Chapter 13
Debugging

Abstract Even experienced programmers create *bugs*—coding errors that take time to correct. Syntax has to be precisely correct and programming is a complicated task. Computers do exactly what the program says; they are unforgiving. Fortunately, there are tools for locating, understanding, and handling coding errors. Integrated Developments Environments (IDEs), such as PythonWin, have syntax checking and debugging tools. The `arcpy` package has functions for gathering information about errors and the Python language itself has functions and keywords for handling errors gracefully. This chapter covers error debugging and related topics. The three types of programming errors are: *syntax errors*, *exceptions*, and *logic errors*.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Check for syntax errors.
- Recognize exceptions.
- Run code in debug mode.
- Watch variables as code is running.
- Step through code one line at a time.
- Set stopping points (breakpoints) at crucial points within the code.
- Run the code to a breakpoint.
- Set conditions for stopping at a breakpoint.
- Design input to test the code.

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_13

13.1 Syntax Errors

Syntax errors are usually caused by mistakes in spelling, punctuation, or indentation. Spotting syntax errors becomes easier with practice. See how quickly you can locate the two syntax errors in the following code:

```
# buggyCode1.py
import os, sys
outputDir = os.path.dirname(sys.argv[1]) + '\outputFiles/
if not os.path.exists(outputDir)
    os.mkdir(outputDir)
```

The end of line 3 is a missing a quotation mark and line 4 is missing a parenthesis.

Chapter 4 introduced the PythonWin syntax check button  , which is a bit like the spelling/grammar checker in a word processing editor. Word processors can automatically identify errors such as misspellings and run-on sentences; Proof readers are needed to catch other writing errors such as misplaced modifiers or split infinitives. Similarly, certain types of programming errors are more difficult to automatically detect than others.

Syntax errors are errors that can be detected just based on parsing the text. *Parsing* is breaking the text down into a set of components that the system can recognize. When the 'Run' button is pushed, the IDE first attempts to parse the code. Syntax errors confound the parsing process. If a syntax error is detected, the parser stops and the script will not run. Remember—the parser jumps to the line or near the line where the syntax error occurred to provide a clue.

13.2 Exceptions

The parser can only detect violations of syntax rules. If the syntax rules are adhered to, Python will run (or attempt to run) the script. If it encounters an error when attempting to execute a line of code, it throws an exception and stops the run at that point. Chapter 2 discussed Python built-in exceptions such as `NameErrors` and the traceback errors they print. Can you see why the following script throws a `NameError` exception?

```
# buggyCode2.py
import arcpy, sys, os
arcpy.env.workspace = sys.argv[1]
fc = arcpy.ListFeatureClasses()
for fc in fcs:
```

```
# Append Buffer to the output name.
fcBuffer = os.path.splitext(fc)[0] + 'Buffer.shp'
# Call buffer tool with required input,
#    output, and distance arguments.
arcpy.Buffer_analysis(fc, fcBuffer, '1 elephant')
```

Don't be distracted by the elephant—we'll get to that in a minute. Look earlier in the script. Python executes each line of code in order. Before it even gets close to the elephant, it reaches an undefined variable, throws an exception, stops the run, and prints a traceback with the following exception:

```
NameError: name 'fcs' is not defined
```

The variable `'fcs'` is used on line 5, but it's not defined. If you have already run a script that assigned a value to `'fcs'` during this PythonWin session, you won't see this error. Remember to test scripts with a fresh PythonWin session before sharing code. Modify line 4 to define `'fcs'` as follows:

```
fcs = arcpy.ListFeatureClasses()
```

Now we're ready to talk about the elephant. Save and run the script again. This time it reaches the Buffer tool call, tries to create a buffer, can't understand the buffer distance argument, and prints a traceback with the following exception:

```
ExecuteError: Failed to execute. Parameters are not valid.
ERROR 000728: Field 1 elephant does not exist within table
Failed to execute (Buffer).
```

The required buffer distance parameter of the Buffer tool can be specified as a linear unit or a field name (for a field whose values specify a buffer distance for each feature). Since 'elephant' is not a recognized unit of measure, it searches for a field named '1 elephant'. No such field exists and so it fails to execute and throws an `ExecuteError`. The `ExecuteError` is a special error thrown by `arcpy` whenever a geoprocessing tool encounters an error.

