Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 2

Python and Monkey

Introduction

This is it, the exciting stuff! Our Python environment is setup and broke in. We've played with a few traditional programs, answered some probing questions. Hopefully, we've found out that there is quite a bit we don't know yet but we have the taste now. We have the taste and we want more.

Perfect. Aboslutely perfect.

Two things I want to draw your attention to before we move forward. First, beginning with this chapter, we are going to progressively build out one project. All of the **Examples**, **Exercises**, and **Questions**, and will serve a common objective: getting a python to chase a monkey.

There is a lot of ground work to cover and this chapter outlines the major building blocks in our Python programming foundation. Accordingly, there are call-aheads to later chapters. While there is no harm in reading ahead, we should be cautious to work in order. Second, our second step in programming will be onto an unusual path. This will require some patience on our part as develop enough of a framing context for concepts and constructs to make sense. We need to trust the process here. It will pay off.

For now, let's imagine a calm jungle scene. The wind blows and the green tree canpoy sways. On one side, a python is coiled in slumber. Across the detritus strewn jungle floor, a monkey examines scratches his head and makes some monkey sounds…

Learning Objectives

1. Summarize how object oriented design addresses flaws in procedural programming.
2. Define and create classes and instantiate objects of those classes.

3. Evaluate class design strategies.

Object Oriented Design

Yes, learning object oriented programming first is rather taboo. In fact, you would be hard pressed to find a textbook used in university which does it. I honestly don't know why. My guess is traditional programming assumes students are overwhelmed by the strangeness of object orientation. I disagree.

I think learning classes and objects first is natural since these are the way we have to model persons, places, and things. In other words, nouns. That's an important obversation for us to grasp: classes are nouns, thus model *things*.

Let's start with a question: what is a *noun*? We should come up with an answer before reading on. Take a moment.

Go it?

Okay, we can ask this question because programming languages are a type of formal language. While Python does not have a noun per se, it does have a syntax that we can imagine as a noun. That noun-like syntax is the foundation, the bedrock of modern programming - the **class**.

We should think of a **class** as a noun with its *nounness* written out. By nounness, I mean the elements which make it a noun as opposed to something else. For instance, nouns have names. . . python, monkey, or tree for common nouns. Likewise, we can have nouns such as Phil or Marvin for proper nouns. There are other elements defining nounness which we'll explore later. For the time being, let's say we instantiate **classes** (common nouns) as **objects** (proper nouns).

I think the power and beauty of object oriented design rests in being able to model real-world things in software. We can express what precisely power and beauty mean in four steps.

First, object oriented design is intuitive. We should find it intuitive because that is how we interact with things in the world.

Following naturally, I next suggest we will find object orientation carries with it a natural pull towards robust organization. Because we build top-down instead of bottom-up, we almost always are starting with the outer most constraints. I think this forces us to organize in a manner that is at once logical and representative of our world.

Furthermore, the power comes from the encapsulation and inheritance principles associated with object oriented design. Meanwhile, the beauty comes from the ability to develop custom form and function through the object orientation. Before we explore those topics, we need to explore **classes** and **objects** a bit more.

Classes and Objects

Remember, at one level of resolution, we can define a **class** as a noun. For our purpose, let's take a python (the snake, not the language) as a thing and then model it in software. The resultant class would look something like the code in **Example 1**.

Example 1

```
(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")
```

This example demonstrates an entity we call `Python()` which is the common noun part of the `Phil = new Python()` instantiation. Ideally, we model a real-world python in our `Python` class. The means to do so stem from the **properies** (data) and **methods** (actions) which we can see in lines two and four in **Example 1**.

The idea that a class is a noun is half of the truth. The full truth is that we are defining custom *types* as we model a person, place, or thing. We can see this in how we *instantiate* a class as an **object**. Let's pause to think about the possible consequences of this truth.

Normally, our first foray into *types* comes when we learn about **variables**. We explore variables as the focus of chapter 4 but suffice it to say that variables have *types* such as integer or string. As we can see here, it is the underlying *type* which provides the noun substructure in the programmatic context. Thus, by defining `class Python:` we are actually defining a *type* in identical fashion to how integer defines `42` as a discrete numeric value.

