Chapter 7

Making Decisions

In This Chapter

- Using the `if` statement to make simple decisions
- Performing more advanced decision making with the `if...else` statement
- Creating multiple decision levels by nesting statements

The ability to make a decision, to take one path or another, is an essential element of performing useful work. Math gives the computer the capability to obtain useful information. Decisions make it possible to do something with the information after it's obtained. Without the capability to make decisions, a computer would be useless. So any language you use will include the capability to make decisions in some manner. This chapter explores the techniques that Python uses to make decisions.

Think through the process you use when making a decision. You obtain the actual value of something, compare it to a desired value, and then act accordingly. For example, when you see a signal light and see that it's red, you compare the red light to the desired green light, decide that the light isn't green, and then stop. Most people don't take time to consider the process they use because they use it so many times every day. Decision making comes naturally to humans, but computers must perform the following tasks every time:

1. Obtain the actual or current value of something.
2. Compare the actual or current value to a desired value.
3. Perform an action that corresponds to the desired outcome of the comparison.

Making Simple Decisions Using the if Statement

The `if` statement is the easiest method for making a decision in Python. It simply states that if something is true, Python should perform the steps that follow. The following sections tell you how you can use the `if` statement to make decisions of various sorts in Python. You may be surprised at what this simple statement can do for you.

Understanding the if statement

You use if statements regularly in everyday life. For example, you may say to yourself, "If it's Wednesday, I'll eat tuna salad for lunch." The Python `if` statement is a little less verbose, but it follows precisely the same pattern. Say you create a variable, `TestMe`, and place a value of 6 in it, like this:

```
TestMe = 6
```

You can then ask the computer to check for a value of 6 in `TestMe`, like this:

```
if TestMe == 6:
   print("TestMe does equal 6!")
```

Every Python `if` statement begins, oddly enough, with the word *if.* When Python sees `if`, it knows that you want it to make a decision. After the word *if* comes a condition. A *condition* simply states what sort of comparison you want Python to make. In this case, you want Python to determine whether `TestMe` contains the value `6`.

Notice that the condition uses the relational equality operator, `==`, and not the assignment operator, `=`. A common mistake that developers make is to use the assignment operator rather than the equality operator. You can see a list of relational operators in Chapter 6.

The condition always ends with a colon (:). If you don't provide a colon, Python doesn't know that the condition has ended and will continue to look for additional conditions on which to base its decision. After the colon come any tasks you want Python to perform. In this case, Python prints a statement saying that `TestMe` is equal to `6`.

Using the if statement in an application

It's possible to use the `if` statement in a number of ways in Python. However, you immediately need to know about three common ways to use it:

- Use a single condition to execute a single statement when the condition is true.
- Use a single condition to execute multiple statements when the condition is true.
- Combine multiple conditions into a single decision and execute one or more statements when the combined condition is true.

The following sections explore these three possibilities and provide you with examples of their use. You see additional examples of how to use the `if` statement throughout the book because it's such an important method of making decisions.

Working with relational operators

A *relational operator* determines how a value on the left side of an expression compares to the value on the right side of an expression. After it makes the determination, it outputs a value of `true` or `false` that reflects the truth value of the expression. For example, `6 == 6` is `true`, while `5 == 6` is `false`. Table 6-3 contains a listing of the relational operators. The following steps show how to create and use an `if` statement. This example also appears with the downloadable source code as `SimpleIf1.py`.

1. **Open a Python Shell window.**

   You see the familiar Python prompt.

2. **Type** TestMe = 6 **and press Enter.**

   This step assigns a value of `6` to `TestMe`. Notice that it uses the assignment operator and not the equality operator.

3. **Type** if TestMe == 6: **and press Enter.**

   This step creates an `if` statement that tests the value of TestMe using the equality operator. You should notice two features of the Python Shell at this point:

   - The word *if* is highlighted in a different color than the rest of the statement.
   - The next line is automatically indented.

4. **Type** print("TestMe does equal 6!") **and press Enter.**

   Notice that Python doesn't execute the `if` statement yet. It does indent the next line. The word *print* appears in a special color because it's a function name. In addition, the text appears in another color to show you that it's a string value. Color coding makes it much easier to see how Python works.

