35

Python

Objectives

- To understand basic Python data types.
- To understand string processing and regular expressions in Python.
- To use exception handling.
- To perform basic CGI tasks in Python.
- To construct programs that interact with MySQL databases using the Python Database Application Programming Interface (DB-API).

Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern.
Alfred North Whitehead

No rule is so general, which admits not some exception.
Robert Burton

Outline

35.1 Introduction

Python is an interpreted, cross-platform, object-oriented language that can be used to write large-scale Internet search engines, small administration scripts, GUI applications, CGI scripts and more. The creator of the language, Guido van Rossum, combined a clean syntax with popular elements from several existing languages to produce Python.

Python is a freely distributed technology whose open-source nature has encouraged a wide base of developers to submit modules that extend the language. Using Python's core modules and those available for free on the Web, programmers can develop applications that accomplish a great variety of tasks. Python's interpreted nature facilitates rapid application development (RAD) of powerful programs. GUI applications, in particular, can be developed and tested quickly using Python's interface to Tcl/Tk (among other GUI toolkits).

For this chapter, we assume that the reader has installed Python 2.0 or later. *ActivePython* is the industry-standard distribution of Python for Windows, Solaris and Linux platforms. ActivePython is available as a free download from `www.activestate.com/Products/ActivePython`. Follow the instructions available at this site to install and configure Python on your computer. Several other distributions of Python are available at `www.python.org`.

35.1.1 First Python Program

In this section, we examine a simple Python program and explain how to work with the Python programming environment. Python can execute programs stored in files, or can run in **interactive mode**, where users enter lines of code one at a time. Among other things, interactive mode enables program writers to test small blocks of code quickly and contributes to a relatively rapid development time for most Python projects.

Figure 35.1 is a simple Python program that prints the text `Welcome to Python!` to the screen. Lines 1–2 contain single-line comments that describe the program. Comments in Python begin with the # character; Python ignores all text in the current line after this character. Line 4 uses the `print` statement to write the text `Welcome to Python!` to the screen.

```
1  # Fig. 35.1: fig35_01.py
2  # A first program in Python
3
4  print "Welcome to Python!"
```

```
Welcome to Python!
```

Fig. 35.1 Simple Python program.

Python statements can be executed in two ways. The first is by typing statements into a file (as in Fig. 35.1). Python files typically end with `.py`, although other extensions (e.g., `.pyw` on Windows) can be used. Python is then invoked on the file by typing

```
python file.py
```

at the command line, where `file.py` is the name of the Python file. [*Note:* To invoke Python, the system's `PATH` variable must be set properly to include the `python` executable. If you installed ActivePython, the appropriate variable should already be set. If you are using a different version of Python, please consult its documentation for information on setting up the correct variable.] The output box of Fig. 35.1 contains the results of invoking Python on `fig35_01.py`.

Python statements can also be interpreted interactively. Typing

```
python
```

at the command prompt runs Python in interactive mode.

Error-Prevention Tip 35.1

In interactive mode, Python statements can be entered and interpreted one at a time. This mode is often useful when debugging a program (i.e., discovering and removing errors in the program).

Figure 35.2 shows Python running in interactive mode on Windows. The first two lines display information about the version of Python being used. The third line begins with the **Python prompt** (>>>). A Python statement is interpreted by typing the statement at the Python prompt and pressing the **Enter** or **Return** key.

```
Python 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>> print "Welcome to Python!"
Welcome to Python!
>>> ^Z
```

Fig. 35.2 Python in interactive mode.

The `print` statement on the third line prints the text `Welcome to Python!` to the screen. After printing the text to the screen, the Python prompt is displayed again (line 5), and Python waits for the user to enter the next statement. We exit Python by typing *Crtl-Z* (on Microsoft Windows systems) and pressing the *Return* key. [*Note:* On UNIX and Linux systems, *Ctrl-D* exits Python.]

35.1.2 Python Keywords

Before we discuss Python programming in more detail, we present a list of Python's **keywords** (Figure 35.3). These words have special meanings in Python and cannot be used as variable names, function names or other objects.

Python keywords

and	continue	else	for	import	not	raise
assert	def	except	from	in	or	return
break	del	exec	global	is	pass	try
class	elif	finally	if	lambda	print	while

Fig. 35.3 Python keywords.

A list of Python keywords can also be obtained from the **keyword module**. Figure 35.4 illustrates how to obtain the list of Python keywords in interactive mode. [*Note*: We discuss modules further in Section 35.4.]

```
Python 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>> import keyword
>>> print keyword.kwlist
['and', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try',
'while']
>>>
```

Fig. 35.4 Printing Python keywords in interactive mode.

Python is a case-sensitive language. This means that Python treats variable `x` (lowercase) and variable `X` (uppercase) as two different variables. Similarly, the statement

```
Def = 3
```

is a valid Python statement, but the statement

```
def = 3
```

causes a syntax error, because `def` is a keyword and, therefore, not a valid variable name.

Good Programming Practice 35.1

Using variable or function names that resemble keywords (e.g., variable `Def`*) or Python functions (e.g.,* `list`*) may confuse the program writer and readers. Avoid using such variable or function names.*

35.2 Basic Data Types, Control Statements and Functions

This section introduces basic data types, control statements and functions, using a simple program (Fig. 35.5). In this program, we define two functions that use control statements to perform the operations of these functions.

Line 5 contains the function definition header for function `greatestCommonDivisor`. This function computes the **greatest common divisor** of two numbers—the largest integer that divides evenly into both numbers. The keyword `def` marks the beginning of the function definition. The function takes two parameters: `x` and `y`. The list of parameters is placed inside parentheses (`()`), and the parameter list is followed by a **colon** (`:`).

```
1  # Fig. 35.5: fig35_05.py
2  # Program to illustrate basic data types, control statements and
3  # functions.
4
5  def greatestCommonDivisor( x, y ):
6     gcd = min( x, y )
7
8     while gcd >= 1:
9
10       if ( x % gcd ) == ( y % gcd ) == 0:
11          return gcd
12       else:
13          gcd -= 1
14
15 def determineColor( color ):
16
17    if color == "green":
18       print "You entered green!"
19    elif color == "purple":
20       print "You entered purple!"
21    else:
22       print "You did not enter green or purple."
23
24 number1 = int( raw_input( "Enter a positive integer: " ) )
25 number2 = int( raw_input( "Enter a positive integer: " ) )
26
27 print "The greatest common divisor is", \
28    greatestCommonDivisor( number1, number2 )
29
30 for entry in range( 5 ):
31    colorChoice = raw_input( "\nEnter your favorite color: " )
32    determineColor( colorChoice )
```

Fig. 35.5 Data types, control statements and functions. (Part 1 of 2.)

```
Enter a positive integer: 2
Enter a positive integer: 30
The greatest common divisor is 2

Enter your favorite color: yellow
You did not enter green or purple.

Enter your favorite color: green
You entered green!

Enter your favorite color: black
You did not enter green or purple.

Enter your favorite color: purple
You entered purple!

Enter your favorite color: red
You did not enter green or purple.
```

Fig. 35.5 Data types, control statements and functions. (Part 2 of 2.)

Common Programming Error 35.1

Forgetting to place a colon after a function definition header or after a control statement is a syntax error.

Line 6 calls Python function **`min`** on parameters `x` and `y`. This function returns the smaller of the two values. We assign the value returned by `min` to local variable `gcd`.

Note that line 6 is indented. Unlike many other languages, Python determines the beginning and end of a statement based on whitespace. Each new line begins a new statement. The indentation in line 6 marks the beginning of the code block that belongs to function `greatestCommonDivisor`. Groups of statements that belong to the same block of code are indented by the same amount. The language does not specify how many spaces to indent, only that the indentation must be consistent.

Common Programming Error 35.2

Inconsistent indentation in a Python program causes syntax errors.

Line 8 describes the beginning of a Python **`while` loop**. The code in the `while` block executes as long as `gcd` is greater than or equal to 1.

Line 10 is a Python **`if`** statement. If the specified condition is true (i.e., the condition evaluates to any nonzero value), the code in the `if` block (i.e., the indented code that follows the `if` statement) executes. The statement in line 10 uses the **modulo operator (%)** to determine whether parameters `x` and `y` can be divided evenly by variable `gcd`. The statement illustrates the fact that Python comparison expressions can be "chained." This code is identical to

```
if ( x % gcd ) == 0 == ( y % gcd ):
```

and to

```
if x % gcd == 0 and y % gcd == 0:
```

Chaining occurs left to right; therefore, the preceding expression is more efficient than the expression presented in the code, because the preceding expression may save a division operation.

If the expression in line 10 is true, we have found the greatest common divisor. The **`return`** keyword (line 11) exits the function and returns the specified value.

If the expression in line 10 is false (i.e., the condition evaluates to zero), the code in the `else` block (lines 12–13) executes. This code decrements variable `gcd` by 1, using the `-=` statement and has the same effect as the statement

```
gcd = gcd - 1
```

Python defines several such operators, including `+=`, `-=`, `*=`, `/=`, `%=` (modulo division) and `**=` (exponentiation). [*Note:* These operators are new as of Python 2.0; using these operators in Python 1.5.2 or earlier causes a syntax error.]

Function `determineColor` (lines 15–22) takes parameter `color`, which contains a string. Lines 17–22 use the **`if...elif...else` control statement** to evaluate expressions based on the value of the parameter. If the value of parameter `color` is equal to the string `"green"` (line 17), the function prints `"You entered green!"` If the value of `color` is equal to the string `"purple"` (line 19), the function prints `"You entered purple!"` If the value of `name` does not match either of these strings (line 21), the function prints `"You did not enter green or purple."` Function `determineColor` illustrates simple Python string comparisons. We discuss string comparison and other string manipulations in Section 35.4.

Line 24 calls Python function `raw_input` to get input from the user. This function takes an optional string argument that is displayed as a prompt to the user. The `raw_input` function returns a string. The Python function `int` takes as an argument a noninteger type and returns an integer representation of the argument. We store the integer returned from function `int` in local variable `number1`. Line 25 retrieves a value for `number2` in a similar fashion.

Common Programming Error 35.3

A numerical value obtained via the `raw_input` function must be converted from a string to the proper numerical type. Manipulating a string representation of a numerical value may result in a logical or syntactical error.

Lines 27–28 print the greatest common divisor of the two numbers to the screen. The backslash character (\) at the end of line 27 is a **line-continuation character** that allows us to continue a statement on the next line. The **comma** (`,`) that follows the string informs Python that we want to print additional items after the string. In this case, the additional item is the integer value returned by the call to function `greatestCommonDivisor`. Note from the output that Python automatically inserts a space between the last character in the string and the integer value.

Common Programming Error 35.4

Forgetting to include a line-continuation character (\) at the end of a statement that continues onto the next line is a syntax error.

Line 30 begins a Python `for` loop. The call to Python function `range` with an argument of `5` returns the values `0`, `1`, `2`, `3` and `4`. [*Note:* The function actually returns a **list** that contains these values. We discuss lists in Section 35.3.] The `for` loop iterates through these values and, on each iteration, assigns a value to variable `entry` and then executes the state-

ments in the `for` block (lines 31–32). Thus, the statements in the `for` block execute five times. These statements retrieve a string from the user and pass it to function `determineColor`. Note the "`\n`" **escape sequence** at the beginning of the string in line 31. This is a special Python character that prints a **newline** to the screen. A newline causes the cursor (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen. The program exits after calling function `determineColor` on five user-defined strings. Figure 35.6 lists some common Python escape sequences.

