MODULE 4

CHAPTER 1

ORGANIZING FILES

THE SHUTIL MODULE

The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the shutil functions, you will first need to use `import shutil`.

Copying Files and Folders

The shutil module provides functions for copying files, as well as entire folders.

Calling `shutil.copy(source, destination)` will copy the file at the path *source* to the folder at the path *destination*. (Both *source* and *destination* are strings.) If *destination* is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

Enter the following into the interactive shell to see how `shutil.copy()` works:

```
>>> import shutil, os
>>> os.chdir('C:\\')
❶ >>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
❷ >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

The first `shutil.copy()` call copies the file at *C:\spam.txt* to the folder *C:\delicious*. The return value is the path of the newly copied file. Note that since a folder was specified as the destination ❶, the original *spam.txt* filename is used for the new, copied file's filename. The second `shutil.copy()` call ❷ also copies the file at *C:\eggs.txt* to the folder *C:\delicious* but gives the copied file the name *eggs2.txt*.

While `shutil.copy()` will copy a single file, `shutil.copytree()` will copy an entire folder and every folder and file contained in it. Calling `shutil.copytree(source, destination)` will copy the folder at the path *source*, along with all of its files and subfolders, to the folder at the path *destination*. The *source* and *destination* parameters are both strings. The function returns a string of the path of the copied folder.

Enter the following into the interactive shell:

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

The shutil.copytree() call creates a new folder named *bacon_backup* with the same content as the original *bacon* folder. You have now safely backed up your precious, precious bacon.

Moving and Renaming Files and Folders

Calling shutil.move(*source*, *destination*) will move the file or folder at the path *source* to the path *destination* and will return a string of the absolute path of the new location.

If *destination* points to a folder, the *source* file gets moved into *destination* and keeps its current filename. For example, enter the following into the interactive shell:

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named *eggs* already exists in the *C:* directory, this shutil.move() calls says, "Move *C:\bacon.txt* into the folder *C:\eggs*."

If there had been a *bacon.txt* file already in *C:\eggs*, it would have been overwritten. Since it's easy to accidentally overwrite files in this way, you should take some care when using move().

The *destination* path can also specify a filename. In the following example, the *source* file is moved *and* renamed.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

This line says, "Move *C:\bacon.txt* into the folder *C:\eggs*, and while you're at it, rename that *bacon.txt* file to *new_bacon.txt*."

Both of the previous examples worked under the assumption that there was a folder *eggs* in the *C:* directory. But if there is no *eggs* folder, then move() will rename *bacon.txt* to a file named *eggs*.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

Finally, the folders that make up the destination must already exist, or else Python will throw an exception. Enter the following into the interactive shell:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  File "C:\Python34\lib\shutil.py", line 521, in move
    os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'
```

Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, you use the shutil module.

- Calling os.unlink(*path*) will delete the file at *path*.
- Calling os.rmdir(*path*) will delete the folder at *path*. This folder must be empty of any files or folders.
- Calling shutil.rmtree(*path*) will remove the folder at *path*, and all files and folders it contains will also be deleted.

a Python program that was intended to delete files that have the *.txt* file extension but has a typo (highlighted in bold) that causes it to delete *.rxt* files instead:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        os.unlink(filename)
```

If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        #os.unlink(filename)
        print(filename)
```

Now the os.unlink() call is commented, so Python ignores it. Instead, you will print the filename of the file that would have been deleted. Running this version of the program first will show you that you've accidentally told the program to delete *.rxt* files instead of *.txt* files.

Safe Deletes with the send2trash Module

Since Python's built-in shutil.rmtree() function irreversibly deletes files and folders, it can be dangerous to use. A much better way to delete files and folders is with the third-party send2trash module. You can install this module by running pip install send2trash from a Terminal window. (See Appendix A for a more in-depth explanation of how to install third-party modules.)

Using send2trash is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycle bin instead of permanently deleting them. If a bug in your program deletes something with send2trash you didn't intend to delete, you can later restore it from the recycle bin.

After you have installed send2trash, enter the following into the interactive shell:

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a')  # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

Walking a Directory Tree

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go. Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

Let's look at the *C:\delicious* folder with its contents, shown in Figure 9-1.

