8

HOW TO USE CLASSES AND OBJECTS

Why is a giraffe like a sidewalk? Because a giraffe and a sidewalk are both *things*, which are known in the English language as nouns and in Python as *objects*. In programming, objects are a way to organize code and break things down to more easily work with complex ideas. (We used an object in Chapter 4 when we worked with the `turtle` module's `Turtle` object.)

To fully understand how objects work in Python, we need to think about types of objects. Let's start with giraffes and sidewalks.

A giraffe is a type of mammal, which is a type of animal. A giraffe is also an animate object—it's alive.

There's not much to say about a sidewalk other than it's not a living thing. Let's call it an inanimate object (in other words, it's not alive). The terms *mammal*, *animal*, *animate*, and *inanimate* are all ways of classifying things.

Breaking Things into Classes

In Python, objects are defined by *classes*, which classify objects into groups. For example, the tree diagram in Figure 8-1 shows the classes that giraffes and sidewalks fit into based on our preceding definitions.

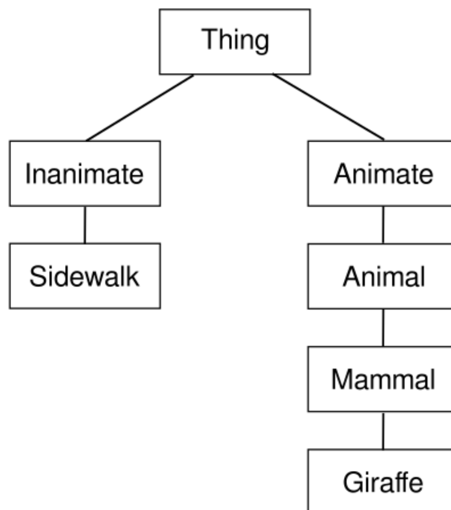


Figure 8-1: Tree diagram of some classes

The main class is `Thing`. Below the `Thing` class, we have `Inanimate` and `Animate`. These classes are further broken down into `Sidewalk` for `Inanimate`, and `Animal`, `Mammal`, and `Giraffe` for `Animate`.

We can use classes to organize bits of code. For example, consider the `turtle` module. All the things Python's turtle module can do—such as moving forward, moving backward, turning left, and turning right—are functions

in the `Turtle` class. An object is a member of a class, and we can create any number of objects for a class—which we'll get to shortly.

Now let's create the same set of classes shown in our tree diagram, starting from the top. We define classes using the `class` keyword followed by a name. `Thing` is the broadest class, so we'll create it first:

```
>>> class Thing:
        pass
```

We name the class `Thing` and use the `pass` statement to let Python know we're not going to give any more information. The `pass` keyword is used when we want to provide a class or function but don't want to fill in the details at the moment.

Next, we'll add the other classes and build some relationships between them.

Children and Parents

If a class is a part of another class, it's considered a *child* of that class, and the other class is its *parent*. Classes can be both *children of* and *parents to* other classes. In our tree diagram, the class above another class is its parent and the class below it is its child. For example, `Inanimate` and `Animate` are both children of the class `Thing`, meaning that `Thing` is their parent.

To tell Python that a class is a child of another class, we add the name of the parent class in parentheses after the name of our new class, like this:

```
>>> class Inanimate(Thing):
        pass

>>> class Animate(Thing):
        pass
```

Here, we create a class called `Inanimate` and tell Python that its parent class is `Thing`. Next, we create a class called `Animate` and tell Python that its parent class is also `Thing`.

Let's create the `Sidewalk` class with the parent class `Inanimate` like so:

```
>>> class Sidewalk(Inanimate):
        pass
```

We can organize the `Animal`, `Mammal`, and `Giraffe` classes using their parent classes as well:

```
>>> class Animal(Animate):
         pass

>>> class Mammal(Animal):
         pass
```

```
>>> class Giraffe(Mammal):
        pass
```

Adding Objects to Classes

We now have a bunch of classes, but what about putting some more information into those classes? Say we have a giraffe named Reginald. We know he belongs in the `Giraffe` class, but what do we use—in programming terms—to describe the single giraffe called Reginald? We call Reginald an object (also known as an *instance*) of the `Giraffe` class. To "introduce" Reginald to Python, we use this little snippet of code:

```
>>> reginald = Giraffe()
```

This code tells Python to create an object of the `Giraffe` class and assign it to the `reginald` variable. Like when we call a function, the class name is followed by parentheses. Later in this chapter, we'll see how to create objects and use parameters in the parentheses.