Escape sequence	Meaning
`\n`	Newline (line feed).
`\r`	Carriage return.
`\t`	Tab.
`\'`	Single quote.
`\"`	Double quote.
`\b`	Backspace.
`\\`	Backslash.

Fig. 35.6 Escape sequences.

35.3 Tuples, Lists and Dictionaries

In addition to basic data types that store numerical values and strings, Python defines three data types for storing more complex data: the `list` (a sequence of related data), the **tuple** (pronounced *too-ple*; a list whose elements may not be modified) and a **dictionary** (a list of values that are accessed through their associated keys). These data types are high-level implementations of simple data structures that enable Python programmers to manipulate many types of data quickly and easily. Some Python modules (e.g., `Cookie` and `cgi`) use these data types to provide simple access to their underlying data structures. Figure 35.7 is a program that illustrates tuples, lists and dictionaries.

```
1  # Fig. 35.7: fig35_07.py
2  # A program that illustrates tuples, lists and dictionaries.
3
4  # tuples
5  aTuple = ( 1, "a", 3.0 )       # create tuple
6  firstItem = aTuple[ 0 ]        # first tuple item
7  secondItem = aTuple[ 1 ]       # second tuple item
8  thirdItem = aTuple[ 2 ]        # third tuple item
9
10 print "The first item in the tuple is", firstItem
11 print "The second item in the tuple is", secondItem
12 print "The third item in the tuple is", thirdItem
13 print
14
```

Fig. 35.7 Tuples, lists and dictionaries. (Part 1 of 3.)

```
15  firstItem, secondItem, thirdItem = aTuple
16  print "The first item in the tuple is", firstItem
17  print "The second item in the tuple is", secondItem
18  print "The third item in the tuple is", thirdItem
19  print
20
21  aTuple += ( 4, )
22  print "Used the += statement on the tuple"
23  print
24
25  # print the tuple
26  print "The raw tuple data is:", aTuple
27  print "The items in the tuple are:"
28
29  for item in aTuple:   # print each item
30      print item,
31
32  print      # end previous line
33  print      # blank line
34
35  # lists
36  aList = [ 1, 2, 3 ]      # create list
37  aList[ 0 ] = 0           # change first element of list
38  aList.append( 5 )        # add item to end of list
39
40  print "The raw list data is:", aList   # print list data
41  print
42
43  aList += [ 4 ]     # add an item to the end of the list
44  print "Added an item to the list using the += statement"
45  print
46
47  # print each item in the list
48  print "The items in the list are:"
49
50  for item in aList:
51      print item,
52
53  print      # end previous line
54  print      # blank line
55
56  # dictionaries
57  aDictionary = { 1 : "January", 2 : "February", 3 : "March",
58                  4 : "April", 5 : "May", 6 : "June", 7 : "July",
59                  8 : "August", 9 : "September", 10 : "October",
60                  11 : "November" }
61  aDictionary[ 12 ] = "December"          # add item to dictionary
62
63  print "The raw dictionary data is:", aDictionary
64  print "\nThe entries in the dictionary are:"
65
66  for item in aDictionary.keys():
67      print "aDictionary[ ", item, " ] = ", aDictionary[ item ]
```

Fig. 35.7 Tuples, lists and dictionaries. (Part 2 of 3.)

```
The first item in the tuple is 1
The second item in the tuple is a
The third item in the tuple is 3.0

The first item in the tuple is 1
The second item in the tuple is a
The third item in the tuple is 3.0

Used the += statement on the tuple

The raw tuple data is: (1, 'a', 3.0, 4)
The items in the tuple are:
1 a 3.0 4

The raw list data is: [0, 2, 3, 5]

Added an item to the list using the += statement

The items in the list are:
0 2 3 5 4

The raw dictionary data is: {12: 'December', 11: 'November', 10: 'October', 9:
'September', 8: 'August', 7: 'July', 6: 'June', 5: 'May', 4: 'April', 3:
'March', 2: 'February', 1: 'January'}

The entries in the dictionary are:
aDictionary[  12  ] =  December
aDictionary[  11  ] =  November
aDictionary[  10  ] =  October
aDictionary[  9  ] =  September
aDictionary[  8  ] =  August
aDictionary[  7  ] =  July
aDictionary[  6  ] =  June
aDictionary[  5  ] =  May
aDictionary[  4  ] =  April
aDictionary[  3  ] =  March
aDictionary[  2  ] =  February
aDictionary[  1  ] =  January
```

Fig. 35.7 Tuples, lists and dictionaries. (Part 3 of 3.)

Line 5 creates a tuple with elements `1`, `"a"` and `3.0`. Tuples are created as a comma-separated list of values inside parentheses. A tuple is used most often to contain combinations of many data types (e.g., strings, integers, other tuples). Lines 6–8 use the `[]` operator to access specific elements through an index. The first element in a tuple has index 0.

Tuple element contents are **immutable**—they cannot be modified. So the statement

```
aTuple[ 0 ] = 0
```

produces a runtime error similar to

```
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: object doesn't support item assignment
```

Common Programming Error 35.5

Attempting to change an immutable data structure is a syntax error.

Attempting to access a value at a nonexistent element is also an error. The statement

```
print aTuple[ 10 ]
```

produces a runtime error similar to

```
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IndexError: tuple index out of range
```

because `aTuple` does not have a tenth element.

Common Programming Error 35.6

Trying to access an out-of-range element (i.e., an element at an index that does not exist) produces a runtime error.

Line 15 **unpacks** the items of the tuple into three variables. This statement produces the same results as lines 6–8. Line 21 has the effect of adding an element to the end of variable `aTuple`. The right-hand side of the += statement must be a tuple; therefore, we must specify a **one-element tuple**, or **singleton**, on the right-hand side of the statement. The value `( 4, )` is a one-element tuple. The comma after the tuple element value is mandatory, because the value `( 4 )` is an integer.

Because tuples are immutable, the += statement actually creates a new tuple that combines the tuple on the left side of the += sign (i.e., `aTuple`) with the tuple on the right side of the += sign (i.e., `( 4, )`) to create a new tuple. The new tuple is stored in variable `aTuple`.

The output of line 26 shows how the `print` statement handles a variable that is a tuple. Lines 29–30 use a `for` loop to print each element in variable `aTuple`.

The statement in line 29 assigns the first element in `aTuple` (i.e., `aTuple[ 0 ]`) to variable `item`. Line 30 then prints the value of variable `item` to the screen. The `for` loop iterates over each element in the tuple, assigns the element to variable `item` and executes the code in line 30.

By default, the `print` statement writes a newline character (e.g., a carriage return) at the end of its output; however, the comma in line 30 tells Python not to print the newline character. In the next iteration of the `for` loop, the `print` statement writes text to the screen on the same line as the previous `print` statement. Lines 32–33 print a new line and a blank line to the screen, respectively, after all the elements in the tuple have been displayed

Line 36 creates a list that contains elements `1`, `2` and `3`. Python lists are similar to tuples, except that Python lists are **mutable** (they may be altered). Line 37 demonstrates this fact by assigning the value `0` to the element in the list at index 0. Line 38 adds an element to the end of a list by calling list method `append`. Lists also support several other methods (Fig. 35.8).

The output from the statement in line 40 shows how the `print` statement handles a variable that is a list. Line 43 adds the integer `4` to variable `aList`, using the += statement. The value on the right side of the += statement must be a list (or another sequence, such as a string or tuple). In this case, the list contains one element. The `for` statement (lines 50–51) prints each element of the list to the screen.

Method	Purpose
`append(` *item* `)`	Inserts *item* at the end of the list.
`count(` *item* `)`	Returns the number of occurrences of *item* in the list.
`extend(` *newList* `)`	Inserts *newList* at the end of the list.
`index(` *item* `)`	Returns the index of the first occurrence of *item* in the list. If the element is not in the list, a `ValueError` exception occurs. [*Note:* We discuss exceptions in Section 35.5]
`insert(` *index*, *item* `)`	Inserts *item* at position *index*.
`pop(` [*index*] `)`	Removes and returns the last element in the list. If the optional parameter *index* is specified, removes and returns the element at position *index*.
`remove(` *item* `)`	Removes the first occurrence of *item* from the list. If *item* is not in the list, a `ValueError` exception occurs.
`reverse()`	Reverses the items in the list.
`sort(` [*function*] `)`	Sorts the items in the list. Optional parameter *function* is a comparison function that may be user defined.

Fig. 35.8 Python list methods.

Lines 57–60 create a Python dictionary. Each entry in a dictionary has two parts—a **key** and a **value**—and a dictionary consists of a set of zero or more comma-separated key-value pairs. A value in a dictionary is manipulated using its key. The key must be of an immutable data type (e.g., a number, a string or a tuple that contains only immutable data types); dictionary values may be any data type. Each key-value pair takes the form **key : value**.

Line 61 illustrates how to add a new element to a dictionary by using the `[]` operator. Because a value must be accessed using its corresponding key, each key in a dictionary must be unique. For example, the statements

```
month = { 11 : "November" }
month[ 11 ] = "Nov."
```

would create a dictionary and then change the value associated with key `11` from `"November"` to the abbreviation `"Nov."`.

Lines 66–67 use a `for` loop to print each key-value pair in variable `aDictionary`. Method `keys` returns an unordered list of all the keys in the dictionary. Dictionaries also support several other methods (Fig. 35.9). The `for` loop iterates over each key and prints the key and its corresponding value. Each value in the dictionary is accessed using the `[]` operator (line 67).

Method	Description
`clear()`	Deletes all items from the dictionary.

Fig. 35.9 Python dictionary methods. (Part 1 of 2.)

Method	Description
`copy()`	Creates a copy of the dictionary.
`get(` *key* [, *falseValue*] `)`	Returns the value associated with *key*. If *key* is not in the dictionary and if *falseValue* is specified, returns the specified value.
`has_key(` *key* `)`	Returns `1` if *key* is in the dictionary; returns `0` if *key* is not in the dictionary.
`items()`	Returns a list of tuples that are key-value pairs.
`keys()`	Returns a list of keys in the dictionary.
`setdefault(` *key* [, *falseValue*] `)`	Behaves similarly to method `get`. If key is not in the dictionary and *falseValue* is specified, inserts the key and the specified value into the dictionary.
`update(` *otherDictionary* `)`	Adds all the key-value pairs from *otherDictionary* to the current dictionary.
`values()`	Returns a list of values in the dictionary.

Fig. 35.9 Python dictionary methods. (Part 2 of 2.)

35.4 String Processing and Regular Expressions

Programmers use string processing to accomplish a variety of tasks. System administration scripts can use Python modules and strings to process text files. Web programmers can use Python CGI scripts to compose and modify Web pages, to validate user-entered data from an XHTML form or to aggregate and display data from a variety of sources. This section discusses simple string processing in Python, including the use of **regular expressions**. A regular expression string defines a pattern with which text data can be compared. Regular expressions are used to search through strings, text files, databases, and so on. Regular expressions are not part of the core Python language, but regular expression processing capability is available through the standard Python `re` **module**.

