Chapter 11
Debugging and error handling

11.1 Introduction

This chapter discusses debugging procedures and provides a review of the most common Python errors. Error-handling procedures are also discussed, including how to get the most out of `try-except` statements.

No matter how careful you are in writing code, errors are bound to happen. There are three main types of errors you will encounter in Python: *syntax errors*, *exceptions*, and *logic errors*. Syntax errors prevent code from running. With an exception, a script will stop running midprocess. A logic error means the script will run but produce undesired results.

11.2 Recognizing syntax errors

Syntax errors pertain to spelling, punctuation, and indentation. Common syntax errors result from misspelled keywords or variables, missing punctuation, and inconsistent indentation. See if you can spot the error in the following code:

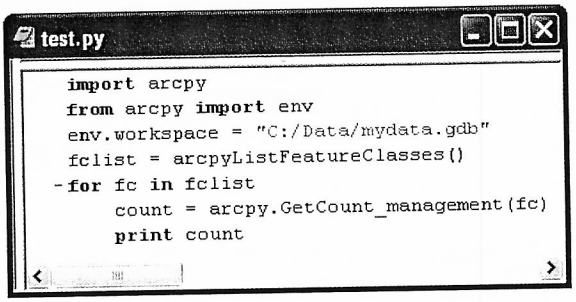
```
import arcpy
from arcpy import env
env.workspace = "C:/Data/mydata.gdb"
fclist = arcpy.ListFeatureClasses()
for fc in fclist
    count = arcpy.GetCount_management(fc)
    print count
```

The colon (:) at the end of the first line of the `for` loop is missing. The following syntax error is displayed when this code is run in the Python window:

```
Parsing error SyntaxError: invalid syntax
```

PythonWin has a built-in checking process, which works somewhat like a spell checker in a word-processing application. The process is enabled by clicking the Check button on the PythonWin Standard toolbar. This checks the current script file without running it.

Consider the preceding example code, which is shown in the figure.



When you click the Check button, a message appears on the PythonWin status bar: `Failed to run script - syntax error - invalid syntax`. The cursor is also moved to the line where the first syntax error was detected—in this case, line 5, as shown in the figure.

Changing line 5 to the following code fixes the syntax error:

```
for fc in fclist:
```

Running this code produces the following message: `Python and the TabNanny successfully checked the file 'test.py'`. This means there are no syntax errors and the code will run when you click the Run button.

The Check button runs a syntax checker and TabNanny, which checks for inconsistent indentation and spacing. For example, the following block of code uses inconsistent indentation:

```
for fc in fclist:
    count = arcpy.GetCount_management(fc)
     print count
```

>>> TIP

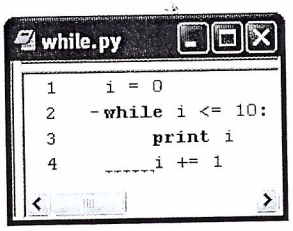
Remember that you can make the line numbers visible in PythonWin by clicking View > Options > Editor on the menu bar, and then increasing the margin width for line numbers.

Using this code produces the following message:

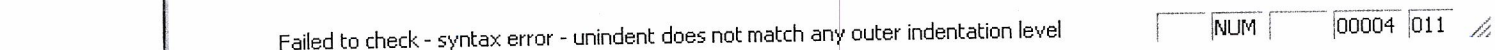
```
Failed to check - syntax error - unexpected indent
```

Similar errors can occur if your indentation uses a combination of spaces and tabs. Especially when you copy from other sources, such as Microsoft Word, Microsoft PowerPoint, or PDF, blocks of code may appear to be indented correctly visually but may actually use a combination of spaces and tabs.

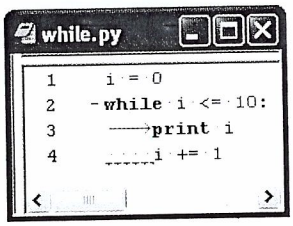
Consider the example code in the figure.



Visually, the block of code appears to be aligned, but TabNanny has underlined some of the whitespace in red. When a check is run, an error message appears, like the example in the figure.



To reveal the nature of the error, on the PythonWin menu bar, click View > Whitespace. The type of characters used for the indentation is displayed in the script window.



The arrow in the script window indicates the use of a tab, and the dots indicate spaces. Indentation should be consistent, so the tab should be replaced by four spaces.

Note: Although Python reports the line where a syntax error occurs, sometimes the actual syntax error occurs on a line that's above the one reported.

>>> TIP

Copying code from other sources such as Word and .pdf files will likely introduce other types of errors, including quotation marks. In general, therefore, it is not recommended to copy code from these types of files.

