**Class basics**

Classes can be very useful for storing complicated objects with their own methods and variables. Defining a class is much like defining a function, but we use the classkeyword instead. We also use the word object in parentheses because we want our classes to *inherit* the object class. This means that our class has all the properties of an object, which is the simplest, most basic class. Later we'll see that classes can inherit other, more complicated classes. An empty class would look like this:

class ClassName(object): # class statements go here

**Instructions**

**1.**

Define a new class named "Car". For now, since we have to put something inside the class, use the passkeyword.

|  |
| --- |
| class Car(object):  pass |

**Create an instance of a class**

We can use classes to create new objects, which we say are **instances** of those classes.

Creating a new instance of a class is as easy as saying:

newObject = ClassName()

**Instructions**

**1.**

Below your Car class, create a new object named my\_car that is an instance of Car.

|  |
| --- |
| class Car(object):  pass  my\_car = Car() |

**Class member variables**

Classes can have *member variables* that store information about each class object. We call them member variables since they are information that belongs to the class object.

Creating member variables and assigning them initial values is as easy as creating any other variable:

class ClassName(object): memberVariable = "initialValue"

**Instructions**

**1.**

Inside your Car class, replace the pass statement with a new member variable named condition and give it an initial value of the string "new".

|  |
| --- |
| class Car(object):  condition = "new"  my\_car = Car() |

**Calling class member variables**

Each class object we create has its own set of member variables. Since we've created an object my\_car that is an instance of the Carclass, my\_car should already have a member variable named condition. This attribute gets assigned a value as soon as my\_car is created.

**Instructions**

**1.**

At the end of your code, use a printstatement to display the conditionof my\_car.

Hint

Since the attribute conditionbelongs to the object my\_car, you'll need to use *dot notation* to access the object's member variable: my\_car.condition.

|  |
| --- |
| class Car(object):  condition = "new"  my\_car = Car()  print my\_car.condition |

**Initializing a class**

There is a special function named \_\_init\_\_() that gets called whenever we create a new instance of a class. It exists by default, even though we don't see it. However, we can define our own \_\_init\_\_() function inside the class, overwriting the default version. We might want to do this in order to provide more input variables, just like we would with any other function.

The first argument passed to \_\_init\_\_()must always be the keyword self - this is how the object keeps track of itself internally - but we can pass additional variables after that.

In order to assign a variable to the class (creating a member variable), we use *dot notation*. For instance, if we passed newVariable into our class, inside the \_\_init\_\_() function we would say:

self.new\_variable = new\_variable

**Instructions**

**1.**

Define the \_\_init\_\_() function of the Car class to take four inputs: self, model, color, and mpg. Assign the last three inputs to member variables of the same name by using the selfkeyword.

Then, modify the object my\_car to provide the following inputs at initialization:

model = "DeLorean" color = "silver" mpg = 88

You don't need to include the selfkeyword when you create an instance of a class, because selfgets added to the beginning of your list of inputs automatically by the class definition.

Hint

Creating an instance of a class with many initialization variables looks the same as calling a function with many inputs; put all the values in parentheses, separated by commas.

In the body of \_\_init\_\_(), you'd set the model like this:

def \_\_init\_\_(self, model, color, mpg): self.model = model

|  |
| --- |
| class Car(object):  condition = "new"  def \_\_init\_\_(self, model, color, mpg):  self.model = model  self.color = color  self.mpg = mpg  my\_car = Car("DeLorean", "silver", 88)  print my\_car.condition |

# Referring to member variables

Calling class member variables works the same whether those values are created within the class (like our car's condition) or values are passed into the new object at initialization. We use dot notation to access the member variables of classes since those variables belong to the object.

For instance, if we had created a member variable named new\_variable, a new instance of the class named new\_objectcould access this variable by saying:

new\_object.new\_variable

**Instructions**

**1.**

Now that you've created my\_carprint its member variables:

* First print the model of my\_car. Click "Stuck? Get a hint!" for an example.
* Then print out the color of my\_car.
* Then print out the mpg of my\_car.