Figure 35.10 demonstrates the use of strings in Python. Lines 5–6 assign the value `"This is a string."` to variable `string1` and print that value to the screen. In lines 8–9, we assign a similar value to variable `string2` and print that string.

```
1  # Fig. 35.10: fig35_10.py
2  # Program to illustrate use of strings
3
4  # simple string assignments
5  string1 = "This is a string."
6  print string1
7
8  string2 = "This is a second string."
9  print string2
10
```

Fig. 35.10 Using strings in Python. (Part 1 of 2.)

```
11  # string concatenation
12  string3 = string1 + " " + string2
13  print string3
14
15  # using operators
16  string4 = '*'
17  print "String with an asterisk: " + string4
18  string4 *= 10
19  print "String with 10 asterisks: " + string4
20
21  # using quotes
22  print "This is a string with \"double quotes.\""
23  print 'This is another string with "double quotes."'
24  print 'This is a string with \'single quotes.\''
25  print "This is another string with 'single quotes.'"
26  print """This string has "double quotes" and 'single quotes.'"""
27
28  # string formatting
29  name = raw_input( "Enter your name: " )
30  age = raw_input( "Enter your age: " )
31  print "Hello, %s, you are %s years old." % ( name, age )
```

```
This is a string.
This is a second string.
This is a string. This is a second string.
String with an asterisk: *
String with 10 asterisks: **********
This is a string with "double quotes."
This is another string with "double quotes."
This is a string with 'single quotes.'
This is another string with 'single quotes.'
This string has "double quotes" and 'single quotes.'
Enter your name: Brian
Enter your age: 33
Hello, Brian, you are 33 years old.
```

Fig. 35.10 Using strings in Python. (Part 2 of 2.)

In line 12, three strings—`string1`, `" "` and `string2`—are concatenated with operator +. We then print this new string (`string3`).

Lines 16–17 create and print a string with a single character—an asterisk. Line 18 uses the *= statement to concatenate `string4` to itself 10 times. We print the resulting string in line 19. Python also defines the += statement for strings, which effectively concatenates two strings. [*Note:* Since strings are immutable, the *= and += statements actually create new strings to perform their respective operations.]

Lines 22–26 illustrate the use of quotes in a string. Line 22 shows one method of displaying double quotes inside a string. The double quotes are displayed using the **escape character** (\). If we omit the escape character, then Python interprets the double-quote character as marking the end of the string, rather than as a character within the string itself. Line 23 presents another method of displaying double quotes inside a string. Note that the entire string is contained within single quotes (`'`). Python strings may be contained within

either double quotes or single quotes. As line 23 demonstrates, if a string is contained within single quotes, then double quotes within the string do not need to be "escaped" with the backslash character. Similarly, if a string is contained within double quotes (line 25), then single quotes within the string do not need to be escaped.

If we do not want to escape quote characters in a string, we can place the entire string within pairs of three consecutive double-quote characters (line 26). This is called a **triple-quoted string**—triple-quoted strings may alternatively be surrounded by sets of three consecutive single-quote characters (`'''`). We use triple-quoted strings later in this chapter to output large blocks of XHTML from CGI scripts.

In lines 29–30, we use Python function `raw_input` to input the user's name and age. In line 31, we format a string to incorporate the input data. The `%` **format character** acts as a placeholder in the string. The format character `s` indicates that we want to place another string within the current string at the specified point. Figure 35.11 lists several format characters for use in string formatting. [*Note*: See Appendix E on number systems for a discussion of the numeric terminology in Fig. 35.11.]

Symbol	Meaning
`c`	Single character (i.e., a string of length 1).
`s`	String.
`d`	Signed decimal integer.
`u`	Unsigned decimal integer.
`o`	Unsigned octal integer.
`x`	Unsigned hexadecimal integer (using format `abcdef`).
`X`	Unsigned hexadecimal integer (using format `ABCDEF`).
`f`	Floating-point number.
`e, E`	Floating-point number (using scientific notation).
`g, G`	Floating-point number (using least-significant digits).

Fig. 35.11 String-format characters.

At the end of line 31, we use the `%` operator to indicate that the formatting characters in the string are to be replaced with the values listed between the parentheses. Python constructs the string from left to right by matching a placeholder with the next value specified between parentheses and replacing the formatting character with that value.

Figure 35.12 presents some of Python's regular expression operations. Line 4 **imports the `re` (regular expression) module**. A `module` contains data and functions that a program can use to accomplish a specific task. After importing a module, the program can make use of the data and functions it contains. In our example, importing the `re` module enables us to access data and functions that facilitate regular-expression processing.

Line 8 compiles the regular expression `"Test"`, using the `re` module's `compile` function. This method returns an object of type `SRE_Pattern`, which represents a compiled regular expression.

```
# Fig. 35.12: fig35_12.py
# Program searches a string using the regular expression module.

import re

searchString = "Testing pattern matches"

expression1 = re.compile( r"Test" )
expression2 = re.compile( r"^Test" )
expression3 = re.compile( r"Test$" )
expression4 = re.compile( r"\b\w*es\b" )
expression5 = re.compile( r"t[aeiou]", re.I )

if expression1.search( searchString ):
   print '"Test" was found.'

if expression2.match( searchString ):
   print '"Test" was found at the beginning of the line.'

if expression3.match( searchString ):
   print '"Test" was found at the end of the line.'

result = expression4.findall( searchString )

if result:
   print 'There are %d words(s) ending in "es":' % \
      ( len( result ) ),

   for item in result:
      print " " + item,

print
result = expression5.findall( searchString )

if result:
   print 'The letter t, followed by a vowel, occurs %d times:' % \
      ( len( result ) ),

   for item in result:
      print " " + item,

print
```

```
"Test" was found.
"Test" was found at the beginning of the line.
There are 1 words(s) ending in "es":  matches
The letter t, followed by a vowel, occurs 3 times:  Te  ti  te
```

Fig. 35.12 Regular expressions to search a string.

Software Engineering Observation 35.1

If a program uses a regular expression string many times, compiling that string can speed up the regular expression comparisons.

Figure 35.13 lists the most popular regular expression symbols recognized by the `re` module. Unless otherwise specified, regular expression characters `*` and `+` match as many occurrences of a pattern as possible. For example, the regular expression `hel*o` matches strings that have the letters `he`, followed by any number of `l`'s, followed by an `o` (e.g., `"heo"`, `"helo"`, `"hello"`, `"hellllo"`).

Character	Matches
`^`	Beginning of string.
`$`	End of string.
`.`	Any character, except a newline.
`*`	Zero or more occurrences of the pattern.
`+`	One or more occurrences of the preceding pattern.
`?`	Zero or one occurrence of the preceding pattern.
`{m, n}`	Between `m` and `n` occurrences of the preceding pattern.
`\b`	Word boundary (i.e., the beginning or end of a word).
`\B`	Nonword boundary.
`\d`	Digit ([0–9]).
`\D`	Nondigit.
`\w`	Any alpha-numeric character.
`[...]`	Any character defined by the set.
`[^...]`	Any character not defined by the set.

Fig. 35.13 `re` module's regular expression characters.

Lines 9–12 of Fig. 35.12 use a few of these symbols to compile four regular expression patterns. The expression in line 9 (`expression2`) matches the string `"Test"` at the beginning of a line. The expression in line 10 (`expression3`) matches the string `"Test"` at the end of a line. The expression in line 11 (`expression4`) matches a word that ends with `"es"`. The expression in line 12 (`expression5`) matches the letter `t`, followed by a vowel. Line 12 illustrates the optional second argument that function `compile` may take. This argument is a flag that describes how the regular expression will be used when the expression is matched against a string. The **`re.I`** flag means that case is ignored when using the regular expression to process a string.

The `r` character before each string in lines 8–12 indicates that the string is a **raw string**. Python handles backslash characters in raw strings differently than in "normal" strings. Specifically, Python does not interpret backslashes as escape characters. Writing all regular expressions as raw strings can help programmers avoid writing regular expressions that may be interpreted in a way that was not intended. For example, without the raw-string character, the regular expression string in line 11 would have to be written as `\\b\\w*es\\b`, because `\b` is a backspace to Python, but a word boundary in regular expressions.

Line 14 uses the `SRE_Pattern`'s **`search` method** to test `searchString` against the regular expression `expression1`. The `search` method returns an `SRE_Match` object. If

`search` does not find any matching substrings, it returns **`None`**. `None` is a Python type whose value indicates that no value exists. In a Python `if` statement, `None` evaluates to false; therefore, we only need to test the return value to determine whether any matches were found. If a match is found, we print an appropriate message.

Line 17 uses `SRE_Pattern`'s **`match` method** to test `searchString` against regular expression `expression2`. The `match` method returns an `SRE_Match` object only if the string matches the pattern exactly.

Line 23 uses `SRE_Pattern`'s **`findall` method** to store in variable `result` a list of all the substrings in `searchString` that match the regular expression `expression4`. If `findall` returns any matches, we print a message that indicates how many words were found (lines 25–27) by using Python function `len`. When run on a list, function `len` returns the number of elements in the list. Lines 29–30 print each item in the list, followed by a space.

Lines 33–40 perform similar processing with `expression5` to print all the substrings in `searchString` that match the pattern of the letter `t` followed by a vowel. Remember that `expression5` was compiled using the `re.I` flag. Thus the letter `t` or the vowels in `searchString` can be either lowercase or uppercase. We end the program by printing a new line.

35.5 Exception Handling

In an interpreted language such as Python, errors pose a unique problem, because many errors that would be caught at compilation time for a compiled language are not caught until runtime in an interpreted language. These errors cause **exceptions** in Python. **Exception handling** enables programs and programmers to identify an error when it occurs and take appropriate action. Exception handling is geared to situations in which a code block detects an error but is unable to deal with it. Such a block of code will **raise an exception**. The programmer can write code that **catches the exception** and handles the error in a "graceful" manner.

Python accomplishes exception handling through the use of **`try`…`except` blocks**. Any code that causes an error raises an exception. If this code is contained in a `try` block, the corresponding `except` block then catches the exception (i.e., handles the error). The core Python language defines a hierarchy of exceptions. A Python `except` block can catch one of these exceptions, or a subset of these exceptions, or it can specify none of these exceptions, in which case the code block catches all exceptions. Figure 35.14 shows how dividing by zero raises a `ZeroDivisionError` exception.

```
Python 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>>
```

Fig. 35.14 Interactive session illustrating a `ZeroDivisionError` exception.

Figure 35.15 presents a simple program that illustrates exception handling in Python. The program requests two numbers from the user, then attempts to divide the first number by the second.

```
1  # Fig. 35.15: fig35_15.py
2  # A simple program that illustrates exceptions.
3
4  def getFloat():
5     return float( raw_input( "Enter a number: " ) )
6
7  number1 = number2 = None
8
9  while number1 == None:
10    try:
11       number1 = getFloat()
12    except ValueError:
13       print "Value entered was not a number"
14
15 while number2 == None:
16    try:
17       number2 = getFloat()
18    except ValueError:
19       print "Value entered was not a number"
20
21 try:
22    result = number1 / number2
23 except ZeroDivisionError:
24    print "Cannot divide by zero!"
25 else:
26    print "The result of division is: %f" % result
```

```
Enter a number: g
Value entered was not a number
Enter a number: 45
Enter a number: 0
Cannot divide by zero!
```

Fig. 35.15 Exception handling in Python.

Lines 4–5 define function `getFloat`, which prompts the user for a number and returns the number that the user enters. This function gets user input through Python function `raw_input` and then obtains the user-entered value as a floating-point value with Python function `float`.

Line 7 creates two variables (`number1` and `number2`) and assigns `None` to both. Lines 9–19 use `while` loops to store user-entered values in these variables by using function `getFloat`, with exception handling. In lines 9 and 15, we use the == comparison operator to test whether the program has received a valid number. Lines 10–11 define a `try` block. Any code in the `try` block that raises an exception will be "caught" and handled in the corresponding `except` block (lines 12–13). The `try` block calls function `getFloat` to get the user input.

