8 HOW TO USE CLASSES AND OBJECTS

Why is a giraffe like a sidewalk? Because both a giraffe and a sidewalk are *things*, known in the English language as *nouns* and in Python as *objects*.

The idea of *objects* is an important one in the world of computers. Objects are a way of organizing code in a program and breaking things down to make it easier to think about complex ideas. (We used an object in Chapter 4 when we worked with the turtle—`Pen`.)

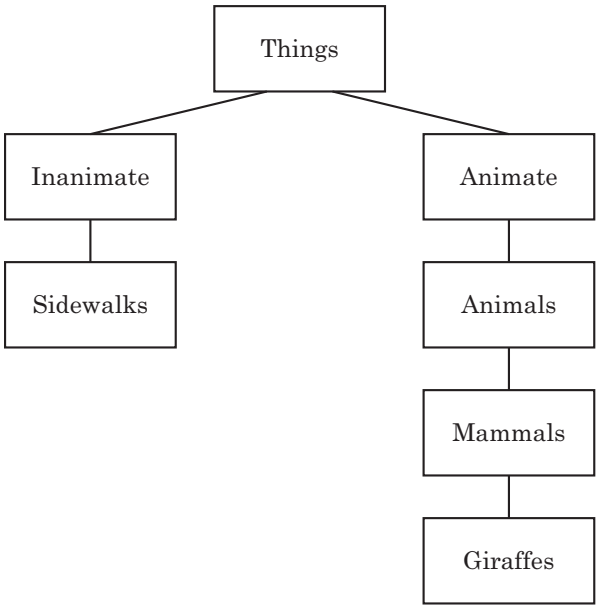
To really understand how objects work in Python, we need to think about types of objects. Let's start with giraffes and sidewalks.

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

Now consider a sidewalk. 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 we can think of as a way to classify objects into groups. Here is a tree diagram of the classes that giraffes and sidewalks would fit into based on our preceding definitions:



The main class is `Things`. Below the `Things` class, we have `Inanimate` and `Animate`. These are further broken down into just `Sidewalks` for `Inanimate`, and `Animals`, `Mammals`, and `Giraffes` for `Animate`.

We can use classes to organize bits of Python code. For example, consider the `turtle` module. All the things that Python's `turtle` module can do—such as moving forward, moving backward, turning left, and turning right—are functions in the `Pen` class. An object can be thought of as a member of a class, and we can create any number of objects for a class—which we will get to shortly.

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

```
>>> class Things:
        pass
```

We name the class `Things` and use the `pass` statement to let Python know that we're not going to give any more information. `pass` 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, then it's 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 `Things`, meaning that `Things` 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(Things):
        pass

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

Here, we create a class called `Inanimate` and tell Python that its parent class is `Things` with the code `class Inanimate(Things)`. Next, we create a class called `Animate` and tell Python that its parent class is also `Things`, using `class Animate(Things)`.

Let's try the same thing with the `Sidewalks` class. We create the `Sidewalks` class with the parent class `Inanimate` like so:

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

And we can organize the `Animals`, `Mammals`, and `Giraffes` classes using their parent classes as well:

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

>>> class Mammals(Animals):
        pass

>>> class Giraffes(Mammals):
        pass
```

ADDING OBJECTS TO CLASSES

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

```
>>> reginald = Giraffes()
```

This code tells Python to create an object in the `Giraffes` class and assign it to the variable `reginald`. Like 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 the class definition, 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 is associated with a class, we do so in the same way that we define any other function, except that 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')
```

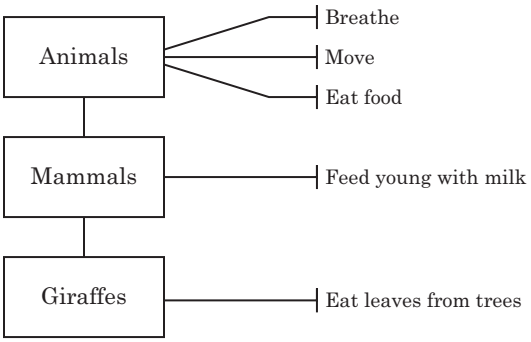
And here are a couple of functions that belong to a class:

```
>>> 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 95. We can add *characteristics* to each class to describe what it is and what it can do. A characteristic is a trait that all of the members of the class (and its children) share.

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



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 `Animals` class will look like this:

```
>>> class Animals(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 the `pass` keyword on the next line, we define a function called `breathe`, and give it one parameter: `self`. 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 will see this parameter in use later.

