Python Chases Monkey

An Introduction to Python Programming

Jason M. Pittman

Chapter 5

Python Sees Monkey

Introduction

A python cannot chase a monkey if the python does not see it. This might be the most Zen thing I've written so far, maybe ever. Despite the philosophical trappings in such a statement, the fact remains true. Thus, even in the language used to express the idea, we find the basis of *conditional* evaluation. Thus, if we were to render the chapter title as a complete sentence, we might come close to, *if a python sees a monkey, the python chases the monkey.* Yes, that reads as exactly what I meant.

Accordingly, the *if* becomes operational important and forms the main thrust this chapter. We will study *if* as the basis for **conditionals** along with a host of associated supporting keywords and operators. While the supporting keywords and operators may seem inconsequential, we'll soon discover just how much power rests in their subtlety. Futhermore, I suggest we'd be doing our python and monkey a great disservice if we explored conditionals without also considering how to repeatedly evaluate the condition. Doing so involves **iteration** or what many call **loops**.

With that, let's rejoin the python as it finishes counting its scales. The tree canopy is now still. Suddenly, the sound of dead leaves and dry twigs snapping resounds through the jungle. The monkey freezes. If the python sees it, the chase will begin...

Learning Objectives

1. Interpret and evaluate logical expressions
2. Construct conditional statements to affect program flow
3. Process logic iteratively with repetition controls

Conditionals

Despite what we might think, programming is much more than simple, step-by-step processing of statements. The reason for this is that not many problems can be solved through mere procedural execution. Instead, we often need to check something and, based on the result, take a different logic path. This is where **conditional** evaluation comes in for us.

Let's say that a **conditional** is a form of decision structure. The decision is one we, the programmer, makes *before* committing code to file. We do so by designing statements using *relational* operators such as `>`, `<`, `==`, and `!=`. These allow us to pose a question and receive a propositional evaluation of **true** or **false**.

Speaking of propositions, we also get access to powerful Boolean operators and associated keywords such as `and`, `or`, `not`, `is` and `in`. These enhance our conditional toolbox by allowing for extremely readable code. At the same time, attention to detail is important as we read, however. It is easy- too easy in my experience- to mistype `==` as `=`. The former is the equivalent of stating *is equal to* whereas the latter is the assignment operator we explored in the previous chapter.

Keep in mind, by using conditional statments we are designing decision informing questions that have propositional outcomes. The desired effect is one of control over what programming statements will be executed. This idea leads us to the first conditional keyword.

If

The fundamental **conditional** component is the `if` keyword. Using the `if` keyword, we can build the basic conditional pattern as follows:

Example 1

```
(1) if something is true:
(2)     do that thing
```

Notice how line one is interrogative in structure. Then, as long as `something` is indeed true, the program will move into the scope defined by the conditional and line two will run. This should cause us to wonder what happens if `something` is actually false. Well, the program continues on as if nothing happened. More technically, line two is skipped over.

In practice, the `something` in **Example 1** will almost always be some variable. I use *variable* loosely here; the better term would be **object**. Further, the `if` keyword causes our program to evaluate whether the value held by our variable-object meets (or does not meet) a stated **condition**. The result, as we've established, is either true or false. The

condition is manufactured by inserting one of the aforementioned operators between the variable-object and the condition.

Example 2

```
(1) if monkey.isSeen is False:
(2)     monkey.eat(banana)
```

Example 2 reveals a practical instance of our fundamental conditional pattern. At the heart of it, all **conditional** statements adhere to this pattern.

One thing to to notice as we explore **conditional** statements operators is there is a loose equivalence between some, say `is` and `==`. Elsewhere, operators such as `in` or `>` are functionally associated with specific **types**. We cannot use `in` as an operator in **Example 2** for instance and we cannot compare two `str` types using `<` or similar operator.

We can however nest conditionals within conditionals.

Example 3

```
(1) if monkey.isSeen is False:
(2)     if monkey.hasBanana is True:
(3)         monkey.eat(banana)
```

Nesting a conditional can be restated as a compound single conditional in many cases. The consequence can be more readable code since the propositional connective (e.g., `and`) is made explicit. Check this out:

Example 4

```
(1) if monkey.isSeen is False and monkey.hasBanana is True
```

If-Else

Often we want to be able to control what happens when our condition is not in addition to when it is met. This is not the same as the nested or compound single `if` as we can observe. The principal difference is the `if-else` has two controlled outcomes as opposed to a single controlled outcome.

Example 5

```
(1) if monkey.isSeen is False:
(2)     monkey.eat(banana)
(3) else:
```

```
(4)     monkey.climb(tree)
```

If-Else-If

We have one last conditional pattern to examine which adds a necessary complexity to our conditional control structure. As we might notice in the `if` and `if-else` patterns, we actually have just a single condition present. There are cases in which we might want additional condition. For instance, the possible decisions we can model for our monkey based on whether it is near a tree might look like the following example.

