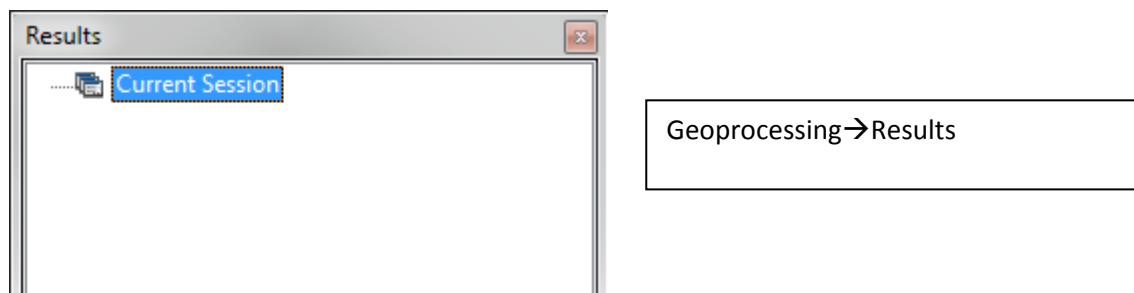


Chapter 7: Tool messaging, results and error handling in Python

7.0 Tool messaging

- During execution of a tool various messages are generated and written to the geoprocessor object.
- These messages include such information as the following:
 - When the operation started and ended
 - The parameter values used
 - General information about the operation's progress (information message)
 - Warnings of potential problems (warning message)
 - Errors that cause the tool to stop execution (error message)
- These messages can be read by your Python script and your code can be designed to appropriately handle any errors or warnings that have been generated.
- Depending on where you are running the tools from, messages appear in the Results window, the Python window, and the progress dialog box.

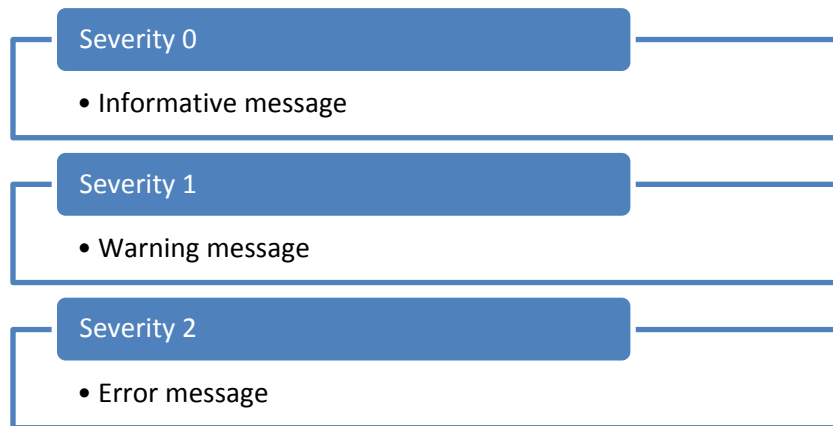


The Results Window

See:

http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/Using_the_Results_window/00210000000130000000/ for more information on the results window

- Classification of messages- All messages have a severity property, either informative, warning, or error. The severity is an integer where 0 = informative, 1 = warning, and 2 = error.