If the user does not enter a numerical value at the prompt, the `float` function raises a `ValueError` exception, which is caught by the `except` block (lines 12–13). This block prints an appropriate message before program control returns to the top of the `while` loop. Lines 15–19 repeat the same action to get a floating-point value for variable `number2`.

Lines 21–26 print the results of dividing variables `number1` and `number2`. We place the call to `divideNumbers` in the `try` block. As we saw in Fig. 35.14, if a program attempts to divide by zero, it raises a `ZeroDivisionError`. The `except` block in lines 26–27 catches this exception and prints an appropriate message to the screen.

A `try` block may optionally specify a corresponding `else` block (lines 25–26). If the code in the `try` block does not raise an exception, the program executes the code in the `else` block. If an exception is raised in the `try` block, the `else` block is not executed. In our example, the `else` block prints the result of the division.

Good Programming Practice 35.2

In general, we want to minimize the amount of code contained in a `try` *block. Usually, we only place code in a* `try` *block that could raise an exception that we are capable of handling. In the* `else` *block, we place code that we want to run if no exception is raised in the* `try` *block.*

35.6 CGI Programming

Python has many uses on the Web. Modules `cgi` (for access to XHTML forms), `Cookie` (to read and write cookies), `smtplib` (to manipulate SMTP messages), `urllib` (to manipulate Web data), `ftplib` (to perform client-side FTP tasks) and others provide powerful extensions that Web programmers can use to write CGI scripts quickly for almost any task. This section introduces Python CGI programming. Sections 35.7–35.9 present more detailed CGI applications. We assume that the reader has installed and configured the Apache Web server. Apache does not usually need any special configuration to run a Python script; a script need merely be placed in the specified `cgi-bin` directory. XHTML documents should be placed in the server's document root directory. See Chapter 21 for more details.

Figure 35.16 gathers all the CGI environment variables and values and organizes them in an XHTML table that is displayed in a Web browser. Line 1

```
#!c:\Python\python.exe
```

is a **directive** (sometimes called the **pound-bang** or **shebang**) that provides the server with the location of the Python executable. This directive must be the first line in a CGI script. For UNIX-based machines, this value might commonly be

```
#!/usr/bin/python or #!/usr/local/bin/python
```

depending on the actual location of the Python executable. [*Note*: If necessary, be sure to modify the shebang in each of the remaining chapter examples to reflect the actual location of Python on your system.]

```
1  #!c:\Python\python.exe
2  # Fig 35.16: fig35_16.py
3  # Program to display CGI environment variables
4
5  import os
6  import cgi
7
```

Fig. 35.16 Displaying environment variables via CGI. (Part 1 of 2.)

```
 8  print "Content-type: text/html"
 9  print
10
11  print """<!DOCTYPE html PUBLIC
12     "-//W3C//DTD XHTML 1.0 Transitional//EN"
13     "DTD/xhtml1-transitional.dtd">"""
14
15  print """
16  <html xmlns = "http://www.w3.org/1999/xhtml" xml:lang="en"
17     lang="en">
18     <head><title>Environment Variables</title></head>
19        <body><table style = "border: 0">"""
20
21  rowNumber = 0
22
23  for item in os.environ.keys():
24      rowNumber += 1
25
26      if rowNumber % 2 == 0:
27          backgroundColor = "white"
28      else:
29          backgroundColor = "lightgrey"
30
31      print """<tr style = "background-color: %s">
32      <td>%s</td><td>%s</td></tr>""" \
33            % ( backgroundColor, item,
34                cgi.escape( os.environ[ item ] ) )
35
36  print """</table></body></html>"""
```

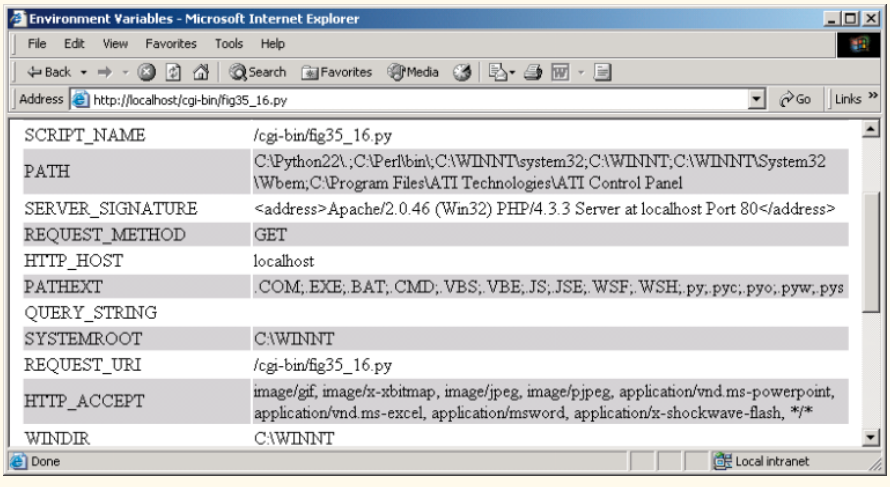


Fig. 35.16 Displaying environment variables via CGI. (Part 2 of 2.)

Line 5 imports the **os module** that allows developers to access information about the operating system. We use the `os` module in this example to retrieve the system's environment variables. Line 6 imports the **`cgi` module** that provides functionalities for writing

CGI scripts. In this example, we use the module to format output; in later examples, we use module `cgi` to perform more complex CGI tasks.

Lines 8–9 print a valid HTTP header. Browsers use HTTP headers to determine how to handle the incoming data, and a valid header must be sent to ensure that the browser displays the information correctly. The blank line below the header is required; without this line, the content will not be delivered properly to the client. Lines 11–13 print the XHTML `DOCTYPE` string to the browser.

The **`environ` data member** (line 23) of module `os` (imported in line 5) holds all the environment variables. This data member acts like a dictionary; therefore, we can access its keys via the `keys` method and its values via the `[]` operator. In lines 31–32, we print a new row in the table for each item returned by method `os.environ.keys`. This row contains the key and the key's value. Notice that we pass each environment variable to function `cgi.escape`. This function formats text in an "XHTML-safe" way—special XHTML characters such as < and & are formatted so that they appear in the document as they should. After we have printed all the environment variables, we close the `table`, `body` and `html` tags (line 36).

35.7 Form Processing and Business Logic

XHTML forms allow users to enter data to be sent to a Web server for processing. Once the server receives the form input, a server program processes the data. Such a program has many uses. For example, it could help people purchase products, send and receive Web-based e-mail, or complete a survey. These types of Web applications allow users to interact with the server. Figure 35.17 uses an XHTML `form` to allow users to input personal information for a mailing list. This type of registration might be used to store user information in a database. [*Note*: The file `fig35_17.html` should be placed in your Web server's document root directory (e.g., `C:\Program Files\Apache Group\Apache2\htdocs`). View the file by loading `http://localhost/fig35_17.html` in your browser.]

```
1  <!DOCTYPE html PUBLIC
2     "-//W3C//DTD XHTML 1.0 Transitional//EN"
3     "DTD/xhtml1-transitional.dtd">
4  <!-- Fig. 35.17: fig35_17.html -->
5
6  <html xmlns = "http://www.w3.org/1999/xhtml" xml:lang="en"
7     lang="en">
8  <head>
9     <title>Sample FORM to take user input in HTML</title>
10 </head>
11
12 <body style = "font-family: Arial, sans-serif; font-size: 11pt">
13
14    <div style = "font-size: 15pt; font-weight: bold">
15       This is a sample registration form.
16    </div>
17       Please fill in all fields and click Register.
18
19    <form method = "post" action = "/cgi-bin/fig35_18.py">
```

Fig. 35.17 XHTML form to collect information from user. (Part 1 of 3.)

```
20          <img src = "images/user.gif" alt = "user" /><br />
21          <div style = "color: blue">
22             Please fill out the fields below.<br />
23          </div>
24
25          <img src = "images/fname.gif" alt = "firstname" />
26          <input type = "text" name = "firstname" /><br />
27          <img src = "images/lname.gif" alt = "lastname" />
28          <input type = "text" name = "lastname" /><br />
29          <img src = "images/email.gif" alt = "email" />
30          <input type = "text" name = "email" /><br />
31          <img src = "images/phone.gif" alt = "phone" />
32          <input type = "text" name = "phone" /><br />
33
34          <div style = "font-size: 8pt">
35             Must be in the form (555)555-5555<br/><br/>
36          </div>
37
38          <img src = "images/downloads.gif" alt = "downloads" /><br />
39          <div style = "color: blue">
40             Which book would you like information about?<br />
41          </div>
42
43          <select name = "book">
44             <option>XML How to Program</option>
45             <option>Python How to Program</option>
46             <option>E-business and E-commerce How to Program</option>
47             <option>Internet and WWW How to Program 3e</option>
48             <option>C++ How to Program 4e</option>
49             <option>Java How to Program 5e</option>
50             <option>Visual Basic How to Program</option>
51          </select>
52          <br /><br />
53
54          <img src = "images/os.gif" alt = "os" /><br />
55          <div style = "color: blue">
56             Which operating system are you
57             currently using?<br />
58          </div>
59
60          <input type = "radio" name = "os" value = "Windows XP"
61          checked = "checked" />
62          Windows XP
63          <input type = "radio" name = "os" value = "Windows 2000" />
64          Windows 2000
65          <input type = "radio" name = "os" value = "Windows 95_98" />
66          Windows 95/98/ME<br />
67          <input type = "radio" name = "os" value = "Linux" />
68          Linux
69          <input type = "radio" name = "os" value = "Other" />
70          Other<br />
71          <input type = "submit" value = "Register" />
72
```

Fig. 35.17 XHTML form to collect information from user. (Part 2 of 3.)

```
73       </form>
74    </body>
75    </html>
```

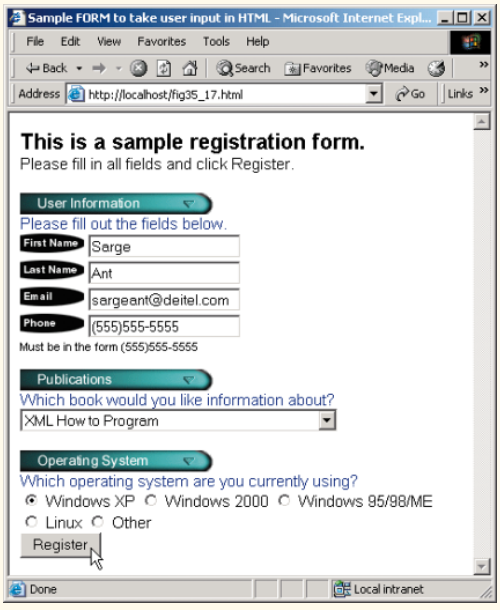


Fig. 35.17 XHTML form to collect information from user. (Part 3 of 3.)

The `form` element (line 19) specifies how the information enclosed by tags `<form>` and `</form>` should be handled. The first attribute, `method = "post"`, directs the browser to send the form's information to the server. The second attribute, `action = "/cgi-bin/fig35_18.py"`, directs the server to execute the `fig35_18.py` Python script, located in the `cgi-bin` directory. The `name`s given to the input items (e.g., `firstname`) in the Web page are important when the Python script is executed on the server. These `name`s allow the script to refer to the individual pieces of data the user submits. When the user clicks the button labeled **Register**, both the input items and the names given to the items are sent to the `fig35_18.py` Python script.

