Object Oriented Programming in Python

Topics

[Introduction to Objects](#_Introduction_to_Objects)

[Creating Classes](#_Creating_Classes)

[Inheritance](#_Inheritance)

[Overloading Methods](#_Overriding_Methods)

[Static and Class Methods](#_Static_Methods)

[Private and Public Classes](#_Private_and_Public)

Reference: <https://www.techwithtim.net/tutorials/python-programming/classes-objects-in-python/private-and-public-classes/>

## Introduction to Objects

### What is an Object?

In python almost everything we create and use is an object. Pretty much any time we declare a variable we are creating a new object of a certain class. Different objects have certain properties and they inherit those properties from the class they belong to. For example objects of type str have methods like .strip() and .split(), Integers can be added and lists can be indexed. These are all properties specific to objects of certain classes.

In this series I will be showing you how we can create our own classes and objects.

**Help() Function**

To be able to have a look at some of the builtin functions in python we can use the help() function. The help function will list all of the methods and attributes of a class and give us descriptions on what everything does.

help**(**int**)**

*# Try running this and see what happens*

**Definitions**

**Instance:** Whenever we create a new object we are said to be creating an instance of a class. For example typing the command x = 1 translates to: creating a new instance of the int class with the value 1 and the name x.

**Method:** You can think of a method as a function that is specific to certain objects and classes. Methods are created within a class and are only visible to instances of that class. An example of a method is .strip(). It can only be used on objects of class str as it is specific to the str class. All methods must be called on an instance of a class. We cannot simply type strip() as we need to include an instance followed by a period before it.

**Attributes:** An attribute is anything that is specific to a certain object. For example the object has an attribute color. We can change that color and modify it and if we create a new turtle object it can have a different color.

# Creating Classes

When we create a class we are essentially creating our own data type. To do this we must first come up with a name for our class and populate it with some methods.

**class** Dog**():**

**def** \_\_init\_\_**(**self**):**

**pass**

**def** speak**(**self**):**

**pass**

The name of this class is "Dog" and currently it has two methods: \_\_init\_\_ and speak.

Typically when we create a class we create a method that is known as a **constructor**. A constructor is what will be called automatically when we create a new instance of that class. To create a constructor we must use the method name \_\_init\_\_(). We do not need to call this method by doing something like "instance.\_\_init\_\_()" because when we first create a Dog object it will be automatically called.

To create a new instance of our Dog class we can do the following:

**class** Dog**():**

**def** \_\_init\_\_**(**self**):**

**print(**"Created a dog"**)**

**def** speak**(**self**):**

**pass**

tim **=** Dog**()** *# This will print "Created a dog" because \_\_init\_\_ will automatically be called*

## The Self Keyword

You may have noticed that each of my methods above contain the keyword self as a parameter. For now all of the methods we make must do just this. When we call methods on an instance of the class the name of that instance is automatically passed to the method as the argument self. This allows us to access and change attributes that are specific to this instance.

To create a new attribute we must use the self keyword in the following way.

**class** Dog**():**

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

self**.**name **=** name *# self.name is now an attribute of the Dog class and has a specific value for each instance*

**def** speak**(**self**):**

**pass**

tim **=** Dog**(**"Tim"**)**

Since the \_\_init\_\_ methods now takes two arguments (self and name) we must pass a name when we create a new Dog object.

Since this name is specific to each instance when we can create multiple Dog objects they all may have different names.

**class** Dog**():**

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

self**.**name **=** name *# self.name is now an attribute of the Dog class and has a specific value for each instance*

**def** speak**(**self**):**

**print(**self**.**name**)**

tim **=** Dog**(**"Tim"**)**

dog2 **=** Dog**(**"Fred"**)**

tim**.**speak**()** *# Prints "Tim"*

dog2**.**speak**()** *# Prints "Fred"*

## Changing Attributes

Now that we know how to create attributes it's time to talk about accessing and changing them. We can change attributes in two ways. It is best practice to change an attribute within the class like so.

