Chapter 14
Error Handling

Abstract Data corruption and locking can cause geoprocessing scripts to crash and throw exceptions. Other influences, such as user input values, can also cause scripts to crash. The topic of this chapter is using error handling structures to control script behavior when exceptions occur. Error handling can suppress those alarming trace-back messages that exceptions throw and provide a smoother way to proceed.

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

- Handle potential errors from user input.
- Use error handling keywords, `try` and `except`.
- Anticipate named exceptions.
- Print geoprocessing messages.
- Avoid crashing in the midst of a loop.
- Identify when to use error handling instead of conditional blocks.

Chapter 13 discussed syntax errors, exceptions, and logic errors and how to debug logic errors. The focus of debugging is to remove errors in the code. The programmer needs to remove syntax errors and test the code to expose and remove logic errors. Exception errors, however, can come in two forms. Some exceptions are due to errors in the code; Others are due to outside factors, such as user input.

In the first case, the programmer needs to repair them. In the second case, the programmer should write code to anticipate and handle the error. This chapter presents Python structures for handling exceptions. To clarify the difference between exceptions that should be repaired and exceptions that should be handled, take the following map algebra script as an example:

```
# multRast.py
import arcpy, sys

# Set multiplication factor
factor = float(sys.argv[1])

arcpy.env.overwriteOutput = True
arcpy.env.worspace = 'C:/gispy/data/ch14/rastTester.gdb'
arcpy.CheckOutExtension('Spatial')

# Get raster list & multiply each by a factor.
rasterList = arcpy.ListRasters()
for rasterImage in rasterList:
    rasterObj = arcpy.Raster(rasterImage)
    rastMult = rasterObj * factor
    rastMult.save(rasterImage + '_out')
del rastMult
```

'multRast.py' should apply a constant multiplier to all the rasters in the directory, but it throws the following exception:

```
for rasterImage in rasterList:
TypeError: 'NoneType' object is not iterable
```

This error occurs because of a sneaky typo that caused a series of cascading events leading to this exception. Have you spotted it? The problem occurs because 'workspace' is spelled wrong in the following line of code:

```
arcpy.env.worspace = 'C:/gispy/data/ch14/rastTester.gdb'
```

The script didn't give an error, when it reached this line. It just silently created a new (useless) variable `arcpy.env.worspace`. Since the value of `arcpy.env.workspace` was not set by this script, the default value—empty string—is used for the `arcpy` workspace setting.

When `ListRasters` is called without setting the workspace, it returns the Python built-in constant `None`, meaning the `rasterList` variable is set to `None`. Then the script tries to loop over the raster list, which is when the exception is finally thrown ('NoneType' object is not iterable).

This exception must be avoided by correcting the typo. However, other exceptions could occur due to user input. This script is designed to take a numerical argument. But suppose users are dealing with rasters representing elevation in meters. They might try to run the script with an argument such as "0.001 meters". Given this input, the script throws the following exception:

```
factor = float(sys.argv[1])
ValueError: invalid literal for float(): 0.001 meters
```

Python can't cast the input to a float, since it contains non-numeric components. This is the other flavor of exception—it is not occurring because of an error in the code. It is due to an outside input that is beyond the control of the programmer. In this example, the script stops running abruptly when the exception is thrown. However, the script can be modified to handle this exception more gracefully, that is, it can be *caught*. This chapter starts with some simple scripts to introduce Python error handling syntax and then demonstrates geoprocessing script applications.

14.1 try/except Structures

Exceptions can be caught by using Python `try` and `except` keywords to group lines of code into blocks. The code that could generate the exception is placed within the `try` block and contingency code (what to do in case of trouble) can be placed inside an `except` block. `try` and `except` blocks require colons and indentation, just like other Python code blocks such as conditional blocks and looping blocks. When the exceptions are caught, no traceback messages are printed in the Interactive Window (unless the code is run in debug mode); Instead the script can print a more intelligible message or it can perform some alternative action. For example, when a user passes "5 meters" as an argument into the following script, they receive a clear message and the code exits gracefully:

```
# doubleMyNumber.py
import sys

try:
    number = float(sys.argv[1])
    product = 2*number
    print 'The doubled number is {0}'.format(product)
except:
    print 'An error occurred.'
    print 'Please enter a numerical argument.'
print 'Good bye!'
```