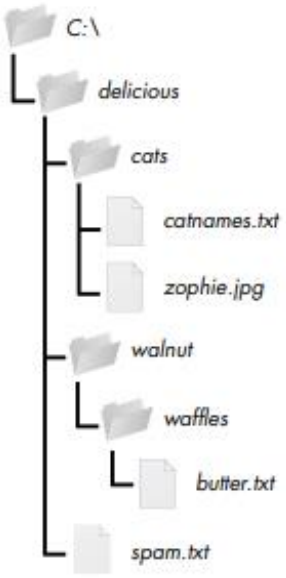

Figure 9-1: An example folder that contains three folders and four files

Here is an example program that uses the `os.walk()` function on the directory tree from Figure 9-1:

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)


    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': '+ filename)

    print('')
```

The `os.walk()` function is passed a single string value: the path of a folder. You can use `os.walk()` in a `for` loop statement to walk a directory tree, much like how you can use the `range()` function to walk over a range of numbers. Unlike `range()`, the `os.walk()` function will return three values on each iteration through the loop:

1. A string of the current folder's name
2. A list of strings of the folders in the current folder
3. A list of strings of the files in the current folder

When you run this program, it will output the following:

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

COMPRESSING FILES WITH THE ZIPFILE MODULE

- Compressing a file reduces its size, which is useful when transferring it over the Internet. And 204 Chapter 9 since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one.
- This single file, called an archive file, can then be, say, attached to an email. Your Python programs can both create and open (or extract) ZIP files using functions in the zipfile module.
- Say you have a ZIP file named example.zip that has the contents shown in Figure 9-2. You can download this ZIP file from http:// nostarch.com/automatestuff/ or just follow along using a ZIP file already on your computer.

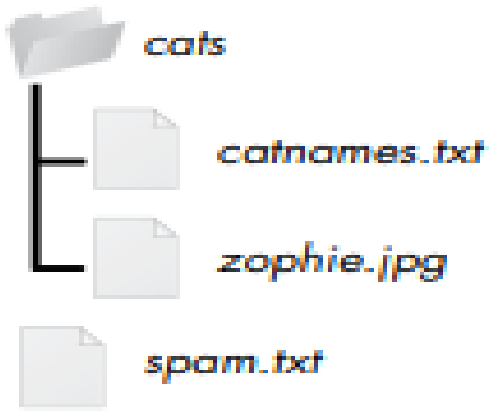

Figure 9-2: The contents of example.zip

Reading ZIP Files

To read the contents of a ZIP file, first you must create a `ZipFile` object (note the capital letters *Z* and *F*). `ZipFile` objects are conceptually similar to the `File` objects you saw returned by the `open()` function in the previous chapter: They are values through which the program interacts with the file. To create a `ZipFile` object, call the `zipfile.ZipFile()` function, passing it a string of the *.zip* file's filename. Note that `zipfile` is the name of the Python module, and `ZipFile()` is the name of the function.

For example, enter the following into the interactive shell:

```
>>> import zipfile, os
>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
❶ >>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

After running this code, the contents of *example.zip* will be extracted to *C:*. Optionally, you can pass a folder name to `extractall()` to have it extract the files into a folder other than the current working directory. If the folder passed to the `extractall()` method does not exist, it will be created. For instance, if you replaced the call at ❶ with `exampleZip.extractall('C:\\delicious')`, the code would extract the files from *example.zip* into a newly created *C:\delicious* folder.

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

Creating and Adding to ZIP Files

To create your own compressed ZIP files, you must open the `ZipFile` object in *write mode* by passing `'w'` as the second argument. (This is similar to opening a text file in write mode by passing `'w'` to the `open()` function.)

When you pass a path to the `write()` method of a `ZipFile` object, Python will compress the file at that path and add it into the ZIP file. The `write()` method's first argument is a string of the filename to add. The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; you can always just set this value to `zipfile.ZIP_DEFLATED`. (This specifies the *deflate* compression algorithm, which works well on all types of data.) Enter the following into the interactive shell:

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

This code will create a new ZIP file named *new.zip* that has the compressed contents of *spam.txt*.

Project: Renaming Files with American-Style Dates to European-Style Dates

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY). This boring task could take all day to do by hand! Let's write a program to do it instead.

Here's what the program does:

- It searches all the filenames in the current working directory for American-style dates.
- When one is found, it renames the file with the month and day swapped to make it European-style.

This means the code will need to do the following:

- Create a regex that can identify the text pattern of American-style dates.
- Call `os.listdir()` to find all the files in the working directory.
- Loop over each filename, using the regex to check whether it has a date.
- If it has a date, rename the file with `shutil.move()`.

For this project, open a new file editor window and save your code as *renameDates.py*.