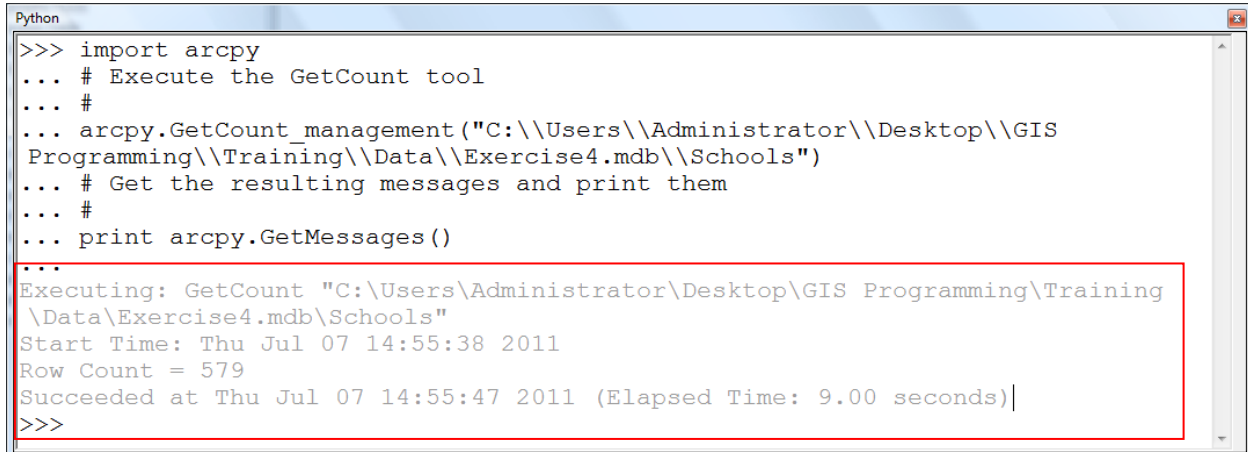
- Informative message
 - An informative message is just that—information about execution.
 - It is never used to indicate problems.
 - Only general information, such as a tool's progress, what time a tool started or completed, output data characteristics, or tool results, is found in informative messages.
- Warning message
 - Warning messages are generated when a tool experiences a situation that may cause a problem during its execution or when the result may not be what you expect.
 - For example, defining a coordinate system for a dataset that already has a coordinate system defined generates a warning.
 - You can take action when a warning is returned, such as canceling the tool's execution or making another parameter choice.
 - Contain six digit code
- Error message
 - Error messages indicate a critical event that prevented a tool from executing. Errors are generated when one or more parameters have invalid values, an invalid path to data or when a critical execution process or routine has failed.
 - Contain six digit code
- These ID codes provide additional information on their causes and how they can be dealt with.

- When error or warning codes are shown in the tool or progress dialog box, Python window, or Result window, they have a link that allows you to go directly to the additional help for that message. A summary of codes can be found at http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/000001_Angle_should_be_greater_than_0_and_less_than_180/00vp00000002000001/
- Multiple messages may be generated. These are stored in a list and can be retrieved.

7.1 Getting messages

- Messages from the last tool executed are maintained by ArcPy and can be retrieved using the [GetMessages\(\)](#) function.
- GetMessages() returns a single string containing all the messages from the tool that was last executed.
- The returned messages can be filtered to include only those with a certain severity using the severity option.
- When using ArcPy, the **first message gives the tool executed, and the last message gives the ending and elapsed time for the tool's execution.**
- The tool's second and last messages always give the start and end time, respectively, for the tool's execution.

Example: getCount.py



```

Python
>>> import arcpy
... # Execute the GetCount tool
... #
... arcpy.GetCount_management("C:\\Users\\Administrator\\Desktop\\GIS
Programming\\Training\\Data\\Exercise4.mdb\\Schools")
... # Get the resulting messages and print them
... #
... print arcpy.GetMessages()
...
Executing: GetCount "C:\\Users\\Administrator\\Desktop\\GIS Programming\\Training
\\Data\\Exercise4.mdb\\Schools"
Start Time: Thu Jul 07 14:55:38 2011
Row Count = 579
Succeeded at Thu Jul 07 14:55:47 2011 (Elapsed Time: 9.00 seconds)
>>>

```

GetCount_management returns the total number of rows for a feature class, table, layer, or raster.

