Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 4

Python Counts Scales

Introduction

Our python is awake and ready to chase the monkey. Before setting off, our python needs to count its scales. Why would a python count its scales? Maybe the python wants to make sure none are missing so that there's no loss of traction. Alternatively, the python may realize the reason to count its scales is to facilitate the examples in this chapter. After all, scales and the counting of them are representative of **variables** and **properties**.

We may not realize it but we already know what variables are and how to use them. Basic algebra introduced us to the idea that some things can act as placeholders for other things. The things that stand for other things are variables. Further, when we say things like, *some number x* or *for a value y*, we are using varaibles. Furthermore, we've worked with properties in earlier chapters albeit simple versions of properities.

Before you panic, variables in programming are much, much easier to deal with than variables in mathematics. Strictly speaking, we don't manipulate variables in programming. We don't solve for them. Rather, there are specific programming considerations which extend the definition and increase the usefulness. We'll get to know these better by observing python counting its scales.

Learning Objectives

1. Articulate key characterstics of variables and how they are important for storing data
2. Summarize the purpose of properties and variable scope
3. Implement methods to set and retrieve stored values

Variables

On one hand, a variable is like a python scale because variables and scales are discrete constructs. There is no half scale or two and a half variables. There is only one and it is entirely itself. On the other hand, counting scales represents how variables are used. That is, we store values in them.

You may recognize a statement like `x = 2`. Here, we can draw upon our algebra knowledge to identify the `x` as a variable. Unlike algebra where we attempt to solve the equation, here we are assigning the value `2` to the variable `x`. The dead give away is the `=` symbol which is our assignment operator.

The technical explanation is a bit more arcane but bears discussion. Strictly speaking, variables provide a reference or pointer to an object. A dirty pythonic secret is that *everything* is an object. Accordingly, we can store *anything* in a variable. For convenience, we call the variable through an assigned symbolic name and thus can access the held object.

Speaking of *assignment* - we have learned yet another Python **operator** in the form of the assignment symbol `=`. Briefly, this operator takes the value on the right of it and assigns it to the object on the left. In the context of this chapter, the specific type of object on the left is a **variable**.

Let's connect all of this up conceptually.

Remember before when we learned that everything is an object in Python? Good. We can then say then types are classes and variables are instances of those classes.

Types

In math, we used discrete variables to hold continous values. Despite the diffence between say a positive and negative number, the variable is just a variable. Comparatively, variables in programming can be configured to hold only certain **types** of values. The **only** is operationally important and something for us to give due attention.

How and when variable types can be declared or changed is what defines the kind of type system associated with a programming language. I'm going to tell you this might be one of the most misunderstood areas of programming. The language used to describe typing systems is unfortunatley not as clear as the types themselves. The direct result is **variable typing** is repsonsible for the majority of errors.

There are two kinds of type systems: *strong* and *weak*. Strong typed variables can be trusted to not encounter unexpected value type changes when our program runs. Check this out:

Example 1

```
(1) Phil = "has scales"
(2) Phil = Phil + 2
```

What might we guess the outcome to be from line two? We should treat this as a basic experiment; fire up the Python interpreter and run the code. See - that's the beauty of programming... we never have to guess. We can prototype (something Python is exceedingly good at!).

If `Phil` is a python, and thus a *strong* typed snakey snake, he will not be happy. In contrast, believe it or not, some languages such as `Perl` allow for such implicit typing shenangins because they have a *weak* type system. This is technically the source of the *loosely* typed concept.

Moving forward, both *strong* and *weak* forms can be *static* or *dynamic*.

As we demonstrated in our **Example 1** experiment, Python is indeed strongly typed. I think the fact that Python is dynamically typed leads people to consider the language to be *loosely* typed. This is strictly and emphatically incorrect.

I think the confusion comes from and how we might overcome it. You see, in other languages such as `C` or `C++`, we define the variable type when we declare the variable. For example, here's a standard `C` way to declare an integer variable and initiatlize it to a value of zero:

```
int scales = 0
```

The pythonic way to do the same thing would be `scales = 0`. Do you see the difference?