Both of the exceptions thrown by 'buggyCode2.py' can be corrected by changing the code (by defining `fcs` and specifying a valid linear unit). Other factors that can cause exceptions to be thrown may be beyond the programmer's control. For example, the script may encounter a corrupted file in the list which it can't buffer. Python structures for handling these situations will be presented in Chapter 14.

13.3 Logic Errors

Errors do not always generate exceptions. A program containing logic errors can run smoothly to completion and have inaccurate results. Because logic errors are not detectable by the computer, we have to understand the problems we're solving and inspect the code closely when we perceive unexpected results. The following code for normalizing the time-series dates contains a logic error:

```
# buggyCode3.py
# normalize data time steps
timeSteps = [2011, 2009, 2008, 2005, 2004, 2003, 2001, 1999]
# normalize to values between 0 and 1
maxv = max(timeSteps)
minv = min(timeSteps)
r = maxv - minv
# list comprehension with buggy expression
normalizedList = [v - minv/r for v in timeSteps]
print normalizedList
```

A set of numbers are normalized by subtracting the minimum value and dividing by the range. The output should be numbers between 0 and 1, but the results are the following list:

```
>>> [1845, 1843, 1842, 1839, 1838, 1837, 1835, 1833]
```

'buggyCode3.py' runs without errors because the code doesn't violate any syntax rules and it doesn't throw any exceptions; it also gives the wrong result. Can you identify the two mistakes in the list comprehension expression? First, the missing parentheses in the numerator cause the arithmetic operation to be performed in an unintended order. The division is performed first. In this case, our minimum value is 1999 and the range is 12, so 1999 is divided by 12, giving 166. This value (166) is then subtracted from each time step (For example, for the first time step, the result is `2011 - 166 = 1845`). The subtraction must be wrapped in parentheses to force it to be calculated first. Replace the list comprehension with the following line of code:

```
normalizedList = [(v - minv)/r for v in timeSteps]
```

Running the script with the modified list comprehension prints the following:

```
>>> [1, 0, 0, 0, 0, 0, 0, 0]
```

The results are mostly zeros when we would expect decimal values ranging between 0 and 1. This brings us to the second mistake; Integer division causes the

division remainders to be discarded. Cast the numerator or denominator to solve this problem. Replace the list comprehension with the following line of code:

```
normalizedList = [(v - minv)/float(r) for v in timeSteps]
```

and the truly normalized list is printed as follows:

```
>>> [1.0, 0.8333333333333334, 0.75, 0.5, 0.4166666666666667,
0.3333333333333333, 0.16666666666666666, 0.0]
```

> **Tip** Use a small tractable sample dataset for preliminary testing to catch logic errors early.

The small set of numbers in the time step list made it easy to evaluate the results. When you're trying new code, it's good practice to test the functionality incrementally on a very small dataset so that you can predict the desired outcome and compare it to the results produced by the code. Had this been run on a large input file of dates, unearthing the bugs would take longer. In fact, you may not notice the problem until you perform other calculations on the output. Even then you may not notice the mistake. Further calculations might mask the error. Unlike syntax errors or exceptions, your only means of detecting logic errors is inspecting the results. By testing each piece as you build it, you can be more confident that the overall result is correct.

> **Tip** Test code incrementally as you build each new piece of functionality.

Sometimes errors are not revealed by one test. It's good practice to test multiple scenarios. The following script is supposed to remove names from a list, but the code is poorly designed:

```
# buggyLoop.py
# Remove feature classes whose names do not contain the given tag.
tag = 'zones'
fcs = [u'data1',u'data2',u'data3',u'fireStations',
       u'park_data',u'PTdata4',u'schoolzones',
       u'regions1',u'regions2',u'workzones']

print 'Before loop: {0}'.format(fcs)
for fcName in fcs:
    if tag in fcName:
        fcs.remove(fcName)
print 'After loop: {0}'.format(fcs)
```

This code removes items containing the word ‘zones’. The script prints the following:

```
>>> Before loop: [u'data1', u'data2', u'data3', u'fireStations',
u'park_data', u'PTdata4', u'schoolzones', u'regions1', u'regions2',
u'workzones']
After loop: [u'data1', u'data2', u'data3', u'fireStations', u'park_data',
u'PTdata4', u'regions1', u'regions2']
```