- Individual messages can be retrieved using the [GetMessage](#) function.
- This function has one parameter, which is the index position of the message.
- The [GetMessageCount](#) function returns the number of messages from the last tool executed. The example below shows how to print information about which tool was executed along with ending and elapsed times for the tool.

```
Python
>>> arcpy.GetMessage(0)
u'Executing: GetCount "C:\\Users\\Administrator\\Desktop\\GIS Programming
\\Training\\Data\\Exercise4.mdb\\Schools"'
>>> arcpy.GetMessage(1)
u'Start Time: Thu Jul 07 14:55:38 2011'
>>>
```

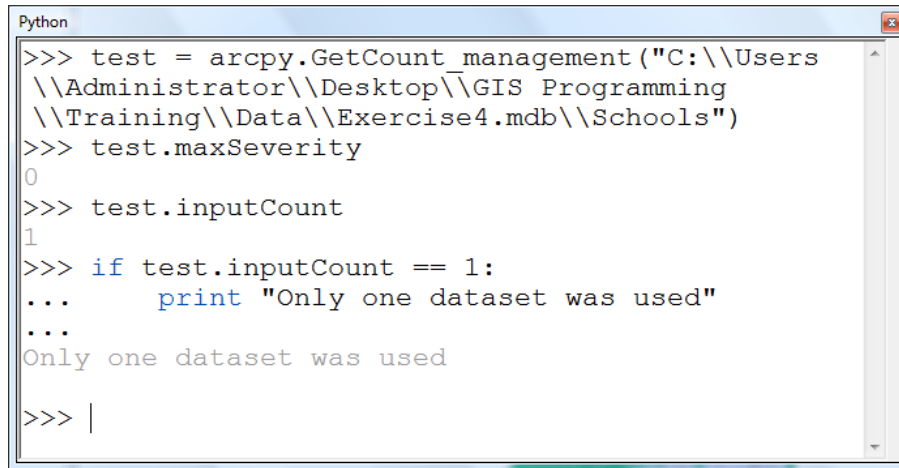
```
Python
>>> arcpy.GetMessageCount()
4
>>>
```

- Messages can be filtered by severity using the GetMessages (severity)

```
Python
>>> arcpy.GetMessages(0)
u'Executing: GetCount "C:\\Users\\Administrator
\\Desktop\\GIS Programming\\Training\\Data
\\Exercise4.mdb\\Schools"\nStart Time: Thu Jul
07 14:55:38 2011\nRow Count = 579\nSucceeded at
Thu Jul 07 14:55:47 2011 (Elapsed Time: 9.00
seconds)'
>>> arcpy.GetMessages(1)
''
>>> arcpy.GetMessages(3)
''
>>>
```

7.2 Retrieving messages from the Result Object

- Unlike getting messages from ArcPy, messages on a Result object can be maintained even after running multiple tools.
- The Result object supports several of the same functions used to get and interpret geoprocessing tool messages.

A screenshot of a Python command prompt window. The window title is "Python". The code being executed is as follows:

```
>>> test = arcpy.GetCount_management("C:\\Users\\Administrator\\Desktop\\GIS Programming\\Training\\Data\\Exercise4.mdb\\Schools")
>>> test.maxSeverity
0
>>> test.inputCount
1
>>> if test.inputCount == 1:
...     print "Only one dataset was used"
...
Only one dataset was used
>>> |
```

- Try out a few of the functions at http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/Understanding_message_types_and_severity/002z0000000p000000/
- <http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/000v000000n7000000.htm>

