Python oop

**OOP and the Web**

As we saw in the previous video, OOP allows us to create a blueprint for the purpose of reproducing objects. While games are great use cases for OOP, there are many places you'll see OOP in your work as a web developer. Like in game development, OOP is used in web development as a way of producing many copies of similar objects. Truly understanding Object Oriented Programming can be very challenging. We'll build one piece at a time, from basic to complex. Don't give up if it doesn't make sense at first. Understanding OOP is key to being successful as a web developer, particularly when using frameworks like Django.

Okay, enough high-level talk. Let's create something useful. I'd like to build a class called User.

**User**

**Note: The following code is the finished implementation of our blueprint for creating instances of users. We will build this together as we go, so don't worry if you don't understand every line quite yet.**

Let's say we want to create a web app and will need lots of instances of users. We can start from scratch and create each user, but this clearly isn't going to scale. Maybe we could have some sort of system for creating a user based upon a set of instructions. What attributes and actions are common across all instances of users?

It may seem like having a blueprint means that each user must be the same. The blueprint will only dictate what qualities (attributes) we must track for each user and what each user object should be able to do. When a new user is created, a value must be filled in for the name attribute.

This is the finalized code of the class User that we are going to write incrementally throughout the following tabs:

class User(object):

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

self.name = name

self.email = email

self.logged = True

def login(self):

self.logged = True

print self.name + " is logged in."

return self

def logout(self):

self.logged = False

print self.name + " is not logged in"

return self

def show(self):

print "My name is {}. You can email me at {}".format(self.name, self.email)

return self

# Class Structure

You can easily create a class in Python by typing the keyword class followed by the name of your class and **(object)**.

class User(object):

pass

Copy

You should notice that we define one parameter object. When the parameter for a class is object it simply means that this class inherits from the object class. We'll get into inheritance a little more later on. You'll notice the keyword pass in the body - it is used as a placeholder where some code is required. It means do nothing.

More generally, a class looks like this:

class ClassName(object):

#attributes and methods here (we'll talk more about these in a moment)

Copy

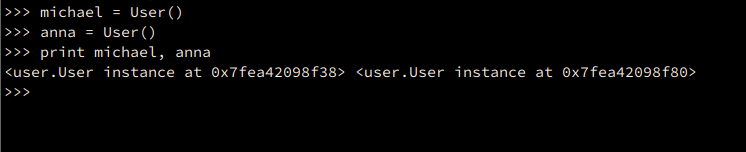
Think of the class as a blueprint for creating something. Once we've finished our blueprint we can create **instances** of this class. The User class we defined above is a blueprint for creating users. We create a new instance by using the class name as if it were a function. Let's go ahead and make instances of our User class

michael = User()

anna = User()

Copy

Let's print these and see what the output is:



The output we see here gives you information about what class your object belongs to and where it is stored in memory. Later, we'll talk about how to manipulate the way this output is displayed.

It can be helpful to think of a class as a blueprint and an object as the thing we make based on that blueprint.

Classes include two types of things:

* **Attributes:** Characteristics shared by all instances of the class type. Take our User class, for example. All users have a name and an email. You might be wondering how each user can have a different name and email. We'll show you in the following tab.
* **Methods:** Actions that an object can perform. A user, for example, might be able to make a purchase. A method is like a function that belongs to a class. It's some instructions that will only work when called on an object that has been created from that class. We'll show you how shortly.

In the following tabs we'll talk more about attributes and methods. Remember, objects have two important features: storage of information and ability to execute some logic.

## Attributes and Methods

Objects can store two different types of information: attributes and methods. Attributes are characteristics of an object, while methods are things an object can do. For example: I am a person. Like other people I have hair. Like other people I can walk. In our User example a user has an email and a password. A User can login, logout and register. Here's how our user class looks when we add some attributes and methods:

# declare a class and give it name User

class User(object):

# the \_\_init\_\_ method is called every time a new object is created

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

# set some instance variables. just like any variable we can call these anything

self.name = name

self.email = email

self.logged = False

# this is a method we created to help a user login

def login(self):

self.logged = True

print self.name + " is logged in."

return self

#now create an instance of the class

new\_user = User("Anna","anna@anna.com")

print new\_user.email

Copy

We've already been over the first line. We know how to declare a class. In the next line we declare the \_\_init\_\_ method. The init method is called every time a new object is created. Above, you'll notice that the new object creation includes passing parameters. Anything we pass in here is passed into the init method: new\_user = User("Anna","anna@anna.com")

We'll talk about init in more detail later. Next, we'll set some values that belong to each object to be equal to the values we passed in: self.email = email

More about self later. After the init method is another method called login. All the method does for now is print a string. The logic we need to log a user into a web app is pretty complicated. It requires a database, for one. We'll be writing the code for that later in Flask.