But what does the `reginald` object do? Well, nothing at the moment. To make our objects useful, when we create our classes, we also need to define functions that can be used with the objects in that class. Rather than just using the `pass` keyword immediately after defining the class, we can add function definitions.

Defining Functions of Classes

Chapter 7 introduced functions as a way to reuse code. When we define a function that's associated with a class, we do so in the same way that we define any other function, except we indent it beneath the class definition. For example, here's a normal function that isn't associated with a class:

```
>>> def this_is_a_normal_function():
        print('I am a normal function')
```

And here are a couple of functions that are defined for a class:

```
>>> class ThisIsMySillyClass:
        def this_is_a_class_function():
            print('I am a class function')
        def this_is_also_a_class_function():
            print('I am also a class function. See?')
```

Adding Class Characteristics as Functions

Consider the child classes of the `Animate` class we defined on page 83. We can add characteristics to each class that describe what it is and what it can do. A *characteristic* is a trait all members of the class (and its children) share.

For example, what do all animals have in common? To start with, they breathe, move, and eat. What about mammals? Mammals feed their young with milk, and they also breathe, move, and eat. We know giraffes eat leaves from high up in trees. And like all mammals, they feed their young with milk, breathe, move, and eat. When we add these characteristics to our tree diagram, we get something like Figure 8-2.

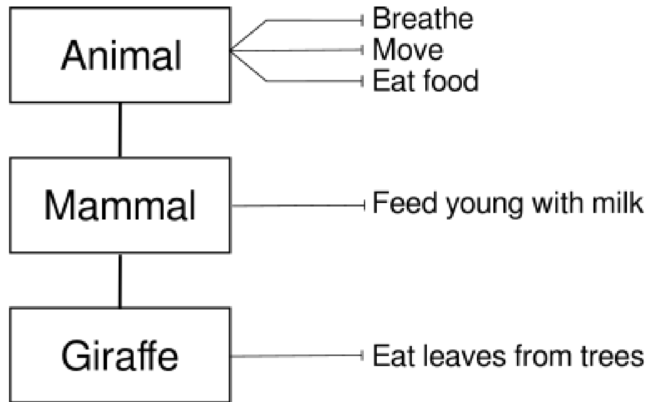


Figure 8-2: Classes with characteristics

These characteristics can be thought of as actions, or functions—things that an object of that class can do.

To add a function to a class, we use the `def` keyword. So the `Animal` class will look like this:

```
>>> class Animal(Animate):
        def breathe(self):
            pass
        def move(self):
            pass
        def eat_food(self):
            pass
```

In the first line of this listing, we define the class as we did before, but instead of using `pass` on the next line, we define a function called `breathe` and give it the `self` parameter. The `self` parameter is a way for one function in the class to call another function in the class (and in the parent class). We'll see this parameter in use later.

On the next line, `pass` tells Python we're not going to provide any more information about the `breathe` function because it's going to do nothing for now. Then we add the `move` and `eat_food` functions and use the `pass` keyword for both. We'll recreate our classes shortly and put some proper code in the functions. This is a common way to develop programs.

NOTE *Often, programmers will create classes with functions that do nothing as a way to figure out what the class should do before getting into the details of the individual functions.*

We can also add functions to the `Mammal` and `Giraffe` classes. Each class will be able to use the characteristics (or functions) of its parent, meaning you don't need to make one really complicated class. Instead, you can put your functions in the highest parent class where the characteristic applies. (This makes your classes simpler and easier to understand.)

```
>>> class Mammal(Animal):
        def feed_young_with_milk(self):
            pass

>>> class Giraffe(Mammal):
        def eat_leaves_from_trees(self):
            pass
```

In the above code, the `Mammal` class provides a function `feed_young_with_milk`. The `Giraffe` class is a child class (or *subclass*) of `Mammal` and provides another function: `eat_leaves_from_trees`.