Hint

To print my\_car's model, you'd type:

print my\_car.model

|  |
| --- |
| class Car(object):  condition = "new"  def \_\_init\_\_(self, model, color, mpg):  self.model = model  self.color = color  self.mpg = mpg  my\_car = Car("DeLorean", "silver", 88)  print my\_car.condition  print my\_car.model  print my\_car.color  print my\_car.mpg |
| new  DeLorean  silver  88 |

# Creating class methods

Besides member variables, classes can also have their own methods. For example:

class Square(object): def \_\_init\_\_(self, side): self.side = side def perimeter(self): return self.side \* 4

The perimeter() class method is identical to defining any other function, except that it is written inside of the Square class definition.

Just like when we defined \_\_init\_\_(), you need to provide self as the first argument of any class method.

**Instructions**

**1.**

Inside the Car class, add a method named display\_car to Car that will reference the Car's member variables to return the string, "This is a [color] [model] with [mpg] MPG." You can use the str() function to turn your mpg into a string when creating the display string.

Replace the individual printstatements with a single printcommand that displays the result of calling my\_car.display\_car()

Hint

Remember, in order to access member variables of a class (even while inside of that class), we have to use the self keyword and dot notation to specify that we mean the member variable that belongs to the class.

|  |
| --- |
| class Car(object):  condition = "new"  def \_\_init\_\_(self, model, color, mpg):  self.model = model  self.color = color  self.mpg = mpg    def display\_car(self):  print "This is a %s %s with %s MPG." % (self.color, self.model, str(self.mpg))  my\_car = Car("DeLorean", "silver", 88)  my\_car.display\_car() |

# Modifying member variables

We can modify variables that belong to a class the same way that we initialize those member variables. This can be useful when we want to change the value a variable takes on based on something that happens inside of a class method.

**Instructions**

**1.**

Inside the Car class, add a method drive\_car that sets self.conditionto the string "used".

Remove the call to my\_car.display\_car() and instead print only the condition of your car.

Then drive your car by calling the drive\_car method.

Finally, print the condition of your car again to see how its value changes.

|  |
| --- |
| class Car(object):  condition = "new"  def \_\_init\_\_(self, model, color, mpg):  self.model = model  self.color = color  self.mpg = mpg  def display\_car(self):  print "This is a %s %s with %s MPG." % (self.color, self.model, str(self.mpg))  def drive\_car(self):  self.condition = "used"  my\_car = Car("DeLorean", "silver", 88)  print my\_car.condition  my\_car.drive\_car()  print my\_car.condition |
| new  used |

# Inheritance

One of the benefits of classes is that we can create more complicated classes that inherit variables or methods from their **parent classes**. This saves us time and helps us build more complicated objects, since these **child classes** can also include additional variables or methods.

We define a "child" class that inherits all of the variables and functions from its "parent" class like so:

class ChildClass(ParentClass): # new variables and functions go here

Normally we use object as the parent class because it is the most basic type of class, but by specifying a different class, we can inherit more complicated functionality.

**Instructions**

**1.**

Create a class ElectricCar that inherits from Car. Give your new class an \_\_init\_\_() method of that includes a battery\_type member variable in addition to the model, color and mpg.

Then, create an electric car named my\_car with a "molten salt"battery\_type. Supply values of your choice for the other three inputs (model, color and mpg).

Hint

Redefining a method of a "child" class is as simple as including a definition for that function inside the "child" class; this version will take precedence over the inherited version.

Remember to include the selfkeyword as the first input when you define the \_\_init\_\_() method!

|  |
| --- |
| class Car(object):  condition = "new"  def \_\_init\_\_(self, model, color, mpg):  self.model = model  self.color = color  self.mpg = mpg    def display\_car(self):  print "This is a %s %s with %s MPG." % (self.color, self.model, str(self.mpg))    def drive\_car(self):  self.condition = "used"    class ElectricCar(Car):  def \_\_init\_\_(self, model, color, mpg, battery\_type):  self.model = model  self.color = color  self.mpg = mpg  self.battery\_type = battery\_type  my\_car = ElectricCar("DeLorean", "silver", 88, "molten salt") |

# Overriding methods

Since our ElectricCar is a more specialized type of Car, we can give the ElectricCar its own drive\_car() method that has different functionality than the original Car class's.