5. **Press Enter.**

   The Python Shell outdents this next line and executes the `if` statement, as shown in Figure 7-1. Notice that the output is in yet another color. Because `TestMe` contains a value of `6`, the `if` statement works as expected.

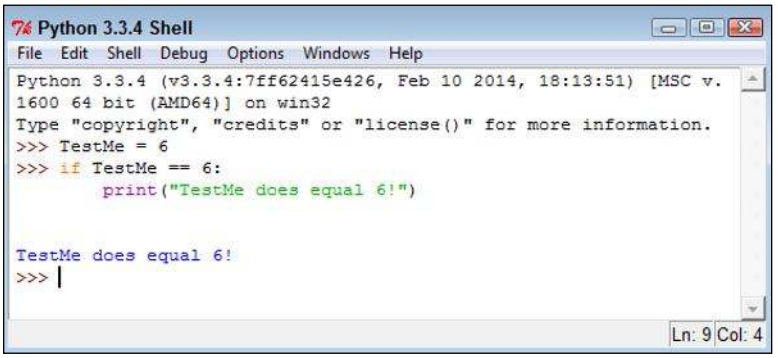


Figure 7-1: Simple `if` statements can help your application know what to do in certain conditions.

Performing multiple tasks

Sometimes you want to perform more than one task after making a decision. Python relies on indentation to determine when to stop executing tasks as part of an `if` statement. As long as the next line is indented, it's part of the `if` statement. When the next line is outdented, it becomes the first line of code outside the `if` block. A *code block* consists of a statement and the tasks associated with that statement. The same term is used no matter what kind of statement you're working with, but in this case, you're working with an `if` statement that is part of a code block. This example also appears with the downloadable source code as `SimpleIf2.py`.

1. **Open a Python Shell window.**

   You see the familiar Python prompt.

2. **Type the following code into the window — pressing Enter after each line:**

```
TestMe = 6
if TestMe == 6:
   print("TestMe does equal 6!")
   print("All done!")
```

Notice that the shell continues to indent lines as long as you continue to type code. Each line you type is part of the current `if` statement code block.

When working in the shell, you create a block by typing one line of code after another. If you press Enter twice in a row without entering any text, the code block is ended, and Python executes the entire code block at one time.

3. **Press Enter.**

   Python executes the entire code block. You see the output shown in Figure 7-2.

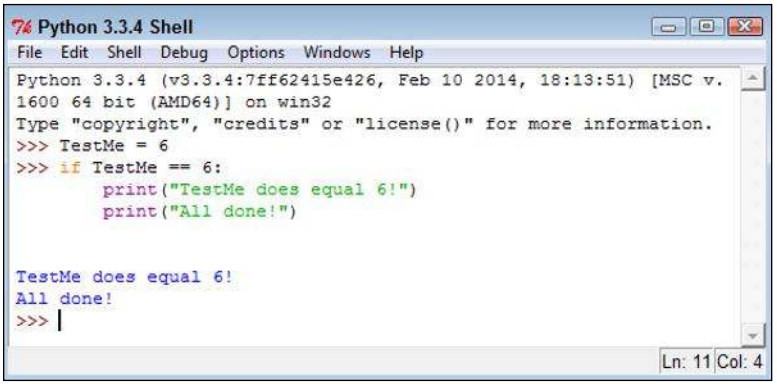


Figure 7-2: A code block can contain multiple lines of code — one for each task.

Making multiple comparisons using logical operators

So far, the examples have all shown a single comparison. Real life often requires that you make multiple comparisons to account for multiple requirements. For example, when baking cookies, if the timer has gone off and the edges are brown, it's time to take the cookies out of the oven.

In order to make multiple comparisons, you create multiple conditions using relational operators and combine them using logical operators (see Table 6-4). A *logical operator* describes how to combine conditions. For example, you might say `x == 6 and y == 7` as two conditions for performing one or more tasks. The `and` keyword is a logical operator that states that both conditions must be true.

