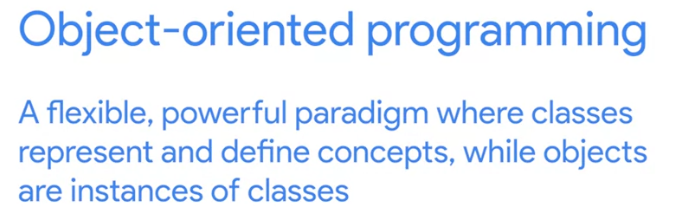
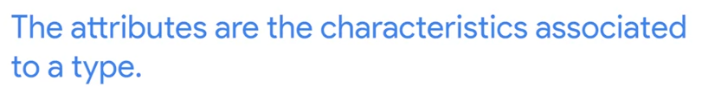
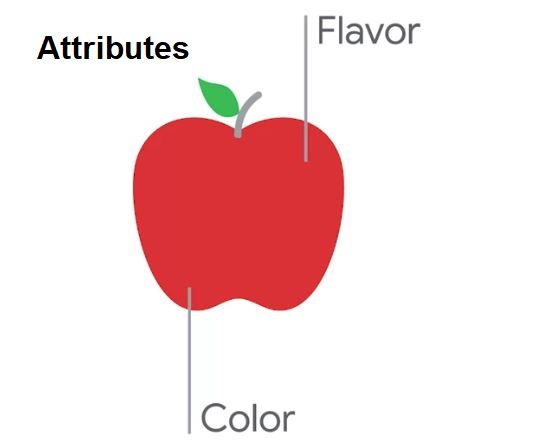
**Object-Oriented Programming**