Example 6

```
(1) if monkey.nearTree is False:
(2)     monkey.run()
(3) elif monkey.nearTree is True:
(4)     monkey.climb(tree)
(5) else:
(6)     monkey.freeze()
```

I need to the point out lines five and six. While we can easily program the `if-else-if` pattern without the *fall-through* `else` and the code will function. However, imagine what might happen if somehow our monkey object doesn't have the `nearTree` property set. In real life, a `freeze()` method is common when we don't know how to react to a given situation and that is precisely what we've developed here for monkey.

Of course, the idea lacking is the aspect of repetition. After all, we don't freeze once or climb a single tree. No, we have a series of such decision making events across time. Fortunately, we can model that too using **iteration**.

Iteration (Loops)

What many people refer to as a **loop** in programming, we are going to refer to as **iteration**. The reason for this seemingly innocuous distinction is so we can explore the concept of **iteration** separately from the implementation of the concept which relies on keywords related to the mechanics of a programming **looping**. Put simply, *iteration* is an idea of how a program iterates over some code statement(s) whereas *looping* is the implementation of the idea manifest in keywords.

More particularly, we can state Python has two types of **iteration**: definite and indefinite. While both represent a means to repeat code statements (i.e., blocks of *logic*), the amount of repetition is precisely what the definiteness refers to in this context. Therein, **definite iteration** indicates a predefined amount of repetition. In contrast, **indefinite iteration** requires a conditional as the terminal factor.

In other words, when we program a statement to the effect of *execute the `print()` statement 10 times* we are using definite iteration. A statement along the lines of *`print()` until `False`* then is an example of indefinite iteration.

Fortunately, iteration is not just theoretical. The ideas of definite and indefinite iteration align with the practical application through a **loop**. Python makes the two iteration practicals straightforward to remember as there are only two keywords: `for` or `while`.

For

The `for` keyword is tightly coupled to **definite iteration**. The implementation pattern is a dead giveaway of both the idea and the keyword:

Example 7

```
(1) for a_variable in some_iterable:
(2)     statement using a_variable
```

The definite iteration keyword `for` is clearly visible. That is certainly a major part of the pattern. However, we have to pay attention to `in` as an operator and `iterable` as an object of the iteration. Keeping the notion that Python is meant to be read more than written, we can interpret (and read) line one literally.

Example 7 reveals a way we can take advantage of Python's dynamic type system. That is, we are iteratively and sequentially assigning values from the object *some_iterable* to the variable *a_variable*. Then, on line two, we use the values from *some_iterable* in some statement one by one until we have gotten to the end of the values in *some_iterable*. That's right- Python handles the definite iteration on our behalf so that we don't have to manually handle iterator values as we might in `C` or `C++` (e.g., `int i = 0; i++`)

Most often, we will use a `for` loop to iterate over a list or tuple. We'll learn about those data structures in a later chapter and the built-in function. For now, we should consider a functional example, something like:

Example 8

```
(1) names = ["Phil", "Peetie", "Piotr"]
(2) for name in names:
(3)     print(name)
```

If we get curious- and we should- we can execute an identical process without `for` in our interpreter. Doing so may give us valuable insight.

While

The 'while' keyword is directly associated with **indefinite iteration**. The *indefiniteness* stems from the lack of definitive iterative range. In **Example 8**, we know the iteration will cease when the last `name` gets pulled from the list. Compare that to the following example and try to figure out *when* the iteration will end.

Example 9

```
(1) while name is not "Phil":
(2)     print("I'm not Phil")
```

As long as `name` doesn't hold the value `Phil`, the iteration will continue. In fact, what **Example 9** demonstrates is an *infinite loop* potentially. Such behavior is possible because this type of iteration uses an implicit **conditional** to exit the loop. Thus, with **indefinite iteration** we have to exercise an abundance of care in making certain we have a means to control when the fun stops so to speak. Often such control occurs within the logic block encased in the scope of the iteration. Typically we see something like:

Example 10

```
(1) number_of_bananas = 12
(2)
(3) while number_of_bananas > 0:
(4)     monkey.eat(banana)
(5)     number_of_bananas = number_of_bananas - 1
```

Excercises

1. Implement a `Monkey` class which includes a property that allows us to implement the code in **Example 2** within a method called `Shines`.
2. In the `Monkey` class from **Exercise 1**, implement a **definite iterator** in a class method `Greeting` such that the monkey greets us by output each item in the list, `["Hi", "my", "name", "is", "Murry"]`.

Questions

1. Is the **conditional** `value is True` is equivalent to `value is not False`?
2. What form of programming construct is `.isSeen` in **Example 2**?
3. How many iterations will the loop in **Example 10** execute before exiting?