Chapter 11

Working with Strings

In This Chapter

- Considering the string difference
- Using special characters in strings
- Working with single characters
- Performing string-specific tasks
- Finding what you need in a string
- Modifying the appearance of string output

Your computer doesn't understand strings. It's a basic fact. Computers understand numbers, not letters. When you see a string on the computer screen, the computer actually sees a series of numbers. However, humans understand strings quite well, so applications need to be able to work with them. Fortunately, Python makes working with strings relatively easy. It translates the string you understand into the numbers the computer understands, and vice versa.

In order to make strings useful, you need to be able to manipulate them. Of course, that means taking strings apart and using just the pieces you need or searching the string for specific information. This chapter describes how you can build strings using Python, dissect them as needed, and use just the parts you want after you find what's required. String manipulation is an important part of applications because humans depend on computers performing that sort of work for them (even though the computer has no idea of what a string is).

After you have the string you want, you need to present it to the user in an eye-pleasing manner. The computer doesn't really care how it presents the string, so often you get the information, but it lacks pizzazz. In fact, it may be downright difficult to read. Knowing how to format strings so that they look nice onscreen is important because users need to see information in a form they understand. By the time you complete this chapter, you know how to create, manipulate, and format strings so that the user sees precisely the right information.

Understanding That Strings Are Different

Most aspiring developers (and even a few who have written code for a long time) really have a hard time understanding that computers truly do only understand 0s and 1s. Even larger numbers are made up of 0s and 1s. Comparisons take place with 0s and 1s. Data is moved using 0s and 1s. In short, strings don't exist for the computer (and numbers just barely exist). Although grouping 0s and 1s to make numbers is relatively easy, strings are a lot harder because now you're talking about information that the computer must manipulate as numbers but present as characters.

There are no strings in computer science. Strings are made up of characters, and individual characters are actually numeric values. When you work with strings in Python, what you're really doing is creating an assembly of characters that the computer sees as numeric values. That's why the following sections are so important. They help you understand why strings are so special. Understanding this material will save you a lot of headaches later.

Defining a character using numbers

To create a character, you must first define a relationship between that character and a number. More important, everyone must agree that when a certain number appears in an application and is viewed as a character by that application, the number is translated into a specific character. One of the most common ways to perform this task is to use the American Standard Code for Information Interchange (ASCII). Python uses ASCII to translate the number 65 to the letter *A*. The chart at `http://www.asciitable.com/` shows the various numeric values and their character equivalents.

Every character you use must have a different numeric value assigned to it. The letter *A* uses a value of 65. To create a lowercase *a,* you must assign a different number, which is 97. The computer views *A* and *a* as completely different characters, even though people view them as uppercase and lowercase versions of the same character.

The numeric values used in this chapter are in decimal. However, the computer still views them as 0s and 1s. For example, the letter *A* is really the value 01000001 and the letter *a* is really the value 01100001. When you see an *A* onscreen, the computer sees a binary value instead.

Having just one character set to deal with would be nice. However, not everyone could agree on a single set of numeric values to equate with specific characters. Part of the problem is that ASCII doesn't support characters used by other languages; also, it lacks the capability to translate special characters into an onscreen presentation. In fact, character sets abound. You can see a number of them at `http://www.i18nguy.com/unicode/codepages.html`. Click one of the character set entries to see how it assigns specific numeric values to each character. Most characters sets do use ASCII as a starting point.

Using characters to create strings

Python doesn't make you jump through hoops to create strings. However, the term *string* should actually give you a good idea of what happens. Think about beads or anything else you might string. You place one bead at a time onto the string. Eventually you end up with some type of ornamentation — perhaps a necklace or tree garland. The point is that these items are made up of individual beads.

The same concept used for necklaces made of beads holds true for strings in computers. When you see a sentence, you understand that the sentence is made up of individual characters that are strung together by the programming language you use. The language creates a structure that holds the individual characters together. So, the language, not the computer, knows that so many numbers in a row (each number being represented as a character) defines a string such as a sentence.