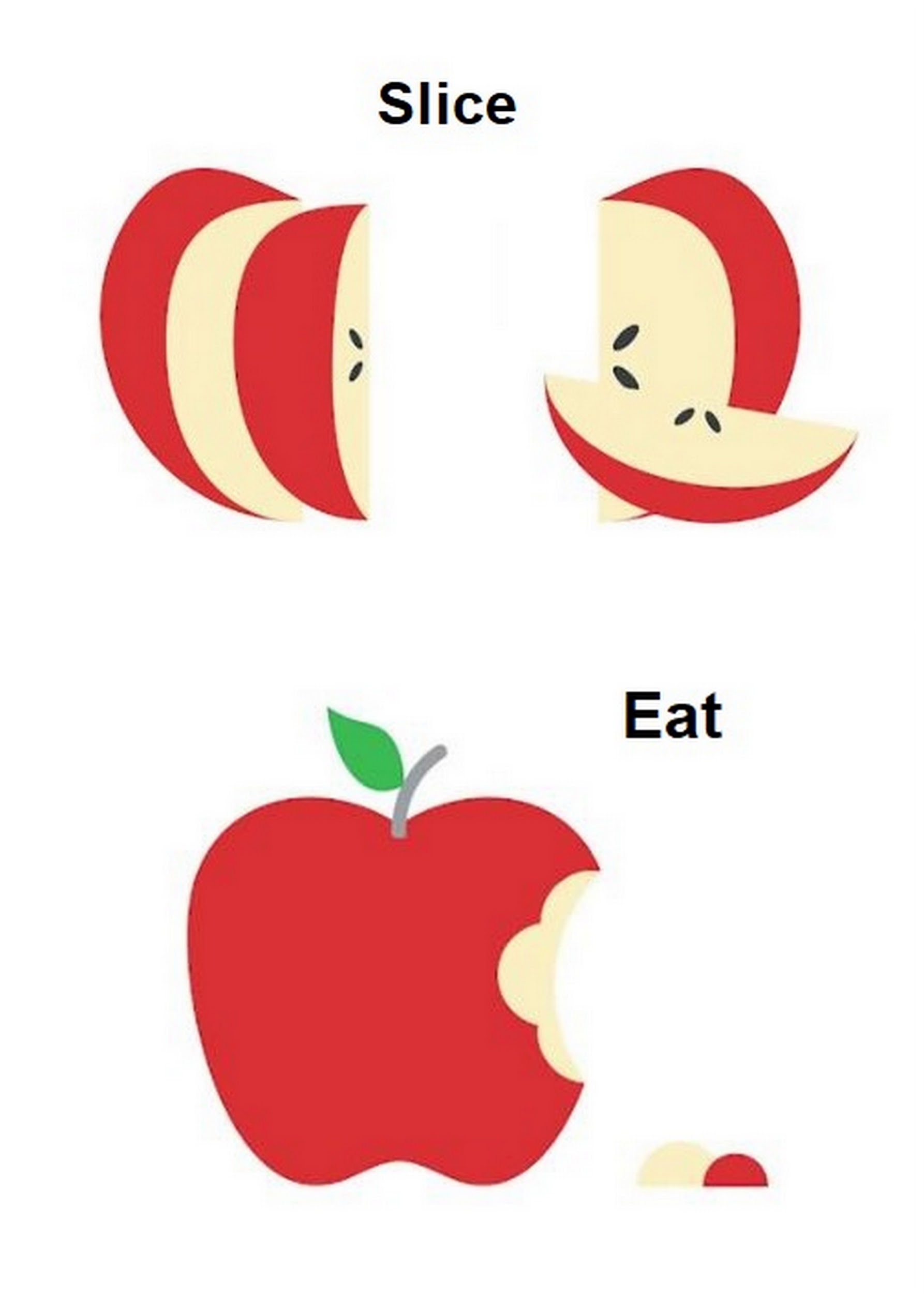




Using the apple example, ***attributes*** are the characteristics associated with the apple.



***Methods*** depend on what we want to do with the apple. Like ***Slice method*** and ***Eat method***.

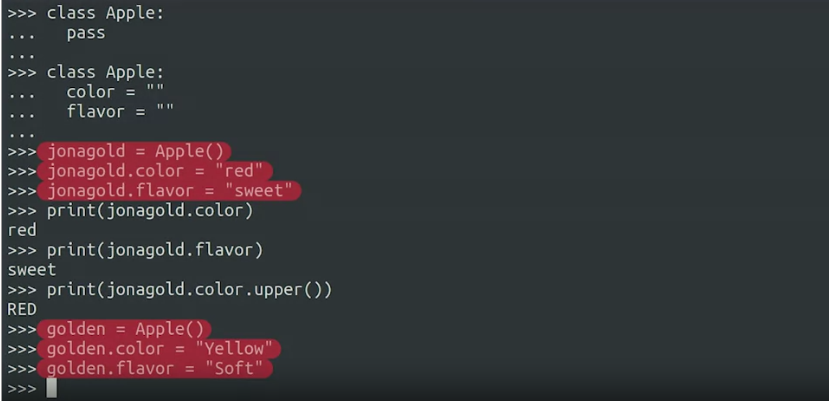


# 

# Classes and Objects in Python

To see the valid methods of a list, use ***dir(object)*** function. And to see how to use those methods, use ***help(object)*** function.

Defining New Classes

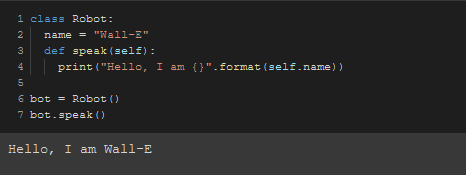


Both "jonagold" and "golden" are instances of the Apple class.

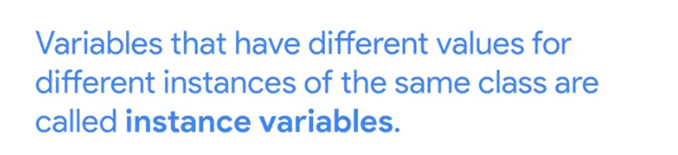
### **Classes and Methods**

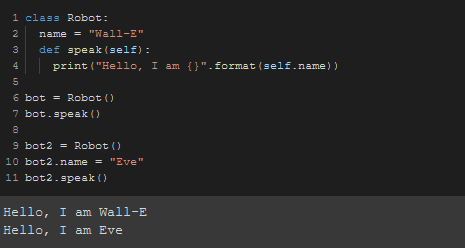
**Self:** self represents the instance of the class. By using the “self” keyword we can access the attributes and methods of the class in python.

It binds the attributes with the given arguments. The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes.



In this example, the speak method printed the name “Wall-E” which was the name attribute that we set.





