The __init__ Method and del Function

by Sophia

WHAT'S COVERED

In this lesson, we will explore initializing objects from classes and deleting objects when they are no longer needed. Specifically, this lesson covers:

1. The __init__ Method

Every time an object is created from a class (class instantiation), Python calls the `__init__` method, which is short for initialize. The **__init__ method**'s sole purpose is to initialize (assign values) to the object's attributes, and it is only used within a class. If you are familiar with other programming languages like Java and C++, the `__init__` method functions the same as those language's "constructors." We would not be able to set our own default values for the attributes without the use of the `__init__` method.

BIG IDEA

The `__init__` is not required but it is a good practice to use. If we want to set attributes to specific values and not depend on system-defined default values, we need to use this method.

The way that the `__init__` method name is written needs to be specific.

EXAMPLE

```
__init__
```

The name should be `__init__`. That is two underscore characters followed by the word "init" and then followed by two more underscore characters.

Here is the full syntax to create the `__init__` method.

EXAMPLE

```
def __init__(self, <otherParameters>):
```

You may recognize this syntax as it’s very similar to the creation of any function. First, we use the `def` keyword to define the method. Then comes the `__init__` initializer name, which is followed by parameters surrounded by parentheses and finally a colon (:).

The first parameter will always be `self` followed by any other parameters that need to be declared so the class can accept them.

Remember from the past lesson that this first parameter is typically called `self`. This `self` is a variable that represents the instance of the class (the object that’s being created at the moment). This variable binds the class’s attributes to the arguments that it is receiving from this new object.

A note on the name `self`: we can potentially change the name of `self` to anything we want. The name `self` has no special meaning to Python. Using the word `self` is an industry best practice and is typically used since it is self-explanatory of what it represents.

In the class `PeopleCounter` example within the last lesson, we did not include the `__init__` method. Here is the example again.

EXAMPLE

```
class PeopleCounter:
   x = 0

   def anotherOne(self) :
     self.x = self.x + 1
     print("So far",self.x)
```

Here we set the attribute value of `x` to 0. If we wanted to include different values for `x` on new instances, we would not be able to. We would need to include the `__init__` method to allow different values if that is what we intend when we first initialize the object.

Having a class with attributes with default value(s) is only unique to that class. So for our example `PeopleCounter` class, the variable `x` is only unique to the `PeopleCounter` class. Let’s create a new class called `User` that we can use for an application/program that creates user login information.

First, we’ll create a class called `User`. Then, we'll define the `__init__` method with the `self` parameter and two additional parameters, `uname` for username and `pword` for password. That way we can assign these variables whenever we want to create a user.

EXAMPLE

```
class User:
  def __init__(self, uname, pword):
```

This code won’t do anything specific yet since we started a method but did not define what would be next. At this point, if the code was executed, the `def __init__` method line executes and we would have an empty object named `self` inside of the class. The `uname` and `pword` parameters would hold whatever argument data that we're passing in, which we'll get to in a bit. This object that is currently empty with no data doesn't do any good. The next step, then, is to ensure that the unique information that we want to be in the object is assigned to it. This is done by assigning values to each of the object's attributes.

Remember that this is meant to be a `User` class about a user and we want to have attributes that reflect that kind of data. Let's say that we want each user to have a username attribute that contains the user's username along with a password attribute to store the password. We can define and populate those attributes by doing the following.

EXAMPLE

```
class User:
  def __init__(self, uname, pword):
    self.username = uname
    self.password = pword
```

The first indented line of our `__init__` method will create a new attribute named `username` for the new instance (the self or current instance) and will assign to it whatever argument value is passed into the `uname` attribute when the class is called. The second line creates an attribute named `password` for the new `self` instance and will assign to it whatever argument value is passed in as the `pword` attribute.

We used a couple of conventions when setting up our new class which is important to call out:

- The class name should be capitalized.
- Attributes, on the other hand, should use the same standard as used for creating variable names.

That completes our class creation using the `__init__` method. Next, we will create an instance of the class.

TERM TO KNOW

`__init__`
The `__init__` method's sole purpose is to initialize (assign values) to the object's attributes and it is only used within a class.