You may wonder why it's important to even know how Python works with characters. The reason is that many of the functions and special features that Python provides work with individual characters, and it's important to know that Python sees the individual characters. Even though you see a sentence, Python sees a specific number of characters.

Unlike most programming languages, strings can use either single quotes or double quotes. For example, "Hello There!" with double quotes is a string, as is 'Hello There!' with single quotes. Python also supports triple double and single quotes that let you create strings spanning multiple lines. The following steps help you create an example that demonstrates some of the string features that Python provides. This example also appears with the downloadable source code as `BasicString.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:**

```
print('Hello There (Single Quote)!')
print("Hello There (Double Quote)!")
print("""This is a multiple line
string using triple double quotes.
You can also use triple single quotes.""")
```

Each of the three `print()` function calls demonstrates a different principle in working with strings. It's equally acceptable to enclose the string in either single or double quotes. When you use a triple quote (either single or double), the text can appear on multiple lines.

3. **Choose Run➪Run Module.**

You see a Python Shell window open. The application outputs the text. Notice that the multiline text appears on three lines (see Figure 11-1), just as it does in the source code file, so this is a kind of formatting. You can use multiline formatting to ensure that the text breaks where you want it to onscreen.

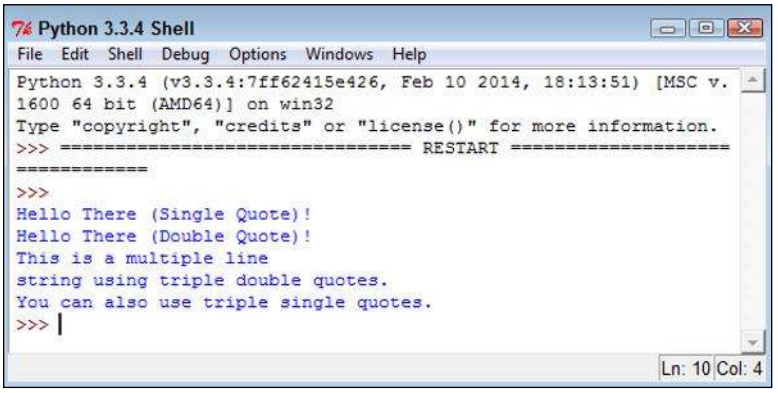

Figure 11-1: Strings consist of individual characters that are linked together.

Creating Stings with Special Characters

Some strings include special characters. These characters are different from the alphanumeric and punctuation characters that you're used to using. In fact, they fall into these categories:

- **Control:** An application requires some means of determining that a particular character isn't meant to be displayed but rather to control the display. All the control movements are based on the *insertion pointer,* the

line you see when you type text on the screen. For example, you don't see a tab character. The tab character provides a space between two elements, and the size of that space is controlled by a tab stop. Likewise, when you want to go to the next line, you use a carriage return (which returns the insertion pointer to the beginning of the line) and linefeed (which places the insertion pointer on the next line) combination.

- **Accented:** Characters that have accents, such as the acute (‘), grave (`), circumflex (^), umlaut or diaeresis (¨), tilde (~), or ring (˚), represent special spoken sounds, in most cases. You must use special characters to create alphabetical characters with these accents included.
- **Drawing:** It's possible to create rudimentary art with some characters. You can see examples of the box-drawing characters at `http://jrgraphix.net/r/Unicode/2500-257F`. Some people actually create art using ASCII characters as well (`http://www.asciiworld.com/`).
- **Typographical:** A number of typographical characters, such as the pilcrow (¶),are used when displaying certain kinds of text onscreen, especially when the application acts as an editor.
- **Other:** Depending on the character set you use, the selection of characters is nearly endless. You can find a character for just about any need. The point is that you need some means of telling Python how to present these special characters.

A common need when working with strings, even strings from simple console applications, is control characters. With this in mind, Python provides escape sequences that you use to define control characters directly (and a special escape sequence for other characters).