Figure 35.18 takes user information from `fig35_17.html` and sends a Web page to the client indicating that the information was received. Line 6 `import`s the `cgi` module, which provides functionality for writing CGI scripts in Python, including access to XHTML form values.

```
1  #!c:\Python\python.exe
2  # Fig. 35.18: fig35_18.py
3  # Program to read information sent to the server from the
4  # form in the form.html document.
5
6  import cgi
```

Fig. 35.18 Python program to process XHTML form data from user. (Part 1 of 4.)

```
7  import re
8
9  # the regular expression for matching most US phone numbers
10 telephoneExpression = \
11    re.compile( r'^\(\d{3}\)\d{3}-\d{4}$' )
12
13 def printContent():
14    print "Content-type: text/html"
15    print
16    print """
17 <html xmlns = "http://www.w3.org/1999/xhtml" xml:lang="en"
18    lang="en">
19    <head><title>Registration results</title></head>
20       <body>"""
21
22 def printReply():
23    print """
24       Hi <span style = "color: blue; font-weight: bold">
25       %(firstName)s</span>.
26       Thank you for completing the survey.<br />
27       You have been added to the <span style = "color: blue;
28       font-weight: bold">%(book)s </span> mailing list.<br /><br />
29
30       <span style = "font-weight: bold">
31       The following information has been saved in our database:
32       </span><br />
33
34       <table style = "border: 0; border-width: 0;
35          border-spacing: 10">
36       <tr><td style = "background-color: yellow">Name </td>
37           <td style = "background-color: yellow">Email</td>
38           <td style = "background-color: yellow">Phone</td>
39           <td style = "background-color: yellow">OS</td></tr>
40
41       <tr><td>%(firstName)s %(lastName)s</td><td>%(email)s</td>
42           <td>%(phone)s</td><td>%(os)s</td></tr>
43       </table>
44
45       <br /><br /><br />
46
47       <div style = "text-align: center; font-size: 8pt">
48       This is only a sample form.
49       You have not been added to a mailing list.
50       </div></center></body></html>
51       """ % personInfo
52
53 def printPhoneError():
54
55    print """<span style = "color: red; font-size 15pt">
56       INVALID PHONE NUMBER</span><br />
57       A valid phone number must be in the form
58       <span style = "font-weight: bold">(555)555-5555</span>
59       <span style = "color: blue"> Click the Back button,
```

Fig. 35.18 Python program to process XHTML form data from user. (Part 2 of 4.)

```
60          enter a valid phone number and resubmit.</span><br /><br />
61          Thank You.</body></html>"""
62
63    def printFormError():
64
65       print """<span style = "color: red; font-size 15pt">
66          FORM ERROR</span><br />
67          You have not filled in all fields.
68          <span style = "color: blue"> Click the Back button,
69          fill out the form and resubmit.</span><br /><br />
70          Thank You.</body></html>"""
71
72    printContent()
73
74    form = cgi.FieldStorage()
75
76    try:
77       personInfo = { 'firstName' : form[ "firstname" ].value,
78                      'lastName' : form[ "lastname" ].value,
79                      'email' : form[ "email" ].value,
80                      'phone' : form[ "phone" ].value,
81                      'book' : form[ "book" ].value,
82                      'os' : form[ "os" ].value }
83    except KeyError:
84       printFormError()
85
86    if telephoneExpression.match( personInfo[ 'phone' ] ):
87       printReply()
88    else:
89       printPhoneError()
```

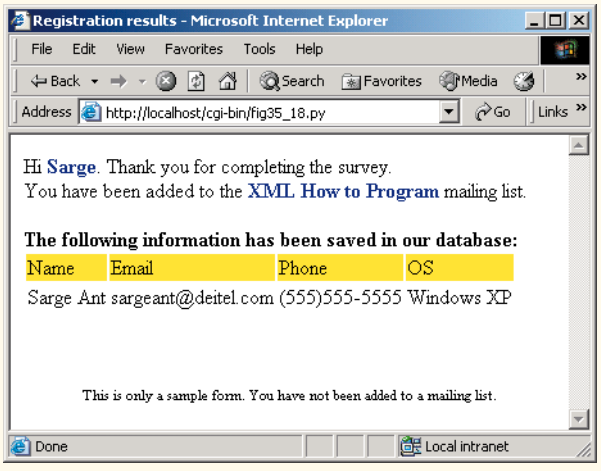


Fig. 35.18 Python program to process XHTML form data from user. (Part 3 of 4.)

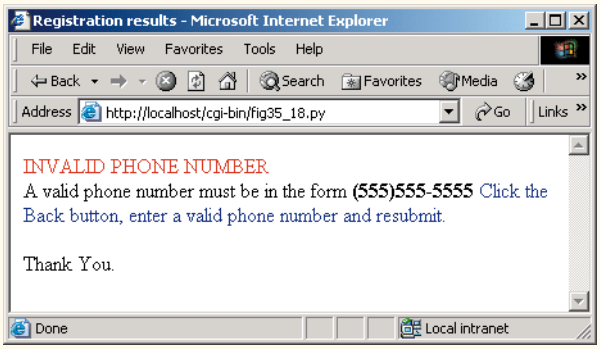


Fig. 35.18 Python program to process XHTML form data from user. (Part 4 of 4.)

Line 72 begins the main portion of the script and calls function `printContent` to print the proper HTTP header and XHTML `DOCTYPE` string. Line 74 creates an instance of class `FieldStorage` and assigns the instance to variable `form`. This class contains information about any posted forms. The `try` block (lines 76–82) creates a dictionary that contains the appropriate values from each defined element in `form`. Each value is accessed via the `value` **data member** of a particular `form` element. For example, line 78 assigns the `value` of the `lastName` field of `form` to the dictionary key `'lastName'`.

If the value of any element in `form` is `None`, the `try` block raises a `KeyError` exception, and we call function `printFormError`. This function (lines 63–70) prints a message in the browser that tells the user the form has not been completed properly and instructs the user to click the **Back** button to fill out the form and resubmit it.

Line 86 tests the user-submitted phone number against the regular expression `telephoneExpression` (compiled in lines 10–11). If the expression's `match` method does not return `None`, we call the `printReply` function (discussed momentarily). If the `match` method does return `None` (i.e., the phone number is not in the proper format), we call function `printPhoneError`. This function (lines 53–61) displays a message in the browser that informs the user that the phone number is in improper format and instructs the user to click the **Back** button to change the phone number and resubmit the form.

If the user has filled out the form correctly, we call function `printReply` (lines 22–51). This function thanks the user and displays an XHTML `table` with the information gathered from the form. Note that we format the output with values from the `personInfo` dictionary. For example, the beginning of line 25

```
%(firstName)s
```

inserts the value of the string variable `firstName` into the string after the percent sign (%). Line 51 informs Python that the string variable `firstName` is a key in the dictionary `personInfo`. Thus, the text at the beginning of line 25 is replaced with the value stored in `personInfo['firstName']`.

35.8 Cookies

When a client visits a Web site, the server for that Web site may **write a cookie** to the client's machine. This cookie can be accessed by servers within the Web site's domain at a later time. Cookies are usually small text files used to maintain **state information** for a particular client. State information may contain a username, password or specific information that might be helpful when a user returns to a Web site. Many Web sites use cookies to store a client's postal zip code. The zip code is used when the client requests a Web page from the server. The server might send the current weather information or news updates for the client's region. The scripts in this section write cookie values to the client and retrieve the values for display in the browser.

Figure 35.19 is an XHTML `form` that asks the user to enter a name, height and favorite color values. These values are passed to the `fig35_20.py` script, which writes the values in a client-side cookie. [*Note*: Be sure to place the file `fig35_19.html` in your Web server's document root directory. View the file by loading `http://localhost/fig35_19.html` in your browser.]

```
1  <!DOCTYPE html PUBLIC
2     "-//W3C//DTD XHTML 1.0 Transitional//EN"
3     "DTD/xhtml1-transitional.dtd">
4  <!-- Fig. 35.19: fig35_19.html -->
5
6  <html xmlns = "http://www.w3.org/1999/xhtml" xml:lang = "en"
7     lang = "en">
8     <head>
9        <title>Writing a cookie to the client computer</title>
10    </head>
11
12 <body style = "background-image: images/back.gif;
13    font-family: Arial,sans-serif; font-size: 11pt" >
14
15    <span style = "font-size: 15pt; font-weight: bold">
16       Click Write Cookie to save your cookie data.
17    </span><br />
18
19    <form method = "post" action = "/cgi-bin/fig35_20.py">
20       <span style = "font-weight: bold">Name:</span><br />
21       <input type = "text" name = "name" /><br />
22       <span style = "font-weight: bold">Height:</span><br />
23       <input type = "text" name = "height" /><br />
24       <span style = "font-weight: bold">Favorite Color</span><br />
25       <input type = "text" name = "color" /><br />
26       <input type = "submit" value = "Write Cookie" />
27    </form>
28
29 </body>
30 </html>
```

Fig. 35.19 XHTML form to get cookie values from user. (Part 1 of 2.)

Fig. 35.19 XHTML form to get cookie values from user. (Part 2 of 2.)

Figure 35.20 is the script that retrieves the form values from `fig35_19.html` and stores them in a client-side cookie. Line 6 imports the `Cookie` module. This module provides capabilities for reading and writing client-side cookies.

Lines 9–15 define function `printContent`, which prints the content header and XHTML `DOCTYPE` string to the browser. Line 17 retrieves the form values by using class `FieldStorage` from module `cgi`. We handle the form values with a `try`…`except`…`else` block. The `try` block (lines 19–22) attempts to retrieve the form values. If the user has not completed one or more of the form fields, the code in this block raises a `KeyError` exception. The exception is caught in the `except` block (lines 23–28), and the program calls function `printContent`, then outputs an appropriate message to the browser.

```
1  #!C:\Python\python.exe
2  # Fig. 35.20: fig35_20.py
3  # Writing a cookie to a client's machine
4
5  import cgi
6  import Cookie
7  import time
8
9  def printContent():
10    print "Content-type: text/html"
11    print
12    print """
13 <html xmlns = "http://www.w3.org/1999/xhtml" xml:lang="en"
14    lang="en">
15    <head><title>Cookie values</title></head>"""
16
17 form = cgi.FieldStorage()  # get form information
18
```

Fig. 35.20 Writing a cookie to a client's machine. (Part 1 of 3.)

```
try:  # extract form values
   name = form[ "name" ].value
   height = form[ "height" ].value
   color = form[ "color" ].value
except KeyError:
   printContent()
   print """<body><h3>You have not filled in all fields.
   <span style = "color: blue"> Click the Back button,
   fill out the form and resubmit.<br /><br />
   Thank You. </span></h3>"""
else:

   # construct cookie expiration date and path
   expirationFormat = "%A, %d-%b-%y %X %Z"
   expirationTime = time.localtime( time.time() + 300 )
   expirationDate = time.strftime( expirationFormat,
      expirationTime )
   path = "/"

   # construct cookie contents
   cookie = Cookie.SimpleCookie()

   cookie[ "Name" ] = name
   cookie[ "Name" ][ "expires" ] = expirationDate
   cookie[ "Name" ][ "path" ] = path

   cookie[ "Height" ] = height
   cookie[ "Height" ][ "expires" ] = expirationDate
   cookie[ "Height" ][ "path" ] = path

   cookie[ "Color" ] = color
   cookie[ "Color" ][ "expires" ] = expirationDate
   cookie[ "Color" ][ "path" ] = path

   # print cookie to user and page to browser
   print cookie

   printContent()
   print """<body style = "background-image: /images/back.gif;
   font-family: Arial,sans-serif; font-size: 11pt">
   The cookie has been set with the following data: <br /><br />

   <span style = "color: blue">Name:</span> %s<br />
   <span style = "color: blue">Height:</span> %s<br />
   <span style = "color: blue">Favorite Color:</span>
   <span style = "color: %s"> %s</span><br />""" \
   % ( name, height, color, color )

   print """<br /><a href= "fig35_21.py">
   Read cookie values</a>"""

print """</body></html>"""
```

Fig. 35.20 Writing a cookie to a client's machine. (Part 2 of 3.)