* Python / Django

* Python OOP

* Python OOP

* Self and Init



[Hello, Cadence C](http://learn.codingdojo.com/m/7/3187/18926)

[CHECKLIST](http://learn.codingdojo.com/m/7/3187/18926)

# Self and \_\_init\_\_()

To review:

1. **A class:** Instructions for how to build many objects that share characteristics.
2. **An object:** A data type built according to specifications provided by the class definition.
3. **An attribute:** A value. Think of an attribute as a variable that is stored within an object.
4. **A method:** A set of instructions. Methods are functions that are associated with an object. Any function included in the parent class definition can be called by an object of that class.

### Self

In the previous tab you might have noticed that the first parameter in our method was "self": def login(self,id):. This will be true for any method we create. However, when we call methods, we do not have to pass in any arguments. This is known as implicit passage of self. We can now change the state of the single object by making modifications only to self. The self parameter includes all the information about the individual object that has called the method. Without self, every time we changed one object's attributes, we'd change the attribute for all the items of that type. Try running the following code:

class User(object):

name = "Anna"

anna = User()

print "anna's name: ", anna.name

User.name = "Bob"

print "anna's name after change:", anna.name

bob = User()

print "bob's name:", bob.name

Copy

I can do this instead to change the name attribute for a single instance: anna.name = "Anna". But imagine having to do so for each instance you create. There's an easier way! This is where \_\_init\_\_() comes in!

### \_\_init\_\_()

Python's \_\_init\_\_ method is something that's known as **a magic method**. Magic methods are automatically created and sometimes invoked when a new instance of a class is created. One you'll be working with in most classes you build is \_\_init\_\_.

\_\_init\_\_ is useful because it allows us to set some attributes when a new instance is created. Because we know that the init method will run immediately, we can pass in some parameters to that \_\_init\_\_ method upon object creation:

class User(object):

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

self.name = name

self.email = email

self.logged = False

user1 = User("Anna Propas", "anna@anna.com")

print user1.name

print user1.logged

print user1.email

Copy

Above, we are able to create objects that share characteristics but are still individual. When the object is created we can specify the values we'd like to assign to each attribute. In the \_\_init\_\_ method we are assigning the values we passed in as values of the attributes of each individual object.

Note: We can create two objects that are identical in regards to the data they store, but they will be different. This is because each object will be stored in different places in memory and is referred to according to that unique placement.

The above concepts can be challenging to learn and confusing at first. It'll get easier over time, but first you have to challenge yourself to learn something tough. Stick with it!

## Chaining Methods

So far, you have learned how to create objects and use methods within that object. Looking back at the User example, you might create something like this:

user1.login()

user1.show()

user1.logout()

Copy

This takes up a lot of space and we're repeating our call to user1 many times. There's a way to do this just once and keep attaching new method calls to the end of the previous one, like so:

user1.login().show().logout()

Copy

This might look familiar. That's because you saw something like this when you worked with jQuery. The way we're able to do this with Python is by asking each method to return self. That means, if you remember back to your work with functions, that each method call will now be equal to the instance that called it.

For example if user1.login() returned its own instance (user1) then you could chain another method to user1.login(), doing something like user1.login().show()

If you remember from the previous tab, self refers to the individual instance of the object. We can allow chaining of methods like so:

class User(object):

def login(self):

// your code goes here...

return self

Copy

**"return self"** returns its own instance allowing you to chain methods after calling that method.

The practice of having OOP return its own instance is pretty common and this can be done with other languages (JavaScript, Ruby, PHP, C++, Java) as well. Note that in some languages, you do not **return self**, rather you **return this**.

# Modules & Packages

## Modules

Modules are simply Python files with the .py extension which implement a set of functions. Modules are imported using the import command.

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

Now if we want to import the urllib module, which enables us to create read data from URLs, we can simply import the module:

# import the library

import urllib

# NOTE: there's a urllib version for each version of Python:

# Use urllib2 if you're using Python 2

# Use urllib3 for Python 3

# Finally, use it...

urllib.urlopen(...)

Copy

Notice how we used urllib as a variable to refer to our module and then we called the functions using dot notation.