But when `tag = 'zones'` is replaced with `tag = 'data'`, the code no longer works as expected:

```
>>> Before loop: [u'data1', u'data2', u'data3', u'fireStations',
u'park_data', u'PTdata4', u'schoolzones', u'regions1', u'regions2',
u'workzones']
After loop: [u'data2', u'fireStations', u'PTdata4', u'schoolzones',
u'regions1', u'regions2', u'workzones']
```

In this case, the `remove` statement confounds the proper course of iteration. Each time an item is removed, `fcName` skips the next item when it gets updated. Step through this code with the debugger and watch `fcName` and `fcs` to view this peculiar behavior.

In fact, making alterations to a list while you’re looping through it is never a good idea. The loop needs to use a second list to store the results, as in the following code:

```
print 'Before loop: {0}'.format(fcs)
fcsOut = []
for fcName in fcs:
    if tag not in fcName:
        fcsOut.append(fcName)
print 'After loop: {0}'.format(fcsOut)
```

Logic errors come in many forms, but they all arise from the same basic problem: your code is doing something other than what you desire and expect. Once you perceive a mistake in the results, it may take time to discover the source. Many beginning programmers avoid using IDE’s built-in debugger and try to use the “poor man’s debugger”—debugging by simply printing variable values throughout the code. This is effective some of the time, but in many cases, using the debugging functionality reveals the problem more quickly. The remainder of this chapter discusses debugging toolbar buttons and how to use breakpoints to improve debugging technique.

13.4 PythonWin Debugging Toolbar

When code raises exception errors or contains logic errors, you often need to look more closely at variable values. With an IDE such as PythonWin, code can be run in the standard way, using the 'Run' button or it can be run in debug mode. Chapter 11 showed how to use the debugger to step through the code one line at a time and watch variable values change as the code executes. Recall these points from the debugger discussion in Chapter 11:

- To make the toolbar visible: View>Toolbars>Debugging (Some buttons are grayed out, but become active when a debugging session commences)
- Click the 'Step over' button to start debugging and step one line at a time through the code.
- Generally, you don't want to click the 'Step (in)' button, but if you do by mistake, click the 'Step out' button until you return to your script. We'll revisit these two buttons in Chapter 8.
- The arrow points at the current line of code while you are stepping using the debugger. This points to the next line of code that has not yet been executed.
- Click the 'Watch' button to open the Watch panel and double-click on 'New Item' to add a variable to the watch list.
- Click the 'Close' button to stop the debugging session at any time before the script completes.

So far, you're familiar with half the debugging toolbar buttons. Figure 13.1 depicts the toolbar buttons clustered by related functionality (watch, breakpoint, step, and go/close). Table 13.1 explains each cluster, listing each button's name and usage. You have already used the 'Watch' button, the three 'Step (in)' buttons, and the 'Close' button. Now we'll address the unfamiliar buttons in Table 13.1.

The 'Stack view' button opens a panel that displays the values of all objects that are currently defined within the PythonWin session. Each object expands to

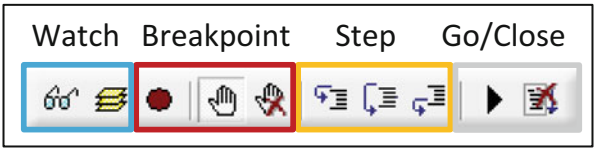


Figure 13.1 Debugging toolbar buttons clusters.

Table 13.1 PythonWin debugging toolbar buttons.

Icon	Key	Name	Usage
		Watch	Display the value of Python code expressions during a debugging session.
		Stack view	Displays the values of all the currently available Python objects.
		Breakpoint list	Lists each breakpoint and set breakpoint conditions.
	F9	Toggle breakpoint	Place the cursor on a line of code and click this button to toggle a breakpoint on this line.
		Clear breakpoints	Removes all breakpoints.
	F11	Step	Execute one line of code. Step inside any functions called in this next line of code.
	F10	Step over	Execute the next line of code without stepping inside any functions called in this line of code.
	Shift+F11	Step out	Step out of a function.
	F5	Go	Run in debug mode to the first (or next) breakpoint.
	Shift+F5	Close	Stop the debugging session without executing another line of code. This does not close PythonWin or any scripts.

show the contained objects. This complex tree structure may not be of interest to beginners. The next three buttons relate to using breakpoints, which are extremely useful for programmers of all skill levels.