**class** Dog**():**

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

self**.**name **=** name

self**.**age **=** age

**def** speak**(**self**):**

**print(**"I am"**,** self**.**name**,** "and I am"**,** self**.**age**,** "years old"**)**

**def** change\_age**(**self**,** age**)**

self**.**age **=** age *# This changes the age attribute*

tim **=** Dog**(**"Tim"**,** **5)**

tim**.**change\_age**(7)**

tim**.**speak**()** *# This will print "I am Tim and I am 7 years old"*

The second way to change attribute is to do so outside of the class, like so:

**class** Dog**():**

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

self**.**name **=** name

self**.**age **=** age

**def** speak**(**self**):**

**print(**"I am"**,** self**.**name**,** "and I am"**,** self**.**age**,** "years old"**)**

tim **=** Dog**(**"Tim"**,** **5)**

tim**.**age **=** **7**

tim**.**speak**()** *# This will print "I am Tim and I am 7 years old"*

**Note:** We can add new attributes to a class from outside of the \_\_init\_\_ method. However, we must do this within another method in the class. (Watch video if confused)

# 

# Inheritance

When we create classes we can inherit methods and attributes from other already existing classes. This allows us to create a variety of different sub-classes or child classes based off of what is known as a parent class or super class.

The following words are all equivalent and mean a class that inherits from another:  
- child class  
- subclass  
- derived class  
- concrete class

The following words are all equivalent and mean a class that is inherited from:  
- parent class  
- super class  
- abstract class

When we inherit from a class all the methods and attributes of the parent class are passed down to the child class.

**class** Animal**():**

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

self**.**name **=** name

self**.**age **=** age

**def** speak**(**self**):**

**print(**"I am"**,** self**.**name**,** "and I am"**,** self**.**age**,** "years old"**)**

**class** Dog**(**Animal**):**

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

self**.**name **=** name

self**.**age **=** age

self**.**type **=** "dog"

*# Since we inherit from the animal class we can use the method speak on Dog objects*

tim **=** Dog**(**"Tim"**,** **5)**

tim**.**speak**()** *# This will print "I am Tim and I am 5 years old"*

## Calling a Super-Classes Constructor

Often times when we create sub-classes we want to initialize them in a similar same way as the parent class. To save ourselves from repeating code we can call the parents \_\_init\_\_ function from inside of our childs class \_\_init\_\_ like so:

**class** Animal**():**

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

self**.**name **=** name

self**.**age **=** age

**def** speak**(**self**):**

**print(**"I am"**,** self**.**name**,** "and I am"**,** self**.**age**,** "years old"**)**

**class** Dog**(**Animal**):**

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

super()**.**\_\_init\_\_**(**name**,** age**)** *# This will call the Animal classes constructor method*

tim **=** Dog**(**"Tim"**,** **5)**

tim**.**speak**()** *# This will print "I am Tim and I am 5 years old"*

# Overriding Methods

Sometimes when we inherit from a parent class we want to have methods or attributes that have the same name as a method in the parent class but that have a different functionality. If we create a method or attribute inside of our child class with the same name as one in the parent it will override the parent class.

**class** Animal**():**

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

self**.**name **=** name

self**.**age **=** age

**def** speak**(**self**):**

**print(**"I am"**,** self**.**name**,** "and I am"**,** self**.**age**,** "years old"**)**

**class** Dog**(**Animal**):**

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

super**.**\_\_init\_\_**(**name**,** age**)** *# This will call the Animal classes constructor method*

**def** speak**(**self**):**

**print(**"I am a Dog"**)**

tim **=** Dog**(**"Tim"**,** **5)**

tim**.**speak**()** *# This will print "I am a Dog"*

### Example

Below is a realistic example of inheritance and where you may use it.