### Creating Your Own Modules

Writing your own Python modules is very simple. To create a module, we first create a new .py file with the module name in the same directory as the file that will import the module. Then we import it using the import command and the Python file name (without the .py extension)

For example, let's create a module of arithmetic operations:

#file name: arithmetic.py

def add(x, y):

return x + y

def multiply(x, y):

return x \* y

def subtract(x, y):

return x - y

Copy

We can import arithmetic module and run the functions by doing this...

import arithmetic

print arithmetic.add(5, 8)

print arithmetic.subtract(10, 5)

print arithmetic.multiply(12, 6)

Copy

Note: just make sure that the module and the file importing the module are in the same folder/directory.

### Standard (Built-In) Modules

Python comes with a library of standard modules. Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built-in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the winreg module is only provided on Windows systems. One particular module deserves special mention: sys, which is built into every Python interpreter.

List of built-in modules could be found in this [link](https://docs.python.org/2/library/index.html).

#### Exploring Built-In Modules

Two very important functions come in handy when exploring modules in Python - the dir and help functions. We can look for which functions are implemented in each module, by using the dir function:

>>> import urllib

>>> dir(urllib)

['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE', 'URLopener', '\_\_all\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_version\_\_', '\_ftperrors', '\_get\_proxies', '\_get\_proxy\_settings', '\_have\_ssl', '\_hexdig', '\_hextochr', '\_hostprog', '\_is\_unicode', '\_localhost', '\_noheaders', '\_nportprog', '\_passwdprog', ...

>>> help(urllib) # will return a listing of information on the given module

Copy

## Packages

A module is a single file (or files) that are imported under one import. A **package** is a collection of modules in directories that give a package hierarchy.

from my\_package.subdirectory import my\_functions

Copy

Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

sample\_project

|\_\_\_\_\_ python\_file.py

|\_\_\_\_\_ my\_modules

|\_\_\_\_\_ \_\_init\_\_.py

|\_\_\_\_\_ test\_module.py

|\_\_\_\_\_ another\_module.py

|\_\_\_\_\_ third\_module.py

Copy

In the above diagram, the package name is my\_modules.

#### Writing Packages

Each package in Python is a directory which MUST contain a special file called **\_\_init\_\_.py.** This file can be empty, and it indicates that the directory containing it is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called my\_modules, which marks the package name, we can then create a module inside that package called test\_module. We also must not forget to add the ***\_\_init\_\_.py***file inside the my\_modules directory.

To use the module test\_module, we can import it in two ways:

import my\_modules.test\_module

Copy

**OR**

from my\_modules import test\_module

Copy

The \_\_init\_\_.py file can also decide which modules this package will export as an API, while keeping other modules internal, by overriding the \_\_all\_\_ variable, like so:

\_\_init\_\_.py:

\_\_all\_\_ = ["test\_mod

**class User**

By now, you should have a better understanding of the user class we showed you at the beginning of this chapter!

So far we've learned how to do a handful of things:

* Define new classes
* Add attributes that describe the state of our objects
* Add methods that describe the behavior of our objects
* initialize our instances with the \_\_init\_\_() method
* naming and passing default parameters
* Import and create new modules that we can use in our files/classes

Here is our User class with some comments to help you review:

# instantiate class User

class User(object):

# this method to run every time a new object is instantiated

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

# instance attributes

self.name = name

self.email = email

self.logged = True

# login method changes the logged status for a single instance (the instance calling the method)

def login(self):

self.logged = True

print self.name + " is logged in."

return self

# logout method changes the logged status for a single instance (the instance calling the method)

def logout(self):

self.logged = False

print self.name + " is not logged in"

return self

# print name and email of the calling instance

def show(self):

print "My name is {}. You can email me at {}".format(self.name, self.email)

return self

Inheritance

Inheritance is forming new classes using classes that have already been defined. In other words, it allows one class to take on some or even all of its attributes and methods from a parent class. The benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) can override or extend the functionality of base classes (ancestors).

Now, let's look back on our Bike and Car classes. What do Bike and Car have in common? They are both vehicles. We'll create a parent vehicles class and these subclasses: classes Bike, Car, and Airplane. Creating subclasses is useful when you want to create a more specialized version of a current class that you have. If it is just a specialized class, it means that the current functionality will mostly remain the same except for some minor adjustments. By using class inheritance, we don't have to repeat ourselves. Here's how our vehicle class might look:

class Vehicle(object):

def \_\_init\_\_(self, wheels, capacity, make, model):

self.wheels = wheels

self.capacity = capacity

self.make = make

self.model = model

self.mileage = 0

def drive(self,miles):

self.mileage += miles

return self

def reverse(self,miles):

self.mileage -= miles

return self

Copy

Implicit Inheritance

Now we want to create new (sub)classes that are able to use all of the function and properties in the Vehicle class but also have some of their own additional functions and properties.