>>> TIP

Using the TAB key in PythonWin results in four spaces by default. The spacing can be modified by clicking View > Options > Tabs and Whitespace from the menu bar. An actual tab is normally introduced only when copying from other applications.

11.3 Recognizing exceptions

Syntax errors are frustrating, but they are relatively easy to catch compared to other errors. Consider the following example that has the syntax corrected:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data/mydata.gdb"
fclist = arcpy.ListFeatureClasses()
for fc in fclist:
    count = arcpy.GetCount_management(fc)
    print count
```

When you run the script again, it runs without a syntax error. But what if no count is printed? Is that an error? Perhaps the workspace is incorrect, or perhaps there are no feature classes in the workspace.

Rather than referring to these incidents as "errors," it is common for programming languages to discern between a normal course of events and something exceptional. There might be errors, but there might simply be events you might not expect to happen. These events are called *exceptions*. Exceptions refer to errors that are detected while the script is running. When an exception is detected, the script stops running unless the detection is handled properly. Exceptions are said to be *thrown*. If the exception is handled properly—that is, it is *caught*—the program can continue running. Examples of exceptions and proper error-handling techniques are covered later in this chapter.

11.4 Using debugging

When code results in exception errors or logic errors, you may need to look more closely at the values of variables in your script. This can be accomplished using a debugging procedure. Debugging is a methodological process for finding errors in your script. There are a number of possible debugging procedures, from very basic to more complex. Debugging procedures include the following:

- Carefully reviewing the content of error messages
- Adding print statements to your script
- Selectively commenting out code
- Using a Python debugger

Each of these approaches is reviewed in this section in more detail. Keep in mind that most of the time, debugging does not tell you *why* a script did not run properly, but it will tell you *where*—that is, on which line of code it failed. Typically, you still have to figure out why the error occurred.

Carefully reviewing the content of error messages

Error messages generated by ArcPy are usually informative. Consider the following example:

```
import arcpy
arcpy.env.workspace = "C:/Data"
infcs = ["streams.shp", "floodzone.shp"]
outfc = "union.shp"
arcpy.Union_analysis(infcs, outfc)
```

This script carries out a union between two input feature classes, which are entered as a list. The result should be a new output feature class in the same workspace. The error message in PythonWin is as follows:

```
ExecuteError: Failed to execute. Parameters are not valid.
ERROR 000366: Invalid geometry type
Failed to execute (Union).
```

This is a specific error message produced by ArcPy, also referred to as an `ExecuteError` exception. The message is useful because it includes the statement: `Invalid geometry type`. Closer inspection of the input feature classes reveals that one of the inputs (streams.shp) is a polyline feature class, and the Union tool works with polygon features only. So the error message does not tell you exactly what is wrong (that is, it did not say that streams.shp is the geometry type polyline and that the Union tool does not accept this geometry type), but it points you in the right direction.

Not all error messages are as useful. Consider the following script:

```
import arcpy
arcpy.env.workspace = "C:/mydata"
infcs = ["streams.shp", "floodzone.shp"]
outfc = "union.shp"
arcpy.Union_analysis(infcs, outfc)
```

>>> TIP

When a specific error code is included in the error message, such as ERROR 000366, you can learn more about it in ArcGIS Desktop Help. In Help, go to Geoprocessing > Tool errors and warnings, and browse to the specific error by number.

This is, in fact, the same script, but it uses a different workspace (C:\mydata), which does not exist. The error message in PythonWin is as follows:

```
Traceback (most recent call last):
  File "C:\Python27\ArcGIS10.1\Lib\site-packages\PythonWin\pywin\
framework\scriptutils.py", line 325, in RunScript
    exec codeObject in __main__.__dict__
  File "C:\data\myunion.py", line 5, in <module>
    arcpy.Union_analysis(infcs, outfc)
  File "C:\Program Files (x86)\ArcGIS\Desktop10.1\arcpy\arcpy\
analysis.py", line 574, in Union
    raise e
ExecuteError: Failed to execute. Parameters are not valid.
ERROR 000366: Invalid geometry type
Failed to execute (Union).
```

Note: When the same code is run in the Python window in ArcGIS, a different error message results:

```
Runtime error <class 'ArcGISscripting.ExecuteError'>: ERROR 000732: Input
Features: Dataset streams.shp #;floodzone.shp # does not exist or is not
supported
```

The PythonWin error message is rather misleading. It appears to suggest that the error is on line 5 of the code (where the union is carried out) and that there is an issue with the geometry. The error is, in fact, on line 2 where an invalid workspace is defined, and the invalid geometry message results from the fact that no feature classes could be obtained from the nonexistent workspace. Unfortunately, the error-reporting functions can't always report a more specific error message, such as `Workspace does not exist`.

Carefully examining error messages can be useful since they may, in fact, hold the answer to how to fix a problem. But don't stare yourself blind poring over them because the error may be something quite different and the error messages could prove misleading.

Adding print statements to your script

When you have multiple lines of code that contain geoprocessing tools, it may not always be clear on which line an error occurred. In such cases, it may be useful to add `print` statements after each geoprocessing tool or other important steps to confirm they were run successfully. Consider the following code:

```
import arcpy
from arcpy import env
env.overwriteOutput = True
env.workspace = "C:/Data"
arcpy.Buffer_analysis("roads.shp", "buffer.shp", "1000 METERS")
print "Buffer completed"
arcpy.Erase_analysis("buffer.shp", "zone.shp", "erase.shp")
print "Erase completed"
arcpy.Clip_analysis("erase.shp", "wetlands.shp", "clip.shp")
print "Clip completed"
```

Even if the error message is cryptic and not informative, the `print` statements will illustrate which steps have been completed. The error can most likely be traced to the block of code just prior to the `print` statement that did not execute.

`Print` statements can be effective, but they are most useful when you already have a good idea of what might be causing the error. One of the downsides of using `print` statements is that they need to be cleaned up once the error has been fixed, which can be a substantial amount of work.

Selectively commenting out code

You can selectively *comment out* code to see if removing certain lines eliminates the error. If your script has a typical sequential workflow, you would work from the bottom up. For example, the following code illustrates how the lower lines of code are commented out, using double number signs (##), to isolate the error:

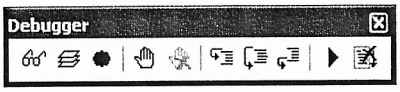
```
import arcpy
from arcpy import env
env.overwriteOutput = True
env.workspace = "C:/Data"
arcpy.Buffer_analysis("roads.shp", "buffer.shp", "1000 METERS")
##arcpy.Erase_analysis("buffer.shp", "streams.shp", "erase.shp")
##arcpy.Clip_analysis("erase.shp", "wetlands.shp", "clip.shp")
```

As with adding print statements, this approach of commenting out lines of code does not identify why an error occurs, but only helps you to isolate where it occurs.

Using a Python debugger

Another, more systematic, approach to debugging code is to use a Python debugger. A debugger is a tool that allows you to step through your code line by line, to place breakpoints in your code to examine the conditions at that point, and to follow how certain variables change throughout your code. Python has a built-in debugger module called `pdb`. It is a bit cumbersome because it lacks a user interface. However, Python editors such as IDLE and PythonWin include a solid debugging environment. In the next example that follows, the PythonWin debugger is used.

In PythonWin, you can turn the Debugger toolbar on and off by clicking View > Toolbars > Debugging from the menu bar.



The tools on the Debugger toolbar are briefly described in table 11.1.

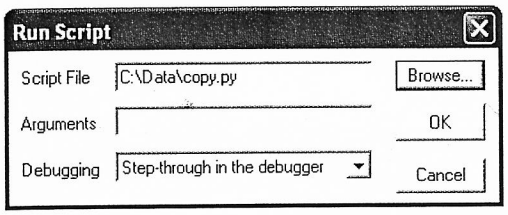
Table 11.1 Tools on the Debugger toolbar

	Watch	Makes the Watch window visible, which allows you to keep track of the values of specifically defined variables in a script
	Stack view	Makes the Stack window visible, which keeps track of all variables in a script
	Breakpoint list	Makes the Breakpoint list window visible, which lists all the current breakpoints in a script
	Toggle Breakpoint	Turns a breakpoint on or off at the cursor location in the current script
	Clear All Breakpoints	Removes all breakpoints from the current script
	Step (or Step Into)	Runs the current line of code and moves to the next line, which can be in a different module, function, or method
	Step Over	Runs the current line of code, and if it includes a Python module, function, or method, it runs it, and then returns to the next line of code in the original script
	Step Out	Runs the current Python module, function, or method, and then returns to the next line of code in the original script
	Go	Runs a script until the next breakpoint or until the last line of code is reached
	Close	Stops the execution of code and exits the debugger to return to the script

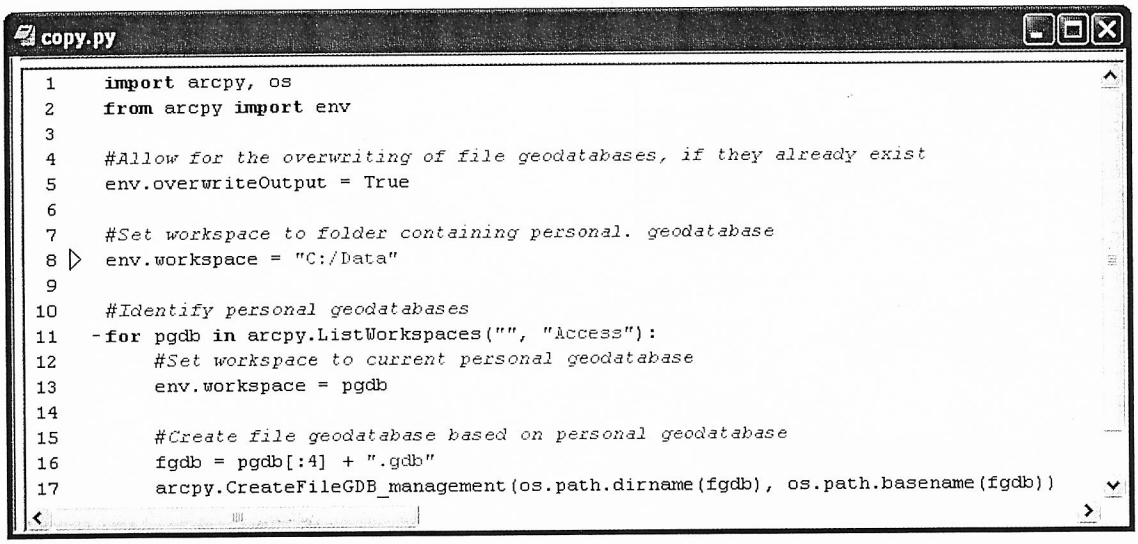
A typical debugging procedure in PythonWin is as follows:

1. Check for any syntax errors and save the script.

2. Run the script, but this time select a Debugging option on the Run Script dialog box—for example, "Step-through in the debugger".

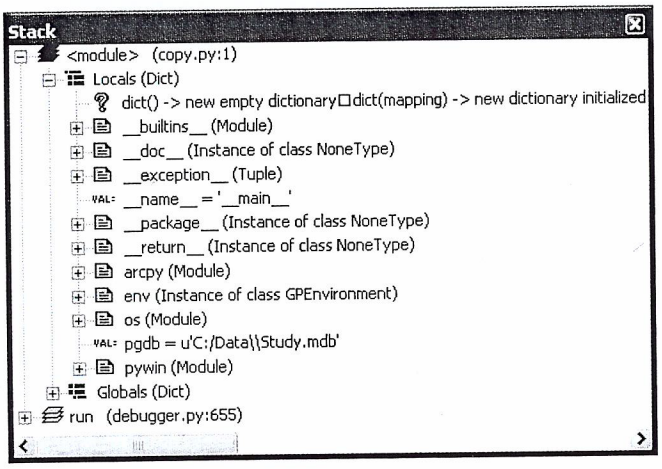


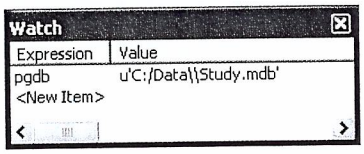
3. Use the Step tool to go through your script line by line. Keep track of any error messages in the Interactive Window. The yellow arrow that appears in the window indicates the current line of code.



4. Use the Step Over and Step Out tools to skip ahead. For example, if the current line of code contains a call to a different module, function, or method, using the Step tool results in stepping into that procedure. Using the Step Over tool will run the line of code without stepping into that procedure—you are stepping over the details of that procedure. Once you step into a procedure, you can use the Step Out tool to step out of the procedure without stepping through the rest of the lines of code. This allows you to fast-forward and return to the next line of code that called the procedure.

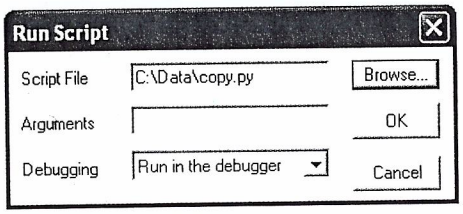
5. While stepping through the script code, open the Watch and Stack windows to keep track of variables. The Stack window keeps track of all the variables and the Watch window keeps track of only the variables you specify manually.





6. When you find an error, use the Close tool to stop the execution of the script. Then fix the error and run the script again.

When scripts get longer, going through a script line by line can be a bit cumbersome. Instead, you can place breakpoints in the script using the Toggle Breakpoint tool. The next time you run the script, you can select the "Run in the debugger" option.



Then the debugger will stop the script at only the predefined breakpoints and run the lines in between breakpoints in one step instead of stopping at every line.

copy.py