To explain: the difference is in a language like `C` we explicitly tell the variable what type of value to hold whereas in Python we use a value to implicitly inform the variable what type is being given to it. Said another way, the difference is like us reading a definition for a vocabularly word versus us deriving the meaning by reading the vocabularly word in a sentence.

With all that said, Python the language has seven categories of *built-in* types. We should learn to recognize and use three of those categories: **text**, **numeric**, and **boolean**.

Text

Python includes the `str` type which is associated with text. While straightfoward, we need to be aware of the difference between a `str` type using single quotations versus a `str` using double quotation marks. The difficulty here is the two forms of `str` can be used interchangeably except for when one condition isp resent in our text. To illustrate this, try the following example in the interactive interpreter:

Example 2

```
(1)  print("I'm a python with over 9000 scales")
(2)  print('I'm a python with over 9000 scales')
```

Line two ought to return an error condition. We receive an error because Python reads text contained in single quotes *literally* which means special characters are interpreted. By contrast, double qoutes signals that Python we treat everything as part of the text.

I suggest a simple rule of thumb (or scale!) we should follow is to wrap text in double quotes unless we have a specific reason to use single quotes.

Numeric

The most diverse type is the numerics. While languages such as `C` have multiple subversions of say the `int` type, Python provides just the main numeric types of `int`, `float`, and `complex`. This simplifies everything a great deal.

Think of our python `Phil` counting scales. Scales are **integers** and fractional scales would be **floats**. The difference is a mathematical definition best understood through a simple example.

Example 3

```
(1) numberOfScales = 0
(2) bananas = 2.5
```

Notice the lack of quotes wrapping the values in **Example 3**; we don't use quotation marks because numeric values are not text values. Neither are **Boolean** values either which is our next consideration.

Boolean

In the science of computing, almost everything revolves around the binary concept of `True` or `False`. As programming is a practical expression of computer science, languages such as Python have the ability to express this concept using a specific type we refer to as **Boolean**. More specifically, Python has a 'bool' type.

Again, think of our python `Phil` counting scales. Scales are either present (`True`) or not present (`False`). See **Example 4** for how we might handle such a proposition.

Example 4

```
(1) hasScales = True
(2) isMonkey = False
```

I should call attention to the lack of quotes again. We're not using `True` or `False` as text strings so quotes are not necessary (or useful).

As a side note, I suggest we should get into the habit of naming our `bool` type variables with some appropriate conjugation of `is` prefix. By doing so, we communicate in our source code that the variable is an *explicit* **Boolean** type. Later, when we start to explore **conditionals** we will see how many objects in Python have an *implicit* truth value. For now, we should exercise prudence and not confuse implicit types with something related but distinct called **type casting**.

Type Casting

There are occasions in which we want to forego the implicitness of the dynamic typing system and make a type explicit. Well, saying explicit here might be misleading. What we occasionally need to do is *convert* from one type to another. In my observation, the most common programming situation requiring such behavior is when we take input from a user (default type is `str`) but want to use the input numerically (i.e., `int`).

There are two ways to cast a type (to another type). Consider the following:

Example 5

```
(1) int_scales = 100
(2) str_scales = "100"
```

Based on our discussion of **types**, we should be able to recognize line one as a numeric integer and line two as a text string. Yet, the values *appear* to be the same. This is a pain point for us if we're not paying close attention.

If we assume we cannot change line two, then we have to convert the value upon use. This is where explicitly converting from one type to another comes into play. Fortunately, Python has built-in **functions** directly mapped to the **type** names. There are more than we might first imagine and not all of the functions may work the way we would expect. This is a good opportunity to take a break and go read the language specification for built-in functions.