Step 1: Create a Regex for American-Style Dates

The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates. The to-do comments will remind you what's left to write in this program. Typing them as `TODO` makes them easy to find using IDLE's CTRL-F find feature. Make your code look like the following:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

❶ import shutil, os, re

  # Create a regex that matches files with the American date format.
❷ datePattern = re.compile(r"""^(.*?) # all text before the date
      ((0|1)?\d)-                     # one or two digits for the month
```

```
    ((0|1|2|3)?\d)-                 # one or two digits for the day
    ((19|20)\d\d)                   # four digits for the year
    (.*?)$                          # all text after the date
❸   """, re.VERBOSE)

# TODO: Loop over the files in the working directory.

# TODO: Skip files without a date.

# TODO: Get the different parts of the filename.

# TODO: Form the European-style filename.

# TODO: Get the full, absolute file paths.

# TODO: Rename the files.
```

From this chapter, you know the `shutil.move()` function can be used to rename files: Its arguments are the name of the file to rename and the new filename. Because this function exists in the `shutil` module, you must import that module ❶.

But before renaming the files, you need to identify which files you want to rename. Filenames with dates such as *spam4-4-1984.txt* and *01-03-2014eggs.zip* should be renamed, while filenames without dates such as *littlebrother.epub* can be ignored.

You can use a regular expression to identify this pattern. After importing the `re` module at the top, call `re.compile()` to create a `Regex` object ❷. Passing `re.VERBOSE` for the second argument ❸ will allow whitespace and comments in the regex string to make it more readable.

The regular expression string begins with `^(.*?)` to match any text at the beginning of the filename that might come before the date. The `((0|1)?\d)` group matches the month. The first digit can be either `0` or `1`, so the regex matches `12` for December but also `02` for February. This digit is also optional so that the month can be `04` or `4` for April. The group for the day is `((0|1|2|3)?\d)` and follows similar logic; `3`, `03`, and `31` are all valid numbers for days. (Yes, this regex will accept some invalid dates such as `4-31-2014`, `2-29-2013`, and `0-15-2014`. Dates have a lot of thorny special cases that can be easy to miss. But for simplicity, the regex in this program works well enough.)

While 1885 is a valid year, you can just look for years in the 20th or 21st century. This will keep your program from accidentally matching nondate filenames with a date-like format, such as *10-10-1000.txt*.

The `(.*?)$` part of the regex will match any text that comes after the date.

Step 2: Identify the Date Parts from the Filenames

Next, the program will have to loop over the list of filename strings returned from `os.listdir()` and match them against the regex. Any files that do not

have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables. Fill in the first three TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--

# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)

    # Skip files without a date.
❶   if mo == None:
❷       continue

❸   # Get the different parts of the filename.
    beforePart = mo.group(1)
    monthPart  = mo.group(2)
    dayPart    = mo.group(4)
    yearPart   = mo.group(6)
    afterPart  = mo.group(8)

--snip--
```

If the Match object returned from the search() method is None ❶, then the filename in amerFilename does not match the regular expression. The continue statement ❷ will skip the rest of the loop and move on to the next filename.

Otherwise, the various strings matched in the regular expression groups are stored in variables named beforePart, monthPart, dayPart, yearPart, and afterPart ❸. The strings in these variables will be used to form the European-style filename in the next step.

To keep the group numbers straight, try reading the regex from the beginning and count up each time you encounter an opening parenthesis. Without thinking about the code, just write an outline of the regular expression. This can help you visualize the groups. For example:

```
datePattern = re.compile(r"""^(1) # all text before the date
    (2 (3) )-                     # one or two digits for the month
    (4 (5) )-                     # one or two digits for the day
    (6 (7) )                      # four digits for the year
    (8)$                          # all text after the date
    """, re.VERBOSE)
```

Here, the numbers 1 through 8 represent the groups in the regular expression you wrote. Making an outline of the regular expression, with just the parentheses and group numbers, can give you a clearer understanding of your regex before you move on with the rest of the program.

Step 3: Form the New Filename and Rename the Files

As the final step, concatenate the strings in the variables made in the previous step with the European-style date: The date comes before the month. Fill in the three remaining `TODO`s in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--

    # Form the European-style filename.
❶   euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart +
                   afterPart

    # Get the full, absolute file paths.
    absWorkingDir = os.path.abspath('.')
    amerFilename = os.path.join(absWorkingDir, amerFilename)
    euroFilename = os.path.join(absWorkingDir, euroFilename)

