Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 3

Python Wakes Up

Introduction

Okay, maybe creating pythons and monkeys wasn't the most exciting programming experience you can imagine. That's because the python and monkey didn't do anything. They simply existed. I reckon a philospher would find existence worthy of course-level inquirty but we programmers need action, we need movement. Well, we ought to be interested in existence too but just not in the same context as philosphy.

Anyway, in view of the last chapter largely centering around nouns, it is a natural segue to take a deep descent into verbs now. With this, we get the action we crave and we get our python and monkey to do something. If we want to refresh or more adequately prepare, there is the overview call-ahead in the **methods** section of chapter 2.

We're going to expand our knowledge concerning methods in this chapter. Not only are we going to explore different types of methods but we can going to juxtapose methods and a very similar construct called **functions**. Further, we're going to emphasize two important aspects of object-oriented design as a natural byproduct of how we model and create behaviors for our animals.

In the meantime, imagine we're back in the jungle scene. The wind is still gently blowing the tree canpoy back and forth. Where our python was once coiled in slumber, it now stirs and begins to rouse. The monkey takes notice of the slight movement...

Learning Objectives

1. Explain how functions and methods are used to encapsulate logic
2. Describe components needed to define functions and methods
3. Implement functions and methods according to object oriented design principles

Functions

A good place to begin is with a definition. We can take a a **function** to be a discrete unit of work. Technically speaking, there are three types of functions which we refer to as *built-in*, *user-defined*, and *lamba*. We're going to focus primarily on the user-defined type although we constantly use the built-in. Lamba functions are an advanced topic and best left for a later discussion.

Let's give give a python the ability to move by developing a user-defined function.

Example 1

```
(1) def slither(newX, newY):
(2)     x += newX
(3)     y += newY
(4)
```

We can put this **function** anywhere in a program and then call it using a statement such as `slither(10, 20)`. If we imagine the `x` and `y` to be coordinates, we just moved a python. Kind of. There is a fair amount of context and we certainly lack a complete program in which this function might be useful. That aside, there are five programming constructs contained in those three lines of code we want to know more about.

First, we have the `def` keyword. As a keyword, `def` communicates our intention to declare or *define* a function to (a) those people reading the code as well as (b) to the `Python` interpreter.

We also learned how to pass **parameters** to a function. Parameters are values we want to use *within* the function but originate from outside of the function. We have to provide these when we *call* or use the function. So, given **Example 1**, we might call the function like `slither(1, 1)`.

Next, we have somewhat of a two-for-one: we have the function **name** and the entirey of line one is a **signature**. The name is close to arbitrary. In earlier versions of Python, we were stuck with letters and numbers with underscores. However, in modern Python we can use just about any legal character. The construct elements on the right of the name add detail, kind of like an operational suffix. The two are so intertwined we always define functions using parentheses and we always call functions likewise.

Lastly, we experienced a new operator here, `+=`. We read this as, *increase the value of the left variable by the value on the right.* Python has the standard **operators** we would expect a programming language to possess. Rather than memorize a list, I think we'd better served by just using them. Worse case, we can find them by asking simple but precise questions such as *how do I add to integers in Python* and we'll find `x = 1; y = 1; x + y = 2`.

Those five constructs collectively define a **function**. Technically, the same is true for a **method** but we'll get to that in a little bit. In the meantime, there are two types of functions we should know since the difference is largely based on implementation, not discrete elements.

Pure

Sometimes develop functions that do not modify the values passed through parameters, does not produce an effect. If we pass the same parameter twice, the function does the same thing twice. Exercise caution here as these functions can have a `return` statement. Think of a **function** as such `strlen()`. Such a function can be called **pure**.

Modifiers

More commonly, we develop functions that modify values passed as parameters. These are referred to as **modifiers** or *modifying functions*. As a clear difference compared to **pure** functions, think of something like `time(today)`. Normally, the dead giveaway that we need a **modifier** function is when we want to modify an object which is passed as a parameter but the calling code maintains a reference so changes inside the function are manifested outside as well.

Waking the python

Certainly, we could wake our python using **functions**. However, we'd be stuck either in a simple procedural programming context or, at best, forcing object-oriented program constructs into procedural framework. Neither is desirable if you want to get to a place where a python is awake and maybe chasing a monkey. For that, we need **methods**. Fortunately, everything we've learned so far carries forward.

Methods

Let's look at an example before we dive into technical details. In comparing **Example 1** above and **Example 2** below, what would can we assert as differences?