```
1   import arcpy, os
2   from arcpy import env
3
4   #Allow for the overwriting of file geodatabases, if they already exist
5   env.overwriteOutput = True
6
7   #Set workspace to folder containing personal. geodatabase
8   env.workspace = "C:/Data"
9
10  #Identify personal geodatabases
11  for pgdb in arcpy.ListWorkspaces("", "Access"):
12      #Set workspace to current personal geodatabase
13      env.workspace = pgdb
14
15      #Create file geodatabase based on personal geodatabase
16      fgdb = pgdb[:4] + ".gdb"
17      arcpy.CreateFileGDB_management(os.path.dirname(fgdb), os.path.basename(fgdb))
```

The breakpoints can be turned on and off by placing the cursor in the desired line of code and using the Toggle Breakpoint tool. To clear all the breakpoints in the current script, use the Clear All Breakpoints tool.

11.5 Using debugging tips and tricks

Following are some general tips and tricks that will help you to debug your scripts:

- Remember that ArcGIS for Desktop applications often place a lock on a file, which may prevent a script from overwriting the file.
- When working with very large files, first try your code on a small file with similar properties.
- Watch where the values of variables are changing by inserting print statements or breakpoints in the code.
- Place breakpoints inside blocks of code where repetition should be occurring.
- If PythonWin does not stop running while you are debugging, you can interrupt the code by right-clicking the PythonWin icon in the notification area, at the far-right corner of the taskbar, and then clicking "Break into running code". This will result in a `KeyboardInterrupt` exception in the Interactive Window without closing PythonWin. ➔

11.6 Error handling for exceptions

Although debugging procedures can contribute to writing correct code, exception errors are still likely to occur in your scripts. Exceptions refer to errors that are detected as the script is running. One key reason for this is that many scripts rely on user input, and you can't always control the input other users will provide. Well-written scripts, therefore, include error-handling procedures to handle exceptions. Error-handling procedures are written to avoid having a script fail and not provide meaningful feedback.

To handle exceptions, you could use conditional statements to check for certain scenarios, which is analogous to using an `if` statement. You have already encountered some in previous chapters. For example, the existence of a path can be determined in Python using a built-in Python function such as `os.path.exists`. For catalog paths, you can use the `Exists` function to determine whether data exists. For example, the following code determines whether a shapefile exists:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data"
shape_exist = arcpy.Exists("streams.shp")
print shape_exist
```