7.3 Outputting messages to the user

- Using several functions
 - addMessage(message string)
 - addWarning(message string)
 - addError(message string)
- These messages are immediately returned to the application or script that is executing
- Example: outputMessages1.py

```
outputMessages1.py - Notepad
File Edit Format View Help
import arcpy

fc = arcpy.GetParameterAsText(0)
result = arcpy.GetCount_management(fc)

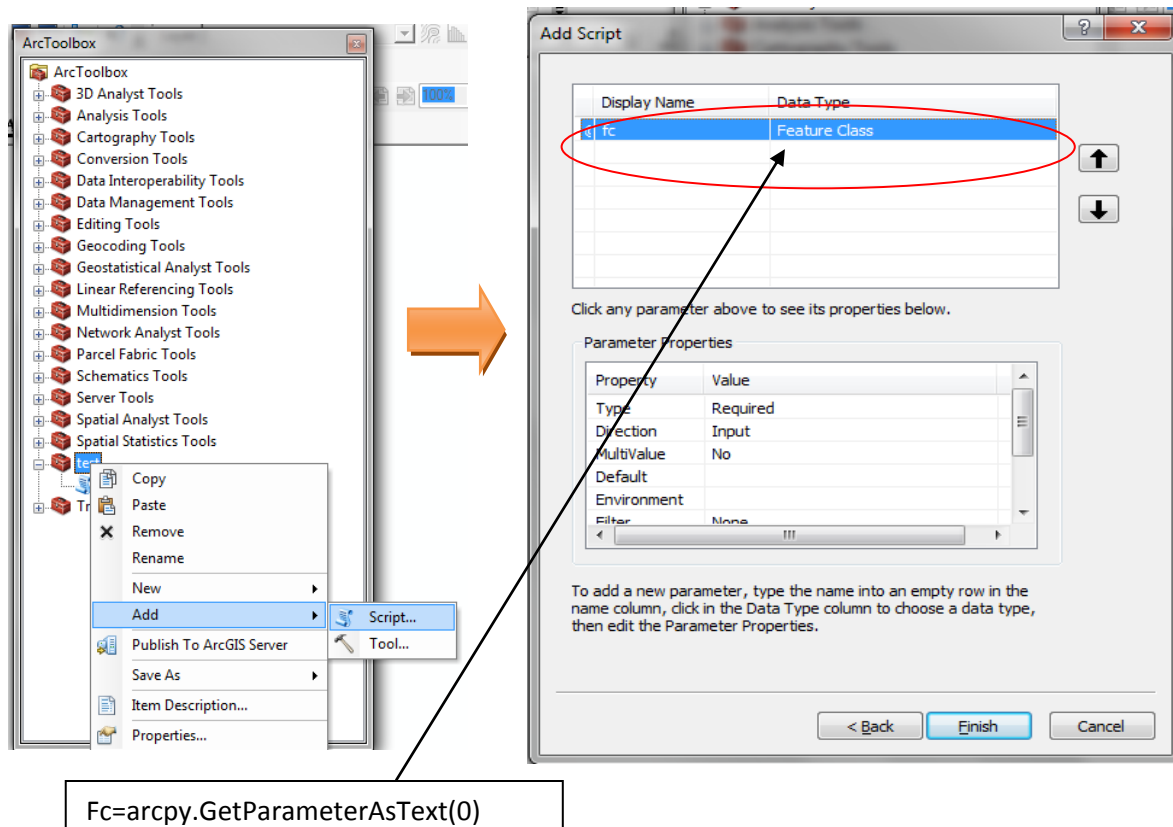
# Get the count from GetCount's Result object
#
featurecount = int(result.getOutput(0))

if featurecount == 0:
    arcpy.AddError(fc + " has no features.")
else:
    arcpy.AddMessage(fc + " has " + str(featurecount))
```

Get this above script to run!

Hint:

1. GetParameterAsText only works in ArcGIS
2. See Figures below



REMEMBER: To add a script you MUST first have a script prepared

See:

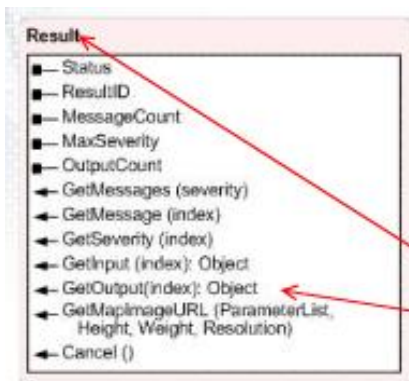
<http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/AddMessage/000v00000005000000/>
for more.