One of the most common uses for making multiple comparisons to determine when a value is within a certain range. In fact, *range checking,* the act of determining whether data is between two values, is an important part of making your application secure and user friendly. The following steps help

you see how to perform this task. In this case, you create a file so that you can run the application multiple times. This example also appears with the downloadable source code as `SimpleIf3.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
Value = int(input("Type a number between 1 and 10: "))

if (Value > 0) and (Value <= 10):
   print("You typed: ", Value)
```

   The example begins by obtaining an input value. You have no idea what the user has typed other than that it's a value of some sort. The use of the `int()` function means that the user must type a whole number (one without a decimal portion). Otherwise, the application will raise an *exception* (an error indication; Chapter 9 describes exceptions). This first check ensures that the input is at least of the correct type.

   The `if` statement contains two conditions. The first states that `Value` must be greater than 0. You could also present this condition as `Value >= 1`. The second condition states that `Value` must be less than or equal to 10. Only when `Value` meets both of these conditions will the `if` statement succeed and print the value the user typed.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open with a prompt to type a number between 1 and 10.

4. **Type** 5 **and press Enter.**

   The application determines that the number is in the right range and outputs the message shown in Figure 7-3.

5. **Repeat Steps 3 and 4, but type** 22 **instead of 5.**

   The application doesn't output anything because the number is in the wrong range. Whenever you type a value that's outside the programmed range, the statements that are part of the `if` block aren't executed.

6. **Repeat Steps 3 and 4, but type** 5.5 **instead of 5.**

   Python displays the error message shown in Figure 7-4. Even though you may think of 5.5 and 5 as both being numbers, Python sees the first number as a floating-point value and the second as an integer.

7. **Repeat Steps 3 and 4, but type** Hello **instead of 5.**

   Python displays about the same error message as before. Python doesn't differentiate between types of wrong input. It only knows that the input type is incorrect and therefore unusable.

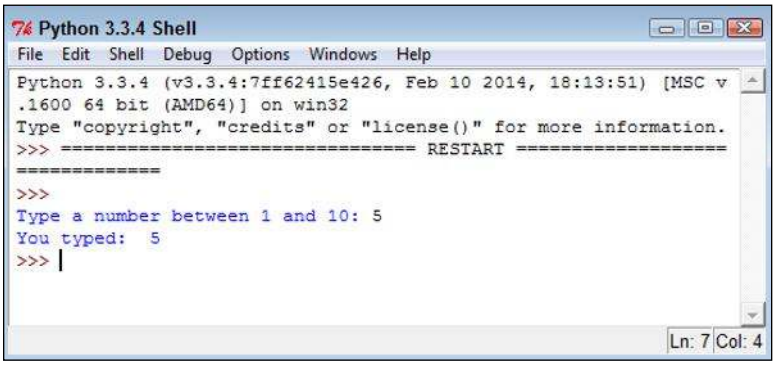


Figure 7-3: The application verifies the value is in the right range and outputs a message.

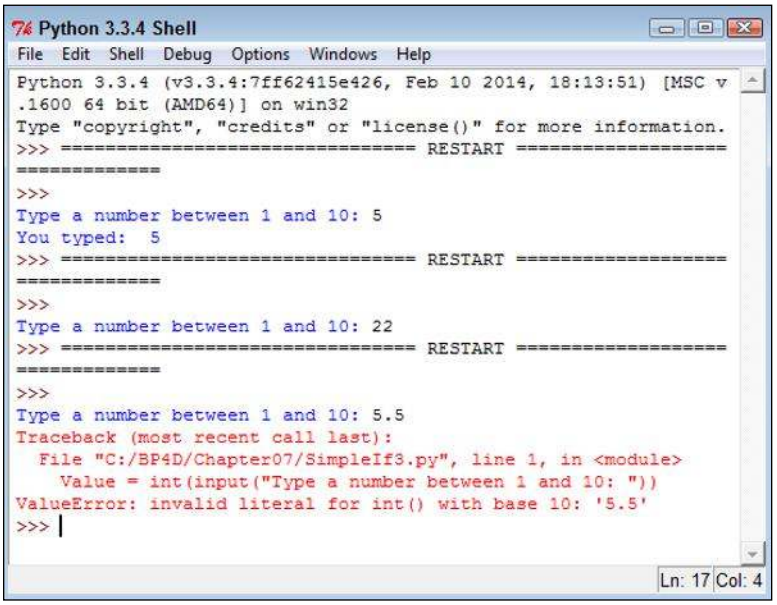
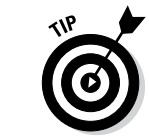


Figure 7-4: Typing the wrong type of information results in an error message.



The best applications use various kinds of range checking to ensure that the application behaves in a predictable manner. The more predictable an application becomes, the less the user thinks about the application and the more time the user spends on performing useful work. Productive users tend to be a lot happier than those who constantly fight with their applications.

Choosing Alternatives Using the if...else Statement

Many of the decisions you make in an application fall into a category of choosing one of two options based on conditions. For example, when looking at a signal light, you choose one of two options: press on the brake to stop or press the accelerator to continue. The option you choose depends on the conditions. A green light signals that you can continue on through the light; a red light tells you to stop. The following sections describe how Python makes it possible to choose between two alternatives.

Understanding the if...else statement

With Python, you choose one of two alternatives using the `else` clause of the `if` statement. A *clause* is an addition to a code block that modifies the way in which it works. Most code blocks support multiple clauses. In this case, the `else` clause enables you to perform an alternative task, which increases the usefulness of the `if` statement. Most developers refer to the form of the `if` statement that has the `else` clause included as the `if...else` statement, with the ellipsis implying that something happens between `if` and `else`.

Sometimes developers encounter problems with the `if...else` statement because they forget that the `else` clause always executes when the conditions for the `if` statement aren't met. It's important to think about the consequences of always executing a set of tasks when the conditions are false. Sometimes doing so can lead to unintended consequences.

Using the if...else statement in an application

The `SimpleIf3.py` example is a little less helpful than it could be when the user enters a value that's outside the intended range. Even entering data of the wrong type produces an error message, but entering the correct type of data outside the range tells the user nothing. In this example, you discover the means for correcting this problem by using an `else` clause. The following steps demonstrate just one reason to provide an alternative action when the condition for an `if` statement is false. This example also appears with the downloadable source code as `IfElse.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
Value = int(input("Type a number between 1 and 10: "))

if (Value > 0) and (Value <= 10):
   print("You typed: ", Value)
else:
   print("The value you typed is incorrect!")
```