Here's how it works:

- If an exception occurs in the `try` block, the rest of the `try` block is skipped and execution jumps to the `except` block. For example, if the input is "5 meters", the script prints the following:

```
>>> An error occurred.
Please enter a numerical argument.
Good bye!
```

- If no exceptions occur, the except block is skipped. For example, if the input is 5, the script prints the following:

```
The doubled number is 10.0
Good bye!
```

When any exception is thrown, the execution moves to the `except` block. This means the same action is taken regardless of whether the input is "5 meters" or the user does not supply an argument. The behavior in case of exceptions can be controlled more precisely by using named exceptions.

14.1.1 Using Named Exceptions

Traceback errors report the name of the exception, such as a `TypeError` or `ValueError`. Exceptions names can be used with `except` blocks to provide special handling for specific errors by placing the name of the exception behind the `except` keyword. For example, the following code catches a `ValueError`:

```
# doubleMyNumberV2.py
import sys

try:
    number = float(sys.argv[1])
    product = 2*number
    print 'The doubled number is {0}'.format(product)
except ValueError:
    print 'Input must be numerical.'
print 'Good bye!'
```

This code catches ValueError exceptions. If the user enters "5 meters", the script prints:

```
>>> Input must be numerical. Good bye!
```

Scripts use named exceptions to anticipate particular vulnerabilities. To do so, the programmer needs to determine which name to use. The 'Python Library Reference' can be used to look up built-in exceptions; But creating code samples that cause an exception is also a good approach. For example, the following code throws a `ZeroDivisionError` exception:

```
>>> 1/0
Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

And the following code throws a `ValueError` exception:

```
>>> float('5 meters')
Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
ValueError: invalid literal for float(): 5 meters
```

The examples demonstrate that a `ZeroDivisionError` should be used as the named exception to catch errors where division by zero might occur and a `ValueError` should be used as the named exception to catch errors generated by calling the built-in `float` function on a non-numeric input.

The `except ValueError` statement in 'doubleMyNumbersV2.py' handles `ValueError` exceptions only. Exceptions with other names will not be caught in this `except` block. For example, if the user does not supply an argument, the script prints an `IndexError` when it tries to read the user argument:

```
number = float(sys.argv[1])
IndexError: list index out of range
```

To handle more than one type of named error, a script can use multiple `except` blocks.

14.1.2 Multiple except Blocks

When named errors are used, a `try` block can have multiple `except` blocks. Then, if an error occurs, the execution jumps to the corresponding named exception block. For example, the following script uses two named `except` blocks:

```
# slopeTry.py
import sys

rise = sys.argv[1]
run = sys.argv[2]
```

```
try:
    print 'Rise: {0} Run: {1}'.format(rise, run)
    slope = float(rise)/float(run)
    print 'Slope = rise/run'
except ZeroDivisionError:
    slope = 'Undefined (line is vertical)'
except ValueError:
    print 'Usage: <numeric rise> <numeric run>'
    slope = 'Not found'

print 'Slope:', slope
```

Here's how this one works:

- If no exceptions occur, the `try` block is completed and both the except blocks are skipped. For example, if the input is 1 3, the script prints the following:

```
>>> Rise: 1 Run: 3
Slope = rise/run
Slope: 0.333333333333
```

- If a `ZeroDivisionError` occurs in the `try` block, the rest of the `try` block is skipped and execution jumps to the `except ZeroDivisionError` block. For example, if the input is 5 0, the script prints the following:

```
>>> Rise: 5 Run: 0
Slope: Undefined (line is vertical)
```

- The script prints the statement before the division and then when the division fails, the `ZeroDivisionError` block sets the slope to '`Undefined (line is vertical)`'.
- If a `ValueError` occurs in the `try` block, the rest of the `try` block is skipped and execution jumps to the `except ValueError` block. For example, if the input is one three, the script prints the following:

```
>>> Rise: one Run: three
Usage: <numeric rise> <numeric run>
Slope: Not found
```

- The script prints the statement before the division and then when the conversion to float fails, execution jumps to the `ValueError` block and prints script usage instructions and sets the slope to '`Not found`'.

'slopeTry.py' handles three cases: successful slope computation, division by zero, and failure to cast to float. The last line of code prints the `slope` variable. Since the exceptions were caught, this line is executed in all three cases, so the script contains an assignment statement for `slope` in all three blocks.

'slopeTry.py' demonstrates using two named exception blocks. In fact, many other variations are possible as long as they comply with the following syntax rules:

- Any number of named `except` blocks can be used with one `try` block.
- At most one unnamed `except` block can be used with a single `try` block.
- An unnamed `except` block can be used along with named `except` blocks.
- When an unnamed `except` block is used with named `except` blocks, the unnamed except is placed last.
- A `try` block needs at least one `except` block (or a `finally` block, not discussed here, can be used as an alternative to an `except` block).

14.1.3 Error Handling Gotcha

Catching exceptions can shield users from encountering mysterious traceback messages. However, suppressing the Traceback messages can also potentially lead to pitfalls during code development. For example, the try/except structure in the following code hides an error that should be corrected by the programmer:

```
# cubeMyNumber.py
import sys
try:
    number = float(sys.argv[1])
    cube = number**3
    print 'The cubed number is {0}'.format()  # missing argument
except:
    print 'Input must be numerical.'
print 'Good bye!'
```

This script takes one numeric argument; however, even if the user provides a valid argument, an exception is raised. For example, if the input is 5, the script prints the following:

```
>>> Input must be numerical argument.
Good bye!
```

Something is wrong with the code, but in this case, suppressing the traceback message makes the problem more difficult to find. Can you find what is really wrong with the code? The message in the exception block anticipates that the user enters a non-numeric argument, but some other exception is causing the code to jump to the `except` block and print the 'numerical argument' message which doesn't help to uncover the problem. The Python traceback module can be used during code development to overcome this glitch.

The Python built-in `traceback` module can force traceback messages to print. The traceback `print_exc` method prints the most recent exception. To use the `print_exc` traceback method, import the `traceback` module and then call the `print_exc` method inside the except clause as in the following code:

```
# cubeMyNumberV2.py
import sys, traceback
try:
    number = float(sys.argv[1])
    cube = number**3
    print 'The cubed number is {0}'.format()
except:
    print 'Input must be numerical.'
    traceback.print_exc()
print 'Good bye!'
```

If this script is run with a valid argument, the raised exception is printed. For example, if the input is 5, the script prints the following:

```
Input must be numerical.
Traceback (most recent call last):
File 'C:\\example_scripts\cubeMyNumberV2.py', line 6, in <module>
print 'The doubled number is {0}'.format() #missing arg
IndexError: tuple index out of range
Good bye!
```

This traceback helps us to see that the string `format` is called incorrectly; it is missing a needed argument and throws an `IndexError` exception. It is true that an error occurred, but this is not the kind of error that should be handled with try/except blocks in the script. It should instead be repaired by the programmer. Line 6 should be corrected as follows:

```
print 'The cubed number is {0}'.format(cube)
```

This 'gotcha' is less likely to occur when using named exceptions, although, as 'cubeMyNumber.py' demonstrates, confusion can still occur.

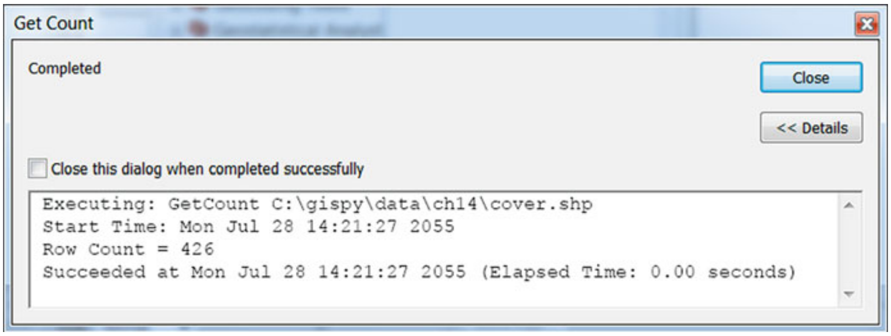
Note The amount of code placed inside an exception handling block should be minimal so that other exceptions are not masked.

14.2 Geoprocessing and Error Handling

Now that you know the basic approach for Python error handling, you can apply it to geoprocessing. This section describes how to get feedback from geoprocessing calls, how to use a named exception to catch geoprocessing errors, and how to use these components within a batch geoprocessing script.

14.2.1 Getting Geoprocessing Messages

When a tool is run from ArcToolbox, a Geoprocessing Window is opened. This window shows progress as the tool works and when it stops running, it displays a success or failure message, as in the following screen shot of a successful Get Count (Data Management) tool run:



```
Executing: GetCount C:\gispy\data\ch14\cover.shp
Start Time: Mon Jul 28 14:21:27 2055
Row Count = 426
Succeeded at Mon Jul 28 14:21:27 2055 (Elapsed Time: 0.00 seconds)
```

When geoprocessing tools are called by a Python script, a success or failure message is also generated. The `arcpy` package provides functions (`GetMessages`, `GetMessage`, and `GetMessageCount`) for accessing these messages. The `arcpy GetMessages` function returns the message from the most recent tool call. If no tools have been called, it returns an empty string.

```
>>> arcpy.GetMessages()
''
```

No arguments are required and it is usually used with a print statement. The following code calls the Get Count (Data Management) tool and the `GetMessages` function:

```
>>> inputFile = 'C:/gispy/data/ch14/cover.shp'
>>> count = arcpy.GetCount_management(inputFile)
>>> print arcpy.GetMessages()
Executing: GetCount C:/gispy/data/ch14/cover.shp
Start Time: Mon Jul 28 20:30:10 2055
Row Count = 426
Succeeded at Mon Jul 28 20:30:10 2055 (Elapsed Time: 0.00 seconds)
```

This message contains the same information printed in the Geoprocessing Window when the tool is called from ArcToolbox. It reports the tool name and arguments, when the tool was called, any values it returns (e.g., Row Count = 426), and successful completion.

The first line of the message contains a list of all arguments used to call the function (both those specified explicitly and the default values used during the run). In the following code, only the three required arguments are used to call the Buffer tool, but the message also shows the default values that were used for the optional arguments:

```
>>> inputFile = 'C:/gispy/data/ch14/parkLines.shp'
>>> outputFile = 'C:/gispy/scratch/buffer.shp'
>>> arcpy.Buffer_analysis(inputFile, outputFile, '1 mile')
<Result 'C:/gispy/data/ch14\\buffer.shp'>
>>> print arcpy.GetMessages()
Executing: Buffer C:/gispy/data/ch14/parkLines.shp
C:/gispy/scratch\buffer.shp "1 Miles" FULL ROUND NONE #
Start Time: Tue Mar 12 21:05:10 2055
Succeeded at Tue Mar 12 21:05:10 2055 (Elapsed Time: 0.00 seconds)
```

The `arcpy GetMessage` function, which takes a line number as an argument, can be used to print individual lines of the message. Line numbers are zero-based. For example, the following code prints the first line of the message:

```
>>> print arcpy.GetMessage(0)
Executing: Buffer C:/gispy/data/ch14/park.shp C:/gispy/scratch\
buffer.shp "1 Miles" FULL ROUND NONE #
```

The message line count varies depending on the geoprocessing tool. The `GetMessageCount` function returns the line count:

```
>>> arcpy.GetMessageCount()
3
```

The line count minus one is the last valid index. The following code prints the last line of the message:

```
>>> print arcpy.GetMessage (arcpy.GetMessageCount() - 1)
Succeeded at Tue Mar 12 21:05:10 2055 (Elapsed Time: 0.00 seconds)
```

If the most recent tool call failed, the message reports failure. For example, the following code calls the Get Count tool with no arguments, which throws an `ExecuteError` exception:

```
>>> count = arcpy.GetCount_management()
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
File "C:\Program Files (x86)\ArcGIS\Desktop10.1\arcpy\arcpy\
management.py",
line 13637, in GetCount
raise e
ExecuteError: Failed to execute. Parameters are not valid.
```

This time the `GetMessages` function returns a failure message:

```
>>> print arcpy.GetMessages()
Executing: GetCount #
Start Time: Tue Mar 12 20:43:25 2055
Failed to execute. Parameters are not valid.
ERROR 000735: Input Rows: Value is required
Failed to execute (GetCount).
Failed at Tue Mar 12 20:43:25 2055 (Elapsed Time: 0.00 seconds)
```

The failure message generated by `GetMessages` is usually easier to interpret than the traceback message thrown by failed geoprocessing tools. For example, in addition to stating that the parameters are not valid, the message from the failed Get Count tool run states that a value is required for the input rows. Messages like this can be used along with error handling to provide informative feedback in geoprocessing scripts.

14.2.2 The arcpy Named Exception

When `arcpy` geoprocessing tool calls fail, they throw an `ExecuteError`. This is not one of Python's built-in exceptions; Instead, this is a special exception thrown by `arcpy`. Since it's an `arcpy` property, it can be referred to with dot notation as `arcpy.ExecuteError` and it can be used in a named `except` block as in Example 14.1.

Example 14.1

```
# bufferTry.py
import arcpy, sys, os
arcpy.env.overwriteOutput = True
try:
    inputFile = sys.argv[1]
    buff = os.path.splitext(inputFile)[0] + 'Buff.shp'
    arcpy.Buffer_analysis(inputFile, buff, '1 mile')
    print '{0} created.'.format(buff)
except arcpy.ExecuteError:
    print arcpy.GetMessages()
except IndexError:
    print 'Usage: <full path shapefile name>
```

In case a geoprocessing tool fails, the tool failure information can be printed by calling `GetMessages` in the `arcpy.ExecuteError` exception block. For example, if "C:/gispy/scratch/bogus.shp" (a non-existent file) is used as input, the Buffer tool call throws an exception. The code execution jumps to the `arcpy.ExecuteError` block and the following message is printed:

```
Executing: Buffer C:/bogus.shp C:\bogusBuff.shp "1 Miles" FULL
ROUND NONE #
Start Time: Tue Mar 12 22:33:06 2055
Failed to execute. Parameters are not valid.
ERROR 000732: Input Features: Dataset C:/gispy/scratch/bogus.shp
does not exist or is not supported
Failed to execute (Buffer).
Failed at Tue Mar 12 22:33:06 2055 (Elapsed Time: 0.00 seconds)
```

14.3 Catching Exceptions in Loops

When a script performs geoprocessing on a batch of files, unhandled errors will halt the process. For example, if a Buffer tool call fails on a file in a batch of hundreds, the files that come after that one in the list won't be processed, even if they are valid. One bad apple can spoil the bunch. A main advantage of using error handling in scripts is so that batch processing can continue even if a tool call fails for one file.

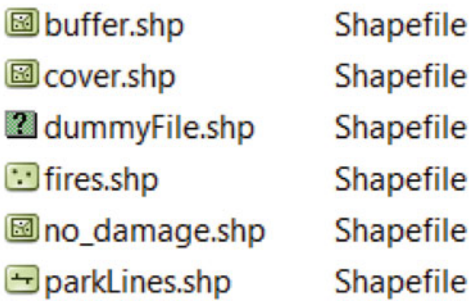
try/except blocks should both go inside the loop with the `try` block wrapped around the geoprocessing calls as in Example 14.2. This script catches geoprocessing tool errors by using a named exception.

Example 14.2

```
# bufferLoopTry.py
# Purpose: Buffer the feature classes in a workspace.
# Usage: No arguments needed.
import arcpy, os
arcpy.env.overwriteOutput = True
arcpy.env.workspace = 'C:/gispy/data/ch14'
outDir = 'C:/gispy/scratch/'
fcs = arcpy.ListFeatureClasses()
distance = '500 meters'