The `Exists` function can be used for feature classes, tables, datasets, shapefiles, workspaces, layers, and files in the current workspace. The function returns a Boolean value indicating whether the element exists.

Besides determining whether data exists, you can determine whether the data is the right type by using the `Describe` function. For example, if your script requires a feature class, you can use the `datsetType` property to determine whether it is a feature class.

Writing conditional statements for every possible error is tedious. And it is impossible to foresee every error. In the example code earlier in this section, you would have to check the following: (1) whether the workspace is valid, (2) whether there is at least one feature class in the workspace, and (3) whether there is a feature class with at least one feature. This could easily double the code in the script.

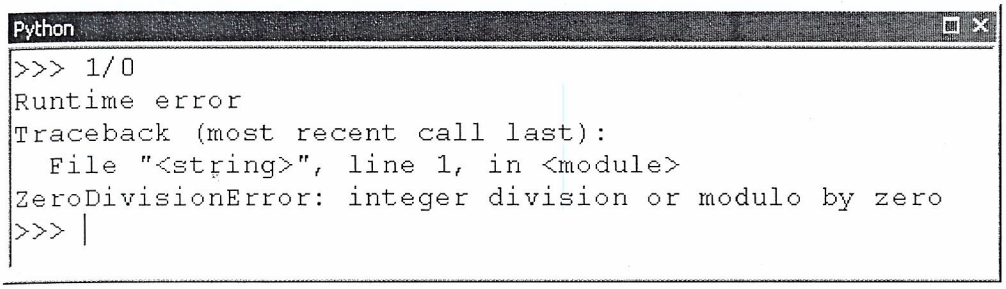
There are two strategies to check for errors and report them in a meaningful manner:

1. Use Python exception objects inside `try-except` statements.
2. Report messages using the ArcPy messaging functions.

A powerful alternative to conditional statements is Python exception objects. When Python encounters an error, it *raises*, or *throws*, an exception. This typically means the script stops running. If such an exception object is not

handled, or *caught*, the script terminates with a runtime error, sometimes also referred to as a *traceback*.

Consider the simple example of trying to divide by zero. In the Python window, it results in a runtime error, as shown in the figure.



The following sections will illustrate how exceptions are raised and how the `try-except` statement can be used to effectively trap errors.