   As before, the example obtains input from the user and then determines whether that input is in the correct range. However, in this case, the `else` clause provides an alternative output message when the user enters data outside the desired range.

   Notice that the `else` clause ends with a colon, just as the `if` statement does. Most clauses that you use with Python statements have a colon associated with them so that Python knows when the clause has ended. If you receive a coding error for your application, make sure that you check for the presence of the colon as needed.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open with a prompt to type a number between 1 and 10.

4. **Type** 5 **and press Enter.**

   The application determines that the number is in the right range and outputs the message shown previously in Figure 7-3.

5. **Repeat Steps 3 and 4, but type** 22 **instead of 5.**

   This time the application outputs the error message shown in Figure 7-5. The user now knows that the input is outside the desired range and knows to try entering it again.

Using the if...elif statement in an application

You go to a restaurant and look at the menu. The restaurant offers eggs, pancakes, waffles, and oatmeal for breakfast. After you choose one of the items, the server brings it to you. Creating a menu selection requires something like an `if...else` statement, but with a little extra oomph. In this case, you use the `elif` clause to create another set of conditions. The `elif` clause is a combination of the `else` clause and a separate `if` statement. The following steps describe how to use the `if...elif` statement to create a menu. This example also appears with the downloadable source code as `IfElif.py`.

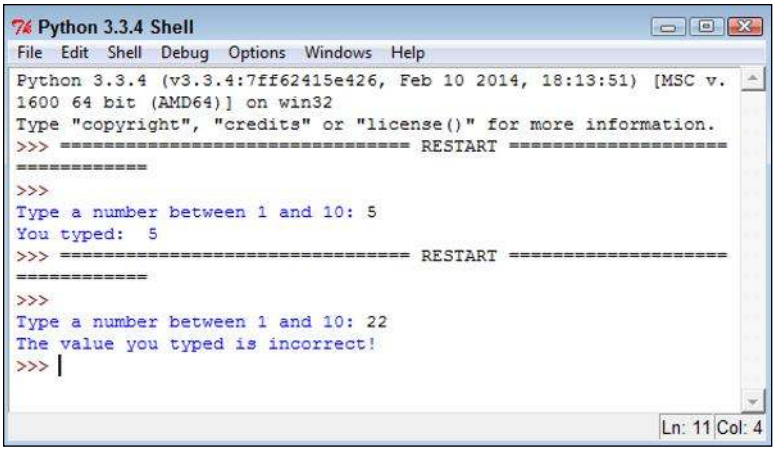


Figure 7-5: It's always a good idea to provide feedback for incorrect input.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
print("1. Red")
print("2. Orange")
print("3. Yellow")
print("4. Green")
print("5. Blue")
print("6. Purple")