Why Use Classes and Objects?

We've now added functions to our classes, but why use classes and objects at all when you could just write normal functions called `breathe`, `move`, `eat_food`, and so on?

To answer that question, we'll use our giraffe called `Reginald`, which we created earlier as an object of the Giraffe class, like this:

```
>>> reginald = Giraffe()
```

Because `reginald` is an object, we can call (or run) functions provided by the `Giraffe` class and its parent classes. We call functions on an object by using the dot (`.`) operator and the name of the function. To tell Reginald to move or eat, we can call the functions like this:

```
>>> reginald = Giraffe()
>>> reginald.move()
>>> reginald.eat_leaves_from_trees()
```

Suppose Reginald has a giraffe friend named Harold. Let's create another `Giraffe` object called `harold`:

```
>>> harold = Giraffe()
```

Because we're using objects and classes, we can tell Python which giraffe we're talking about when we want to run the `move` function. For example, if we want to make Harold move but leave Reginald in place, we could call the move function using our `harold` object, like this:

```
>>> harold.move()
```

In this case, only Harold would be moving.

Let's change our classes a little to make this more obvious. We'll add a `print` statement to each function instead of using `pass`:

```
>>> class Animal(Animate):
        def breathe(self):
            print('breathing')
        def move(self):
            print('moving')
        def eat_food(self):
            print('eating food')
>>> class Mammal(Animal):
        def feed_young_with_milk(self):
            print('feeding young')

>>> class Giraffe(Mammal):
        def eat_leaves_from_trees(self):
            print('eating leaves')
```

Now when we create our `reginald` and `harold` objects and call functions on them, we can see something actually happen:

```
>>> reginald = Giraffe()
>>> harold = Giraffe()
>>> reginald.move()
moving
>>> harold.eat_leaves_from_trees()
eating leaves
```

On the first two lines, we create the variables `reginald` and `harold`, which are objects of the `Giraffe` class. Next, we call the `move` function on `reginald`, and Python prints `moving` on the following line. In the same way, we call the `eat_leaves_from_trees` function on `harold`, and Python prints `eating leaves`. If these were real giraffes, rather than objects in a computer, one giraffe would be walking and the other would be eating.

NOTE *Functions defined for classes are actually called* `methods`*. The terms are almost interchangeable except that methods can only be called on objects of a class. Another way of saying this is that a method is associated with a class but a function is not. Given they are almost the same, we'll use the term* `function` *in this book.*

Objects and Classes in Pictures

Let's try taking a more graphical approach to objects and classes and return to the `turtle` module we toyed with in Chapter 4. When we use `turtle.Turtle()`, Python creates an object of the `Turtle` class that is provided by the `turtle` module (similar to our `reginald` and `harold` objects). We can create two `Turtle` objects (named Avery and Kate) just as we created two giraffes:

```
>>> import turtle
>>> avery = turtle.Turtle()
```

```
>>> kate = turtle.Turtle()
```

Each turtle object (`avery` and `kate`) is a member of the `Turtle` class.

Now here's where objects start to become powerful. Having created our `Turtle` objects, we can call functions on each and they will draw independently. Try this code:

```
>>> avery.forward(50)
>>> avery.right(90)
>>> avery.forward(20)
```

With this series of instructions, we tell Avery to move forward 50 pixels, turn right 90 degrees, and move forward 20 pixels so she finishes facing downward. Remember that turtles always start off facing to the right.

Now it's time to move Kate.

```
>>> kate.left(90)
>>> kate.forward(100)
```

We tell Kate to turn left 90 degrees and then move forward 100 pixels so she ends facing up. So far, we have a line with arrowheads moving in two different directions, with the head of each arrow representing a different turtle object: Avery is pointing down and Kate is facing up (see Figure 8-3).