**class** Veichle**:**

**def** \_\_init\_\_**(**self**,** price**,** color**):**

self**.**color **=** color

self**.**price **=** price

self**.**gas **=** **0**

**def** fillUpTank**(**self**):**

self**.**gas **=** **100**

**def** emptyTank**(**self**):**

self**.**gas **=** **0**

**def** gasLeft**(**self**):**

**return** self**.**gas

**class** Truck**(**Veichle**):**

**def** \_\_init\_\_**(**self**,** price**,** color**,** tires**):**

super**().**\_\_init\_\_**(**price**,** color**)**

self**.**tires **=** tires

**def** beep**(**self**):**

**print(**"Honk honk"**)**

**class** Car**(**Veichle**):**

**def** \_\_init\_\_**(**self**,** price**,** color**,** speed**):**

super**().**\_\_init\_\_**(**price**,** color**)**

self**.**speed **=** speed

**def** beep**(**self**):**

**print(**"Beep Beep"**)**

# Overloading Methods

We often take for granted the fact that you can use operators like +, -, == on python builtin data types. However, in reality this functionality has actually been coded into the classes by python. This means that we can code this functionality into our own classes by creating some special methods.

Take for an example the following class and objects:

**class** Point**():**

**def** \_\_init\_\_**(**self**,** x**=0,** y**=0):**

self**.**x **=** x

self**.**y **=** y

self**.**coords **=** **(**self**.**x**,** self**.**y**)**

**def** move**(**self**,** x**,** y**):**

self**.**x **+=** x

self**.**y **+=** y

p1 **=** Point**(3,** **4)**

p2 **=** Point**(3,** **2)**

p3 **=** Point**(1,** **3)**

p4 **=** Point**(0,** **1)**

If we would like to compare two points for equality we would have to do something like this:

isSame **=** p1**.**x **==** p2**.**x **and** p1**.**y **==** p2**.**y

This is far from elegant and is extremely inefficient. To solve this problem we can overload the default python method \_\_eq\_\_.

**class** Point**():**

**def** \_\_init\_\_**(**self**,** x**=0,** y**=0):**

self**.**x **=** x

self**.**y **=** y

self**.**coords **=** **(**self**.**x**,** self**.**y**)**

**def** move**(**self**,** x**,** y**):**

self**.**x **+=** x

self**.**y **+=** y

**def** \_\_eq\_\_**(**self**,** other**):**

**return** self**.**x **==** other**.**x **and** self**.**y **==** other**.**y

p1 **=** Point**(3,** **4)**

p2 **=** Point**(3,** **2)**

p3 **=** Point**(1,** **3)**

p4 **=** Point**(0,** **1)**

*# Now we can compare points using ==*

isSame **=** p1 **==** p2

**print(**isSame**)** *# Prints False*

There are tons of other python default methods that we can overload. Some of the most used are featured below.

**class** Point**():**

**def** \_\_init\_\_**(**self**,** x**=0,** y**=0):**

self**.**x **=** x

self**.**y **=** y

self**.**coords **=** **(**self**.**x**,** self**.**y**)**

**def** move**(**self**,** x**,** y**):**

self**.**x **+=** x

self**.**y **+=** y

**def** \_\_add\_\_**(**self**,** other**):**

**return** Point**(**self**.**x **+** other**.**x**,** self**.**y **+** other**.**y**)**

**def** \_\_sub\_\_**(**self**,** other**):**

**return** Point**(**self**.**x **+** other**.**x**,** self**.**y **+** other**.**y**)**

**def** \_\_mul\_\_**(**self**,** other**):**

**return** self**.**x **\*** other**.**x **+** self**.**y **\*** other**.**y

p1 **=** Point**(3,** **4)**

p2 **=** Point**(3,** **2)**

p3 **=** Point**(1,** **3)**

p4 **=** Point**(0,** **1)**

p5 **=** p1 **+** p2

p6 **=** p4 **-** p1

p7 **=** p2**\***p3

Now you may notice that when we try to print one of our point objects we get some cryptic text that looks like this



This is because we have not defined how our point should be represented as a string. To do this we must overload the \_\_str\_\_ method.

**class** Point**():**

**def** \_\_init\_\_**(**self**,** x**=0,** y**=0):**

self**.**x **=** x

self**.**y **=** y

self**.**coords **=** **(**self**.**x**,** self**.**y**)**

**def** move**(**self**,** x**,** y**):**

self**.**x **+=** x

self**.**y **+=** y

**def** \_\_str\_\_**(**self**):**

**return** "Point(" **+** str**(**self**.**x**)** **+** ',' **+** str**(**self**.**y**)** **+** ")"

p1 **=** Point**(3,** **4)**

**print(**p1**)** *# This prints Point(3, 4)*

We can also overload the methods below to implement the comparison operators.