for fc in fcs:
    outFile = outDir + os.path.splitext(fc)[0] + 'Buff.shp'
    try:
        arcpy.Buffer_analysis(fc, outFile, distance)
        print 'Created: {0}'.format(outFile)
    except arcpy.ExecuteError:
        print arcpy.GetMessage(2)
```

A geoprocessing tool call can fail for any number of reasons. Data might be locked or corrupted (e.g., a required shapefile support file is missing). Figure 14.1 shows an invalid shapefile 'dummyFile.shp' as it appears in ArcCatalog.



buffer.shp	Shapefile
cover.shp	Shapefile
dummyFile.shp	Shapefile
fires.shp	Shapefile
no_damage.shp	Shapefile
parkLines.shp	Shapefile

Figure 14.1 The ArcCatalog view of the shapefiles in the Chapter 14 data directory.

This file was created by renaming a text file to have an '.shp' extension. The following results are printed when Example 14.2 is run on these files:

```
>>> Created: buffer_buff.shp
Created: coverBuff.shp
ERROR 000229: Cannot open C:/gispy/data/ch14\dummyFile.shp.
Created: firesBuff.shp
Created: no_damageBuff.shp
Created: parkLinesBuff.shp
```

Every file is processed, except 'dummyFile.shp' and an error message is reported. Good programming style only places code inside a loop that needs to go inside the loop. In Example 14.2, the output file name must be updated inside the loop, but the distance remains constant. Along the same lines, the code that goes inside a `try` block should be minimized. In Example 14.2, the Buffer tool call must be inside of the `try` block, but the output file name can be set outside the `try` block.

Another aspect of arranging code comes into play when the batch processing occurs in a `WHILE`-loop. Since try/except blocks can cause the flow of the code to change, it's important to update the iterating variable at the end of the loop outside of both the `try` and `except` blocks. Take Example 14.3, which creates buffers at various distances, up to the maximum specified number of miles. An input of “C:/gispy/data/ch14/cover.shp” and “3” generates 1 mi., 2 mi., and 3 mi. buffer output files and prints the following output:

```
>>> Created: C:/gispy/scratch/cover1Buff.shp
Created: C:/gispy/scratch/cover2Buff.shp
Created: C:/gispy/scratch/cover3Buff.shp
```

The `overwriteOutput` property is set to False so that the script only creates files that don’t already exist. This means the Buffer tool call throws an exception when an output already exists. This exception is caught in the named exception block. If the script is run a second time with `maxDist` set to 5, it prints three exceptions and moves on to create the additional files:

```
>>> ERROR 000725: Output Feature Class: Dataset
C:\gispy\scratch\cover1Buff.shp already exists.
ERROR 000725: Output Feature Class: Dataset C:\gispy\scratch\
cover2Buff.shp already exists.
ERROR 000725: Output Feature Class: Dataset C:\gispy\scratch\
cover3Buff.shp already exists.
Created: cover4Buff.shp
Created: cover5Buff.shp
```

The iterating variable is updated on the last line of code. Notice that this line is inside the `WHILE`-loop, but outside the `except` block. Placing it outside the `WHILE`-loop would cause infinite looping with the iterating variable stuck at 1; Placing it inside the `except` block would lead to infinite looping with the iterating variable stuck at the value of the first iteration where the Buffer tool call does not throw an error.

Example 14.3

```
# bufferLoopDistTry.py
# Purpose: Buffer the input file by the given distance.
# Usage: input_filename numeric_distance
# Example input: C:/gispy/data/ch14/cover.shp 3

import arcpy, sys, os
arcpy.env.workspace = os.path.dirname(sys.argv[1])
fc = os.path.basename(sys.argv[1])
outDir = 'C:/gispy/scratch/'
arcpy.env.overwriteOutput = False
maxDist = float(sys.argv[2])
i = 1
```