    # Rename the files.
❷   print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
❸   #shutil.move(amerFilename, euroFilename)   # uncomment after testing
```

Store the concatenated string in a variable named `euroFilename` ❶. Then, pass the original filename in `amerFilename` and the new `euroFilename` variable to the `shutil.move()` function to rename the file ❸.

This program has the `shutil.move()` call commented out and instead prints the filenames that will be renamed ❷. Running the program like this first can let you double-check that the files are renamed correctly. Then you can uncomment the `shutil.move()` call and run the program again to actually rename the files.

Ideas for Similar Programs

There are many other reasons why you might want to rename a large number of files.

- To add a prefix to the start of the filename, such as adding *spam_* to rename *eggs.txt* to *spam_eggs.txt*
- To change filenames with European-style dates to American-style dates
- To remove the zeros from files such as *spam0042.txt*

Project: Backing Up a Folder into a ZIP File

Say you're working on a project whose files you keep in a folder named *C:\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file "snapshots" of the entire folder. You'd like to keep different versions, so you want the ZIP file's filename to increment each time it is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*,

AlsPythonBook_3.zip, and so on. You could do this by hand, but it is rather annoying, and you might accidentally misnumber the ZIP files' names. It would be much simpler to run a program that does this boring task for you.

For this project, open a new file editor window and save it as *backupToZip.py*.

Step 1: Figure Out the ZIP File's Name

The code for this program will be placed into a function named `backupToZip()`. This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

❶ import zipfile, os

def backupToZip(folder):
    # Backup the entire contents of "folder" into a ZIP file.

    folder = os.path.abspath(folder)   # make sure folder is absolute

    # Figure out the filename this code should use based on
    # what files already exist.
❷   number = 1
❸   while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

❹   # TODO: Create the ZIP file.

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\delicious')
```

Do the basics first: Add the shebang (`#!`) line, describe what the program does, and import the `zipfile` and `os` modules ❶.

Define a `backupToZip()` function that takes just one parameter, `folder`. This parameter is a string path to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the `folder` folder, and add each of the subfolders and files to the ZIP file. Write `TODO` comments for these steps in the source code to remind yourself to do them later ❹.

The first part, naming the ZIP file, uses the base name of the absolute path of `folder`. If the folder being backed up is *C:\delicious*, the ZIP file's name should be *delicious_N.zip*, where $N = 1$ is the first time you run the program, $N = 2$ is the second time, and so on.

You can determine what *N* should be by checking whether *delicious_1.zip* already exists, then checking whether *delicious_2.zip* already exists, and so on. Use a variable named `number` for *N* ❷, and keep incrementing it inside the loop that calls `os.path.exists()` to check whether the file exists ❸. The first nonexistent filename found will cause the loop to `break`, since it will have found the filename of the new zip.

Step 2: Create the New ZIP File

Next let's create the ZIP file. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    # Create the ZIP file.
    print('Creating %s...' % (zipFilename))
❶   backupZip = zipfile.ZipFile(zipFilename, 'w')

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\delicious')
```

Now that the new ZIP file's name is stored in the `zipFilename` variable, you can call `zipfile.ZipFile()` to actually create the ZIP file ❶. Be sure to pass `'w'` as the second argument so that the ZIP file is opened in write mode.

Step 3: Walk the Directory Tree and Add to the ZIP File

Now you need to use the `os.walk()` function to do the work of listing every file in the folder and its subfolders. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--

    # Walk the entire folder tree and compress the files in each folder.
❶   for foldername, subfolders, filenames in os.walk(folder):
        print('Adding files in %s...' % (foldername))
        # Add the current folder to the ZIP file.
❷       backupZip.write(foldername)
```

```
            # Add all the files in this folder to the ZIP file.
❸           for filename in filenames:
                newBase / os.path.basename(folder) + '_'
                if filename.startswith(newBase) and filename.endswith('.zip')
                    continue   # don't backup the backup ZIP files
                backupZip.write(os.path.join(foldername, filename))
        backupZip.close()
        print('Done.')

backupToZip('C:\\delicious')
```

You can use `os.walk()` in a `for` loop ❶, and on each iteration it will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder.

In the `for` loop, the folder is added to the ZIP file ❷. The nested `for` loop can go through each filename in the `filenames` list ❸. Each of these is added to the ZIP file, except for previously made backup ZIPs.

When you run this program, it will produce output that will look something like this:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

The second time you run it, it will put all the files in *C:\delicious* into a ZIP file named *delicious_2.zip*, and so on.

MODULE 4

CHAPTER 2

DEBUGGING

Raising Exceptions

Python raises an exception whenever it tries to execute invalid code. In Chapter 3, you read about how to handle Python's exceptions with try and except statements so that your program can recover from exceptions that you anticipated. But you can also raise your own exceptions in your code. Raising an exception is a way of saying, "Stop running the code in this function and move the program execution to the except statement."

Exceptions are raised with a raise statement. In code, a raise statement consists of the following:

- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function

For example, enter the following into the interactive shell:

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

If there are no try and except statements covering the raise statement that raised the exception, the program simply crashes and displays the exception's error message.