- Outputting all messages using a loop

```
arcpy.Clip_analysis("roads","urban_area","urban_roads")
# Return the resulting messages as script tool output messages
#
x = 0
while x < arcpy.MessageCount:
    arcpy.AddReturnMessage(x)
    x = x + 1
```

7.4. Messages and the Result object

- The Geoprocessor stores results in a result object
- Prior to version 9.3 the Result Object was used exclusively with ArcGIS Server geoprocessing
- Messages are maintained through multiple tool runs



```
import arcpy
inTable = arcpy.GetParameterAsText(0)
result = arcpy.GetCount_management(inTable)
print result.getOutput(0)
```

7.5 Result object status codes

- Contains the status **property** which contains the status on the job currently on the server
- The status is returns as one of several integer values
- You can write Python scripts to read these values and make decisions

Status Codes	
<div>Result<ul style="list-style-type: none">■ Status■ ResultID■ MessageCount■ MaxSeverity■ OutputCount← GetMessages (severity)← GetMessage (index)← GetSeverity (index)← GetInput (index): Object← GetOutput (index): Object← GetMapImageURL (ParameterList, Height, Width, Resolution)← Cancel ()</div>	Status Codes
	0 New
	1 Submitted
	2 Waiting
	3 Executing
	4 Succeeded
	5 Failed
	6 Timed Out
	7 Canceling
	8 Canceled
	9 Deleting
	10 Deleted

7.6 Returning messages with the Result Object

- The results object contains GetMessage and GetMessages **functions**. These are used to either list a specific message or a list of all messages by severity.

Result
<ul style="list-style-type: none">■ Status■ ResultID■ MessageCount■ MaxSeverity■ OutputCount← GetMessages (severity)← GetMessage (index)← GetSeverity (index)← GetInput (index): Object← GetOutput(index): Object← GetMapImageURL (ParameterList, Height, Weight, Resolution)← Cancel ()

- GetMessage (Index) returns a specific message by integer

- GetMessage (Severity) returns all messages by severity (0,1,2)
- MessageCount returns the number of messages on the object.

7.7 Error handling

- No one is perfect as will not all code
- They are unavoidable
- How your code respond to these events also called exceptions is very important
- You should try to handle errors gracefully through Python Error Handling Structures. These examine ArcPy generated Exceptions and act accordingly.
- Without things in place to handle errors appropriately your scripts may immediately fail and frustrate your users. Additionally, your script becomes much more tedious to debug.
- Exception: Unusual or error conditions which occur in your code
- Exceptions statements allow you to trap and handle errors in your code, gracefully recovering from error conditions.
- Exceptions can also be used for:
 - Event notification: signal valid conditions without having to pass result flags around a program or test them explicitly
 - Special case handling: used when a condition happens so rarely that it is hard to justify branching code to handle it. You can eliminate special case code by using exception handlers instead.
- Exceptions are a high level control flow device used primarily for error interception or triggering
- Exceptions can be raised automatically by Python or manually by your programs
- REMEMBER: ArcGIS has error codes assigned to each error which you can use to get assistance about the error.
- A try-except statement can be used to wrap entire programs or just particular portions of code to trap and identify errors.
- If an error occurs within the try statement, an exception is raised, and the code under the except statement is then executed. Using a simple except statement is the most basic form of error handling.
- There are 2 types of try statements:
 - Try/except/else
 - Used to handle exceptions
 - Try/finally
 - Executes finalization code whether exceptions occur or not

Syntax:

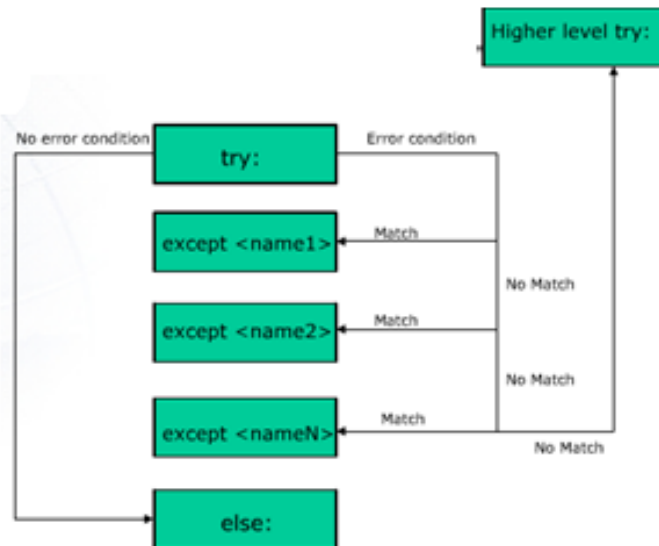
```
try:
    <statements>
except <name>:
    <statements>
except <name>, <data>:
    <statements>
else:
    <statements>
```