13.4.1 Using Breakpoints

Breakpoints are a means to stop the script at a specified line in the program. Once you've reached that line, you can investigate the values of your variables and step through the code to see how it's working. The 'Toggle breakpoint' button  is the most crucial breakpoint button. This allows you to set breakpoints within your code.

The 'Go' button  runs the script to the first (or next) breakpoint it encounters. Conversely, the 'Close' button  (which is more like a stop button) can be used to stop the debugging session when the execution is paused. Despite the name, this button does not close PythonWin nor any of the scripts. It only closes (or stops) the debugging session.

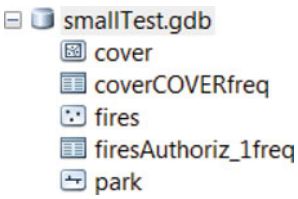
The 'Go' button can be used together with breakpoints to fast-forward the debugger to areas of code that need close inspection. The following code for performing frequency analysis on string fields has an error that is difficult to pinpoint without debugging:

```
# buggyFreq.py
# Purpose: Find frequency of each value in each string field.

import arcpy
arcpy.env.overwriteOutput = True
arcpy.env.workspace = 'C:/gispy/data/ch13/smallTest.gdb'
featureList = arcpy.ListFeatureClasses()

for inputFile in featureList:
    fields = arcpy.ListFields(inputFile, '*', 'String')
    for field in fields:
        fieldName = field.name
    outTable = inputFile + fieldName + 'freq'
    arcpy.Frequency_analysis(inputFile, outTable, fieldName)
    print 'Output table created: {0}'.format(outTable)
```

A frequency table tallies the number of times each value occurs in a field. Running this script creates two frequency tables, one for the 'cover' feature class ('coverCOVERfreq') and one for the 'fires' feature class ('firesAuthoriz_1 freq'):



But then it crashes with the following error:

```
ERROR 000728: Field Authoriz_1 does not exist within table
Failed to execute (Frequency).
```

This code has an indentation error that is causing some lines of codes to be executed at the wrong time. It's often useful to put breakpoints inside loops or conditional constructs. To see the error in this script, place a breakpoint on line 11: Move the cursor to line 11 (any position on the line will do) and then click the 'Toggle breakpoint' button  . A breakpoint circle appears in the margin:

```
 9  -for inputFile in featureList:
10       fields = arcpy.ListFields(inputFile, '*', 'String')
11  -    for field in fields:
12           fieldName = field.name
13       outTable = inputFile + fieldName + 'freq'
```

Use the 'Go' button  to run the code to the breakpoint. The cursor stops on line 11 and won't execute this line of code until debugging is resumed:

```
 9  -for inputFile in featureList:
10       fields = arcpy.ListFields(inputFile, '*', 'String')
11  -    for field in fields:
12           fieldName = field.name
13       outTable = inputFile + fieldName + 'freq'
```

> **Tip** Set watches on input variables.

Display the watch panel  and set watches for the `inputFile`, `fieldName`, and `outputTable` variables. You can also shift the Watch window from its default position on the left to stretch along the bottom of PythonWin. Now resume debugging by the stepping over line 11  . Continue stepping over, one line at a time, and watch the values change until the script stops. You'll first see the frequency table created for the 'COVER' field in the 'cover' feature class.

The next feature class, 'fires', has three string fields 'FireName', 'SizeClass', and 'Authoriz_1'. A frequency table is only created for the last one because of the indentation error. Lines 13–15 should be indented to be inside of the field loop. Instead, the only field name that reaches the Frequency (Analysis) tool is the last field name assigned before leaving the loop.