Back from break? Excellent. Now, let's consider an instance in which we want to convert or *cast* from an an **integer** numeric type to a **string** text type, we would use `str(int)`. More specifically with reference to **Example 5**:

Example 6

```
(1) s = str(int_scales)
(2) print(s + str_scales)
```

When we run **Example 6**, we should expect the output `100100` because the `+` operator is contextualized by Python as a *concatenation* operator, not the *addition* operator. If we want to use the `+` operator as addition, we need Python to infer such based on the

values on either side of the operator. The technical explanation is Python can and will use *implicit* typing.

Example 7

```
(1) int_scales = 100
(2) str_scales = "100"
(3) f = float(str_scales)
(4) print(int_scales + f)
```

Why does line four in **Examples 7** output `200.0` and not `200`? The straightforward answer is Python will implicitly cast to the type which preserves the *most* data. In other words, Python always casts to the more precise or specific type. Thus, with numerics we go from smaller to larger as observed with **Example 7**. The *explicit* type casting will truncate larger to smaller; be careful with statements such as `int(float_value))`.

As an aside, if we're ever curious about what **type** a value is or a variable holds, we have the built-in function `type()` which will return exactly that for us. Pretty neat.

Now, to really understand **type casting**, and the broader idea of **variables**, we should go back through **Example 7** but run `print(int_scales)` as a line five. The ouput should be `100` because we didn't store the ouput from line four in `int_scales`; we just used the variable and the value it holds as a parameter. But what if we wanted to overwrite `int_scales` with the new value? We can do that, no problem, as long we have `int_scales` in scope.

Scope

Think about the scales on our python. The scales on its head belong to its head, not its tail. Thus, the head scales are *scoped* to a region. That's a half baked metaphor but so is the idea of a python chasing a monkey. Let's just keep slithering with it, okay?

If you're looking for a more concrete analogy, we can talk about the nesting of *number sets*. That is, all *natural numbers* belong to the set of *integers* and *integers* exist within the set of *real* numbers. A Venn diagram in prose.

Operationally speaking, there are five scopes to consider: built-in, global, enclosing, and local. The analogy by numbers is appropriate here as long as we focus on the nesting aspect. At the same time, if we want to be proper, we should refer to these are **namespaces**. For the most part, I'll stick with the *scope* term because it has more operational weight. More important than knowing the five scopes are namespaces is knowing that each has a *lifetime*. The lifetime is what technically defines **scope** as we program it statically and experience it during runtime.

Built-in scope

The **built-in** scope is established when the Python interpreter is invokved. This namespace includes all of the built-in objects loaded by default. We have played with a handful of these just in this chapter: **types** such as `int`, `str` and `bool`; functions such as `print()` and `type()`.

Exploring all of the objects included in the built-in namespace is well beyond the scope of our python counting its scales. Suffice it to say, the critical takeaway with built-in scope is dualfold. First, it ends when the Python interpreter closes. This hopefully makes face-value sense; if Python is not running, there is not Python to provide built-in objects. Second, there can only be one instance of this namespace at any given time.

Global scope

Slightly less limited in lifetime is the **global** scope. Global is the boundary within which our *program* exists. It begins when we execute the program as a parameter to the Python interpter such as `python3 myprogram.py` and ends when the program is finished. We might ask what happens if we run the Python interpreter interactively. The answer is how we end up with mutiplie global namespaces running concurrently which is a side-effect of using the `import` statement.

Thus, we can assert that declaring `name = "Phil"` immediately after our shebang line exists at the same scope level as calling up the `Animal` class using an import statement such as `from animal import Animal`.

Enclosing and Local scopes

In comparison to **built-in** and **global** scopes, **enclosing** is somewhat less well defined. This is true because an enclosing namespace is defined dynamically as our program executes based on the presence of specific keywords such as `class` and `def`. **Local** scope works the same way but in a more narrow lifetime context. Let's consider an example to illustrate the interchange between these namespaces.