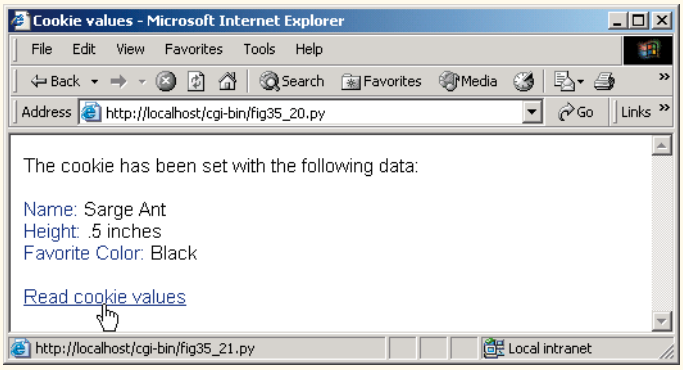


Fig. 35.20 Writing a cookie to a client's machine. (Part 3 of 3.)

The code in the `else` block (lines 29–68) executes after the program successfully retrieves all the form values. Line 32 specifies the format for the expiration value of the cookie. The format characters in this string are defined by the `time` module. For a complete list of `time` tokens and their meanings, visit

```
www.python.org/doc/current/lib/module-time.html
```

The `time` function (line 33) of module `time` returns a floating-point value that is the number of seconds since the **epoch** (i.e., January 1, 1970). We add 300 seconds to this value to set the `expirationTime` for the cookie. We then format the time using the `localtime` function. This function converts the time in seconds to a nine-element tuple that represents the time in local terms (i.e., according to the time zone of the machine on which the script is running). Lines 34–35 call the `strftime` function to format a time tuple into a string. This line effectively formats tuple `expirationTime` as a string that follows the format specified in `expirationFormat`.

Line 39 creates an instance of class `Cookie`. An object of class `Cookie` acts like a dictionary, so values can be set and retrieved using familiar dictionary syntax. Lines 41–51 set the values for the cookie, based on the user-entered values retrieved from the XHMTL form.

Line 54 writes the cookie to the browser (assuming the user's browser has enabled cookies) by using the `print` statement. The cookie must be written before we write the content type (line 56) to the browser. Lines 57–65 display the cookie's values in the browser. We then conclude the `else` block by creating a link to a Python script that retrieves the stored cookie values (lines 67–68).

Figure 35.21 is the CGI script that retrieves cookie values from the client and displays them in the browser. Line 18 creates an instance of class `Cookie`. Line 19 retrieves the cookie values from the client. Cookies are stored as a string in the environment variable `HTTP_COOKIE`. The **`load` method** of class `Cookie` extracts cookie values from a string. If no cookie value exists, then the program raises a `KeyError` exception. We catch the exception in lines 20–22 and print an appropriate message in the browser.

```
#!C:\Python\python.exe
# Fig. 35.21: fig35_21.py
# Program that retrieves and displays client-side cookie values

import Cookie
import os

print "Content-type: text/html"
print
print """
<html xmlns = "http://www.w3.org/1999/xhtml" xml:lang="en"
   lang="en">
   <head><title>Cookie values</title></head>
   <body style =
   font-family: Arial, sans-serif; font-size: 11pt">"""

try:
   cookie = Cookie.SimpleCookie()
   cookie.load( os.environ[ "HTTP_COOKIE" ] )
except KeyError:
   print """<span style = "font-weight: bold">Error reading cookies
   </span>"""
else:
   print """<span style = "font-weight: bold">
   The following data is saved in a cookie on your computer.
   </span><br /><br />"""

   print """<table style = "border-width: 5; border-spacing: 0;
      padding: 10">"""

   for item in cookie.keys():
      print """<tr>
         <td style = "background-color: lavender">%s</td>
         <td style = "background-color: white">%s</td>
      </tr>""" % ( item, cookie[ item ].value )

   print """</table>"""

print """</body></html>"""
```

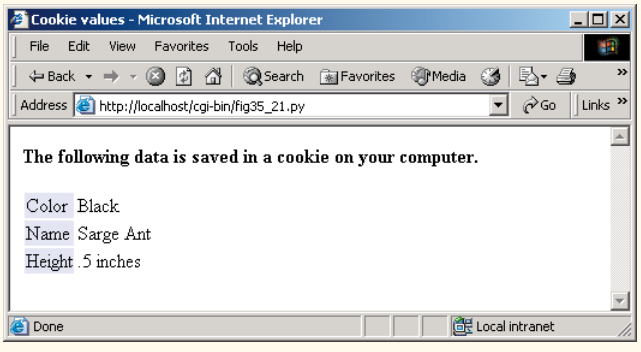


Fig. 35.21 Python CGI script that retrieves and displays client-side cookie values.

If the program successfully retrieves the cookie values, the code in lines 23–37 displays the values in the browser. Cookies act like dictionaries. This means that we can use the `keys` method (line 31) to retrieve the names of all the values in the cookie. Lines 32–35 print these names and their corresponding values in a table.

35.9 Database Application Programming Interface (DB-API)

Python's open-source nature encourages independent developers to contribute additions to the language. In earlier versions of Python, many developers contributed modules that provided interfaces to several databases. Unfortunately, these interfaces rarely resembled one another; if an application developer wanted to change the application's database, the whole program had to be rewritten.

The **Python Database Special Interest Group (SIG)** was formed to develop a specification for **Python database application programming interface (DB-API)**. The specification is now in version 2.0, and modules that conform to this specification exist for many databases. In this section, we illustrate the Python interface to MySQL (**module `MySQLdb`**).

35.9.1 Setup

The next programming example assumes that the user has installed MySQL and the `MySQLdb` module. Please see Chapter 22 for information about MySQL. The `MySQLdb` module must be downloaded from

```
sourceforge.net/projects/mysql-python/
```

and installed on your system. Be sure to download the version of the module appropriate for your version of Python. Run the downloaded file and follow the on-screen instructions to install the module. Next, copy the `books` directory from the Chapter 35 examples directory on the CD to the MySQL data directory (e.g., `C:\mysql\data`).

35.9.2 Simple DB-API Program

The example in this section lets the user choose an author from an XHTML drop-down list. The user then clicks a button to query the database. The database query returns a list of all books by that author.

Figure 35.22 is a CGI script that creates the XHTML author-selection list by querying the database. Line 4 imports the `MySQLdb` module. This provides access to a MySQL database using the Python DB-API.

```
1  #!c:\Python\python.exe
2  # Fig. 35.22: fig35_22.py
3  # A program to illustrate Python's database connectivity.
4  import MySQLdb
5
6  print "Content-type: text/html"
```

Fig. 35.22 Python CGI script to access a MySQL database and create a list of authors. (Part 1 of 2.)

```
print
print """
<html xmlns = "http://www.w3.org/1999/xhtml" xml:lang="en"
   lang="en">
   <head><title>Select Author</title></head>
   <body style =
   font-family: Arial, sans-serif; font-size: 11pt">"""

try:
   connection = MySQLdb.connect( db = "books" )
except OperationalError:
   print "Unable to connect to database: %s" % message
else:
   cursor = connection.cursor()
   cursor.execute( "SELECT * FROM Authors" )
   authorList = cursor.fetchall()

   cursor.close()                # close cursor
   connection.close()            # close connection

   print """
   <form method = "post" action = "/cgi-bin/fig35_23.py">
      <select name = "authorID">"""

   for author in authorList:
      print """<option value = %d>%s, %s</option>"""  \
            % ( author[ 0 ], author[ 2 ], author[ 1 ] )

   print """
      </select>
      <input type = "submit" value = "Execute Query" />
   </ form>"""

print """</body></html>"""
```

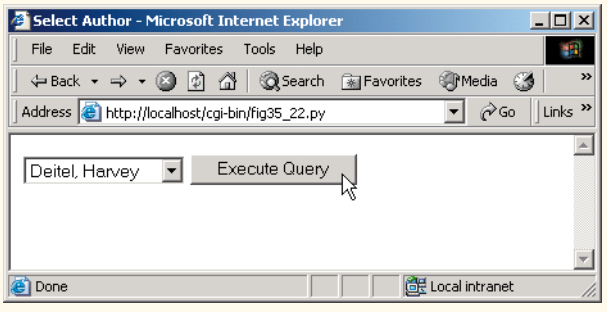


Fig. 35.22 Python CGI script to access a MySQL database and create a list of authors. (Part 2 of 2.)

Lines 6–13 print the HTTP header and XHTML `DOCTYPE` string to the browser. The remainder of the program is contained in a `try…except…else` block. In the `try` block (lines 15–16), we attempt to connect to the MySQL database called `books`. Line 16 connects to the database. The call to `connect` in module `MySQLdb` returns an instance of a `Connection` object. In the call to `connect`, we pass the value `"books"` to the **keyword argument** `db`. A keyword argument is a named argument defined by a function. To pass a value to a named argument, we assign a value to the name inside the function call's parentheses, as in line 16. The `Connection` object returned by the call is stored in local variable `connection`.

If the call to `MySQLdb.connect` succeeds, we have connected to the database. If the call does not succeed, the program receives an `OperationalError` exception. We catch this exception in lines 17–18, where we print an appropriate error message.

If the program does not encounter an `OperationalError` exception, we execute the code in the `else` block (lines 19–38). Line 20 calls the `cursor` method of object `connection`. This method returns a `Cursor` object that allows us to execute queries against the database. We store this object in local variable `cursor`.

Line 21 calls method `execute` to execute an SQL query against the database. The `execute` method takes a valid SQL string as an argument and runs it against the database. The results of the query are stored in object `cursor`. We retrieve the results by calling method `fetchall` (line 22). This method returns a list of all the records that matched the query string we passed to the `execute` call. In our example, method `fetchall` returns a list of all the records from the **Authors** table in the **books** database. We store this list in local variable `authorList`. In lines 24–25, we close the cursor and the connection by calling their respective `close` methods.

Good Programming Practice 35.3

The Python DB-API implementation automatically closes a connection to a database when the program exits; still, we include this code as a matter of good practice.