Often it's the code that calls the function, not the fuction itself, that knows how to handle an expection. So you will commonly see a raise statement inside a function and the try and except statements in the code calling the function. For example, open a new file editor window, enter the following code, and save the program as *boxPrint.py*:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
❶       raise Exception('Symbol must be a single character string.')
    if width <= 2:
❷       raise Exception('Width must be greater than 2.')
    if height <= 2:
❸       raise Exception('Height must be greater than 2.')


    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
❹   except Exception as err:
❺       print('An exception happened: ' + str(err))
```

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOO
O                O
O                O
O                O
OOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

Using the try and except statements, you can handle errors more gracefully instead of letting the entire program crash.

Getting the Traceback as a String

When Python encounters an error, it produces a treasure trove of error information called the *traceback*. The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error. This sequence of calls is called the *call stack*.

Open a new file editor window in IDLE, enter the following program, and save it as *errorExample.py*:

```
def spam():
    bacon()
```

```
def bacon():
    raise Exception('This is the error message.')

spam()
```

When you run *errorExample.py*, the output will look like this:

```
Traceback (most recent call last):
  File "errorExample.py", line 7, in <module>
    spam()
  File "errorExample.py", line 2, in spam
    bacon()
  File "errorExample.py", line 5, in bacon
    raise Exception('This is the error message.')
Exception: This is the error message.
```

From the traceback, you can see that the error happened on line 5, in the `bacon()` function. This particular call to `bacon()` came from line 2, in the `spam()` function, which in turn was called on line 7. In programs where functions can be called from multiple places, the call stack can help you determine which call led to the error.

The traceback is displayed by Python whenever a raised exception goes unhandled. But you can also obtain it as a string by calling `traceback.format_exc()`. This function is useful if you want the information from an exception's traceback but also want an `except` statement to gracefully handle the exception. You will need to import Python's `traceback` module before calling this function.

For example, instead of crashing your program right when an exception occurs, you can write the traceback information to a log file and keep your program running. You can look at the log file later, when you're ready to debug your program. Enter the following into the interactive shell:

```
>>> import traceback
>>> try:
        raise Exception('This is the error message.')
except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close()
        print('The traceback info was written to errorInfo.txt.')

116
The traceback info was written to errorInfo.txt.
```

The `116` is the return value from the `write()` method, since 116 characters were written to the file. The traceback text was written to *errorInfo.txt*.

```
Traceback (most recent call last):
  File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

Assertions

An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. In code, an assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

For example, enter the following into the interactive shell:

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that.'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

Here we've set podBayDoorStatus to 'open', so from now on, we fully expect the value of this variable to be 'open'. In a program that uses this variable, we might have written a lot of code under the assumption that the value is 'open'—code that depends on its being 'open' in order to work as we expect. So we add an assertion to make sure we're right to assume podBayDoorStatus is 'open'. Here, we include the message 'The pod bay doors need to be "open".' so it'll be easy to see what's wrong if the assertion fails.

Later, say we make the obvious mistake of assigning podBayDoorStatus another value, but don't notice it among many lines of code. The assertion catches this mistake and clearly tells us what's wrong.

In plain English, an assert statement says, "I assert that this condition holds true, and if not, there is a bug somewhere in the program." Unlike exceptions, your code should *not* handle assert statements with try and except; if an assert fails, your program *should* crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that's causing the bug.

Assertions are for programmer errors, not user errors. For errors that can be recovered from (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement.

Using an Assertion in a Traffic Light Simulation

Say you're building a traffic light simulation program. The data structure representing the stoplights at an intersection is a dictionary with keys `'ns'` and `'ew'`, for the stoplights facing north-south and east-west, respectively. The values at these keys will be one of the strings `'green'`, `'yellow'`, or `'red'`. The code would look something like this:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

These two variables will be for the intersections of Market Street and 2nd Street, and Mission Street and 16th Street. To start the project, you want to write a `switchLights()` function, which will take an intersection dictionary as an argument and switch the lights.

At first, you might think that `switchLights()` should simply switch each light to the next color in the sequence: Any `'green'` values should change to `'yellow'`, `'yellow'` values should change to `'red'`, and `'red'` values should change to `'green'`. The code to implement this idea might look like this:

```
def switchLights(stoplight):
    for key in stoplight.keys():
        if stoplight[key] == 'green':
            stoplight[key] = 'yellow'
        elif stoplight[key] == 'yellow':
            stoplight[key] = 'red'
        elif stoplight[key] == 'red':
            stoplight[key] = 'green'

switchLights(market_2nd)
```

You may already see the problem with this code, but let's pretend you wrote the rest of the simulation code, thousands of lines long, without noticing it. When you finally do run the simulation, the program doesn't crash—but your virtual cars do!

Since you've already written the rest of the program, you have no idea where the bug could be. Maybe it's in the code simulating the cars or in the code simulating the virtual drivers. It could take hours to trace the bug back to the `switchLights()` function.