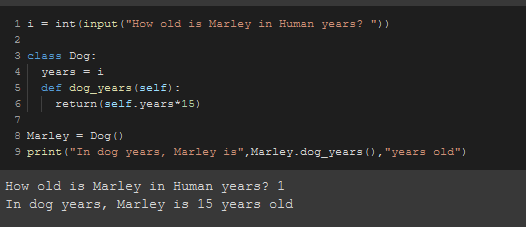
We've now created two instances of the Robot class each of them with their own name.

When calling this speak method each of them prints their name and not the other.

Since methods are just functions that belong to a specific class, they can work as any other function.

So they can receive more parameters and return values if needed.

Let's check out what a method returning a value looks like.



In this case, we've created a method that converts the age of a dog to dog years from human years.

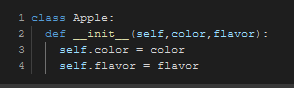
So the value that the method returns to change when we change the years attribute of our instance.

So as the value of the years attribute changes, the return value of the dog years method changes too.

# **Constructors and Other Special Methods**

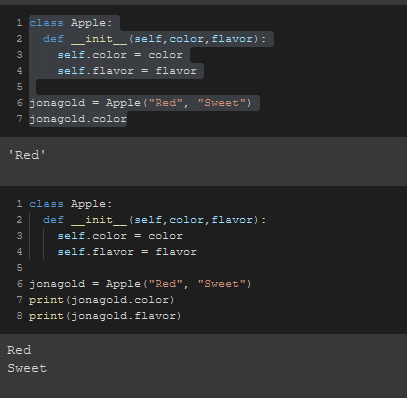
**A constructor is a special kind of method** that Python calls when it instantiates an object using the definitions found in your class.

The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the **\_\_init\_\_()** method is called the constructor and is always called when an object is created.

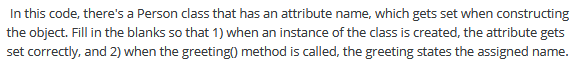


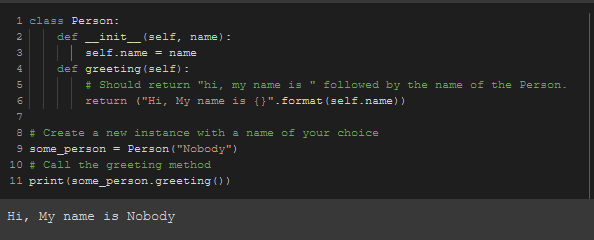
Here, we've defined a constructor, one very important special method.

This method on top of the self variable that represents the instance receives two more parameters: color and flavor.



So now by adding a constructor method that sets the attributes, we can create the class and have its values set right when it's created.



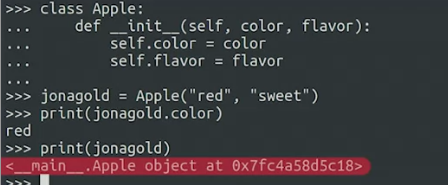


Constructors aren't the only special methods we can write. When we use the ***STR or print functions*** to convert an object to a string, we are using a super-useful special method.

But before we go ahead and define one, let's see what happens when we don't define it.

We just tried to print our apple instance, and we got a very weird message.

We have the words apple and object in there, but what's the rest of it?



Well, when we don't specify a way to print an object, Python uses the default method that prints

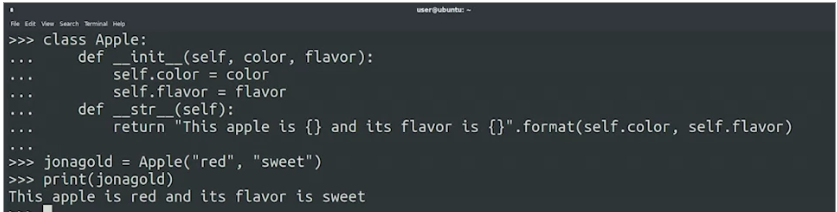
the position where the object is stored in the computer's memory. This is definitely not what we wanted.

If you ever try and print something and Python prints a random string of numbers and letters,

you'll know that it's likely using the default representation, which is the position of the object in the computer's memory.

So how do we tell Python to print something that makes sense for us?

We use the special STR method which returns the string that we want to print.