An *escape sequence* literally escapes the common meaning of a letter, such as *a,* and gives it a new meaning (such as the ASCII bell or beep). The combination of the backslash (\) and a letter (such as *a*) is commonly viewed as a single letter by developers — an *escape character* or *escape code.* Table 11-1 provides an overview of these escape sequences.

Table 11-1 Python Escape Sequences

Escape Sequence	*Meaning*
\newline	Ignored
\\	Backslash (\)
\'	Single quote (‘)
\"	Double quote (")

(continued)

Table 11-1 *(continued)*

Escape Sequence	*Meaning*
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\u*hhhh*	Unicode character (a specific kind of character set with broad appeal across the world) with a hexadecimal value that replaces *hhhh*
\v	ASCII Vertical Tab (VT)
ooo	ASCII character with octal numeric value that replaces *ooo*
\x*hh*	ASCII character with hexadecimal value that replaces *hh*

The best way to see how the escape sequences work is to try them. The following steps help you create an example that tests various escape sequences so that you can see them in action. This example also appears with the downloadable source code as `SpecialCharacters.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:**

```
print("Part of this text\r\nis on the next line.")
print("This is an A with a grave accent: \xC0.")
print("This is a drawing character: \u2562.")
print("This is a pilcrow: \266.")
print("This is a division sign: \xF7.")
```

   The example code uses various techniques to achieve the same end — to create a special character. Of course, you use control characters directly, as shown in the first line. Many special letters are accessible using a hexadecimal number that has two digits (as in the second and fifth lines). However, some require that you rely on Unicode numbers (which always require four digits), as shown in the third line. Octal values use three digits and have no special character associated with them, as shown in the fourth line.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application outputs the expected text and special characters, as shown in Figure 11-2.

The Python shell uses a standard character set across platforms, so the Python Shell should use the same special characters no matter which platform you test. However, when creating your application, make sure to test it on various platforms to see how the application will react. A character set on one platform may use different numbers for special characters than another platform does. In addition, user selection of character sets could have an impact on how special characters displayed by your application appear. Always make sure that you test special character usage completely.

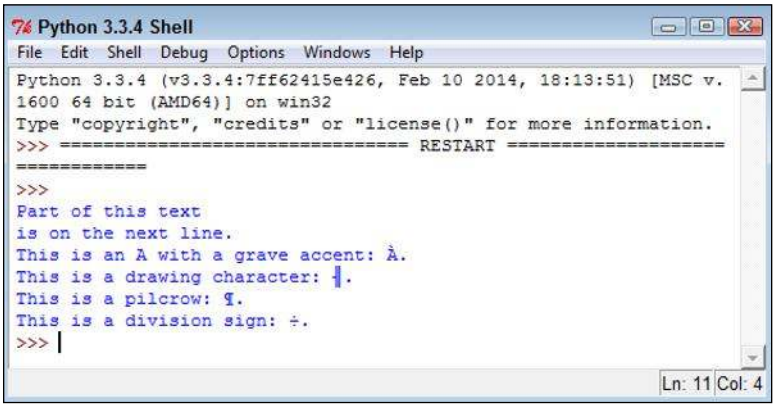

Figure 11-2: Use special characters as needed to present special information or to format the output.

Selecting Individual Characters

Earlier in the chapter, you discover that strings are made up of individual characters. They are, in fact, just like beads on a necklace — with each bead being an individual element of the whole string.

Python makes it possible to access individual characters in a string. This is an important feature because you can use it to create new strings that contain only part of the original. In addition, you can combine strings to create new results. The secret to this feature is the square bracket. You place a square bracket with a number in it after the name of the variable. Here's an example:

```
MyString = "Hello World"
print(MyString[0])
```

In this case, the output of the code is the letter *H*. Python strings are zero-based, which means they start with the number *0* and proceed from there. For example, if you were to type `print(MyString[1])`, the output would be the letter *e*.