← Code inside try statement executes

← Named exceptions can be caught and statements executed

← If no named exceptions meet the condition, an 'else' statement executes

- The else statement is OPTIONAL



The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.
- Example: exception2.py

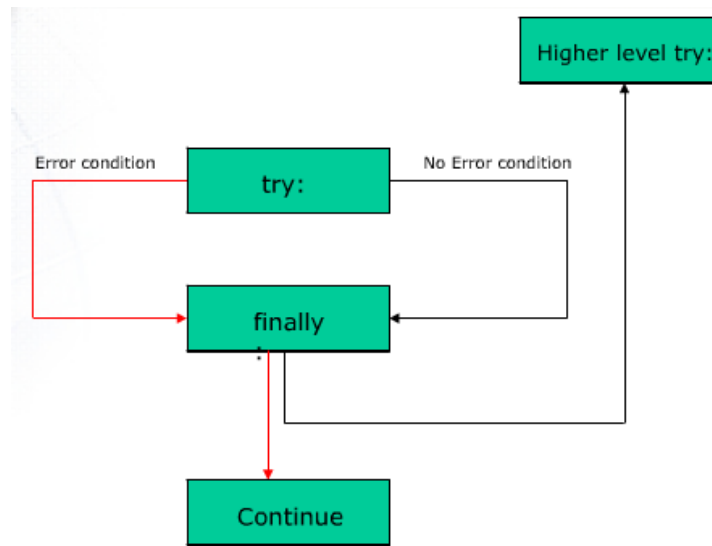
```
Python
>>>
>>> import arcpy
... try:
...     # Execute the Buffer tool
...     #
...     arcpy.Buffer_analysis("c:/transport/roads.shp", "c:/transport/roads_buffer.shp")
... except:
...     print "Ok...hold up, you need to fix me"
...
Ok...hold up, you need to fix me
>>> |
```

- Example: exception.py
- In the following code, [Buffer](#) fails because the required Distance parameter has not been used and the location to the datasets are invalid. Instead of failing without explanation, the except statement is used to trap the errors, then fetch and print the error messages generated by Buffer. Note that the except block is only executed if Buffer returns an error.

