them their default value (or state). This method must have at least one argument as well as the selfvariable, which refers to the object itself (e.g., Dog).

class Dog:

# Initializer / Instance Attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

Similarly, the self variable is also an instance of the class. Since instances of a class have varying values we could state Dog.name = name rather than self.name = name. But since not all dogs share the same name, we need to be able to assign different values to different instances. Hence the need for the special selfvariable, which will help to keep track of individual instances of each class.

**NOTE**: You will never have to call the \_\_init\_\_() method; it gets called automatically when you create a new ‘Dog’ instance.

**Python Object Inheritance**

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called *child classes*, and the classes that child classes are derived from are called *parent classes*.

It’s important to note that child classes override *or* extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent’s attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an object, which generally all other classes inherit as their parent.

Extending the Functionality of a Parent Class

Create a new file called *dog\_inheritance.py*:

# Parent class

class Dog:

# Class attribute

species = 'mammal'

# Initializer / Instance attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instance method

def description(self):

return "{} is {} years old".format(self.name, self.age)

# instance method

def speak(self, sound):

return "{} says {}".format(self.name, sound)

# Child class (inherits from Dog class)

class RussellTerrier(Dog):

def run(self, speed):

return "{} runs {}".format(self.name, speed)

# Child class (inherits from Dog class)

class Bulldog(Dog):

def run(self, speed):

return "{} runs {}".format(self.name, speed)

# Child classes inherit attributes and

# behaviors from the parent class

jim = Bulldog("Jim", 12)

print(jim.description())

# Child classes have specific attributes

# and behaviors as well

print(jim.run("slowly"))

Overriding the Functionality of a Parent Class

Remember that child classes can also override attributes and behaviors from the parent class. For examples:

>>>

>>> class Dog:

... species = 'mammal'

...

>>> class SomeBreed(Dog):

... pass

...

>>> class SomeOtherBreed(Dog):

... species = 'reptile'

...

>>> frank = SomeBreed()

>>> frank.species

'mammal'

>>> beans = SomeOtherBreed()

>>> beans.species

'reptile'

The SomeBreed() class inherits the species from the parent class, while the SomeOtherBreed() class overrides the species, setting it to reptile.