Back to **Example 1**, we should take a few moments to describe what is going on because this is the heart of how object oriented programming addresses flaws in procedural programming. As silly as the simple example may be, we really do have a software model of a snake now. Thus, another way to think about classes is as blueprints. That is, unformed but fully modeled solutions. Accordingly, when we build an instance of a class we are creating or instantiating an object into our computer's memory.

Example 2

```
(1) class Python:
(2)     property = "value"
(3)
```

```
    def __init__(self, value):
        print(value)

    def method(self):
        print("I am a sssneaky ssssnake")

Phil = Python()
```

Programmatically, we are asserting `Phil = new Python()` wherein `Phil` becomes a specific `Python()` through *object instantiation.* What do you think about that?

We can say `Phil` is a `Python()` which is itself a type of snake which is a type of animal. Similar, we can group the attributes defining each of those constructs within it. This harkens back to how a noun has elements defining its nounness. More specifically, we should observe two things in **Example 2**.

First, our class Python establishes what the object `Phil` is at a core, fundamental level. Whatever elements- chiefly, methods and properties- are defined in `Python()` are now *in* `Phil`. Access to those **methods** and **properties** is done through `Phil`. Further, if we read the statement literally, we can assert that we are assigning `Python()` to `Phil`. This type of statement is the heart of programming and a concept we will return to quite frequently.

There is one final note for us to consider. A consequence of `Phil` being a specific type of Python is that we can repeat the object instantiation to create more pythons if we want or need more snakes. We simply need to change the proper noun portion, the object name. Take `Presley = Python()` for instance or `Piotr = Python()`.

On one hand, these are all instantiations of the `Python()` **class** and thus are identical. Yet, on the other hand, these are discrete **objects** in memory and thus represent unique instances of `Python()`. Again, this is a demonstrative difference compared to procedural programming where we are forced into singular entities by design. We can probe this more by examining **methods** and **properties**.

Methods

We'll dive into **methods** in Chapter 6. For the time being, I want to mention a few things about **methods** that are stricly related to what we have learned so far about **classes** and **objects**. In doing so, I want to create a bridge to **properties** and setup a call ahead for the detailed **method** discussion

Principally, **methods** are verbs to the nouns we define as **classes**. An easy verb to imagine might be `slither()` for our `class Python`. We can build anything as a **method**, however we want to use them to model reasonable behaviors for our noun. In this way, the behavior or action is tied to the thing.

Deepining the tie between **methods** and **classes**, we can look at the use of the `self` notation in **Example 1** and **Example 2**. The `self` notation is how our program internally distinguishes between `Phil` and `Piotr` as two instances of the `Python()` class. While we could use any term here (I'm personally tempted to use `this` based on my experience with `C#`), the common convention is `self` due to the strong internal reference present. This begins to make even more sense when we consider **properties** as part of our growing **class** knowledge.

Properties

If a **class** is a noun and a **method** is a verb, a **property** is an adjective. Moreover, if we use **methods** to express behavior, we likewise want to use **properties** to express characteristics. Chapter 4 contains our thorough review of the topic. Meanwhile, we can work on understanding how to implement and use **properties** by envisioning the qualities of a python we'd communicate to another person.

Foundationally, we need to disambiguate **property** and **attribute** because there will be a temptation to use the terms interchangeably. In a general sense, the two are the same thing with respect to how we talk about using them. However, the technical programming implementation is substantively different. Let's compare and examine.

Let's start with a somewhat vanilla attribute implementation. #### Example 3

```
class Python:

    def __init__(self, length=0):
        self.length = length

```

With our `Phil` object instantiated, we can interact with the `.length` attribute directly as `Phil.length = 6` to assign a value or `print(Phil.length)` to output the value. This is perfectly valid and fine. That's not the same as a **property** though.