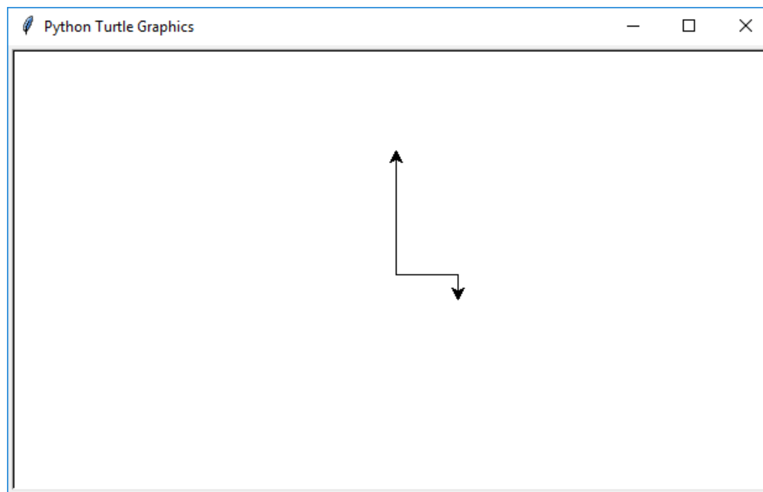
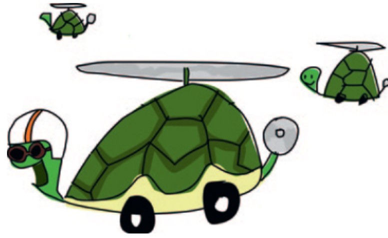


Figure 8-3: Kate and Avery

Now let's add another turtle, Jacob, and move him without bugging Kate or Avery.

```
>>> jacob = turtle.Turtle()
>>> jacob.left(180)
>>> jacob.forward(80)
```



First, we create a new `Turtle` object called `jacob`, then we turn him left 180 degrees and move him forward 80 pixels. Our drawing with three turtles should look like Figure 8-4.

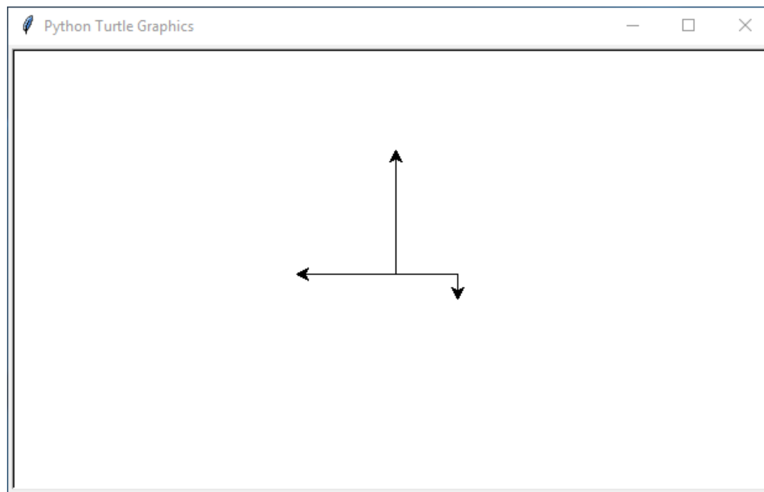


Figure 8-4: Kate and Avery and Jacob

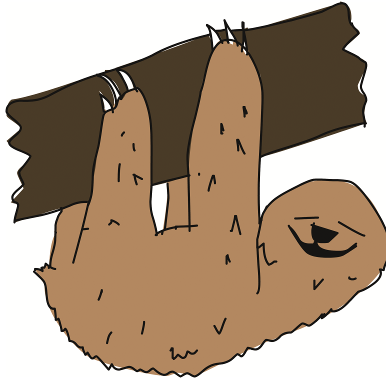
Every time we call `turtle.Turtle()` to create a turtle, we add a new, independent object. Each object is still an instance of the `Turtle` class, and we can use the same functions on each object. But because we're using objects, we can move each turtle independently. Like our independent `Giraffe` objects (Reginald and Harold), Avery, Kate, and Jacob are independent Turtle objects. If we create a new object with the same variable name as an object we've already created, the old object won't necessarily vanish.

Other Useful Features of Objects and Classes

Classes and objects make it easy to group functions. They're also really useful when we want to think about a program in smaller chunks.