Choice = int(input("Select your favorite color: "))

if (Choice == 1):
   print("You chose Red!")
elif (Choice == 2):
   print("You chose Orange!")
elif (Choice == 3):
   print("You chose Yellow!")
elif (Choice == 4):
   print("You chose Green!")
elif (Choice == 5):
   print("You chose Blue!")
elif (Choice == 6):
   print("You chose Purple!")
else:
   print("You made an invalid choice!")
```

The example begins by displaying a menu. The user sees a list of choices for the application. It then asks the user to make a selection, which it places inside `Choice`. The use of the `int()` function ensures that the user can't type anything other than a number.

After the user makes a choice, the application looks for it in the list of potential values. In each case, `Choice` is compared against a particular value to create a condition for that value. When the user types 1, the application outputs the message `"You chose Red!"`. If none of the options is correct, the `else` clause is executed by default to tell the user that the input choice is invalid.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open with the menu displayed. The application asks you to select your favorite color.

4. **Type** 1 **and press Enter.**

   The application displays the appropriate output message, as shown in Figure 7-6.

5. **Repeat Steps 3 and 4, but type** 5 **instead of 1.**

   The application displays a different output message — the one associated with the requested color.

6. **Repeat Steps 3 and 4, but type** 8 **instead of 1.**

   The application tells you that you made an invalid choice.

7. **Repeat Steps 3 and 4, but type** Red **instead of 1.**

   The application displays the expected error message, as shown in Figure 7-7. Any application you create should be able to detect errors and incorrect inputs. Chapter 9 shows you how to handle errors so that they're user friendly.

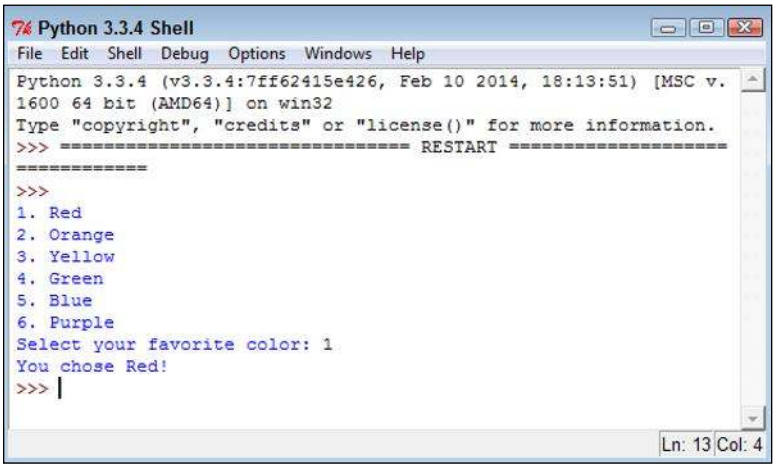


Figure 7-6: Menus let you choose one option from a list of options.

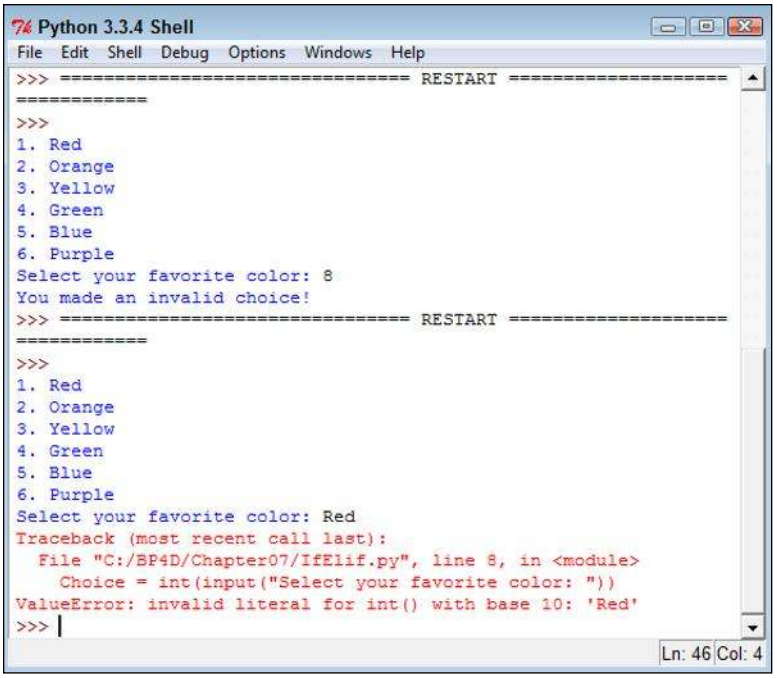


Figure 7-7: Every application you create should include some means of detecting errant input.

No switch statement?

If you've worked with other languages, you might notice that Python lacks a switch statement (if you haven't, there is no need to worry about it with Python). Developers commonly use the switch statement in other languages to create menu-based applications. The `if...elif` statement is generally used for the same purpose in Python.

However, the `if...elif` statement doesn't provide quite the same functionality as a switch statement because it doesn't enforce the use of a single variable for comparison purposes. As a result, some developers rely on Python's dictionary functionality to stand in for the switch statement. Chapter 13 describes how to work with dictionaries.

Using Nested Decision Statements

The decision-making process often happens in levels. For example, when you go to the restaurant and choose eggs for breakfast, you have made a first-level decision. Now the server asks you what type of toast you want with your eggs. The server wouldn't ask this question if you had ordered pancakes, so the selection of toast becomes a second-level decision. When the breakfast arrives, you decide whether you want to use jelly on your toast. This is a third-level decision. If you had selected a kind of toast that doesn't work well with jelly, you might not have had to make this decision at all. This process of making decisions in levels, with each level reliant on the decision made at the previous level, is called *nesting*. Developers often use nesting techniques to create applications that can make complex decisions based on various inputs. The following sections describe several kinds of nesting you can use within Python to make complex decisions.

Using multiple if or if...else statements

The most commonly used multiple selection technique is a combination of `if` and `if...else` statements. This form of selection is often called a *selection tree* because of its resemblance to the branches of a tree. In this case, you follow a particular path to obtain a desired result. The example in this section also appears with the downloadable source code as `MultipleIfElse.py`.

1. **Open a Python File window.**

   You see an editor where you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
One = int(input("Type a number between 1 and 10: "))
Two = int(input("Type a number between 1 and 10: "))

if (One >= 1) and (One <= 10):
   if (Two >= 1) and (Two <= 10):
      print("Your secret number is: ", One * Two)
   else:
      print("Incorrect second value!")
else:
   print("Incorrect first value!")
```