So the STR method lets us print a friendly message instead of a bunch of numbers.

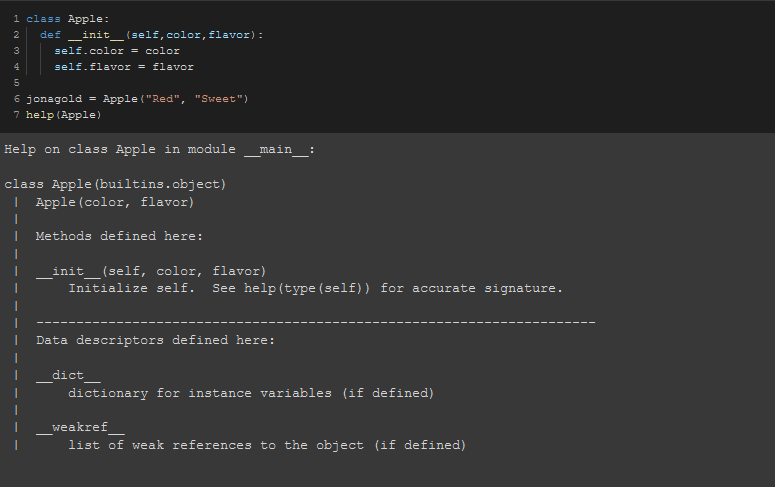
In general, it's a good idea to think ahead and define the STR method when creating objects that you want to print.

# **Documenting Functions, Classes, and Methods (Optional)**

The world of classes and methods can be a little puzzling when you're still learning your way around, and that's why the help function can come in handy. You might remember that we can still use the Python function help to find documentation about classes and methods.

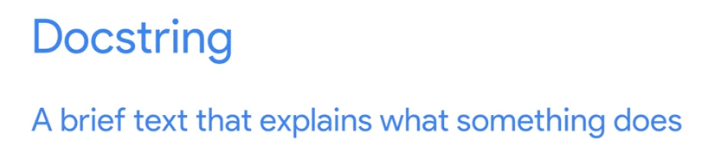
We can also do this on our own classes, methods, and functions.

Using the old ***“Apple”*** example again,

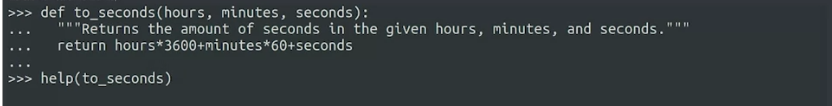


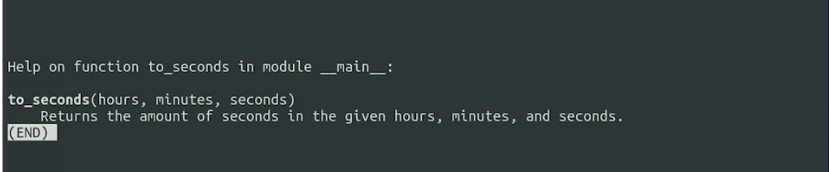
See how when we asked for help on our class we got a list of the methods that are defined in the class

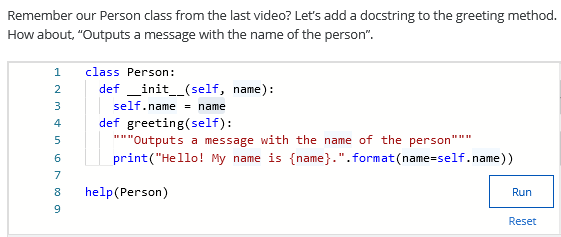
We want our methods, classes, and functions to give us more information when we or someone else use the help function. We can do that by adding a ***docstring***.



So there we have it, we have a function with a docstring in its body. Let's see how we can use the help function to see it.







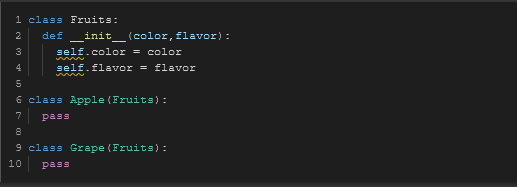
### **Code Reuse**

# **Inheritance**

Inheritance is a powerful feature in object oriented programming. It refers **to defining a new class with little or no modification to an existing class**. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

In other words, this allows us to reduce code duplication by generalizing our code.