You can also obtain a range of characters from a string. Simply provide the beginning and ending letter count separated by a colon in the square brackets. For example, `print(MyString[6:11])` would output the word World. The output would begin with letter 7 and end with letter 12 (remember that the index is zero based).

The following steps demonstrate some basic tasks that you can perform using Python's character-selection technique. This example also appears with the downloadable source code as `Characters.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.**

```
String1 = "Hello World"
String2 = "Python is Fun!"

print(String1[0])
print(String1[0:5])
print(String1[:5])
print(String1[6:])

String3 = String1[:6] + String2[:6]
print(String3)

print(String2[:7]*5)
```

   The example begins by creating two strings. It then demonstrates various methods for using the index on the first string. Notice that you can leave out the beginning or ending number in a range if you want to work with the remainder of that string.

   The next step is to combine two substrings. In this case, the code combines the beginning of `String1` with the beginning of `String2` to create `String3`.

   The use of the `+` sign to combine two strings is called *concatenation*. It's one of the handier operators to remember when you're working with strings in an application.

   The final step is to use a Python feature called *repetition*. You use repetition to make a number of copies of a string or substring.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The applications outputs a series of substrings and string combinations, as shown in Figure 11-3.

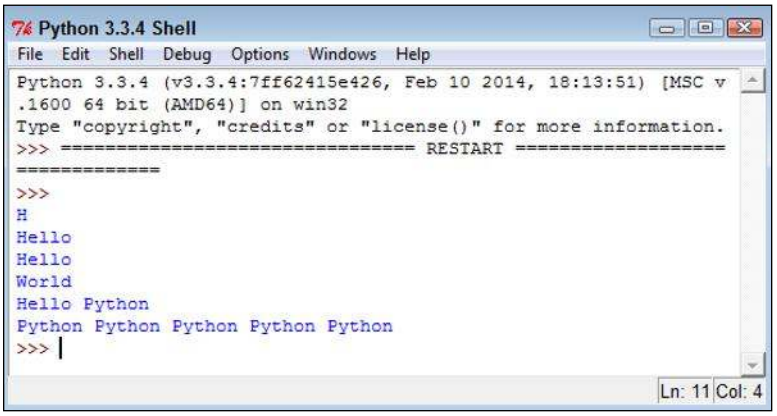

Figure 11-3: You can select individual pieces of a string.

Slicing and Dicing Strings

Working with ranges of characters provides some degree of flexibility, but it doesn't provide you with the capability to actually manipulate the string content or discover anything about it. For example, you might want to change the characters to uppercase or determine whether the string contains all letters. Fortunately, Python has functions that help you perform tasks of this sort. Here are the most commonly used functions:

- `capitalize()`: Capitalizes the first letter of a string.
- `center(`*`width, fillchar`*`=" ")`: Centers a string so that it fits within the number of spaces specified by *`width`*. If you supply a character for *`fillchar`*, the function uses that character. Otherwise, `center()` uses spaces to create a string of the desired width.
- `expandtabs(`*`tabsize`*`=8)`: Expands tabs in a string by replacing the tab with the number of spaces specified by *`tabsize`*. The function defaults to 8 spaces per tab when *tabsize* isn't provided.
- `isalnum()`: Returns `True` when the string has at least one character and all characters are alphanumeric (letters or numbers).
- `isalpha()`: Returns `True` when the string has at least one character and all characters are alphabetic (letters only).
- `isdecimal()`: Returns `True` when a Unicode string contains only decimal characters.