The remainder of the `else` block (lines 29–38) writes the XHTML form that lets the user choose an author and query the database. The `form` is posted to `fig35_23.py`, which queries the database for the user-selected author. Lines 27–29 create the XHTML `select` item from which the user will choose an author, named `authorID`. Lines 31–33 contain a `for` loop that creates an `option` for each author in the database. Each record in `authorList` is a tuple with the following format:

```
(authorID, firstName, lastName, birthYear)
```

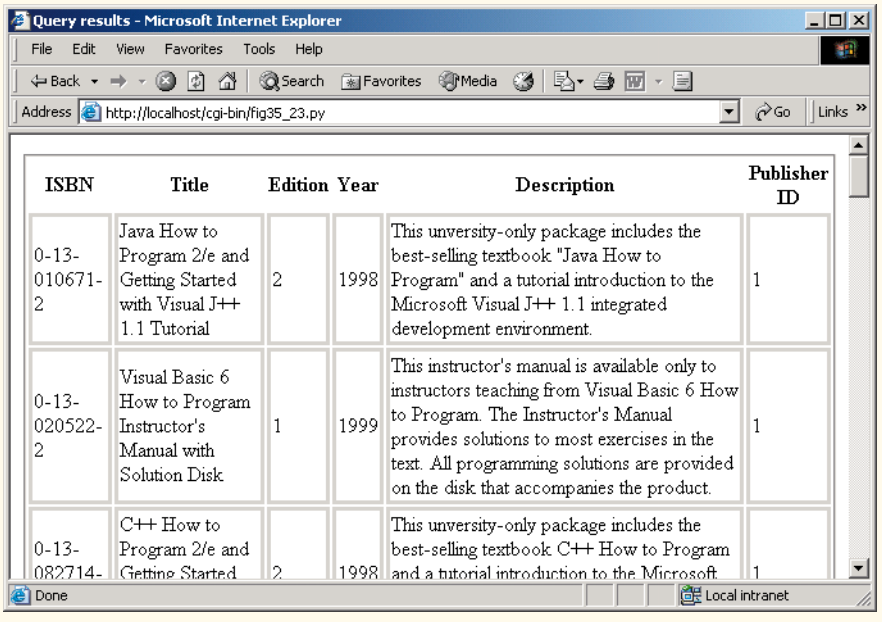
We construct each `option` by assigning a `value` that corresponds to the ID (`author[0]`) and displaying the last name followed by the first name (`author[2]` and `author[1]`, respectively). Lines 35–37 complete the `select` item and add a button to the form so the that user can execute the query.

Figure 35.23 is the CGI script that executes a query based on the author chosen from the form in `fig35_22.html`. Line 10 retrieves the form using the `FieldStorage` class from module `cgi`. Lines 21–30 contain a `try…except` block that attempts to retrieve the `authorID` selected by the user. If the form contains a value for `authorID`, we store it in local variable `authorID`; otherwise, we print an error message to the browser (lines 24–29). Line 30 calls function `sys.exit`, causing the program to terminate.

```
#!c:\Python\python.exe
# Fig. 35.23: fig35_23.py
# A program to illustrate Python's database connectivity.

import cgi
import MySQLdb
import sys

# get results from form
form = cgi.FieldStorage()

print "Content-type: text/html"
print
print """
<html xmlns = "http://www.w3.org/1999/xhtml" xml:lang="en"
   lang="en">
   <head><title>Query results</title></head>
   <body style =
   font-family: Arial, sans-serif; font-size: 11pt">"""

try:
   authorID = form[ "authorID" ].value
except KeyError:
   print """<span style = "color: red size = 15pt">
   FORM ERROR</span><br />
   You did not select an author.<br />
   <span style = "color: blue"> Click the Back button,
   fill out the form and resubmit.<br /><br />
   Thank You.</span></body></html>"""
   sys.exit()

# connect to database and get cursor
try:
   connection = MySQLdb.connect( db = 'books' )
except OperationalError:
   print """<span style = "color: red size = 15pt">
   DATABASE ERROR</span><br /> Unable to connect to database.
   </body></html>"""
   sys.exit()

queryString = """select Titles.* from Titles, AuthorISBN
              where AuthorISBN.AuthorID=%s and
              Titles.ISBN=AuthorISBN.ISBN""" % authorID

cursor = connection.cursor()
cursor.execute( queryString )

results = cursor.fetchall()

cursor.close()                  # close cursor
connection.close()              # close connection

```

Fig. 35.23 Python CGI script to access a MySQL database and create a table of titles, given an author. (Part 1 of 2.)

```
53  # display results
54  print """<table style = "border: groove 2 pt;
55             border-colapse: separate">
56           <tr>
57              <th>ISBN</th>
58              <th>Title</th>
59              <th>Edition</th>
60              <th>Year</th>
61              <th>Description</th>
62              <th>Publisher ID</th>
63           </tr>"""
64
65  for row in results:
66     print "<tr>"
67
68     for entry in row:
69        print '<td style = "border: solid 2pt">%s</td>' % entry
70
71     print "</tr>"
72
73  print """</table></body></html>"""
```



ISBN	Title	Edition	Year	Description	Publisher ID
0-13-010671-2	Java How to Program 2/e and Getting Started with Visual J++ 1.1 Tutorial	2	1998	This university-only package includes the best-selling textbook "Java How to Program" and a tutorial introduction to the Microsoft Visual J++ 1.1 integrated development environment.	1
0-13-020522-2	Visual Basic 6 How to Program Instructor's Manual with Solution Disk	1	1999	This instructor's manual is available only to instructors teaching from Visual Basic 6 How to Program. The Instructor's Manual provides solutions to most exercises in the text. All programming solutions are provided on the disk that accompanies the product.	1
0-13-082714-	C++ How to Program 2/e and Getting Started	2	1998	This university-only package includes the best-selling textbook C++ How to Program and a tutorial introduction to the Microsoft	1

Fig. 35.23 Python CGI script to access a MySQL database and create a table of titles, given an author. (Part 2 of 2.)

We attempt to connect to the MySQL database called **books** in lines 33–39. If we are unable to obtain a connection, we print an error message and call `sys.exit` to exit the program (lines 36–39).

Lines 41–43 construct a query string to execute against the database. This query selects all the columns from table **Title** where the ISBN matches all the ISBNs from table **Autho-**

rISBN that correspond to the `authorID` specified in the form. Lines 45–46 create a cursor for the database and execute the query string against the database. We retrieve the results of the query using method `fetchall` and store the records in local variable `results` (line 48). We then close the cursor and the connection (lines 50–51).

The remainder of the program (lines 54–73) displays the results of the query. We create a table and label the headers with the column names from the database (lines 54–63). Line 65 begins a `for` loop that iterates over each record in local variable `results`. For each record, we create a row in the table (lines 66–71). Each column value has a corresponding entry in the row (lines 68–69). After we have printed all the records, we print a closing table tag (line 73).

In this section, we have illustrated Python's DB-API through a specific implementation of the DBI, module `MySQLdb`. Because `MySQLdb` conforms to the DB-API, the code in our examples would not require many changes to work with another module that conforms to the DB-API. In fact, we could use many other databases, such as Microsoft Access or Informix, because their respective modules (`odbc` and `informixdb`) conform to the DB-API.

35.10 Operator Precedence Chart

This section contains the operator precedence chart for Python (Fig. 35.24). The operators are shown in decreasing order of precedence, from top to bottom.

Operator	Type	Associativity
`' '`	string conversion	left to right
`{ }`	dictionary creation	left to right
`[ ]`	list creation	left to right
`( )`	tuple creation or expression grouping	left to right
`( )`	function call	left to right
`[ : ]`	slicing	left to right
`[ ]`	subscript access	left to right
`.`	member access	left to right
`**`	exponentiation	right to left
`~`	bitwise NOT	left to right
`+` `-`	unary plus unary minus	right to left

`*`
`/`
`%`
	multiplication
	division
	modulus (remainder)
		left to right

`+` `-`	addition subtraction	left to right
`<<` `>>`	left shift right shift	left to right

Fig. 35.24 Python operator precedence chart. (Part 1 of 2.)

Operator	Type	Associativity
&	bitwise AND	left to right
^	bitwise XOR	left to right
|	bitwise OR	left to right
< <= > >= != ==	less than less than or equal greater than greater than or equal not equal equal	left to right
`is`, `is not`	identity	left to right
`in`, `not in`	membership tests	left to right
`not`	boolean NOT	left to right
`and`	boolean AND	left to right
`or`	boolean OR	left to right
`lambda`	lambda expressions (anonymous functions)	left to right

Fig. 35.24 Python operator precedence chart. (Part 2 of 2.)

35.11 Web Resources

`www.python.org`
Python home page. From this site, you can download the latest version of Python for all platforms. The site also posts all the Python documentation and provides links to other resources, such as additional modules, tutorials, search engines, special-interest groups, an event calendar, a job board, mailing lists and archives.

`www.activestate.com/Products/ActivePython`
ActiveState's free distribution of the Python language.

`www.zope.com`
Home page for Zope Corporation, the developers of Zope—a Web application server written in Python.

`www.zope.org`
Home page for Zope and its community.

`starship.python.net`
Provides resources for Python developers. Site members post Python modules and utilities on this site.

`www.python.org/download/download_mac.html`
Provides information on and links to a MacOS version of Python.

`www.vex.net/parnassus`
Contains links to many third-party Python modules, which are freely available for download.

`www.pythonware.com`
Secret Labs AB is a company that offers application-development tools for Python. The Pythonware Web site provides links to Secret Labs AB products and other Python resources.

`starship.python.net/crew/davem/cgifaq/faqw.cgi`
A Python/CGI FAQ.

`www.devshed.com/Server_Side/Python/CGI`
An article/tutorial on writing CGI programs in Python.

`starship.python.net/crew/aaron_watters/pws.html`
Provides instructions for configuring IIS/PWS for Python / CGI scripts.

`www.python.org/doc/howto/regex/regex.html`
Contains a tutorial on using Python regular expressions.

`www.python.org/windows/win32com`
Contains resources for Python/COM development.

`gadfly.sourceforge.net`
Home page for Gadfly, a relational database written in Python.

`aspn.activestate.com/ASPN/Python/Cookbook`
Contains many Python examples to accomplish a variety of tasks.

`www.python.org/windows/win32/odbc.html`
An introduction to Python's `odbc` module can be found at this site.

`starship.python.net/crew/bwilk/access.html`
Contains a few notes on using Python and Microsoft Access.

`www.python.org/doc/Comparisons.html`
Guido van Rossum has posted an essay on this page that compares Python with other popular languages, such as Java, C++ and Perl.

`www.vic.auug.org.au/auugvic/av_paper_python.html`
An overview of Python and lists many uses and features of the language.

`www.networkcomputing.com/unixworld/tutorial/005/005.html`
Contains a tutorial and an introduction to Python.

SUMMARY

- Python is an interpreted, cross-platform, object-oriented language. It is a freely distributed, open-source technology.
- Using Python's core modules and those available for free on the Web, programmers can develop applications that accomplish a variety of tasks.
- Python's interpreted nature facilitates rapid application development (RAD).
- Comments in Python begin with the # character; Python ignores all text in the current line after this character.
- Python statements can be executed in two ways. The statements can be typed into a file on which Python is then invoked. Python statements can also be interpreted dynamically by typing them in at the Python interactive prompt.
- Python keywords have special meanings in Python and cannot be used as variable names, function names and other objects. A list of Python keywords can be obtained from the `keyword` module.
- The keyword `def` marks the beginning of the function definition. The function's parameter list is followed by a colon (`:`).
- Python is a case-sensitive language.
- Python determines the beginning and end of a statement based on whitespace. Each new line begins a new statement, and groups of statements that belong to the same block of code are indented the same amount.
- Keyword `return` causes the function to return the specified value.
- Python function `raw_input` retrieves input from the program user. This function may optionally take a string argument that is a prompt to the user.