```
 while i <= maxDist:
     try:
         outFile = outDir + os.path.splitext(fc)[0] + str(i) + \
                   'Buff.shp'
         distance = str(i) + ' miles'
         arcpy.Buffer_analysis(fc, outFile, distance)
         print 'Created: ', outFile
     except arcpy.ExecuteError:
         print arcpy.GetMessage(3)
     i = i + 1
```

14.4 Discussion

Exceptions come in two flavors, ones which are caused by errors in the code and ones which are caused by some outside influence, such as input data. The first kind should always be resolved by the programmer; The second kind resides in a somewhat gray area. The need for error handling depends on the context. For example, if a script that requires numerical input is being run via a graphical user interface which limits the user input to numerical values, there's no need to handle a `ValueError` for that input. Some situations also require a decision between using conditional constructs and error handling.

Conditional blocks have certain similarities to try/except blocks. With both of these structures, the execution can be diverted depending on conditions. In some situations, conditional blocks could be used to achieve the same effect as try/except blocks. The try/except blocks should be used to handle exceptional cases—when something has gone wrong. When, on the other hand, both `IF` and `ELSE` alternatives are normal acceptable behavior, conditional blocks are more apt. For example, the following code checks if an argument has been provided and if not, it sets a default value:

```
import sys, arcpy
if len(sys.argv) > 1:
    arcpy.env.workspace = sys.argv[1]
else:
    arcpy.env.workspace = 'C:/gispy/data/ch14'
for rast in arcpy.ListRasters():
    print rast
```

The program is designed so that the user can provide an argument, but doesn't strictly need to provide an argument. The program continues with business as usual without arguments, taking the default value for the input workspace instead of an alternative provided by the user. The same result could be accomplished with a try/except block, as in the following code, which tries to access a user argument and catches an `IndexError`:

```
import sys, arcpy
try:
    arcpy.env.workspace = sys.argv[1]
except IndexError:
    arcpy.env.workspace='C:/gispy/data/ch14'
for rast in arcpy.ListRasters():
    print rast
```

In this case, though, conditional blocks are preferable. Nothing has gone wrong; the user has simply chosen to accept the default workspace.

In other scripts, an argument may be required. In this case, running the script without an argument would be considered an error. In fact, the script might need to exit without continuing. The following code uses try/except blocks more appropriately:

```
import sys, arcpy
try:
    arcpy.env.workspace = sys.argv[1]
except IndexError:
    print 'Usage: <input workspace>'
    sys.exit(0)
for rast in arcpy.ListRasters():
    print rast
```

The exception handling block prints instructions for using the script and then forces the script to exit with the `sys` module `exit` method (`sys.exit(0)`); This prevents the code from attempting to list the rasters on an unspecified workspace.

Note try/except blocks should be used judiciously based on desired functionality and expected usage.

Geoprocessing usually involves input data and since the quality of the data can't be guaranteed in most real-world projects, appropriate error handling is an important component of making scripts more reliable and robust.

14.5 Key Terms

`try` and `except` blocks
Named exceptions
`arcpy.GetMessages()`
Catch exceptions
The `arcpy.ExecuteError` exception
`arcpy.GetMessage(index)`
`arcpy.GetMessageCount()`

14.6 Exercises

1. **simplifyOops** Explore two common try/except mistakes as follows:

   (a) The purpose of sample script 'simplifyOops1.py' is to simplify polygons for all valid data within the input directory, using a `FOR`-loop with error handling. But it has some mistakes. Open the script and view the input directory to predict the output it will create.
   (b) Run 'simplifyOops1.py' with no debugging.
   (c) Run the script again using the 'step through in debugger' option.
   (d) Ideally, we want the script to create simplified polygons for all of the valid polygon files and print a warning if it fails for one of the files. `try` and `except` are in the wrong positions for this to happen. Repair the script.
   (e) Predict what will happen when you run the script. Then run it again with no debugging.
   (f) Run the script again stepping through in the debugger. Use the watch window to watch the value of `fc`. Three output shapefiles should be created.
   (g) Now we'll use a different sample script, 'simplifyOop2.py'. The purpose of this script is to simplify polygons for one data file using ten different minimum distance values within a `WHILE`-loop and with error handling. But this script has some mistakes. Open 'simplifyOops2.py' and predict the output it will produce.
   (h) Run 'simplifyOops2.py' and observe the output it creates.
   (i) Run the code again by stepping through in the debugger to see the flow.
   (j) Repair the mistake on the line that reads:
   `minArea = '{0}foot'.format(x)`
   (k) Now run the code by stepping through in the debugger again.
   (l) Watch the value of `x` in the watch window.
   (m) Repair the mistake on the line that reads: `x = x + 1`
   (n) Run the script again and confirm that there are ten output files.

2. **predictTry.py** The sample script, 'predictTry.py' contains multiple named exceptions, as well as conditional blocks. The script requires one argument, the name of an input file. Test your understanding of code flow with try/except blocks by predicting what will be printed by the script for each of the following input scenarios (then check your answers by running the script):

   (a) no arguments
   (b) C:/gispy/data/ch14/predict/bogus.shp (a file that doesn't exist)
   (c) C:/gispy/data/ch14/predict/tree.gif
   (d) C:/gispy/data/ch14/predict/coverPolygons.shp (a Polygon file)
   (e) C:/gispy/data/ch14/predict/firesPoints.shp (a Point file)
   (f) C:/gispy/data/ch14/predict/parkLines.shp (a Polyline file)

3. **dictionaryTry.py** The code in the `try` block of sample script, 'dictionaryTry.py' contains an error, but the `except` block simply prints the message 'An error occurred'. This message is too generic. Add code that uses the Python built-in

traceback module to force the script to print an exception in the exception handling block (Don't correct the error in the `try` block).

4. **fileOpenTry.py** The sample script, 'fileOpenTry.py' opens and reads the contents of a text file. Run the script with a valid text file argument (like example input1 below) to see it work correctly. Then run the script again using the name of a text file that doesn't exist (like example input2), and observe the resulting error. Add a named exception block to the script to handle this kind of exception. Next, print a message inside the exception handling block. Last, add code to force the script to exit by using the `sys` module `exit` function inside the exception handling block. The `###` comments in the script provide guidance.

   Example input1: C:/gispy/data/ch14/cover.prj

   Example output1: (only the first few characters are shown):
   ```
   PROJCS["NAD_1927_StatePlane_Pennsylvania_Sou...
   ```

   Example input2: C:/gispy/data/ch14/dummyFile.prj

   Example output2:
   ```
   Warning: could not open: C:/gispy/data/ch14/dummyFile.prj
   ```

5. **moduloTry.py** The modulo operator (represented by % in Python) finds the remainder of a division operation. For example, 10%6==4 because 4 is the remainder of 10/6. The sample script 'moduloTry.py' takes two input values and finds the modulo. Modify the script to implement error handling with named exception handling blocks to provide the following behavior:

Input	Printed output

25 4	`a: 25 b: 4`
	`c: 1.0`

5 0	`a: 5 b: 0`
	`c: 5.0 mod 0 is undefined`

woods 3	`a: woods b: 3`
	`Usage: <numeric value 1> <numeric value 2>`
	`c: Not found`

6. **exportTry.py** Use the Quick Export (Data Interoperability) tool to export an ArcGIS format data file to a comma separated value file. The script should take two arguments, the full path file name of the data and an output directory. Use error handling in case the tool fails and use the `arcpy GetMessage` function indexes 4 and 3 to print selected portions of the geoprocessing message when the tool call fails. The script should have the following behavior:

   Example input 1: C:/gispy/data/ch14/cover.shp C:/gispy/scratch

   Example output 1:
   ```
   Output created in: CSV, C:/gispy/scratch
   ```

Example input 2: C:/gispy/data/ch14/dummyFile.shp C:/gispy/scratch

Example output 2:

```
Failed to execute (QuickExport).
Feature class 'C:/gispy/data/ch14/dummyFile.shp' is invalid.
```

7. **boxTry.py** Use the Minimum Bounding Geometry (Data Management) tool to find the minimum bounding rectangles for the features in each shapefile in C:/gispy/data/ch14. Create output in 'C:/gispy/scratch'. The script needs no arguments. Use error handling in case the tool fails and use the `arcpy` `GetMessage` function with indexes 2 and 3 to print selected portions of the geoprocessing message when the tool call fails. If the user passes the Chapter 14 data directory, 'C:/gispy/data/ch14', (see Figure 14.1) into the script, it should print the following:

```
bufferBox.shp created.
coverBox.shp created.
ERROR 000229: Cannot open C:/gispy/data/ch14\dummyFile.shp
Failed to execute (MinimumBounding Geometry).
firesBox.shp created.
no_damageBox.shp created.
parkLinesBox.shp created.
```