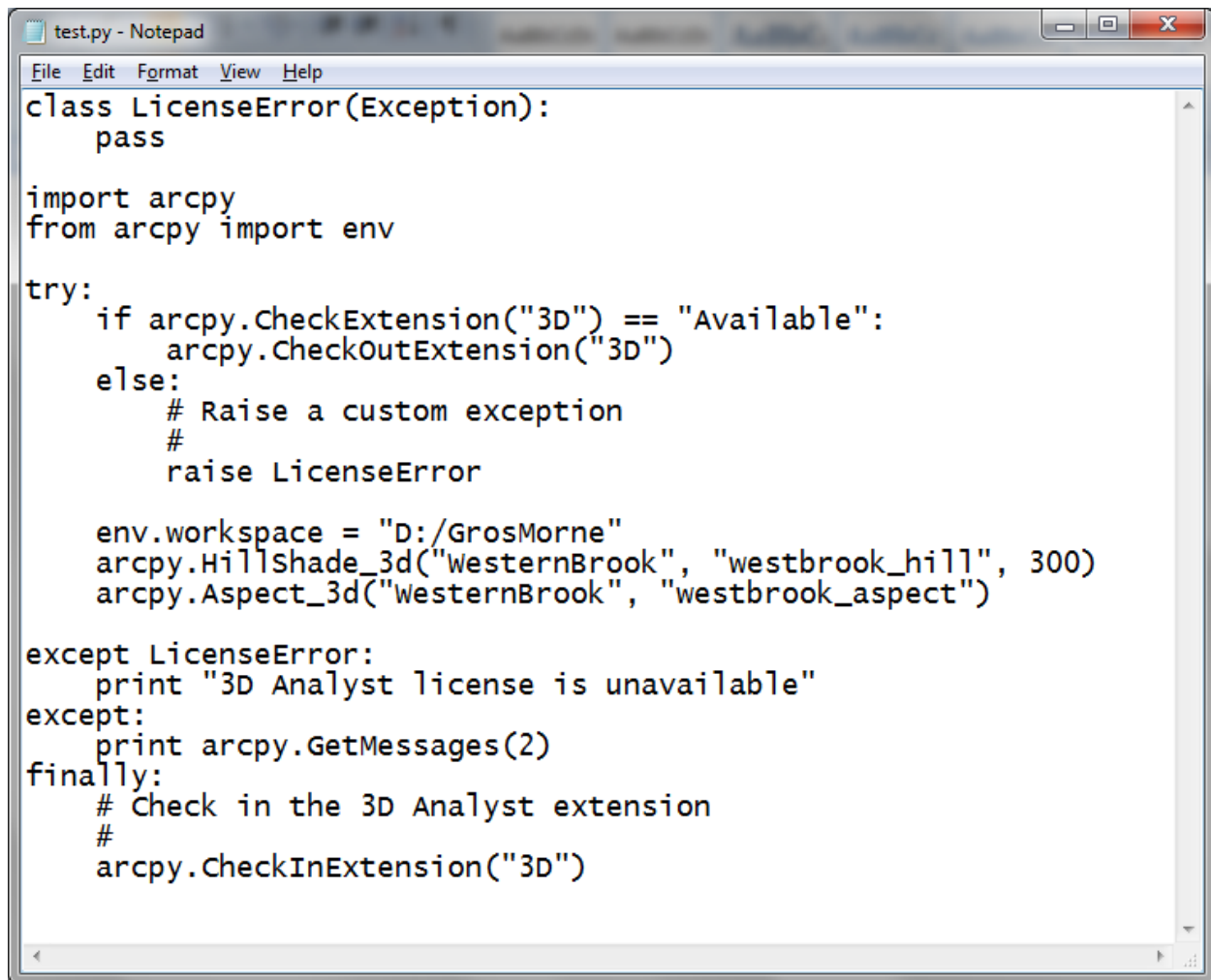
```
Python
>>> import arcpy
... try:
...     # Execute the Buffer tool
...     #
...     arcpy.Buffer_analysis("c:/transport/roads.shp", "c:/transport/roads_buffer.shp")
... except Exception as e:
...     print e.message
...
...     # If using this code within a script tool, AddError can be used to return messages
...     # back to a script tool. If not, AddError will have no effect.
...     arcpy.AddError(e.message)
...
ERROR 000732: Input Features: Dataset c:/transport/roads.shp does not exist or is not supported
ERROR 000735: Distance [value or field]: Value is required
>>>
```

You can click on the error to get more information about the error.

- The other type of try statement is the try/finally statement which allows for finalization actions.
- used for tasks that should be always be executed, whether an exception (error condition) has occurred or not
- when a finally statement is used in a try statement, its block of code is ALWAYS placed (and is always run) at the very end.
- Works in the following way:
 - If an exception occurs, python runs the try block, then the finally block, and then execution continues **pass the entire** try statement.
 - If an exception does not occur during execution, python runs the try block, then the finally block, and then execution **passes to a higher level try** statement.