For comparison, here is **property** implementation of the same idea. #### Example 4

```
class Python:
    _length = 0

    @property
    def length(self):
        return self._length

    @size.setter
    def length(self, value):
        self._length = value
```

Again, we assume we have `Phil()`. Here, we interact with the **property** using **get** and **set** methods to, you guessed it, *get* and *set* property values, respectively. As an illustation, consider `Phil.length` to *get* the value and `Phil.length = 6` to *set*. Yes, there is nothing that stops us from doing `Phil._length = 6`. Be that as it may, I suggest doing so would be contextually and techincally wrong given the overall **class** implementation in **Example 4**.

The reason why I center our attention on the **property** implementation is not because I think **attributes** are incorrect. Rather, I prefer the **property** style implementation because it gives us proper *encapsulation.*

Encapsulation

The object oriented design concept of *encapsulation* gives us the ability to essentially hide or protect values. By values, I mean data. The goal here is to limit complexity because encapsulation limits the coupling of the related components by exposing a single or small set of data accesses.

Let's look back at **Example 4**. We do not access `length` straight through as we would if we called a **variable** in a procedural program. Instead, technically the **class** provides the encapsulation by allowing us to go through it to access `length`. However, I suggest this more by proxy than by direct assertion. This is why the **property** mechanic is important in conjunction with the *getter* and *setter* **methods**. Again, for completeness we could achieve the same functionality using **attributes** because Python as a language empowers us to choose the correct implementation. Yet, if we use the **methods** we have an implementation more aligned with other object oriented paradigms.

Thus, with the object oriented concept of encapsulation we have a clean linking between all of the preceeding constructs. The last thing we should define now is something that brings us all the way back to the chain of *animal* $\rightarrow$ *snake* $\rightarrow$ *python* nouns but in a programmatic context.

Inheritance

The simple explanation of *inheritance* as an object oriented paradigm can be visualized in the *animal* $\rightarrow$ *snake* $\rightarrow$ *python* chain. In such a chain, *animal* is the `base` **class** and *snake* would be a `derived` **class** inheriting from *animal*. Legitimate synonyms are `subclass` for the `derived` and `superclass` for `base`. Let's look at a refactoring of **Example 4** using this logic.

Example 5

```
(1) class Animal:
(2)
```

```
   def __init__(self, name):
       self.name = name

   @property
   def name(self):
       return self.name

class Python(Animal):
    _length = 0

    def __init__(self, name):
       Animal.__init__(self, name)

    @property
    def length(self):
       return self._length

    @size.setter
    def length(self, value):
       self._length = value
```

With an implementation such as **Example 5**, we can call `Phil = Python("Phil")` followed by `Phil.name` and we'd get `Phil` as output. The reason for doing this would be to promote reusability in our code.

Consider this point: we are looking at `Python()` but what about our future `Monkey()`? Monkeys are animals too and assuming we give them a name inheritance keeps us from having duplicate, confusing `name` code across mutiple classes.

I bring this up now because there will be incentive to avoid designing our object oriented projects without giving attention to inheritance. We must resist this path of least effort. Like automation, even though we will spend more time upfront the result will be lower total maintenance overhead and more elegance in our solutions.

Yes, the additional arrow asserts *snake* is also a `base` for *python* which I suppose would work but is confusing in practice. In practice, we assert a *singular* or *multiple* inheritance. Singular represents a basic one-to-one relationship. Mutiple is the opposite of what we probably think- one `derived` **class** inherits from multiple `base` **classes**. By way of illustration, we could assert *animal* → *python* and *example* → *python* so that *python* inherts from both.

This is one of the more advanced concepts introduced in this chapter and is a good place for a break. We'll come back to the principle of inheritance throughout the rest of the book. Until then, let's get some practice in shall we!

Exercises

1. Develop a python class identical to **Example 1** and run the program. What happens?
2. Implement **Example 4** but using `this` instead of `self` and run the program. What happens?
3. Modify the `Animal()` **class** to include the `length` property, remove it from `Python()` and have `Python()` inhert from `Animal()`.

Questions

1. Read the code in **Example 1** and come up with two properties you might define for a Python class. The code **class**, not your school class.
2. What about **methods**? Specify two **methods** your Python **class** might define.
3. Why would we use a **property** instead of an **attribute**?
4. How could multiple inheritance become problematic?
5. If someone calls out **Python**, what are the first and second steps you would take to develop a `Python` class?