11.7 Raising exceptions

Exceptions are raised automatically when something goes wrong. You can also raise exceptions yourself by using the `raise` statement. You can raise a generic exception using the `raise Exception` statement, as follows:

```
>>> raise Exception
Runtime error
Exception
```

You can also add a specific message, as follows:

```
>>> raise Exception("invalid workspace")
Runtime error
Exception: invalid workspace
```

There are many different types of exceptions. You can view all of them by importing the `exceptions` module and using the `dir` function to list all of them:

```
>>> import exceptions
>>> dir(exceptions)
```

Running this code results in a long printout (not shown in entirety here):

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ➔
➔ 'BufferError', 'BytesWarning' ...]
```

Each of these exceptions can be used in the `raise` statement. For example:

```
>>> raise ValueError
Runtime error
ValueError
```

This example is a *named* exception—that is, the specific exception is called by name. Named exceptions allow a script to handle specific exceptions in different ways, which can be beneficial. Using the generic `Exception` is referred to as an *unnamed* exception.

A complete description of built-in Python exceptions can be found in the Python documentation. In the documentation, go to Library Reference and browse to section 6, "Built-in Exceptions." It also includes a hierarchy of errors: for example, `ZeroDivisionError` is one of several types of arithmetic errors (`ArithmeticError`).

>>> TIP

The Python documentation is installed in the same program folder as ArcGIS. For a typical installation, you can find the documentation by clicking the Start button on the taskbar, and then, on the Start menu, clicking All Programs > ArcGIS > Python 2.7 > Python Manuals.

Python v2.7 documentation » The Python Standard Library » previous | next | modules | index

6. Built-in Exceptions

Exceptions should be class objects. The exceptions are defined in the module **exceptions**. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the **exceptions** module.

For class exceptions, in a **try** statement with an **except** clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the **raise** statement. If the exception class is derived from the standard root class **BaseException**, the associated value is present as the exception instance's **args** attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be sub-classed to define new exceptions; programmers are encouraged to at least derive new exceptions from the **Exception** class and not **BaseException**. More information on defining exceptions is available in the Python Tutorial under *User-defined Exceptions*.

The following exceptions are only used as base classes for other exceptions.

exception **BaseException**

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that use **Exception**). If **str()** or **unicode()** is called on an instance of this class, the representation of the argument(s) to the instance are returned or the empty string when there were no arguments. All arguments are stored in **args** as a tuple.

11.8 Handling exceptions

Exceptions in a script can be handled using a `try-except` statement. Handling exceptions is often called *trapping*, or *catching*, the exceptions. When an exception is properly handled, the script does not produce a runtime error but instead reports a more meaningful error message to the user. This means the error is trapped, or caught, before it can cause a runtime error.

Consider the following script that divides two user-supplied numbers:

```
x = input("First number: ")
y = input("Second number: ")
print x/y
```

The script will work fine until zero (0) is entered as the second number, resulting in the following error message:

```
First number: 100
Second number: 0
Traceback (most recent call last):
  File " division.py", line 3, in <module>
    print x/y
ZeroDivisionError: integer division or modulo by zero
```

The `try-except` statement can be used to trap this exception and provide additional error handling, as follows:

```
try:
    x = input("First number: ")
    y = input("Second number: ")
    print x/y
except ZeroDivisionError:
    print "The second number cannot be zero."
```

Notice the structure of the `try-except` statement. The first line of code consists of only the `try` statement, followed by a colon (:). Next is a block of indented code with the procedure you want to carry out. Then comes the `except` statement, which includes a specific exception, followed by a colon (:). Next is a block of indented code that will be carried out if the specific exception is raised. The exception `ZeroDivisionError` is a named exception.

In this example, a simple `if` statement might have been more effective to determine whether the value of y is zero (0). However, for more elaborate code, you might need many such `if` statements, and a single `try-except` statement will be sufficient to trap the error.

Multiple `except` statements can be used to catch different named exceptions. For example:

```
try:
    x = input("First number: ")
    y = input("Second number: ")
    print x/y
except ZeroDivisionError:
    print "The second number cannot be zero."
except TypeError:
    print "Only numbers are valid entries."
```

You can also catch multiple exceptions with a single block of code by specifying them as a tuple:

```
except (ZeroDivisionError, TypeError):
    print "Your entries were not valid."
```

In this case, the error handling is not specific to the type of exception and only a single message is printed, no matter what type of error caused the exception. The error message, in this case, is less specific because it describes several exceptions.

The exception object itself can also be called by providing an additional argument:

```
except (ZeroDivisionError, TypeError) as e:
    print e
```

Running this code allows you to catch the exception object itself and you can print it to see what happened rather than printing a custom error message.

It can be difficult sometimes to predict all the types of exceptions that might occur. Especially in a script that relies on user input, you may not be able to foresee all the possible scenarios. So to catch all the exceptions, no matter what type, you can simply omit the exception class from the `except` statement, as follows:

```
try:
    x = input("First number: ")
    y = input("Second number: ")
    print x/y
except Exception as e:
    print e
```

In this example, the exception is unnamed.

The `try-except` statement can also include an `else` statement, similar to a conditional statement. For example:

```
while True:
    try:
        x = input("First number: ")
        y = input("Second number: ")
        print x/y
    except:
        print "Please try again."
    else:
        break
```

In this example, the `try` block of code is repeated in a `while` loop when an exception is raised. The loop is broken by the `break` statement in the `else` statement only when no exception is raised.

One more addition to the `try-except` statement is the `finally` statement. Whatever the result of previous `try`, `except`, or `else` blocks of code, the `finally` block of code will always be executed. This block typically consists of clean-up tasks and could include checking in licenses or deleting references to map documents.

11.9 Handling geoprocessing exceptions

So far, the exceptions raised have been quite general. A Python script can, of course, fail for many reasons that are not specifically related to a geoprocessing tool as the previous examples illustrate. However, because errors related to geoprocessing tools are somewhat unusual in nature, they warrant more attention.

You can think of errors as falling into two categories: geoprocessing errors and everything else. When a geoprocessing tool writes an error message, ArcPy generates a system error. Specifically, when a geoprocessing tool fails to run, it throws an `ExecuteError` exception, which can be used to handle specific geoprocessing errors. It is not one of the built-in Python exception classes, but it is generated by ArcPy and thus the `arcpy.ExecuteError` class has to be used.

Consider this example:

```
import arcpy
arcpy.env.workspace = "C:/Data"
in_features = "streams.shp"
out_features = "streams.shp"
try:
    arcpy.CopyFeatures_management(in_features, out_features)
except arcpy.ExecuteError:
    print arcpy.GetMessages(2)
except:
    print "There has been a nontool error."
```

The Copy Features tool generates an error because the input and output feature classes cannot be the same, as follows:

```
Failed to execute. Parameters are not valid.
ERROR 000725: Output Feature Class: Dataset C:/Data\zip.shp already ➔
➔ exists.
Failed to execute (CopyFeatures).
```

In the example code, the first `except` statement traps any geoprocessing errors, and the second `except` statement traps any nongeoprocessing errors. This example illustrates how both named and unnamed exceptions can be used in the same script. It is important to first check the named exceptions, such as `except arcpy.ExecuteError`, and then the unnamed exceptions. If the unnamed exceptions were checked first, the statement would catch all exceptions, including any `arcpy.ExecuteError` exceptions. This would mean you would never know whether a named exception (that you put in the script) occurred or not.

In larger scripts, it can be difficult to determine the precise location of an error. You can use the Python `traceback` module to isolate the location and cause of an error.

The `traceback` structure is as follows:

```
try:
    import arcpy
    import sys
    import traceback
    <block of code including geoprocessing tools>
except:
    tb = sys.exc_info()[2]
    tbinfo = traceback.format_tb(tb)[0]
    pymsg = "PYTHON ERRORS:\nTraceback info:\n" + tbinfo + "\nError ➔
➔ Info:\n" + str(sys.exc_type) + ":" + str(sys.exc_value) + "\n"
    arcpy.AddError(pymsg)
    msgs = "ArcPy ERRORS:\n" + arcpy.GetMessages(2) + "\n"
    arcpy.AddError(msgs)
    print pymsg + "\n"
    print msgs
```

In this code, two types of errors are trapped: geoprocessing errors and all other types of errors. The geoprocessing errors are obtained using the ArcPy `GetMessages` function. The errors are returned for use in a script tool (`AddError`) and also printed to the standard Python output (`print`). All other types of errors are retrieved using the `traceback` module. Some formatting is applied and the errors are returned for use in a script tool and printed to the standard Python output.

finally statement. In this example, a custom exception class is created to handle a license error. A license is checked out in the try code block and the license is checked in as part of the finally code block. This ensures the license is checked in, no matter the outcome of running the earlier code blocks, as follows:

```
class LicenseError(Exception):
    pass
import arcpy
from arcpy import env
try:
    if arcpy.CheckExtension("3D") == "Available":
        arcpy.CheckOutExtension("3D")
    else:
        raise LicenseError
    env.workspace = "C:/raster"
    arcpy.Slope_3d("elevation", "slope")
except LicenseError:
    print "3D license is unavailable"
except:
    print arcpy.GetMessages(2)
finally:
    arcpy.CheckInExtension("Spatial")
```

Using the try-except statement for error trapping is very common. The ExecuteError exception class is useful, but in practice, most scripts rely on the simple but effective try-except statement without using specific exception classes.

Sometimes, you will see an entire script wrapped in a try-except statement. It would look something like the following structure in which the try code block could contain hundreds of lines of code:

```
try:
    import arcpy
    import traceback
    ##multiple lines of code here

except:
    tb = sys.exc_info()[2]
    tbinfo = traceback.format_tb(tb)[0]
    pymsg = "PYTHON ERRORS:\nTraceback info:\n" + tbinfo + "\nError
➥ Info:\n" + str(sys.exc_info()[1])
    msgs = "ArcPy ERRORS:\n" + arcpy.GetMessages(2) + "\n"
    arcpy.AddError(pymsg)
    arcpy.AddError(msgs)
```

11.10 Using other error-handling methods

In addition to a `try-except` statement for trapping errors in scripts, several other error-handling methods can be used. Some of them are covered in earlier chapters but warrant further mention here:

- Validating table and field names using the `ValidateTableName` and `ValidateFieldName` functions, respectively (chapter 7).
- Checking for licenses for products using the `CheckProduct` function and for extensions using the `CheckExtension` function (chapter 5).
- Checking for schema locks—many geoprocessing tools will not run properly if schema locks exist on the input.

11.11 Watching for common errors

Following are a number of common errors to look out for when you are scanning your scripts and examining your data.

Common Python code errors

- Simple spelling mistakes
- Forgetting to import modules, such as `arcpy`, `os`, or `sys`
- Case sensitivity—for example, `mylist` versus `myList`
- Paths—for example, using a single backslash (\), such as C:\Data\streams.shp
- Forgetting colons (:) after statements (`for`, `while`, `else`, `try`, `except`)
- Incorrect or inconsistent indentation
- Conditional (==) versus assignment (=) statements

Common geoprocessing-related errors

- Forgetting to determine whether data exists. A small typo in the name of a workspace or feature class will cause a tool to fail. Always double-check that the inputs to a script exist.

- Forgetting to check for overwriting output. The default setting is not to overwrite outputs, so unless this option is specifically cleared, a tool that attempts to overwrite output will not run. A very common scenario is to run a script and it works, but when you run it a second time, it fails—fixing this could be as simple as setting the `overwriteOutput` property of the `env` class to `True`.

- Data is being used in another application. You may be trying to run a script, but it will not run because you are also using the data in ArcMap or ArcCatalog—this is very common because often you are exploring the data that is going to be used in the script. Closing these applications and trying the script again may resolve a script error.

- Not checking the properties of parameters and objects returned by tools. For example, it may sound logical that the Get Count tool produces a count—that is, a number. It actually returns a result object that is printed to the Results window, so you have to use the `getOutput` method to obtain this count. Similarly, distinctions between feature classes and feature layers may seem somewhat trivial, but they may be just the difference between proper tool execution and failure. Carefully examine tool syntax and determine the exact nature of the inputs and outputs.

It is worth noting that many geoprocessing-related errors can be prevented when using script tools. Building a script tool includes validation for preventing invalid parameters. This is covered in chapter 13.

Some of these suggestions may appear rather rudimentary, but the solutions can often be simple if you only knew where to look for them. The syntax of a good Python geoprocessing script is often relatively simple, which is part of the beauty of using Python.

>>> TIP

Remember that geoprocessing scripts don't have to follow Python coding logic alone. They must also obey the rules of the ArcGIS geoprocessing framework.

Points to remember

- Errors in geoprocessing scripts are bound to happen. Although syntax errors are relatively easy to catch, your script may contain other types of errors that prevent proper script execution. Scripts can be made more robust by incorporating error-handling procedures.

- Various debugging methods exist. Relatively simple approaches include carefully examining error messages, adding print statements to the code to review intermediate results, and selectively commenting out code. If these methods are not sufficient to identify and fix errors, a Python debugger can be used such as the PythonWin Debugger. A debugger allows you to carefully step through the code line by line to review error messages and examine the state of variables. Breakpoints can be added to step through larger blocks of code.

- Any debugging procedure will typically identify where the error occurs but not exactly why it occurs. It is therefore good practice to always be aware of common errors, including Python coding errors and ArcGIS geoprocessing errors.

- Basic error-handling procedures include checking whether data exists, determining whether data inputs are the right type, checking for licenses and extensions, and validating table and field names. Typically, an `if` statement is used for this type of error handling.

- It is nearly impossible to anticipate every possible type of errors, and code that checks for such errors would become too cumbersome to write. Whenever something goes wrong in a script, an exception is automatically raised. These exceptions can be trapped using a `try-except` statement. This type of statement makes it possible to identify the type of error or else specific errors. Customized error-handling procedures can be implemented based on the nature of the error. Additional statements, including `else` and `finally`, can be added to the `try-except` statement to ensure efficient error trapping.

- Error messages can be very useful for identifying the nature of the error and how to fix the script. These include both general Python messages and error messages resulting from the ArcPy `ExecuteError` exception class.