2. Creating an Object

Now that we have the class created, let's create some instances of it. The format would look something like this in relation to our defined `User` class.

instanceName = User('usernameValue','passwordValue')

We'll just need to replace the `instanceName` with the variable name that we choose and the `usernameValue` and `passwordValue` with their respective argument values. Let's say that we want to create a user called `account` with the username `sophia` and the password as `myPass`.

EXAMPLE

```
class User:
  def __init__(self, uname, pword):
    self.username = uname
    self.password = pword

account = User('sophia','mypass')
```

If we run this code as is, it'll create an instance of the `User` class passing the arguments `sophia` and `mypass` as argument values and assign that returned object to the variable account. It may help to see more details by printing this out.

EXAMPLE

```
class User:
  def __init__(self, uname, pword):
    self.username = uname
    self.password = pword

account = User('sophia','mypass')

print(account)
print(account.username)
print(account.password)
print(type(account))
```

Here is that output.

```
<__main__.User object at 0x7fca2528cf70>
sophia
mypass
<class '__main__.User'>
```

Let's break down what we are seeing in the output by each `print()` function.

First, we printed out the account variable where we can see that our variable account was created as an instance of the `User` object (class). We can also see where in memory the variable is stored, but you don't need to be concerned with that.

Next, we printed out the username, which we can see in this instance has the value of `sophia`, and the password, which has the value of `mypass`. Finally, we used the `type()` function on the account variable to determine that it is of type class (our `User` class).

BIG IDEA

Object and Instance

It is important to note that we'll see the term "object" and "instance" used through these lessons. They mean the same thing. It’s important to note that an instance of an object is just a way to hold information about an element that’s similar to other elements of the same class.

Properties and Attributes

The terms “ properties” and “attributes” mean the same thing as well. The element's attributes may be unique to other instances. For example, in an application, it would make sense for each of our users to have unique usernames. However, some users may have the same value for their password.

TRY IT

Directions: Try creating an instance/object from a class. Start with our example above and see if you can modify it to something unique for yourself.

3. Deleting an Object

Once we’ve created an object, the object remains in the computer’s memory. Think of it as being a task that you’re thinking about that has to be done. When you're completely finished with the task, in this case, the object, you may want to remove it from your memory. In order to delete an instance, you simply use the following syntax:

```
del <instanceName>
```

The **del keyword** is used to delete objects, and since everything in Python is an object, we can use this keyword to delete anything, including variables, lists, etc. So, if we go back to our `User` object that we created previously (`account`) and delete that object after we output the username, we can see what happens.

EXAMPLE

```
class User:
  def __init__(self, uname, pword):
    self.username = uname
    self.password = pword

account = User('sophia','mypass')
print(account)
print(account.username)
del account
print(account.password)
print(type(account))
```

Here is the new output.

```
<__main__.User object at 0x7072d4a0c370>
```

```
sophia
Traceback (most recent call last):
  File "/home/main.py", line 10, in <module>
    print(account.password)
NameError: name 'account' is not defined
```

Since we deleted the instance of the class after we output the username, our `account` instance no longer exists. This is why we get the exception that the account is not defined since it has been removed.

KEY CONCEPT

Deleting instances is not necessary to do since instances are automatically removed when we exit the block of code where a variable is declared. Delete is typically used if you need to manage memory for performance purposes. Otherwise, allowing Python to manage the objects for you will generally be completely acceptable.

TERM TO KNOW

del
The `del` keyword is used to delete objects, and since everything in Python is an object, we can use this keyword to delete anything, including variables, lists, etc.

SUMMARY

In this lesson, we learned about using the`__init__`method. The `__init__` method allows us to initialize and set up instance variables to be used within an object. We learned how to **create a new object** (instance) from our class, access the variables, and determine the type of the object. We also learned how to **delete an instance of an object** using the `del` keyword, although this is not something that we normally have to do ourselves.

Best of luck in your learning!

TERMS TO KNOW

__init__
The `__init__` method’s sole purpose is to initialize (assign values) to the object’s attributes and it is only used within a class.

del

The `del` keyword is used to delete objects, and since everything in Python is an object, we can use this keyword to delete anything, including variables, lists, etc.