**class** Point**():**

**def** \_\_init\_\_**(**self**,** x**=0,** y**=0):**

self**.**x **=** x

self**.**y **=** y

self**.**coords **=** **(**self**.**x**,** self**.**y**)**

**def** move**(**self**,** x**,** y**):**

self**.**x **+=** x

self**.**y **+=** y

**def** length**(**self**):**

**import** math

**return** math**.**sqrt**(**self**.**x **\*\*** **2** **+** self**.**y**\*\*2)**

**def** \_\_gt\_\_**(**self**,** other**):** *# greater than*

**return** self**.**length**()** **>** other**.**length**()**

**def** \_\_ge\_\_**(**self**,** other**):** *# greater than or equal to*

**return** self**.**length**()** **>=** other**.**length**()**

**def** \_\_lt\_\_**(**self**,** other**):** *# less than*

**return** self**.**length**()** **<** other**.**length**()**

**def** \_\_le\_\_**(**self**,** other**):** *# less than or equal to*

**return** self**.**length**()** **<=** other**.**length**()**

*# We are going to compare points based on their lengths*

p1 **=** Point**(3,** **4)**

p2 **=** Point**(3,** **2)**

p3 **=** Point**(1,** **3)**

p4 **=** Point**(0,** **1)**

isLess **=** p1 **<=** p2 *# This is False*

**print(**isLess**)**

To see a list of all over-loadable methods please [click here.](https://www.techwithtim.net/tutorials/python-programming/classes-objects-in-python/overloading-methods/www.siafoo.net/article/57)

# Static Methods

Static methods are methods within a class that have no access to anything else in the class (no self keyword or cls keyword). They cannot change or look at any object attributes or call other methods within the class. They can be thought of as a special kind of function that sits inside of the class. When we create a static method we must use something called a decorator. The decorator for a static method is "@staticmethod".

**class** myClass**:**

**def** \_\_init\_\_**(**self**):**

self**.**x **=** x

**@staticmethod**

**def** staticMethod**():**

**return** "i am a static method"

*# Notice staticMethod does not require the self parameter*

# Class Methods

Class methods are methods within a class that only have access to class variables and other class methods. They are passed the name of the class and therefore can access anything within the class. Like static methods they cannot access any instance attributes. You can create a class method by using the "@classmethod" decorator.

**class** myClass**:**

count **=** **0**

**def** \_\_init\_\_**(**self**):**

self**.**x **=** x

**@classmethod**

**def** classMethod**(**cls**):**

cls**.**count **+=** **1**

*# The classMethod can access and modify class variables. It takes the class name as a required parameter*

Please refer to the video for further explanation and more detailed examples.

## Private and Public

In other programming languages there is the notion of private and public classes and methods.  
A private class is something that can only be accessed from within a certain file or directory and a private method is something that can only be called from within the class.  
A public class or method is something that can be accessed anywhere.

However, In python this does not exist. Every class and method in python is public and there is no way to change that. We can only simulate creating private classes and methods by using certain notation and conventions.

To declare something as private we use one underscore before the name.

**class** \_Private**:**

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

self**.**name **=** name

**class** NotPrivate**:**

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

self**.**name **=** name

self**.**priv **=** \_Private**(**name**)** *# Even though we decalre something private we can still call and us it*

**def** \_dispaly**(**self**):** *# Private*

**print(**"Hello"**)**

**def** display**(**self**):** *# Public*

**print(**"Hi"**)**

The reason we declare things as private is to tell the programmer not to use them. It is somewhat a warning to the programmer saying that this class or method is private and that they shouldn't mess with it.

Now that you are done learning about classes and objects I recommend you to move onto [intermediate python programming.](https://techwithtim.net/tutorials/python-programming/intermediate-python-tutorials)