   This is simply an extension of the `IfElse.py` example you see in the "Using the if...else statement in an application" section of the chapter. However, notice that the indentation is different. The second `if...else` statement is indented within the first `if...else` statement. The indentation tells Python that this is a second-level statement.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open with a prompt to type a number between 1 and 10.

4. **Type** 5 **and press Enter.**

   The shell asks for another number between 1 and 10.

5. **Type** 2 **and press Enter.**

   You see the combination of the two numbers as output, as shown in Figure 7-8.

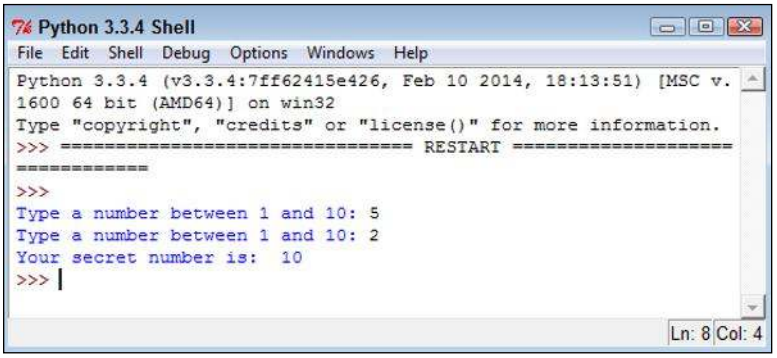


Figure 7-8: Adding multiple levels lets you perform tasks with greater complexity.

This example has the same input features as the `IfElse.py` example. For example, if you attempt to provide a value that's outside the requested range, you see an error message. The error message is tailored for either the first or second input value so that the user knows which value was incorrect.