The next feature class is 'park', which has no string fields. Regardless of this, the Frequency (Analysis) tool is called using the most recent field name ('Authoriz_1'). Since 'park' has no such field, an error occurs. When the error is thrown, the debugging session curtails.

```
 9  -for inputFile in featureList:
10       fields = arcpy.ListFields(inputFile, '*', 'String')
11  -    for field in fields:
12           fieldName = field.name
13       outTable = inputFile + fieldName + 'freq'
14       arcpy.Frequency_analysis(inputFile, outTable, fieldName)
15       print 'Output table created: {0}'.format(outTable)
```

Expression	Value
inputFile	u'park'
fieldName	u'Authoriz_1'
outTable	u'parkAuthoriz_1freq'

The second button in the breakpoint cluster is called ‘Breakpoint list’  . This button toggles a panel with a list of breakpoints and allows you to specify breaking conditions. The Breakpoint List panel, like the Watch panel, is only available while a debugging session is in progress.

> **Tip** Set conditional breakpoints on iterating variables and use values from traceback messages.

Using the breakpoint on line 11 of ‘buggyfreq.py’ improved efficiency by running the code directly to the section that was causing the error. This could be refined even further. Based on the error thrown by ‘buggyfreq.py’, the trouble only began when the field name became ‘Authoriz_1’. To specify this condition, commence a new debugging session by clicking the ‘Step over’ button  and then click the ‘Breakpoint list’ button  . The panel shows a list of breakpoint locations. (Though this example only uses one breakpoint, multiple breakpoints can be set.) Click in the ‘Condition’ column next to the breakpoint ‘buggyfreq.py:11’. Set the condition to check when the field name is ‘Authoriz_1’. Conditions are Python expressions, so two equals signs are used to check for equality.

Condition	Location
fieldName == 'Authoriz_1'	buggyfreq.py: 11

Now run the condition to this breakpoint, using the ‘Go’ button and note that it runs past the fields ‘cover’, ‘FireName’, and ‘SizeClass’ without stopping. The field name is now ‘Authoriz_1’. Click ‘Go’ again and execution stops again, because the code reaches this point again with the field name of ‘Authoriz_1’. Setting conditional breakpoints can be very useful when dealing with a high number of repetitions in a loop.

The third button in the breakpoint cluster removes all the breakpoints  . Note that the Toggle breakpoint button  removes a single breakpoint on the line where the cursor resides. If you intend to only remove one break point, use the toggle breakpoint button. The symbols on these two buttons might lead one to believe that they are exact opposites, but they’re not. Set several more breakpoints in the script and then experiment to see the difference.

Now that we’re familiar with the buttons, well resolve the ‘buggyfreq.py’ example. To repair this script, indent lines 13–15. Indent multiple lines of code efficiently

by selecting the lines and then pressing 'Tab'. Then rerun it to create four tables without throwing errors:

```
>>> Output table created: coverCOVERfreq
Output table created: firesFireNamefreq
Output table created: firesSizeClassfreq
Output table created: firesAuthoriz_1freq
```

> **Tip** Set breakpoints inside loops, conditional constructs, and other structures that control the flow.

Debugging is useful not only for finding bugs, but for understanding the flow of a working script. It allows you to methodically step through and watch variables to discover how code execution proceeds. In summary, this example introduced three additional debugging techniques:

- **Setting breakpoints**: Place your cursor on a line of code to set a breakpoint for that line. A breakpoint allows you to run to that point in the code without stepping through all the previous lines of code one by one. Before you run the code, a breakpoint appears as a circle in the margin next to the line. When you start running in debug mode, it turns pink.
- **Running to breakpoints:** Start the script running in debug mode with the 'Go' button to run to the first breakpoint. If there is more than one break point, it will run to the next one when selected again. If there are no breakpoints, it will run through the entire script non-stop.
- **Setting breakpoint conditions:** Specify stopping conditions for a breakpoint so that it runs directly to the iteration that is causing trouble.

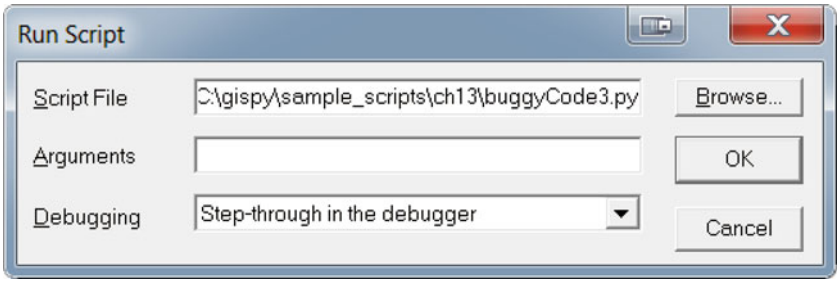
13.5 Running in Debug Mode

So far we have started a debugging session in PythonWin by either selecting 'Step over' or 'Go'. You can also start a debugging session via the 'Run Scripts' dialog box. Your preference may vary depending on the bugs, but here's a list of the options:

Option 1: Make the debugging toolbar viewable. Click 'Step over' (or F10) and start stepping through the code.