For example, consider a huge software application like a word processor or a 3D computer game. It's nearly impossible for most people to fully understand large programs like these because there's so much code. But break these monster programs into smaller pieces and each piece starts to make sense—as long as you know its programming language, of course!

When writing a large program, breaking it up also allows you to divide the work among other programmers. The most complicated programs (like your web browser) were written by many people, or teams of people, working on different parts at the same time around the world. Imagine we want to expand some of the classes we've created in this chapter (`Animal`, `Mammal`, and `Giraffe`), but we have too much work to do, and our friends have offered to help. We could divide the work of writing the code so that one person works on the `Animal` class, another on the `Mammal` class, and still another on the `Giraffe` class.



Inherited Functions

You may realize that whoever ends up working on the `Giraffe` class is lucky, because any functions created by the people working on the `Animal` and `Mammal` classes can also be used by the `Giraffe` class. The `Giraffe` class *inherits* functions from the `Mammal` class, which, in turn, inherits functions from the `Animal` class. In other words, when we create a `Giraffe` object, we can use functions defined in the `Giraffe` class, as well as functions defined in the `Mammal` and `Animal` classes. And, by the same token, if we create a `Mammal` object, we can use functions defined in the `Mammal` class as well as its parent class, `Animal`.

Take a look at the relationship between the `Animal`, `Mammal`, and `Giraffe` classes again. The `Animal` class is the parent of the `Mammal` class, and `Mammal` is the parent of the `Giraffe` class (Figure 8-5).

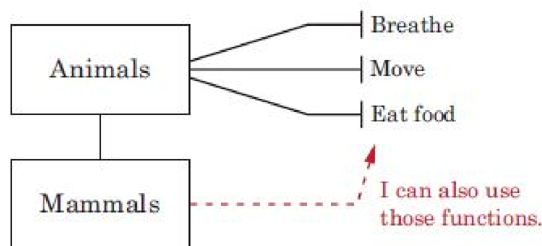


Figure 8-5: Classes and inherited functions

Even though Reginald is an object of the Giraffe class, we can still call the move function we defined in the Animal class because functions defined in any parent class are available to its child classes:

```
>>> reginald = Giraffe()
>>> reginald.move()
moving
```

In fact, all the functions we defined in both the Animal and Mammal classes can be called from our reginald object because the functions are inherited:

```
>>> reginald = Giraffe()
>>> reginald.breathe()
breathing
>>> reginald.eat_food()
eating food
>>> reginald.feed_young_with_milk()
feeding young
```

In this code we create an object of the Giraffe class called reginald. When we call each function it prints a message regardless of whether the function is defined in Giraffe or in a parent class.

Functions Calling Other Functions

When we call functions on an object, we use the object's variable name. For example, we can call the move function on Reginald the giraffe like so:

```
>>> reginald.move()
```

To have a function in the Giraffe class call the move function, we'd use the self parameter. The self parameter is a way for one function in the class to call another function. For example, suppose we add a function called find_food to the Giraffe class:

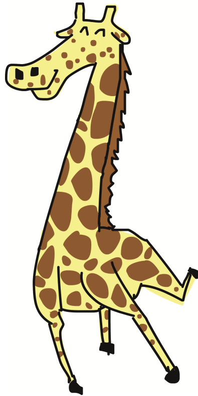
```
>>> class Giraffe(Mammal):
        def find_food(self):
            self.move()
            print('I\'ve found food!')
            self.eat_food()
```

We've created a function that combines two other functions, which is quite common in programming. Often, you'll write a function that does something useful, which you can then use inside another function. (We'll do this in Chapter 11, where we'll write more complex functions to create a game.)

Let's use self to add some functions to the Giraffe class:

```
>>> class Giraffe(Mammal):
        def find_food(self):
```

```
            self.move()
            print('I\'ve found leaves!')
            self.eat_food()
        def eat_leaves_from_trees(self):
            print('tear leaves from branches')
            self.eat_food()
        def dance_a_jig(self):
            self.move()
            self.move()
            self.move()
            self.move()
```



We use the eat_food and move functions from the parent Animal class to define eat_leaves_from_trees and dance_a_jig for the Giraffe class, because these are inherited functions. By adding functions that call other functions, when we create objects of these classes, we can call a single function that does more than just one thing. See what happens when we call the dance_a_jig function below:

```
>>> reginald = Giraffe()
>>> reginald.dance_a_jig()
moving
moving
moving
moving
```

In this code, our giraffe moves four times (that is, the text moving is printed four times).