In Python, we use parentheses in the class declaration to show an inheritance relationship.



For our new fruit classes, we've used that syntax to tell our computer that both the apple and the grape classes inherit from the fruit class.

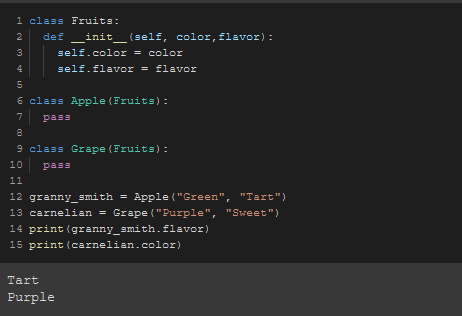
***Because of this, they automatically have the same constructor, which sets the color and flavor attributes.***

You can think of the fruit class as the parent class, and the apple and grape classes as siblings.

Let's see this in action.

First, we create an instance of the apple class. ***Granny\_smith*** = ***Apple***.

And we'll give it two parameters, green as the color and tart as the flavor. And now, an instance of the grape class. Then, to check that this actually worked, let's print the attributes values.



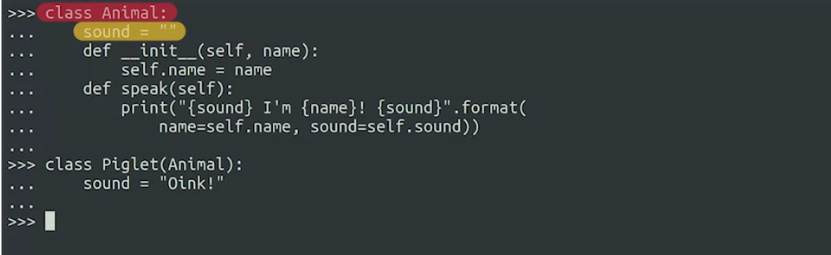
With the inheritance technique, we can use the fruit class to store information that applies to all kinds of fruit, and keep apple or grape specific attributes in their own classes.

For example, we could have an attribute to track how much of an apple is left after it's partially eaten. Of course, this applies to both attributes and methods.

If a class has an attribute or a method defined in it, inheriting classes will have the same attributes and methods defined in them.

But we can also get them to behave differently depending on what we change.

To explore this, let's go back to our piglet example and change it so that there's a base animal class. In this code, we've defined a general class called animal, which has an attribute to store the sound that the animal makes.

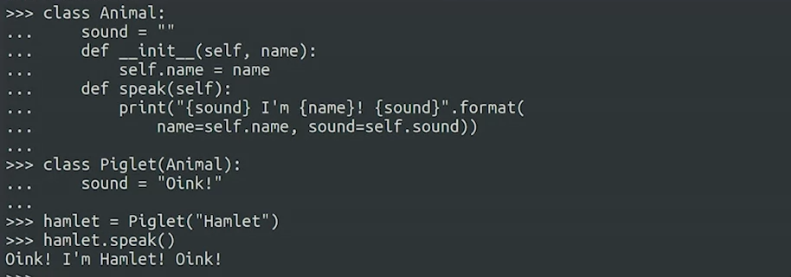


The constructor of the class takes the name that will be assigned to the instance when it's created.

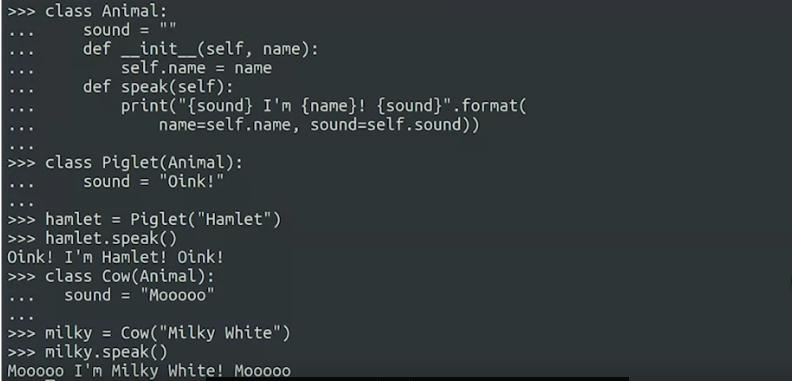
There's also a speak method that prints the name of the animal together with the sound the animal makes.