- `isdigit()`: Returns `True` when a string contains only digits (numbers and not letters).
- `islower()`: Returns `True` when a string has at least one alphabetic character and all alphabetic characters are in lowercase.
- `isnumeric()`: Returns `True` when a Unicode string contains only numeric characters.
- `isspace()`: Returns `True` when a string contains only whitespace characters (which includes spaces, tabs, carriage returns, linefeeds, form feeds, and vertical tabs, but not the backspace).
- `istitle()`: Returns `True` when a string is cased for use as a title, such as *`Hello World`*. However, the function requires that even little words have the title case. For example, *`Follow a Star`* returns `False`, even though it's properly cased, but *`Follow A Star`* returns `True`.
- `isupper()`: Returns `True` when a string has at least one alphabetic character and all alphabetic characters are in uppercase.
- `join(`*`seq`*`)`: Creates a string in which the base string is separated in turn by each character in *`seq`* in a repetitive fashion. For example, if you start with `MyString = "Hello"` and type `print(MyString.join("!*!"))`, the output is `!Hello*Hello!`.
- `len(`*`string`*`)`: Obtains the length of *`string`*.
- `ljust(`*`width, fillchar`*`=" ")`: Left justifies a string so that it fits within the number of spaces specified by *`width`*. If you supply a character for *`fillchar`*, the function uses that character. Otherwise, `ljust()` uses spaces to create a string of the desired width.
- `lower()`: Converts all uppercase letters in a string to lowercase letters.
- `lstrip()`: Removes all leading whitespace characters in a string.
- `max(`*`str`*`)`: Returns the character that has the maximum numeric value in *`str`*. For example, *a* would have a larger numeric value than *A*.
- `min(`*`str`*`)`: Returns the character that has the minimum numeric value in *`str`*. For example, *A* would have a smaller numeric value than *a*.
- `rjust(`*`width, fillchar`*`=" ")`: Right justifies a string so that it fits within the number of spaces specified by *`width`*. If you supply a character for *`fillchar`*, the function uses that character. Otherwise, `rjust()` uses spaces to create a string of the desired width.
- `rstrip()`: Removes all trailing whitespace characters in a string.
- `split(`*`str`*`=" ", `*`num`*`=string.count(`*`str`*`))`: Splits a string into substrings using the delimiter specified by *`str`* (when supplied). The default is to use a space as a delimiter. Consequently, if your string contains *`A Fine Day`*, the output would be three substrings consisting of *`A`*, *`Fine`*, and *`Day`*. You use *`num`* to define the number of substrings to return. The default is to return every substring that the function can produce.

- `splitlines(`*num*`=string.count('\n'))`: Splits a string that contains newline (`\n`) characters into individual strings. Each break occurs at the newline character. The output has the newline characters removed. You can use *num* to specify the number of strings to return.
- `strip()`: Removes all leading and trailing whitespace characters in a string.
- `swapcase()`: Inverts the case for each alphabetic character in a string.
- `title()`: Returns a string in which the initial letter in each word is in uppercase and all remaining letters in the word are in lowercase.
- `upper()`: Converts all lowercase letters in a string to uppercase letters.
- `zfill (`*width*`)`: Returns a string that is left-padded with zeros so that the resulting string is the size of *width*. This function is designed for use with strings containing numeric values. It retains the original sign information (if any) supplied with the number.

Playing with these functions a bit can help you understand them better. The following steps create an example that demonstrates some of the tasks you can perform using these functions. This example also appears with the downloadable source code as `Functions.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:**

```
MyString = "  Hello World  "

print(MyString.upper())

print(MyString.strip())
print(MyString.center(21, "*"))
print(MyString.strip().center(21, "*"))

print(MyString.isdigit())
print(MyString.istitle())

print(max(MyString))

print(MyString.split())
print(MyString.split()[0])
```

   The code begins by creating `MyString`, which includes spaces before and after the text so that you can see how space-related functions work. The initial task is to convert all the characters to uppercase.

Removing extra space is a common task in application development. The `strip()` function performs this task well. The `center()` function lets you add padding to both the left and right side of a string so that it consumes a desired amount of space. When you combine the `strip()` and `center()` functions, the output is different from when you use the `center()` function alone.

You can combine functions to produce a desired result. Python executes each of the functions one at a time from left to right. The order in which the functions appear will affect the output, and developers commonly make the mistake of putting the functions in the wrong order. If your output is different from what you expected, try changing the function order.