If we call the find_food function we get three lines printed:

```
>>> reginald.find_food()
moving
I've found leaves!
eating food
```

Initializing an Object

Sometimes when creating an object, we want to set some values (also called *properties*) for later use. When we *initialize* an object, we're getting it ready to be used.

For example, suppose we want to set the number of spots on our giraffe objects when they're created (or initialized). To do this, we create an `__init__` function (note that there are two underscore characters on each side, for a total of four). The `init` function sets the properties for an object when the object is first created. Python will automatically call this function when we create a new object. Here's how to use it:

```
>>> class Giraffe(Mammal):
        def __init__(self, spots):
            self.giraffe_spots = spots
```

First, we define the `__init__` function with the `self` and `spots` parameters. Just like the other functions we've defined in the class, the `__init__` function also needs to have `self` as the first parameter. Next, we set the `spots` parameter to an object variable (its property) called `giraffe_spots` using the `self` parameter. You might think of this line of code as saying, "Take the value of the `spots` parameter and save it for later (using the `giraffe_spots` object variable)." Just as one function in a class can call another function using the `self` parameter, variables in the class are also accessed using `self`.

Next, if we create a couple of new giraffe objects (called Ozwald and Gertrude) and display their number of spots, you can see the initialization function in action:

```
>>> ozwald = Giraffe(100)
>>> gertrude = Giraffe(150)
>>> print(ozwald.giraffe_spots)
100
>>> print(gertrude.giraffe_spots)
150
```

First, we create an instance of the `Giraffe` class using the parameter value 100. This has the effect of calling the `__init__` function and using 100 for the value of the `spots` parameter. Next, we create another instance of the `Giraffe` class with a value of 150. Lastly, we print the object variable `giraffe_spots` for each of our giraffe objects, and we see that the results are 100 and 150. It worked!

Remember, when we create an object of a class, such as `ozwald` above, we can refer to its variables or functions using the dot operator and the name of the variable or function we want to use (for example, `ozwald.giraffe_spots`). But when we're creating functions inside a class, we refer to those same variables (and other functions) using the `self` parameter (`self.giraffe_spots`).

What You Learned

In this chapter, we used classes to create categories of things and made objects (or instances) of those classes. You learned how the child of a class inherits the functions of its parent, and that even though two objects are of the same class, they're not necessarily clones. For example, two giraffe objects can have their own distinct number of spots. You learned how to call (or run) functions on an object and how object variables are a way of saving values in those objects. Lastly, we used the `self` parameter in functions to refer to other functions and variables. These concepts are fundamental to Python, and you'll see them multiple times throughout the rest of this book.

Programming Puzzles

Give the following examples a try to experiment with creating your own functions. The solutions can be found at *http://python-for-kids.com*.

#1: The Giraffe Shuffle

Add functions to the `Giraffe` class to move the giraffe's left and right feet forward and backward. A function for moving the left foot forward might look like this:

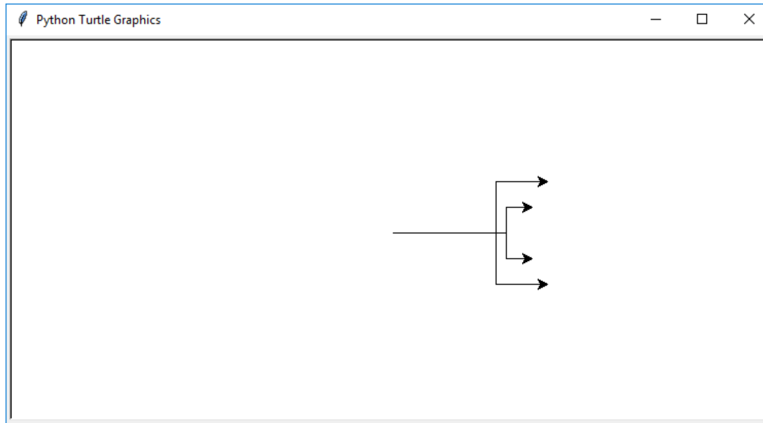
```
>>> def left_foot_forward(self):
        print('left foot forward')
```