- The Python `int` function converts noninteger data types to integers.
- The backslash character (\) is the line-continuation character. Lines may also be continued freely inside nested parentheses, brackets and braces.
- The `\n` escape code is a special character that represents a newline character.
- Tuples are created as a comma-separated list of values in parentheses (`()`). A tuple can contain any data type (e.g., strings, integers, other tuples) and may contain elements of different types.
- Tuples are immutable—after a tuple is created, an element at a defined index cannot be replaced.
- By default, the `print` statement writes a newline character (e.g., a carriage return) at the end of its output. A comma placed at the end of a `print` statement tells Python to leave out the newline.
- Python lists consist of a sequence of zero or more elements.
- Python lists are mutable—an element at an index that has been defined may be replaced.
- Method `append` adds an element to the end of a list.
- Each entry in a dictionary has two parts—the key and the value—and a dictionary consists of a set of zero or more comma-separated key-value pairs.
- A value in a dictionary is accessed through its key. The key must be unique and of an immutable data type (e.g., number, string or tuple that contains only immutable data types); values may be of any data type.
- A regular expression string defines a pattern against which text data can be compared. Regular-expression processing capability is available in the standard Python module `re`.
- Unless otherwise specified, regular expression characters `*` and `+` match as many occurrences of a regular expression as possible.
- Compiling a regular expression string (using `re` method `compile`) speeds up a regular expression comparison that uses that string.
- Strings can be contained in single quotes (`' '`), double quotes (`" "`) or in a set of three single or double quotes (`''' '''` or `""" """`)
- The `%` format character acts like a placeholder in the string. Python defines several format characters for use in string formatting
- Importing a module enables programmers to use the functions it defines.
- An `r` before a string indicates that it is a raw string. Python handles backslash characters in raw strings differently than in "normal" strings—Python does not interpret backslashes as escape characters in raw strings.
- `re` module's `findall` method returns a list of all the substrings in any given string that match a specified regular expression.
- Exception handling enables programs and programmers to identify an error when it occurs and take appropriate action. Python accomplishes exception handling through the use of `try…except` blocks.
- Any code that causes an error raises an exception. If the code that raises an exception is contained in a `try` block, the corresponding `except` block catches the exception (i.e., handles the error).
- An `except` block can and should specify a specific exception to catch.
- A `try` block may optionally specify a corresponding `else` block. If the code in the `try` block does not raise an exception, the program executes the code in the `else` block. If an exception is raised in the `try` block, the `else` block is skipped.
- The pound-bang (`#!`) directive—the directive that specifies the location of the Python executable—must be the first line in a CGI script.

- The `cgi` module provides functionality for writing CGI scripts in Python, including access to XHTML form values.
- `cgi` method `FieldStorage` provides access to XHTML form values.
- The `Cookie` module provides access to cookies.
- An object of class `Cookie` acts like a dictionary, so values can be set and retrieved using familiar dictionary syntax.
- The `time` function of module `time` returns a floating-point value that is the number of seconds since the "epoch" (i.e., the first day of 1970).
- The `load` method of module `Cookie` extracts cookie values from a string. If no cookie value exists, the program raises the `KeyError` exception.

TERMINOLOGY

' (single quote) character
" (double quote) character
""" (triple quote) characters
comment character
#! (pound-bang) directive
% formatting character
% modulo operator
% operator
%= operator
**= statement
*= statement
, (comma) character
. (dot) operator
. operator
/= statement
: (colon) character
: (slice) operator
[] operator
\ (backslash) character
\n escape character
{} characters
+ operator
+= statement
-= statement
`and`
Apache Web server
`append` method of lists
catch an exception
"chained" expression
`cgi` module
CGI scripts
compiling a regular expression
concatenated strings
`connection` object
constructor
`Cookie` class
`Cookie` module
Ctrl-Z/*Ctrl-D* character
`cursor` object
Database Application Programming Interface
Database Special Interest Group (SIG)
debugging
`def`
dictionary
`else`
`environ` data member of module `os`
epoch
escape character
exception handling
expiration value of a cookie
`fetchall` method of class `cursor`
`FieldStorage` class
`findall` method of module `re`
`float` function
`for`
formatting character
`get` method
greatest common divisor
HTTP header
`HTTP_COOKIE` environment variable
`if`
`if...elif`
`if...else`
immutable data type
`import`
importing a module
indentation of statement
`int` function
interactive mode
key-value pair
`KeyError` exception
`keys`
`keyword` module
list

`load` method of class `Cookie`
`localtime` function of module `time`
`match` method of module `re`
`min` function
module
mutable data type
`MySQLdb` module
newline
`None`
`odbc` module
open-source technology
`os` module
Python prompt
query
raise an exception
`range`
rapid-application development (RAD)
raw string
`raw_input`
`re` module
`re.I` flag
regular expression
`return`
`search` method of module `re`
`self` parameter
`SRE_Match` object
`SRE_Pattern` object
`strftime` function of module `time`
string formatting
string manipulation
string processing
Structured Query Language (SQL)
Tcl/Tk
`time` function
`time` module
`time` token
triple-quoted string

SELF-REVIEW EXERCISES

35.1 Fill in the blanks in each of the following statements:

a) Comments in Python begin with the _____ character.
b) Python statements can be executed in two ways. The statements can be typed into a file and then _____, or they can be _____.
c) The keyword _____ marks the beginning of a Python function definition.
d) Function `raw_input` returns a(n) _____.
e) Python defines three data types for storing complex data: _____, _____ and _____.
f) Tuples are _____ (element values at defined indices may not be changed); whereas lists are _____ (element values at defined indices may be changed).
g) Python implements _____ through the use of `try`…`except` blocks.
h) The Python module used to obtain XHTML form contents is _____.
i) Cookies are stored in the environment variable _____.
j) The _____ was formed to develop a specification for Python database application-programming interface (DB-API).

35.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Python is an interpreted language.
b) To exit the Python interactive interpreter, type `exit` at the Python prompt.
c) Forgetting to indent the next line after a colon is a style error.
d) The underscore character (_) marks the continuation of a Python statement onto the next line.
e) Elements must be added to a list by calling list method `append`.
f) A tuple is a valid data type for use as a dictionary key.
g) The pound-bang (`#!`) directive—which tells a server where to find the Python executable—must be the first line in a CGI script.
h) An object of class `Cookie` acts like a dictionary, so values can be set and retrieved using familiar dictionary syntax.

i) The syntax needed to manipulate a database is always dependent on that database.
j) A `Cursor` object is needed to execute a query against a database (for DB-API compliant modules).

35.3 How can a Python CGI script determine a client's IP address?

35.4 For each of the following code examples, identify and correct the error(s):

a)
```
print hello
```
b)
```
aTuple = ( 1, 2 )
aTuple[ 0 ] = 2
```
c)
```
if 0 < 3
   print "0 is less than 3."
```
d)
```
for counter in range( 10 ):
    print counter
```

35.5 Write a one- to three-line block of code for each of the following tasks:

a) Create a string with 50 exclamation points (`!`) using the `*` operator.
b) Print out even numbers from 0 to 100.
c) Convert a user-entered number from a string to an integer.
d) Determine whether a user-entered integer is odd.
e) Concatenate an empty tuple and a singleton with the += statement.

ANSWERS TO SELF-REVIEW EXERCISES

35.1 a) pound (#). b) executed, dynamically interpreted in an interactive session. c) `def`. d) string. e) tuples, lists, dictionaries. f) immutable, mutable. g) exception (or error) handling. h) `cgi`. i) `HTTP_COOKIE`. j) Python Database Special Interest Group.

35.2 a) True. b) False. Type *Crtl-Z* in Microsoft Windows or *Ctrl-D* in Linux/UNIX. c) False. Forgetting to indent the next line after a colon is a syntax error. d) False. The backslash character (\) marks the continuation of a Python statement onto the next line. e) False. Lists can also be augmented by calling the `extend` method or the += statement. f) True. g) True. h) True. i) False. Database modules that conform to the DB-API provide similar syntaxes. j) True.

35.3 A client's IP address is contained in the `REMOTE_ADDR` environment variable of the `os` module.

35.4 a) Logical or syntax error. If the desired result is to output the word "hello," the proper code is `print "hello"`. The code in the problem will print the value of variable `hello`, if a variable by that name exists; the code raises an error if the variable does not exist. b) Runtime error. Tuple values cannot be modified in this way. c) Syntax error. A colon (`:`) must follow the `if` statement. d) Syntax error. The line after the `for` statement must be indented.

35.5 a)
```
theString = '!' * 50
```
b)
```
for item in range( 101 ):
   if item % 2 == 0:
      print item
```
c)
```
number = raw_input( "Enter a number" )
integer = int( number )
```
d)
```
number = int( raw_input( "Enter an integer" ) )
 if number % 2 == 1:
   print "The number is odd."
```
e)
```
emptyTuple = ( )
emptyTuple += ( 1, )
```

EXERCISES

35.6 Describe how input from an XHTML `form` is retrieved in a Python program.

35.7 Figure 35.5 defines function `greatestCommonDivisor` that computes the greatest common divisor of two positive integers. Euclid's algorithm is another method of computing the greatest common divisor. The following steps define Euclid's algorithm for computing the greatest common divisor of two positive integers *x* and *y*:

while y > 0
  z = y
  y = x modulo z
  x = z
return x

Write a function `Euclid` that takes two positive integers and computes their greatest common divisor using Euclid's algorithm.

35.8 Modify functions `greatestCommonDivisor` and `Euclid` from Exercise 35.7 so that each function counts the number of modular divisions performed (i.e., the number of times the function uses the % operator). Each function should return a tuple that contains the calculated greatest common divisor and the number of modular divisions performed. Run each function on the pairs of integers in Fig. 35.25, and fill in the rest of the table. Which function takes fewer modular divisions, on average?

Integer pair	Number of modular divisions for `greatestCommonDivisor`	Number of modular divisions for `Euclid`
1, 101	_____	_____
3, 30	_____	_____
45, 1000	_____	_____
13, 91	_____	_____
100, 1000	_____	_____
2,2	_____	_____
777,77	_____	_____
73,12	_____	_____
26,4	_____	_____
99,27	_____	_____
Average:	_____	_____

Fig. 35.25 Comparing functions `greatestCommonDivisor` and `Euclid`.

35.9 Write a Python program named `states.py` that declares a variable `states` with value `"Mississippi Alabama Texas Massachusetts Kansas"`. Using only the techniques discussed in this chapter, write a program that does the following:

a) Search for a word in variable `states` that ends in `xas`. Store this word in element 0 of a list named `statesList`.
b) Search for a word in `states` that begins with `k` and ends in `s`. Perform a case-insensitive comparison. [*Note:* Passing `re.I` as a second parameter to method compile performs a case-insensitive comparison.] Store this word in element 1 of `statesList`.

c) Search for a word in `states` that begins with `M` and ends in `s`. Store this word in element 2 of the list.
d) Search for a word in `states` that ends in `a`. Store this word in element 3 of the list.
e) Search for a word that begins with `M` in `states` at the beginning of the string. Store this word at element 4 of the list.
f) Output the array `statesList` to the screen.

35.10 In Section 35.6, we discussed CGI environment variables. Write a CGI script that displays a user's IP address in the user's browser.

35.11 Write a CGI script that logs a user into a Web site. The user should be presented with a Web page that contains a form into which users enter their login name and password. The form sends the the user-entered information to a Python script. This script checks a database for the user's login name and validates the user's password. If the login name and password are valid, the Python script writes a `"Login successful"` message to the browser; if the login name and/or password are invalid, the Python script writes a `"Login unsuccessful"` message to the browser.