Some functions work on the string as an input rather than on the string instance. The `max()` function falls into this category. If you had typed `MyString.max()`, Python would have displayed an error. The bulleted list that appears earlier in this section shows which functions require this sort of string input.

When working with functions that produce a list as an output, you can access an individual member by providing an index to it. The example shows how to use `split()` to split the string into substrings. It then shows how to access just the first substring in the list. You find out more about working with lists in Chapter 12.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application outputs a number of modified strings, as shown in Figure 11-4.

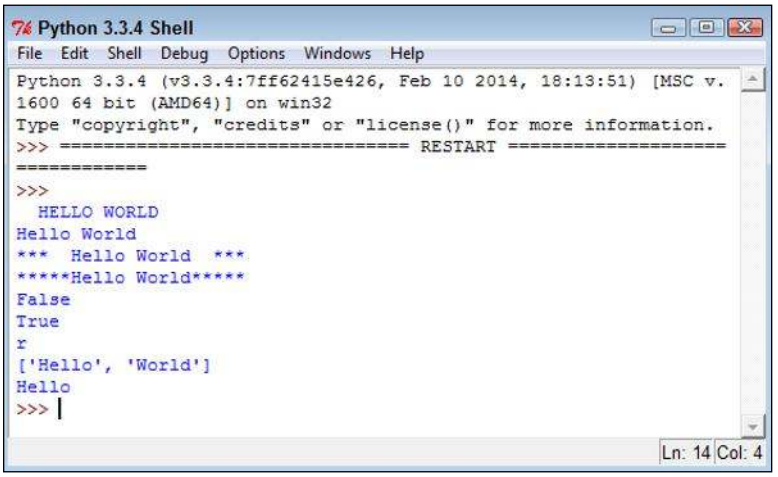

Figure 11-4: Using functions makes string manipulation a lot more flexible.

Locating a Value in a String

There are times when you need to locate specific information in a string. For example, you may want to know whether a string contains the word Hello in it. One of the essential purposes behind creating and maintaining data is to be able to search it later to locate specific bits of information. Strings are no different — they're most useful when you can find what you need quickly and without any problems. Python provides a number of functions for searching strings. Here are the most commonly used functions:

- `count(str, beg= 0, end=len(string))`: Counts how many times *str* occurs in a string. You can limit the search by specifying a beginning index using *beg* or an ending index using *end*.
- `endswith(suffix, beg=0, end=len(string))`: Returns `True` when a string ends with the characters specified by *suffix*. You can limit the check by specifying a beginning index using *beg* or an ending index using *end*.
- `find(str, beg=0, end=len(string))`: Determines whether *str* occurs in a string and outputs the index of the location. You can limit the search by specifying a beginning index using *beg* or a ending index using *end*.
- `index(str, beg=0, end=len(string))`: Provides the same functionality as `find()`, but raises an exception when *str* isn't found.
- `replace(old, new [, max])`: Replaces all occurrences of the character sequence specified by *old* in a string with the character sequence specified by *new*. You can limit the number of replacements by specifying a value for *max*.
- `rfind(str, beg=0, end=len(string))`: Provides the same functionality as `find()`, but searches backward from the end of the string instead of the beginning.
- `rindex(str, beg=0, end=len(string))`: Provides the same functionality as `index()`, but searches backward from the end of the string instead of the beginning.
- `startswith(prefix, beg=0, end=len(string))`: Returns `True` when a string begins with the characters specified by *prefix*. You can limit the check by specifying a beginning index using *beg* or an ending index using *end*.

Finding the data that you need is an essential programming task — one that is required no matter what kind of application you create. The following steps help you create an example that demonstrates the use of search functionality within strings. This example also appears with the downloadable source code as `SearchString.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:**

```
SearchMe = "The apple is red and the berry is blue!"

print(SearchMe.find("is"))
print(SearchMe.rfind("is"))

print(SearchMe.count("is"))

print(SearchMe.startswith("The"))
print(SearchMe.endswith("The"))

print(SearchMe.replace("apple", "car")
      .replace("berry", "truck"))
```

   The example begins by creating `SearchMe`, a string with two instances of the word *is*. The two instances are important because they demonstrate how searches differ depending on where you start. When using `find()`, the example starts from the beginning of the string. By contrast, `rfind()` starts from the end of the string.