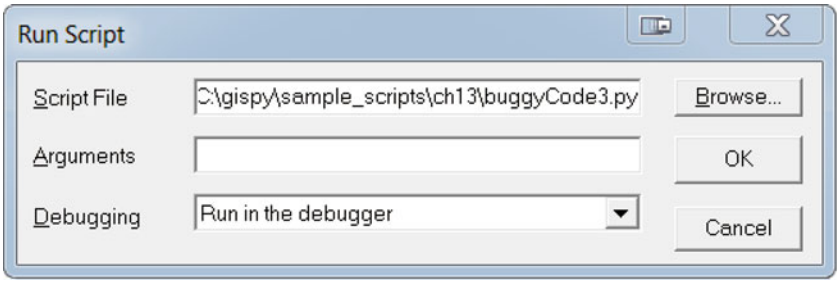
Option 2: Make the debugging toolbar viewable. Set breakpoints and click 'Go' (or F5).

Option 3: Click the 'Run' button and choose 'Step-through in the debugger' on the 'Run Script' form.



The debugging toolbar will automatically appear. Otherwise, this achieves essentially the same effect as option #1. Whichever mode you choose in the combo box remains until you choose another mode or restart PythonWin. In other words, to stop running in debug mode you'll have to choose 'No debugging' in this combo box.

Option 4: Click the 'Run' button and choose 'Run in the debugger' on the 'Run Script' form. This achieves the same effect as option #2, but it's persistent.



There is only one button for stopping a debugging session, but there are several ways in which a session can end:

1. The 'Close' button (or Shift+F5) is selected.
2. The code throws an exception.
3. The code runs to completion.

13.6 PyScripter Debugging Toolbar

The toolbars in most IDEs have very similar buttons and functionality. As an example, Figure 13.2 shows the PyScripter debugging toolbar (visible by default). Table 13.2 lists the PyScripter buttons, their PyScripter names, and their equivalents in PythonWin. PyScripter has a 'Toggle breakpoint' button, but breakpoints can also

Figure 13.2 PyScripter toolbar.

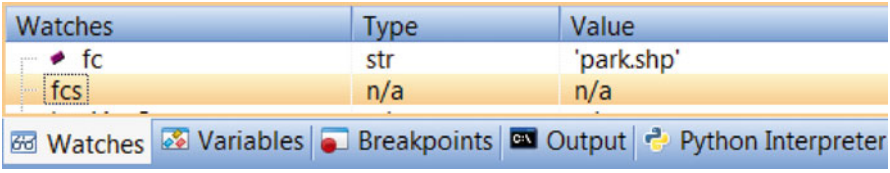
Table 13.2 PyScripter versus PythonWin debugging.

PyScripter	Name	PythonWin
	Run	
	Debug	
	Run to cursor	None
	Step into subroutine	
	Step over next function call	
	Step out of the current subroutine	
	Abort debugging	
	Toggle breakpoint	
	Clear all breakpoints	
Watch window	Watch	
Variables window	Stack view	
Breakpoint window	Breakpoint list	

be toggled by clicking in the margin next to the breakpoint line. PyScripter has a 'Run to cursor' button which has no equivalent in PythonWin.

PythonWin provides the Watch  , Stack view  , and Breakpoint list  buttons on the debugger toolbar to toggle the corresponding panels open/closed during debugging sessions.

PyScripter provides this functionality with windows that are open by default. The 'Watches', 'Variables', and 'Breakpoints' window tabs appear along the bottom of the application:



13.7 Debugging Tips

Debugging skills improve with experience. Here are a few tricks to keep in mind as you get started.