Classes Car, Bike and Airplane all inherit the blueprint of class Vehicle, but in addition, they add their own blueprint. Now to create our new classes in a file called new\_classes.py:

class Vehicle(object):

def \_\_init\_\_(self, wheels, capacity, make, model):

self.wheels = wheels

self.capacity = capacity

self.make = make

self.model = model

self.mileage = 0

def drive(self,miles):

self.mileage += miles

return self

def reverse(self,miles):

self.mileage -= miles

return self

class Bike(Vehicle):

def vehicle\_type(self):

return "Bike"

class Car(Vehicle):

def set\_wheels(self):

self.wheels = 4

return self

class Airplane(Vehicle):

def fly(self, miles):

self.mileage += miles

return self

v = Vehicle(4,8,"dodge","minivan")

print v.make

b = Bike(2,1,"Schwinn","Paramount")

print b.vehicle\_type()

c = Car(8,5,"Toyota", "Matrix")

c.set\_wheels()

print c.wheels

a = Airplane(22,853,"Airbus","A380")

a.fly(580)

print a.mileage

Copy

When we defined each of our classes, we typed Bike(Vehicle), Car(Vehicle), and Airplane(Vehicle). You could read each of these like "Make a class Bike/Car/Airplane that inherits from Vehicle". This is what is known as the implicit inheritance which allows us to use inherited attributes and methods of the Vehicle(parent) class in our new subclasses.

A general skeleton for implicit inheritance:

class Parent(object): # inherits from the object class

# parent methods and attributes here

class Child(Parent): #inherits from Parent class so we define Parent as the first parameter

# parent methods and attributes are implicitly inherited

# child methods and attributes

# Multiple Arguments

What if you want to pass in a variable number of arguments, or want to capture multiple arguments into a single parameter? Placing an asterisk before the name of the parameter after the "normal'' parameters does just that. The asterisk is called a 'splat' operator.

def varargs(arg1, \*args):

print "Got "+arg1+" and "+ ", ".join(args)

varargs("one") # output: "Got one and "

varargs("one", "two") # output: "Got one and two"

varargs("one", "two", "three") # output: "Got one and two, three"

Copy

In this example, the first argument is assigned to the first method parameter as usual. However, the next parameter is prefixed with an asterisk(the "splat" operator we just introduced), which bundles the remaining arguments into a new **tuple**, which is then assigned to that parameter.

If we tested the type of **args** inside our function using type(args) we would get:

def varargs(arg1, \*args):

print "args is of " + str(type(args))

varargs("one", "two", "three") # output: args is of <type 'tuple'>

Copy

**Note** the .join() method is called on a string that glues the values in the tuple together. For example, the tuple of arguments ('two', 'three') was joined as 'two, three' when we called ', '.join(args) .

# Super

Sometimes in your OOP code, you will want to create updated versions of methods that are defined in the parent class, because in addition to your custom code you want specifically to call the parent implementation of that method as well (or instead). In these cases, you would reference that parent object with the keyword ' **super**'. Specifically you reference that parent's method by calling '**super(ChildClassName, self).parent\_method()**'.

## Parent \_\_init\_\_

One thing we may want to do is call the Parent class's \_\_init\_\_ method, but also have our Child class change attributes defined by its Parent class. Say that we wanted each of our sub-classes (Wizard, Ninja, Samurai) to still inherit the attributes of the parent Human class but have more developed attributes than the average Human.  We could do that like this:

from human import Human

class Wizard(Human):

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

super(Wizard, self).\_\_init\_\_() # use super to call the Human \_\_init\_\_ method

self.intelligence = 10 # every wizard starts off with 10 intelligence

def heal(self):

self.health += 10

class Ninja(Human):

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

super(Ninja, self).\_\_init\_\_() # use super to call the Human \_\_init\_\_ method

self.stealth = 10 # every Ninja starts off with 10 stealth

def steal(self):

self.stealth += 5

class Samurai(Human):

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

super(Samurai, self).\_\_init\_\_() # use super to call the Human \_\_init\_\_ method

self.strength = 10 # every Samurai starts off with 10 strength

def sacrifice(self):

self.health -= 5

Copy

If we create instances of each, we'll see that they have the same attributes as the typical Human (the parent class), but they also have beefed up attributes depending on which subclass we instantiated! Notice how we call the \_\_init\_\_ method of the super(parent). Other methods are called the same way with the exception of passing in variables. We left that exercise for you !