   Of course, you won't always know how many times a certain set of characters appears in a string. The `count()` function lets you determine this value.

   Depending on the kind of data you work with, sometimes the data is heavily formatted and you can use a particular pattern to your advantage. For example, you can determine whether a particular string (or substring) ends or begins with a specific sequence of characters. You could just as easily use this technique to look for a part number.

   The final bit of code replaces *apple* with *car* and *berry* with *truck*. Notice the technique used to place the code on two lines. In some cases, your code will need to appear on multiple lines to make it more readable.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application displays the output shown in Figure 11-5. Notice especially that the searches returned different indexes based on where they started in the string. Using the correct function when performing searches is essential to ensure that you get the results you expected.

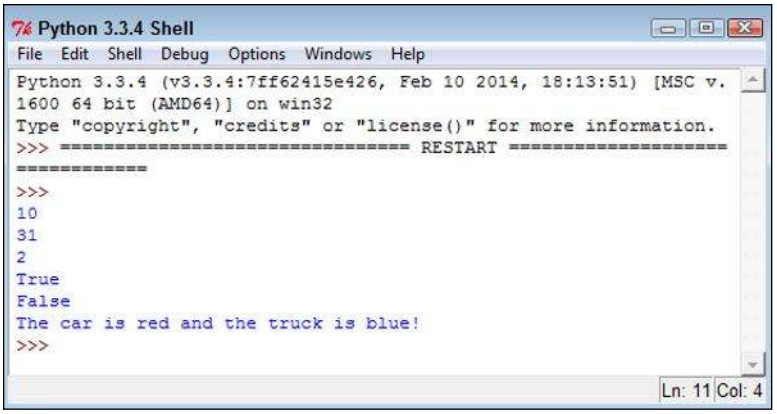

Figure 11-5: Typing the wrong input type generates an error instead of an exception.

Formatting Strings

You can format strings in a number of ways using Python. The main emphasis of formatting is to present the string in a form that is both pleasing to the user and easy to understand. Formatting doesn't mean adding special fonts or effects in this case, but refers merely to the presentation of the data. For example, the user might want a fixed-point number rather than a decimal number as output.

You have quite a few ways to format strings and you see a number of them as the book progresses. However, the focus of most formatting is the `format()` function. You create a formatting specification as part of the string and then use the `format()` function to add data to that string. A format specification may be as simple as two curly brackets `{}` that specify a placeholder for data. You can number the placeholder to create special effects. For example, `{0}` would contain the first data element in a string. When the data elements are numbered, you can even repeat them so that the same data appears more than once in the string.

The formatting specification follows a colon. When you want to create just a formatting specification, the curly brackets contain just the colon and whatever formatting you want to use. For example, `{:f}` would create a fixed-point number as output. If you want to number the entries, the number that precedes the colon: `{0:f}` creates a fixed-point number output for data element one. The formatting specification follows this form, with the italicized elements serving as placeholders here:

```
[[fill]align][sign][#][0][width][,][.precision][type]
```

The specification at `https://docs.python.org/3/library/string.html` provides you with the in-depth details, but here's an overview of what the various entries mean:

- **fill:** Defines the fill character used when displaying data that is too small to fit within the assigned space.
- **align:** Specifies the alignment of data within the display space. You can use these alignments:
  - **<:** Left aligned
  - **>:** Right aligned
  - **^:** Centered
  - **=:** Justified
- **sign:** Determines the use of signs for the output:
  - **+:** Positive numbers have a plus sign and negative numbers have a minus sign.
  - **-:** Negative numbers have a minus sign.
  - **<space>:** Positive numbers are preceded by a space and negative numbers have a minus sign.
- **#:** Specifies that the output should use the alternative display format for numbers. For example, hexadecimal numbers will have a 0x prefix added to them.
- **0:** Specifies that the output should be sign aware and padded with zeros as needed to provide consistent output.
- **width:** Determines the full width of the data field (even if the data won't fit in the space provided).
- **,:** Specifies that numeric data should have commas as a thousands separator.
- **.precision:** Determines the number of characters after the decimal point.
- **type:** Specifies the output type, even if the input type doesn't match. The types are split into three groups:
  - *String:* Use an `s` or nothing at all to specify a string.
  - *Integer:* The integer types are as follows: `b` (binary); `c` (character); `d` (decimal); `o` (octal); `x` (hexadecimal with lowercase letters); `X` (hexadecimal with uppercase letters); and `n` (locale-sensitive decimal that uses the appropriate characters for the thousands separator).

- *Floating point:* The floating-point types are as follows: `e` (exponent using a lowercase e as a separator); `E` (exponent using an uppercase E as a separator); `f` (lowercase fixed point); `F` (uppercase fixed point); `g` (lowercase general format); `G` (uppercase general format); `n` (local-sensitive general format that uses the appropriate characters for the decimal and thousands separators); and `%` (percentage).

The formatting specification elements must appear in the correct order or Python won't know what to do with them. If you specify the alignment before the fill character, Python displays an error message rather than performing the required formatting. The following steps help you see how the formatting specification works and demonstrate the order you need to follow in using the various formatting specification criteria. This example also appears with the downloadable source code as `Formatted.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:**

```
Formatted = "{:d}"
print(Formatted.format(7000))

Formatted = "{:,d}"
print(Formatted.format(7000))

Formatted = "{:^15,d}"
print(Formatted.format(7000))

Formatted = "{:*^15,d}"
print(Formatted.format(7000))

Formatted = "{:*^15.2f}"
print(Formatted.format(7000))

Formatted = "{:*>15X}"
print(Formatted.format(7000))

Formatted = "{:*<#15x}"
print(Formatted.format(7000))

Formatted = "A {0} {1} and a {0} {2}."
print(Formatted.format("blue", "car", "truck"))
```

The example starts simply with a field formatted as a decimal value. It then adds a thousands separator to the output. The next step is to make the field wider than needed to hold the data and to center the data within the field. Finally, the field has an asterisk added to pad the output.

Of course, there are other data types in the example. The next step is to display the same data in fixed-point format. The example also shows the output in both uppercase and lowercase hexadecimal format. The uppercase output is right aligned and the lowercase output is left aligned.

Finally, the example shows how you can use numbered fields to your advantage. In this case, it creates an interesting string output that repeats one of the input values.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application outputs data in various forms, as shown in Figure 11-6.

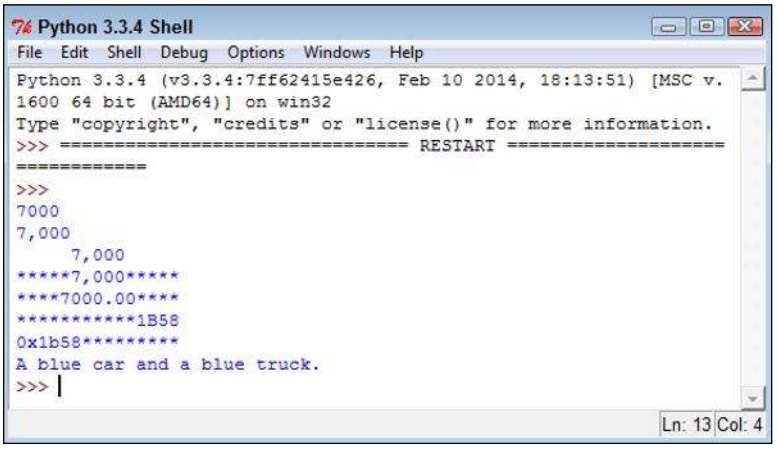

Figure 11-6: Use formatting to present data in precisely the form you want.