**Instructions**

**1.**

Inside ElectricCar add a new method drive\_car that changes the car's condition to the string "like new".

Then, outside of ElectricCar, print the condition of my\_car

Next, drive my\_car by calling the drive\_car function

Finally, print the condition of my\_caragain

Hint

This should be very similar to what you did in the second exercise of this section.

|  |
| --- |
| class Car(object):  condition = "new"  def \_\_init\_\_(self, model, color, mpg):  self.model = model  self.color = color  self.mpg = mpg    def display\_car(self):  print "This is a %s %s with %s MPG." % (self.color, self.model, str(self.mpg))    def drive\_car(self):  self.condition = "used"    class ElectricCar(Car):  def \_\_init\_\_(self, model, color, mpg, battery\_type):  self.model = model  self.color = color  self.mpg = mpg  self.battery\_type = battery\_type  def drive\_car(self):  self.condition = "like new"  my\_car = ElectricCar("DeLorean", "silver", 88, "molten salt")  print my\_car.condition  my\_car.drive\_car()  print my\_car.condition |

|  |
| --- |
| new  like new |

# Building useful classes

Chances are, you won't be designing Car classes in the real world anytime soon. Usually, classes are most useful for holding and accessing abstract collections of data.

One useful class method to override is the built-in \_\_repr\_\_() method, which is short for representation; by providing a return value in this method, we can tell Python how to represent an object of our class (for instance, when using a printstatement).

**Instructions**

**1.**

Define a Point3D class that inherits from object

Inside the Point3D class, define an \_\_init\_\_() function that accepts self, x, y, and z, and assigns these numbers to the member variables self.x, self.y, self.z

Define a \_\_repr\_\_() method that returns "(%d, %d, %d)" % (self.x, self.y, self.z). This tells Python to represent this object in the following format: (x, y, z).

Outside the class definition, create a variable named my\_point containing a new instance of Point3D with x=1, y=2, and z=3.

Finally, print my\_point.

Hint

When defining a new \_\_repr\_\_(), return a string value that uses the member variables of the class to display the 3D point properly. You can use the str() function to put these numbers in the proper string.

For **advanced users:** For more information on \_\_repr\_\_ and other special methods see this [Python documentation](https://docs.python.org/2/reference/datamodel.html#object.__repr__). Note the slight difference between the \_\_repr\_\_ and \_\_str\_\_methods.

|  |
| --- |
| class Point3D(object):  def \_\_init\_\_(self, x, y, z):  self.x = x  self.y = y  self.z = z    def \_\_repr\_\_(self):  return "(%d, %d, %d)" % (self.x, self.y, self.z)    my\_point = Point3D(1, 2, 3)  print my\_point |
| (1, 2, 3) |

Called by the **[repr()](https://docs.python.org/2/library/functions.html" \l "repr" \o "repr)** built-in function and by string conversions (reverse quotes) to compute the “official” string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form <...some useful description...> should be returned. The return value must be a string object. If a class defines [**\_\_repr\_\_()**](https://docs.python.org/2/reference/datamodel.html#object.__repr__) but not [**\_\_str\_\_()**](https://docs.python.org/2/reference/datamodel.html#object.__str__), then [**\_\_repr\_\_()**](https://docs.python.org/2/reference/datamodel.html#object.__repr__) is also used when an “informal” string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

object.**\_\_str\_\_**(*self*)

Called by the **[str()](https://docs.python.org/2/library/functions.html" \l "str" \o "str)** built-in function and by the [**print**](https://docs.python.org/2/reference/simple_stmts.html#print) statement to compute the “informal” string representation of an object. This differs from [**\_\_repr\_\_()**](https://docs.python.org/2/reference/datamodel.html#object.__repr__) in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

object.**\_\_lt\_\_**(*self*, *other*)

object.**\_\_le\_\_**(*self*, *other*)

object.**\_\_eq\_\_**(*self*, *other*)

object.**\_\_ne\_\_**(*self*, *other*)

object.**\_\_gt\_\_**(*self*, *other*)

object.**\_\_ge\_\_**(*self*, *other*)