Example 8

```
(1) count = 1
(2)
(3) def scale_count():
(4)     count = 0
(5)     print(count)
```

In **Example 8**, the variable `count` on line one exists globally (assuming there is not a function, method, or class *enclosing* it). The function or method at line three establishes an **enclosing** namespace within which we have another `count` variable. The *inner* `count`

has a scope **local** to the `scale_count` function. If we run the code, we can see that the `count` value output from a call to `scale_count()` is zero. If we pass the global `count` to a `print()` function we will get `1`.

Wait- what if we want to update the value in the global `count` within the enclosing namespace of the `scale_count` function? This is where **properties** come into play as a way to design our programs.

Properties

A **property** may look like a **variable** but that is only because we're lumping unlike *things* together. The best way I know how to untangle this coiled python is to talk through an example. With that in mind, let's say upfront that a **property** is how we work with a variable we want encapsulated within the class implementation.

Example 9

```
(1)  class Python:
(2)
(3)      def __init__(self, init_name):
(4)          self.__name = init_name
(5)
(6)      def get_name(self):
(7)          return self.__name
(8)
(9)      def set_name(self, new_name):
(10)         self.__name = new_name
```

Here, **Example 9** shows a typical implementation of properties. In some circles, the `__name` construct will be referred to as the property. For our purposes, that's close enough. However, since we have the discussion started let's play it out. Technically, the `__name` *instance* variable is a **field** or **attribute**. The two methods declared on lines six and nine are the **properties** or **property class methods**: one to *get* the value of our **attribute** and another one to *set* the value on demand.

An interesting note is that `__name` is *private*. The field cannot be accessed (read or written to) from outside of the two **properties**. The reason we do this is to abstract (remember: a principle of OOP is *abstraction*) away the logic in the fields. We are free to change the internals without negatively impacting other programs using the publicly exposed **properties**.

Now, let's consider the pythonic way to implement the same design:

Example 10

```
(1)  class Python:
(2)
(3)      def __init__(self, init_name):
(4)          self.name = init_name
(5)
```

Yes, that's the Python way of implementing the same code. This has the advantage of being more readable and less complex. However, we're sacrificing encapsulation since the `name` field is exposed with implicit, public acccess to read and write values. There's a compromise between the two examples. Let's look at **Example 11** which you'll find similar to code we saw earlier in the book.

Example 11

```
(1)  class Python:
(2)
(3)      def __init__(self, init_name):
(4)          self.name = init_name
(5)
(6)      @property
(7)      def name(self):
(8)          return self.__name
(9)
(10)     @name.setter
(11)     def set_name(self, new_name):
(12)         self.__name = new_name
```

The two `@` keywords are **decorators** and inform the Python interpretor that the methods are to be treated like **properties** and `name` is to be treated just like the private attribute in **Example 9**.

There is actually a third way because even the **Example 11** implementation has potential flaws. The details are outside of the scope of how we want our python to count its scales. My advice: if you're going to transition into another language later on, get used to the **property** implementation in **Example 9**. If you're sticking with Python, use the style shown in **Example 10** but be aware of how **Example 11** works.

Exercises

1. Using the Python interpreter, replicate **Example 5** and then attempt to output the result of adding the two variables.
2. Using the Python interpreter, declare `str_scales = "100"` and then write `print(bool(str_scales))` but don't hit `Enter` yet! Guess what you think the output will be. Now, hit that `Enter` key!

3. Create a `Snake` class which incorporates a class level `scale_count` property with a pair of `get` and `set` methods. If you're feeling adventurous, create three identical classes but implement the properties in the three styles shown in the chapter.

Questions

1. How would you correct **Example 2** so the `str` expression was output instead of an error?
2. Can you imagine a different exception to the single quote `str` which results in the same error as **Example 2**?
3. What would you change in **Example 5** so that we got the numeric result of `200`?
4. Why do we get the output we do in **Exercise 2**?
5. Is the function in **Example 8** *pure* or a *modifier*?
6. How do you think we might use **scope** in class design if we're following object-oriented principles?