Example 2

```
(1) def slither(self, newX, newY):
(2)     x += newX
(3)     y += newY
(4)
```

That's right, the only visible difference is the appearance of the `self` parameter in **Example 2**. Otherwise, the two code blocks are identical. However, if we zoom at a

little, we can see another difference which not only explains why we have **self** but also reveals the true difference between a **function** and a **method**.

Example 3

```
(1) class Python:
(2)     def slither(self, newX, newY):
(2)         x += newX
(3)         y += newY
(4)
```

Let's introduce a definition now to cement the difference. Whereas a **function** is a discrete unit of work, a **method** is a function that belongs to a **class**. I think it is fair to articulate that all methods are functions but not all functions are methods. Moreover, as a conseuquence we call functions directly while we call methods indirectly through an object.

By now we should be wondering why we need *methods* if we have *functions* and vice-versa. Put simply, not all `Python` programs are object-oriented. Since **functions** are not associated with **classes**, we use those when we want to develop a discrete block of task-scope work. On the flip, we define **methods** when we need the same programming construct within the context of a **class**.

That's not terribly complicated. However, there are some implicit differences we ought to spend the rest of our time investigating. Let's start with the `self` parameter and the move onto what we can do differently with a **method** in contrast to a **function**.

I should point out: I argue that leveraging object-oriented design principles leads to superior software in all but the most rudimentary problems. Further, if we know how to design using OOP and understand how **methods** operate, we can always opt to have a simple procedural program using **functions** but we cannot always do the opposite.

Self

Before we start discussing the `self` keyword, take a guess at what you think `self` refers to in our code? There's a hint in the phrasing of the question...

Okay, did you surmise that `self` is a reference to the object encapsulating the method? Indeed, the use of `self` is a big clue that we're dealing with object-orientation because only methods will use it. Well, technically **properties** do reference `self` as well but we'll talk about them later.

Init and Constructors

Back to methods proper now; we can talk about a special kind of method called a **constructor**. Take a look at **Example 4** which we first say in the last chapter. More specifically, look at line four.

Example 4

```
(1) class Python:
(2)     property = "value"
(3)
(4)     def __init__(self, value):
(5)         print(value)
(6)
(7)     def method(self):
(8)         print("I am a sssneaky ssssnake")
```

The `__init__` name is a special internal **method** used by Python. Python uses the **method**, calls `__init__` when we *construct* the object of a class containing it. Normally we don't execute logic within the **constructor** but we absolutely use the method to initialize **properties**.

Given that, realize we only need a **constructor** if we have something to initialize. Check this out:

Example 5

```
(1) class Python:
(2)
(3)     def wake(self):
(4)         print("I am awake...")
```

Pretty simple, no? Well, as we increase our design complexity we might find ourselves wanting to selectively wake up the python.

Overloads

If I'm being honest, the way Python handles method **overloading** is strange if you're coming from another language such as `C++` or `C#`. At the same time, if you haven't worked with **overloading** before, this is still going to be strange but for different reasons. Let's try to dispell enough of the strangeness so we can leverage this programming technique.

As a first principle, Python does not allow two **function** or **methods** to have the same name. In the cases when we can bypass this, the *last* one is what gets called. Conversely,

C++ allows us to use the same name as long as the signature is different. Remember function and method signatures from the beginning of the chapter, right?

Example 6

```
(1) class Python:
(2)
(3)     def wake(self, state=None):
(4)         if state is not None:
(5)             print("I'm still sleeping")
(6)         else:
(7)             print("I am awake...")
```

Now, when we call the `Phil.wake()` method we can pass a parameter to conditionally wake `Phil` up.

Polymorphism

Polymorphism is the last of the core object-oriented design tenets for us to examine. The idea is closely related to the **overload** concept because we focus our object design on using single constructs in multiple ways. This works primarily at **class** level but manifests specifically at **method** level. **Example 5** reveals how we can implement this principle.

Example 5

```
(1) class Animal:
(2)
(3)     def setName(self, name):
(4)         self.name = name
(5)
(6)     def getName(self):
(7)         return self.name
(8)
(9) class Python(Animal):
(10)
(11)     def setName(self):
(12)        name = "Phil"
(13)
(14)     def getName(self):
(15)        return self.name
```

Exercises

1. Implement a functioning version of **Example 5**. What value does `getName()` return when called from the `Python()` class?
2. Take the program from **Exercise 1** and implement a **constructor** in the `Python()` class that initializes `name` to `Phi` instead of waiting for the `setName()` method call.

Questions

1. What type of function is represented in **Example 1**?
2. What object-oriented design principle(s) are enforced as a conseuquence by calling methods indirectly through an object?
3. How do you think polymorphism is useful for how we might implement our python and monkey world?