1. If the program is processing data files, use a very simple test file while you're debugging your code. If you don't have one already, create one. For example, if your data is a 1000 by 3000 table, create a 5 by 8 table by using the first few values in the larger table. Run your code on this until you think it's working. Then try it on the larger file.
2. Vary your test input. Code that is functioning correctly for one input, may not be working for another. It's impossible to test every scenario, but here are a few rules of thumb.
   (a) For complex code that takes data as input, test using multiple input files.
   (b) For code that hinges on conditional structures, use input that tests each of the branches in the conditional blocks.
3. Place breakpoints inside loops and conditional constructs. These structures are powerful, but at the same time, they can lead to confusion.
4. Watch variables where they are (supposed to be) changing, by inserting breakpoints or print statements in the code.
5. When working with ArcGIS data, beware of ArcCatalog locks that may throw exceptions and prevent a script from overwriting existing data.
6. Remember that you can 'Break into running code' in PythonWin (or Ctrl + F2 in PyScripter), if the script gets into an infinite loop as described in the Section 10.1 "How to Interrupt Running Code" instructions.

13.8 Key Terms

Bugs
Syntax errors
Parsing
Exceptions
Logic errors
Debugger
Debugging toolbars
Debug mode
Watches
Breakpoints
Breakpoint conditions

13.9 Exercises

1. Use the sample script 'numGames.py' to practice using breakpoints.
   (a) Set breakpoints on lines 5, 7, 11, and 13 (just inside the loop and at each print statement).

(b) Set the argument to the number 6 and select 'Run in the debugger' in the 'Run script' dialog.
(c) Add `x` to the Watch window.
(d) Investigate the script's behavior by clicking 'Go' to run from each breakpoint to the next until the script exits (You'll see the script name 'returned exit code 0' in the feedback bar).
(e) Set the argument to the number −5 and run in the debugger.
(f) Remove the breakpoint on line 5 by clicking the 'Toggle breakpoint' button while the arrow is on line 5.
(g) Select 'Go' four more times. Take a screen shot of the IDE that shows the script, the watch window, the breakpoints, and the arrow where it is now.
(h) Remove all the breakpoints by clicking the 'Clear breakpoints' button.
(i) Start it running again with the 'Go' button.
(j) Stop the infinite loop by clicking on the PythonWin icon in the applications tray and selecting 'Break into running code'. (When this succeeds, you will see a `KeyboardInterrupt` exception in the Interactive Window)
(k) Set breakpoints on lines 5, 7, 11, and 13 again and experiment with various numerical input values. Run to the breakpoints to observe the behavior and determine solutions to the following:
    i. Find an example of an input integer that triggers exactly three print statements.
    ii. Find an example of an input integer that triggers exactly one print statement.
    iii. Find an example of an input integer that triggers exactly two print statements.
    iv. Define the set of numbers that will result in an infinite loop.
    v. Define the set of integers that will result in no print statements.

2. **buggyMerge.py** The sample script 'buggyMerge.py' should merge all the independent dBASE tables in a directory, but it contains a logic error. It works correctly the first time it's run, but if you run it again the merge fails. Set a breakpoint on line 10, use the watch window to watch the `tables` variable, set a breakpoint inside the conditional statement on line 15, and use the debugger to discover and repair the error (so that it can be rerun without breaking).

3. **buggyConditional.py** The sample script 'buggyConditional.py' contains three bugs. Read the following description of how it is supposed to work, then repair the errors, using the debugger.

    - The script takes five arguments:

      One feature class: `arcpy.GetParameterAsText(0)`
      Three numeric values: (Z, AF, T) `arcpy.GetParameterAsText(1),(2),&(3)`
      One output field name: `arcpy.GetParameterAsText(4)`

- The output field is added, if it doesn't already exist.
- The conditional constructs calculate the output field value as follows:
  If the first condition is met, the output value is the sum of `T`, `Z`, and `AF`.
  If the second condition is met, the output value is the sum of `T` and `Z`.
  If the third condition is met, the output value is the sum of `T` and `AF`.
  If none are met, the output value is set to `T`.
- Then the output field is set to the output value using the Calculate Field (Data Management) tool.

In the Watch window, set watches on the input variables and set a breakpoint inside each conditional branch (lines 17, 20, 22, 24, 26). The following sample input should create a field named 'veld' with the value 9 in each row:

C:/gispy/data/ch13/tester.gdb/c1 -2 3 6 veld

Also be sure to test with positive and negative value combinations for Z and AF.