Providing specific error messages is always useful because users tend to become confused and frustrated otherwise. In addition, a specific error message helps you find errors in your application much faster.

Combining other types of decisions

It's possible to use any combination of `if`, `if...else`, and `if...elif` statements to produce a desired outcome. You can nest the code blocks as many levels deep as needed to perform the required checks. For example, Listing 7-1 shows what you might accomplish for a breakfast menu. This example also appears with the downloadable source code as `MultipleIfElif.py`.

Listing 7-1: Creating a Breakfast Menu

```
print("1. Eggs")
print("2. Pancakes")
print("3. Waffles")
print("4. Oatmeal")
MainChoice = int(input("Choose a breakfast item: "))

if (MainChoice == 2):
   Meal = "Pancakes"
elif (MainChoice == 3):
   Meal = "Waffles"

if (MainChoice == 1):
   print("1. Wheat Toast")
   print("2. Sour Dough")
   print("3. Rye Toast")
   print("4. Pancakes")
   Bread = int(input("Choose a type of bread: "))

   if (Bread == 1):
      print("You chose eggs with wheat toast.")
   elif (Bread == 2):
      print("You chose eggs with sour dough.")
   elif (Bread == 3):
      print("You chose eggs with rye toast.")
   elif (Bread == 4):
      print("You chose eggs with pancakes.")
   else:
      print("We have eggs, but not that kind of bread.")

elif (MainChoice == 2) or (MainChoice == 3):
   print("1. Syrup")
   print("2. Strawberries")
   print("3. Powdered Sugar")
   Topping = int(input("Choose a topping: "))

   if (Topping == 1):
      print ("You chose " + Meal + " with syrup.")
   elif (Topping == 2):
      print ("You chose " + Meal + " with strawberries.")
   elif (Topping == 3):
      print ("You chose " + Meal + " with powdered
           sugar.")
   else:
      print ("We have " + Meal + ", but not that
           topping.")

elif (MainChoice == 4):
   print("You chose oatmeal.")

else:
   print("We don't serve that breakfast item!")
```

This example has some interesting features. For one thing, you might assume that an `if...elif` statement always requires an `else` clause. This example shows a situation that doesn't require such a clause. You use an `if...elif` statement to ensure that `Meal` contains the correct value, but you have no other options to consider.

The selection technique is the same as you saw for the previous examples. A user enters a number in the correct range to obtain a desired result. Three of the selections require a secondary choice, so you see the menu for that choice. For example, when ordering eggs, it isn't necessary to choose a topping, but you do want a topping for pancakes or waffles.

Notice that this example also combines variables and text in a specific way. Because a topping can apply equally to waffles or pancakes, you need some method for defining precisely which meal is being served as part of the output. The `Meal` variable that the application defines earlier is used as part of the output after the topping choice is made.

The best way to understand this example is to play with it. Try various menu combinations to see how the application works. For example, Figure 7-9 shows what happens when you choose a waffle breakfast with a strawberry topping.

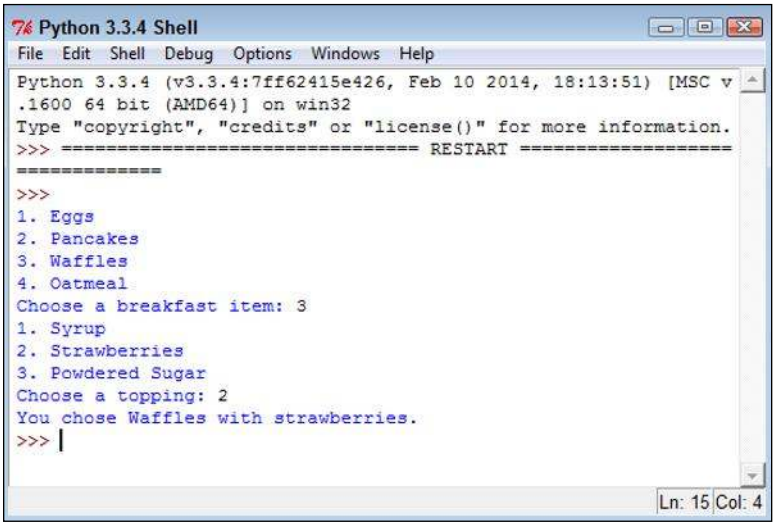


Figure 7-9: Many applications rely on multilevel menus.