But if while writing `switchLights()` you had added an assertion to check that *at least one of the lights is always red*, you might have included the following at the bottom of the function:

```
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
```

With this assertion in place, your program would crash with this error message:

```
Traceback (most recent call last):
  File "carSim.py", line 14, in <module>
    switchLights(market_2nd)
  File "carSim.py", line 13, in switchLights
    assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
❶ AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

Logging

If you've ever put a `print()` statement in your code to output some variable's value while your program is running, you've used a form of *logging* to debug your code. Logging is a great way to understand what's happening in your program and in what order its happening. Python's `logging` module makes it easy to create a record of custom messages that you write. These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time. On the other hand, a missing log message indicates a part of the code was skipped and never executed.

Using the logging Module

To enable the `logging` module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the `#! python` shebang line):

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -  %(levelname)s
-  %(message)s')
```

You don't need to worry too much about how this works, but basically, when Python logs an event, it creates a `LogRecord` object that holds information about that event. The `logging` module's `basicConfig()` function lets you specify what details about the `LogRecord` object you want to see and how you want those details displayed.

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -  %(levelname)s
-  %(message)s')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)'  % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)'  % (n))
    return total

print(factorial(5))
logging.debug('End of program')
```

Here, we use the `logging.debug()` function when we want to print log information. This `debug()` function will call `basicConfig()`, and a line of information will be printed. This information will be in the format we specified in `basicConfig()` and will include the messages we passed to `debug()`. The `print(factorial(5))` call is part of the original program, so the result is displayed even if logging messages are disabled.

The output of this program looks like this:

```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Table 10-1: Logging Levels in Python

Level	Logging Function	Description
DEBUG	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.

Table 10-1 (*continued*)

Level	Logging Function	Description
ERROR	`logging.error()`	Used to record an error that caused the program to fail to do something.
CRITICAL	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Your logging message is passed as a string to these functions. The logging levels are suggestions. Ultimately, it is up to you to decide which category your log message falls into. Enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s -  %(message)s')
>>> logging.debug('Some debugging details.')
2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2015-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The `logging.disable()` function disables these so that you don't have to go into your program and remove all the logging calls by hand. You simply pass `logging.disable()` a logging level, and it will suppress all log messages at that level or lower. So if you want to disable logging entirely, just add `logging.disable(logging.CRITICAL)` to your program. For example, enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
%(levelname)s -  %(message)s')
```

```
>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Since `logging.disable()` will disable all messages after it, you will probably want to add it near the `import logging` line of code in your program. This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a `filename` keyword argument, like so:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='
%(asctime)s -  %(levelname)s -  %(message)s')
```

The log messages will be saved to *myProgramLog.txt*. While logging messages are helpful, they can clutter your screen and make it hard to read the program's output. Writing the logging messages to a file will keep your screen clear and store the messages so you can read them after running the program. You can open this text file in any text editor, such as Notepad or TextEdit.

IDLE's Debugger

The *debugger* is a feature of IDLE that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. By running your program "under the debugger" like this, you can take as much time as you want to examine the values in the variables at any given point during the program's lifetime. This is a valuable tool for tracking down bugs.

To enable IDLE's debugger, click **Debug** ▸ **Debugger** in the interactive shell window. This will bring up the Debug Control window, which looks like Figure 10-1.

When the Debug Control window appears, select all four of the **Stack**, **Locals**, **Source**, and **Globals** checkboxes so that the window shows the full set of debug information. While the Debug Control window is displayed, any time you run a program from the file editor, the debugger will pause execution before the first instruction and display the following:

- The line of code that is about to be executed
- A list of all local variables and their values
- A list of all global variables and their values

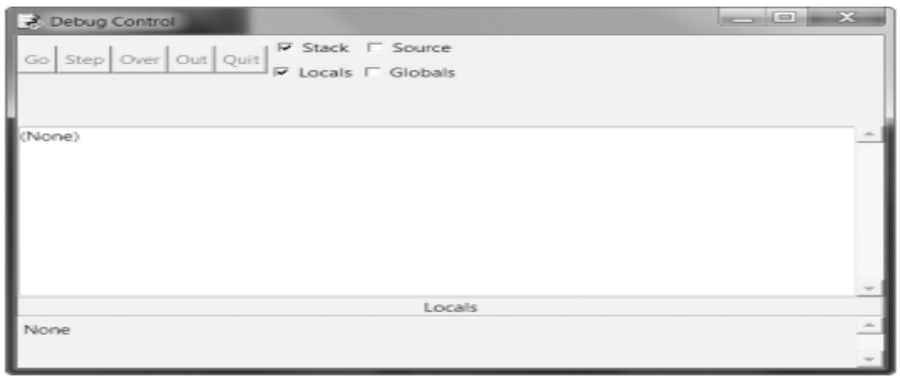

Figure 10-1: The Debug Control window

You'll notice that in the list of global variables there are several variables you haven't defined, such as `__builtins__`, `__doc__`, `__file__`, and so on. These are variables that Python automatically sets whenever it runs a program. The meaning of these variables is beyond the scope of this book, and you can comfortably ignore them.

The program will stay paused until you press one of the five buttons in the Debug Control window: Go, Step, Over, Out, or Quit.

Go

Clicking the Go button will cause the program to execute normally until it terminates or reaches a *breakpoint*. (Breakpoints are described later in this chapter.) If you are done debugging and want the program to continue normally, click the **Go** button.

Step

Clicking the Step button will cause the debugger to execute the next line of code and then pause again. The Debug Control window's list of global and local variables will be updated if their values change. If the next line of code is a function call, the debugger will "step into" that function and jump to the first line of code of that function.

Over

Clicking the Over button will execute the next line of code, similar to the Step button. However, if the next line of code is a function call, the Over button will "step over" the code in the function. The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns. For example, if the next line of code is a `print()` call, you don't

really care about code inside the built-in `print()` function; you just want the string you pass it printed to the screen. For this reason, using the Over button is more common than the Step button.

Out

Clicking the Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you have stepped into a function call with the Step button and now simply want to keep executing instructions until you get back out, click the **Out** button to "step out" of the current function call.

Quit

If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the **Quit** button. The Quit button will immediately terminate the program. If you want to run your program normally again, select **Debug ▸ Debugger** again to disable the debugger.

Debugging a Number Adding Program

Open a new file editor window and enter the following code:

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

Save it as *buggyAddingProgram.py* and run it first without the debugger enabled. The program will output something like this:

```
Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
The sum is 5342
```

The program hasn't crashed, but the sum is obviously wrong. Let's enable the Debug Control window and run it again, this time under the debugger.

When you press F5 or select **Run ▸ Run Module** (with **Debug ▸ Debugger** enabled and all four checkboxes on the Debug Control window checked), the program starts in a paused state on line 1. The debugger will always pause on the line of code it is about to execute. The Debug Control window will look like Figure 10-2.

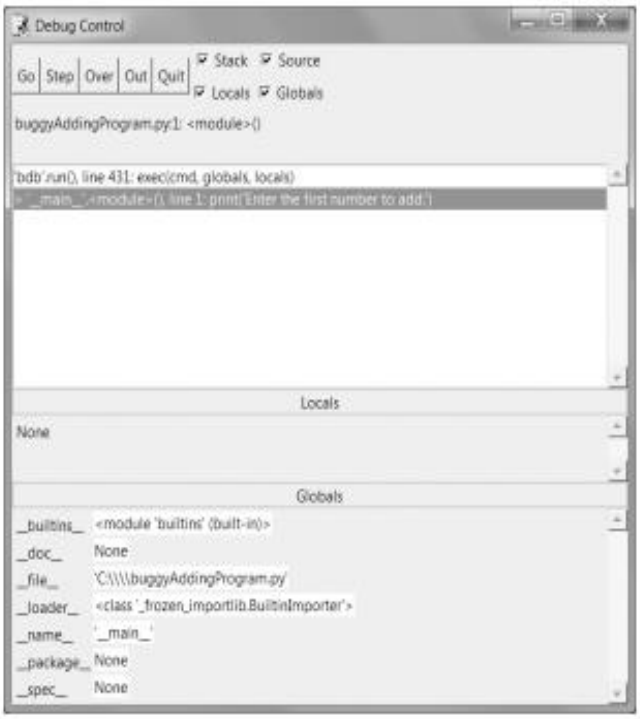

Figure 10-2: The Debug Control window when the program first starts under the debugger

Click the **Over** button once to execute the first `print()` call. You should use Over instead of Step here, since you don't want to step into the code for the `print()` function. The Debug Control window will update to line 2, and line 2 in the file editor window will be highlighted, as shown in Figure 10-3. This shows you where the program execution currently is.

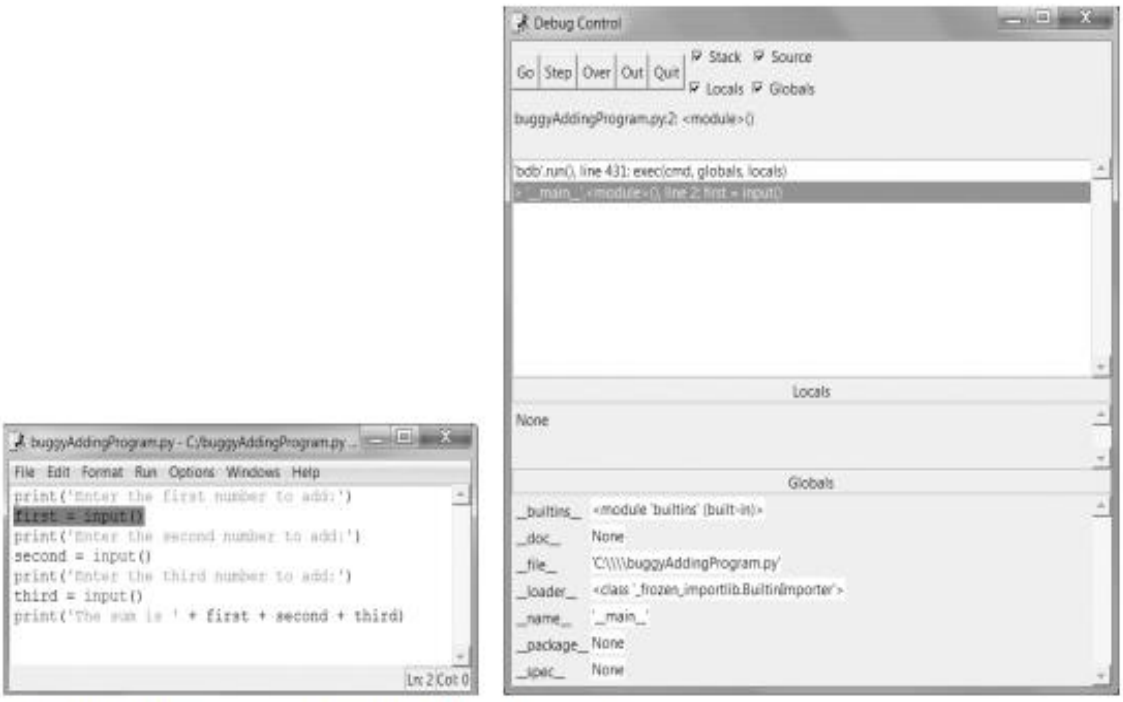

Figure 10-3: The Debug Control window after clicking Over

Click **Over** again to execute the `input()` function call, and the buttons in the Debug Control window will disable themselves while IDLE waits for you to type something for the `input()` call into the interactive shell window. Enter `5` and press Return. The Debug Control window buttons will be reenabled.

Keep clicking **Over**, entering `3` and `42` as the next two numbers, until the debugger is on line 7, the final `print()` call in the program. The Debug Control window should look like Figure 10-4. You can see in the Globals section that the first, second, and third variables are set to string values `'5'`, `'3'`, and `'42'` instead of integer values 5, 3, and 42. When the last line is executed, these strings are concatenated instead of added together, causing the bug.

Figure 10-4: The Debug Control window on the last line. The variables are set to strings, causing the bug.

Stepping through the program with the debugger is helpful but can also be slow. Often you'll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with breakpoints.

Breakpoints

A *breakpoint* can be set on a specific line of code and forces the debugger to pause whenever the program execution reaches that line. Open a new file editor window and enter the following program, which simulates flipping a coin 1,000 times. Save it as *coinFlip.py*.