Then create a function called `dance` to teach our giraffes to dance (the function will call the four foot functions you've just created). The result of calling this new function will be a simple dance:

```
>>> reginald = Giraffe()
>>> reginald.dance()
left foot forward
left foot back
right foot forward
right foot back
left foot back
right foot back
right foot forward
left foot forward
```

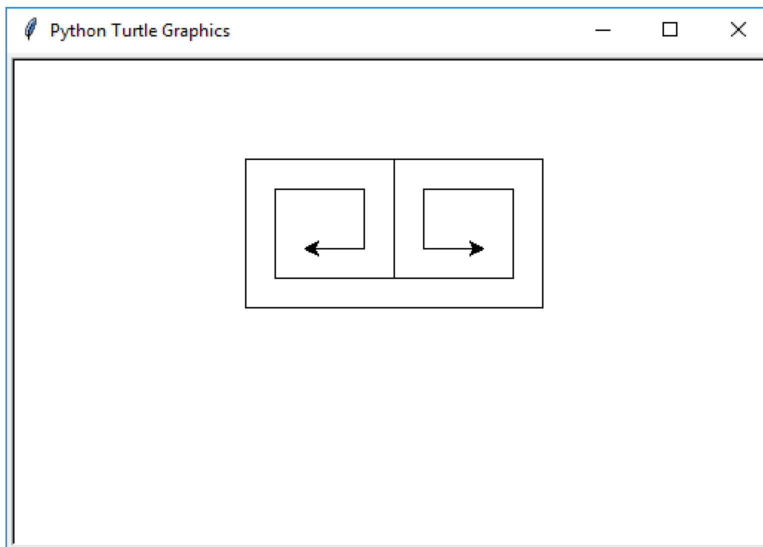
#2: Turtle Pitchfork

Create the following picture of a sideways pitchfork using four `Turtle` objects (the exact length of the lines isn't important). Remember to import the `turtle` module first!



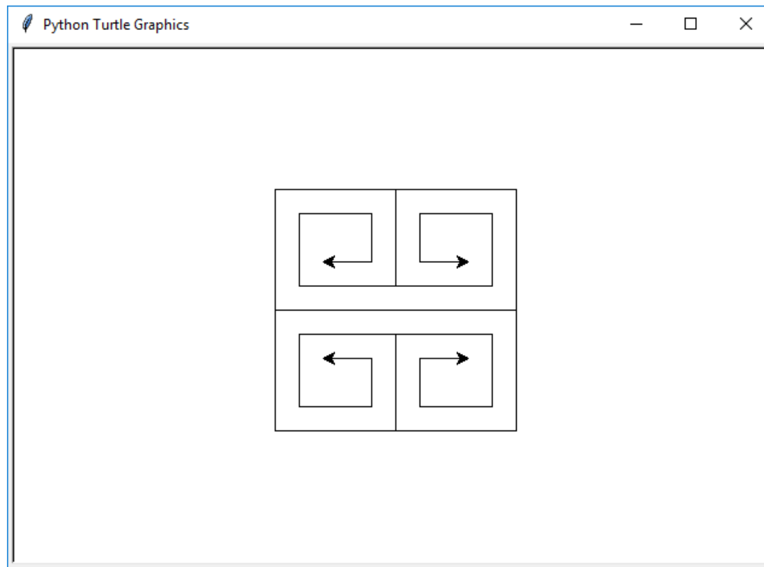
#3: Two Small Spirals

Create the following picture of two small spirals using two `Turtle` objects (again the exact size of the spirals isn't important).



#4: Four Small Spirals

Let's take the two spirals we created in the previous code and make a mirror image to create four spirals, which should look like the following image.


Python Turtle Graphics