On the next line, the `pass` keyword 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 functions `move` and `eat_food`, which also do nothing for now. We'll re-create our classes shortly and put some proper code in the functions. This is a common way to develop programs. 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 other two classes, `Mammals` and `Giraffes`. Each class will be able to use the characteristics (the functions) of its parent. This means that you don't need to make one really complicated class; you can put your functions in the highest parent where the characteristic applies. (This is a good way to make your classes simpler and easier to understand.)

```
>>> class Mammals(Animals):
        def feed_young_with_milk(self):
            pass

>>> class Giraffes(Mammals):
        def eat_leaves_from_trees(self):
            pass
```

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 `Giraffes` class, like this:

```
>>> reginald = Giraffes()
```

Because `reginald` is an object, we can call (or run) functions provided by his class (the `Giraffes` 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 the giraffe to move or eat, we can call the functions like this:

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

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

```
>>> harold = Giraffes()
```

Because we're using objects and classes, we can tell Python exactly which giraffe we're talking about when we want to run the `move` function. For example, if we wanted 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 a bit more obvious. We'll add a `print` statement to each function, instead of using `pass`:

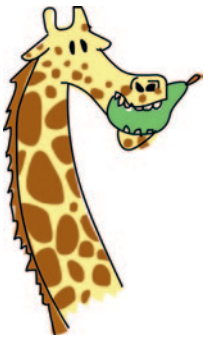
```
>>> class Animals(Animate):
        def breathe(self):
            print('breathing')
        def move(self):
            print('moving')
        def eat_food(self):
            print('eating food')
```

```
>>> class Mammals(Animals):
        def feed_young_with_milk(self):
            print('feeding young')

>>> class Giraffes(Mammals):
        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 = Giraffes()
>>> harold = Giraffes()
>>> 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 `Giraffes` 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.

OBJECTS AND CLASSES IN PICTURES

How about taking a more graphical approach to objects and classes?

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

```
>>> import turtle
>>> avery = turtle.Pen()
>>> kate = turtle.Pen()
```

Each turtle object (`avery` and `kate`) is a member of the `Pen` 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:

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

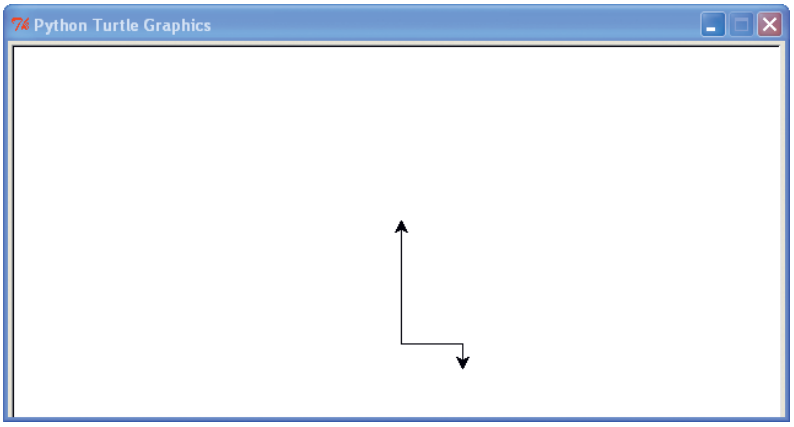
With this series of instructions, we tell Avery to move forward 50 pixels, turn right 90 degrees, and move forward 20 pixels so that she finishes facing downward. Remember that turtles always start off facing to the right.

Now it's time to move Kate.

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

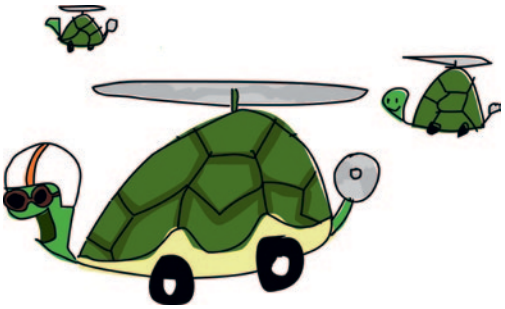
We tell Kate to turn left 90 degrees, and then move forward 100 pixels so that she ends facing up.

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 pointing down, and Kate facing up.

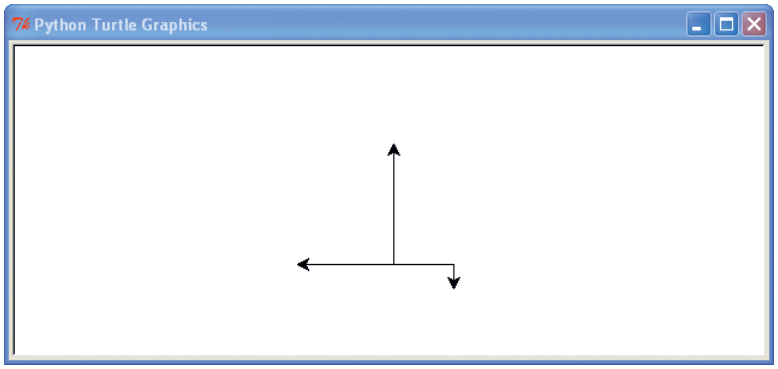


Now let's add another turtle, Jacob, and move him, too, without bugging Kate or Avery.

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



First, we create a new `Pen` object called `jacob`, then we turn him left 180 degrees, and then move him forward 80 pixels. Our drawing now looks like this, with three turtles:



Remember that every time we call `turtle.Pen()` to create a turtle, we add a new, independent object. Each object is still an instance of the class `Pen`, 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. Try it for yourself: Create another Kate turtle and try moving it around.

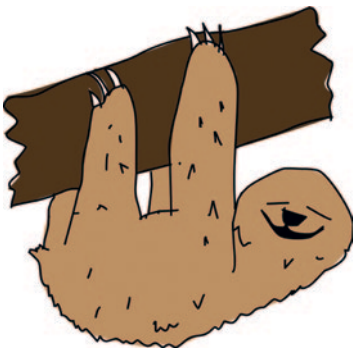
OTHER USEFUL FEATURES OF OBJECTS AND CLASSES

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 really large software application, like a word processor or a 3D computer game. It's nearly impossible for most people to understand large programs like these as a whole because there's just so much code. But break these monster programs into smaller pieces, and each piece starts to make sense—as long as you know the 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 that you use (like your web browser) were written by many people, or teams of people, working on different parts at the same time around the world.

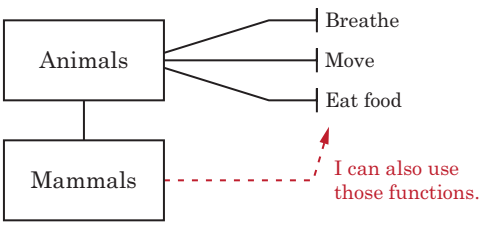
Now imagine that we want to expand some of the classes we've created in this chapter (`Animals`, `Mammals`, and `Giraffes`), but we have too much work to do, and we want our friends to help. We could divide the work of writing the code so that one person worked on the `Animals` class, another on the `Mammals` class, and still another on the `Giraffes` class.



INHERITED FUNCTIONS

Those of you who have been paying attention may realize that whoever ends up working on the `Giraffes` class is lucky, because any functions created by the people working on the `Animals` and `Mammals` classes can also be used by the `Giraffes` class. The `Giraffes` class *inherits* functions from the `Mammals` class, which, in turn, inherits from the `Animals` class. In other words, when we create a giraffe object, we can use functions defined in the `Giraffes` class, as well as functions defined in the `Mammals` and `Animals` classes. And, by the same token, if we create a mammal object, we can use functions defined in the `Mammals` class as well as its parent class `Animals`.

Take a look at the relationship between the `Animals`, `Mammals`, and `Giraffes` classes again. The `Animals` class is the parent of the `Mammals` class, and `Mammals` is the parent of `Giraffes`.



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

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

In fact, all of the functions we defined in both the `Animals` and `Mammals` classes can be called from our `reginald` object because the functions are inherited:

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

FUNCTIONS CALLING OTHER FUNCTIONS

When we call functions on an object, we use the object's variable name. For example, here's how to call the `move` function on Reginald the giraffe:

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

To have a function in the `Giraffes` class call the `move` function, we would use the `self` parameter instead. 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 `Giraffes` class:

```
>>> class Giraffes(Mammals):
        def find_food(self):
            self.move()
            print("I've found food!")
            self.eat_food()
```

We have now created a function that combines two other functions, which is quite common in programming. Often, you will write a function that does something useful, which you can then

use inside another function. (We'll do this in Chapter 13, where we'll write more complicated functions to create a game.)

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

```
>>> class Giraffes(Mammals):
        def find_food(self):
            self.move()
            print("I've found food!")
            self.eat_food()
        def eat_leaves_from_trees(self):
            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 `Animals` class to define `eat_leaves_from_trees` and `dance_a_jig` for the `Giraffes` class because these are inherited functions. By adding functions that call other functions in this way, when we create objects of these classes, we can call a single function that does more than just one thing. You can see what happens when we call the `dance_a_jig` function below—our giraffe moves 4 times (that is, the text "moving" is printed 4 times):

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

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 are getting it ready to be used.

For example, suppose we want to set the number of spots on our giraffe objects when they are created—that is, when they're initialized. To do this, we create an `__init__` function (notice that there are two underscore characters on each side, for a total of four).

This is a special type of function in Python classes and must have this name. The `init` function is a way to set the properties for an object when the object is first created, and Python will automatically call this function when we create a new object. Here's how to use it:

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

First, we define the `init` function with two parameters, `self` and `spots`, with the code `def __init__(self, spots):`. Just like the other functions we have defined in the class, the `init` function also needs to have `self` as the first parameter. Next, we set the parameter `spots` to an object variable (its property) called `giraffe_spots` using the `self` parameter, with the code `self.giraffe_spots = spots`. You might think of this line of code as saying, "Take the value of the parameter `spots` and save it for later (using the object variable `giraffe_spots`)." 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 (Ozwald and Gertrude) and display their number of spots, you can see the initialization function in action:

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

First, we create an instance of the `Giraffes` 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 `Giraffes` class, this time with `150`. Finally, 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 (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, a giraffe object can have its own 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. Finally, 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 again and again as you read the rest of this book.

PROGRAMMING PUZZLES

Some of the ideas in this chapter will start to make sense the more you use them. Try them out with the following examples, and then find the answers at *http://python-for-kids.com/*.

#1: THE GIRAFFE SHUFFLE

Add functions to the `Giraffes` 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 Reginald 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 = Giraffes()
>>> 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 `Pen` objects (the exact length of the lines isn't important). Remember to import the `turtle` module first!