```
import random
heads = 0
for i in range(1, 1001):
❶     if random.randint(0, 1) == 1:
          heads = heads + 1
      if i == 500:
❷         print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

The `random.randint(0, 1)` call ❶ will return `0` half of the time and `1` the other half of the time. This can be used to simulate a 50/50 coin flip where `1` represents heads. When you run this program without the debugger, it quickly outputs something like the following:

```
Halfway done!
Heads came up 490 times.
```

If you ran this program under the debugger, you would have to click the Over button thousands of times before the program terminated. If you were interested in the value of `heads` at the halfway point of the program's execution, when 500 of 1000 coin flips have been completed, you could instead just set a breakpoint on the line `print('Halfway done!')` ❷. To set a breakpoint, right-click the line in the file editor and select **Set Breakpoint**, as shown in Figure 10-5.

Figure 10-5: Setting a breakpoint

You don't want to set a breakpoint on the `if` statement line, since the `if` statement is executed on every single iteration through the loop. By setting the breakpoint on the code in the `if` statement, the debugger breaks only when the execution enters the `if` clause.

The line with the breakpoint will be highlighted in yellow in the file editor. When you run the program under the debugger, it will start in a paused state at the first line, as usual. But if you click Go, the program will run at full speed until it reaches the line with the breakpoint set on it. You can then click Go, Over, Step, or Out to continue as normal.