Then, we have a piglet class that inherits from the animal class.

We set the value of the sound attribute to oink in the piglet class, and that's the only thing we've modified from the original. Everything else is inherited. Let's see this in action.

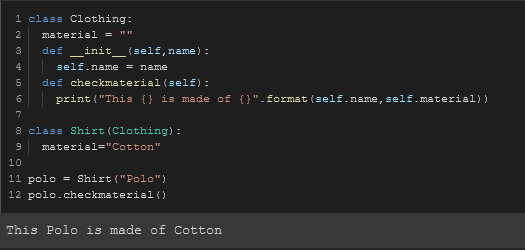


Let's define a new class that also inherits from animal. Let’s say, a cow class. Let's create an instance of this class to make it speak.



So you can see that we can easily define new classes that inherit from the base animal class and use both the attributes and methods that the animal class provides.





Let's think of a different example,

something closer to what you might be doing at your day-to-day job. In a system that handles the employees at your company, you may have a class called employee, which could have the attributes for things like full name of the person, the username used in company systems, the groups the employee belongs to, and so on.

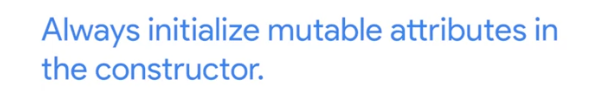
The employee class could have methods to do a bunch of things, like check if an employee belongs to a certain group, or create an email address based on the name and username attributes. The system could also have a manager class.

A manager is an employee, but has additional information associated with it, like the employees that report to a specific manager.

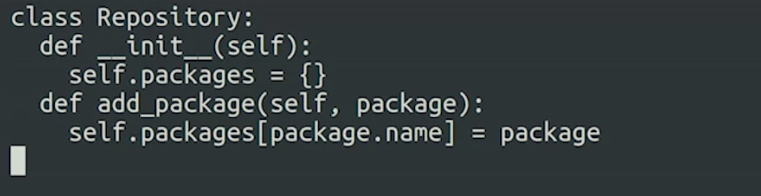
Are you starting to get an idea of the power of inheritance? Inheritance lets you reuse code written for one class in other classes.

# **Composition**

Composition is **a concept that models a has a relationship**. It enables creating complex types by combining objects of other types. This means that a class Composite can contain an object of another class Component . This relationship means that a Composite has a Component .



**RULE OF THUMB**



Now, we can add packages to the dictionary. We could also write a similar method to remove the packages.

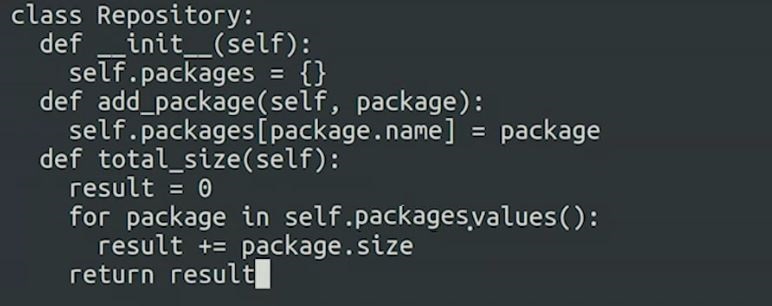
Let's do something more interesting instead.

We said that the packages had a size attribute that holds the size in bytes that the software package requires. So how could we calculate the size of the whole repository?

We need to iterate over the packages contained in the dictionary, adding up all their sizes.

I'd go something like this. We're going to add up all the sizes. So the first thing we need to do, is create a result variable that we'll use to sum up the values.

We have our result initialized. We now need to go through all the packages in the dictionary. Remember, the keys are the package names and the values are the package objects. For our calculation, we only care about the objects. So we'll retrieve them with the values dictionary method.



Now, for each package, we want to add the size to the result variable. We just need to return the result now.

Take a look at the method we've just written. It's a method inside the repository class, that's making use of the values method in the dictionary class and it's accessing the size attribute in the package class. That is the power of composition.

When we have other objects as attributes, we can use all their attributes and methods to get our own code to do what we want.