- In the following example, the 3D Analyst extension is checked back in under a finally clause, ensuring that the extension is always checked back in.



```
test.py - Notepad
File Edit Format View Help
class LicenseError(Exception):
    pass

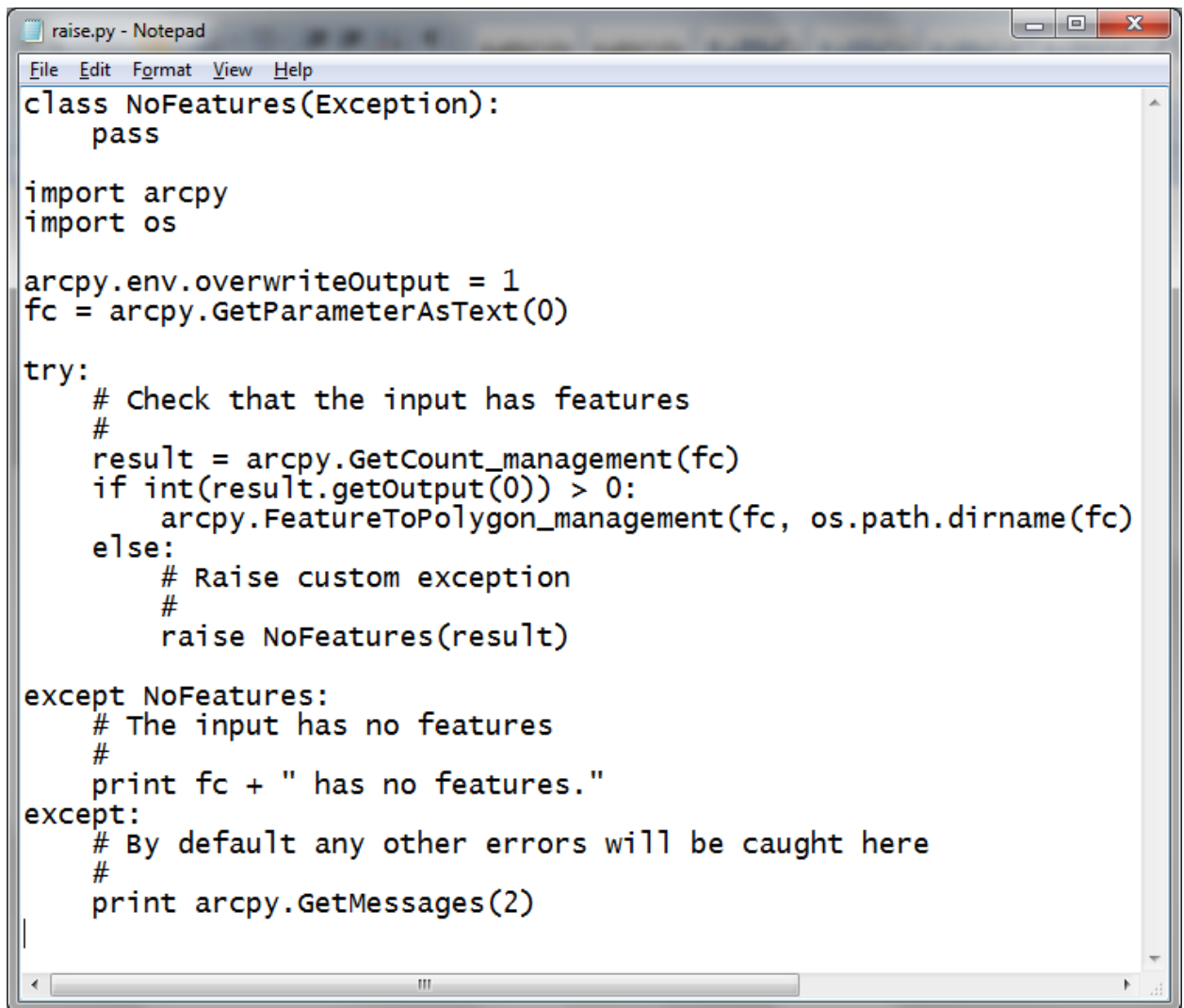
import arcpy
from arcpy import env

try:
    if arcpy.CheckExtension("3D") == "Available":
        arcpy.CheckOutExtension("3D")
    else:
        # Raise a custom exception
        #
        raise LicenseError

    env.workspace = "D:/GrosMorne"
    arcpy.HillShade_3d("WesternBrook", "westbrook_hill", 300)
    arcpy.Aspect_3d("WesternBrook", "westbrook_aspect")

except LicenseError:
    print "3D Analyst license is unavailable"
except:
    print arcpy.GetMessages(2)
finally:
    # Check in the 3D Analyst extension
    #
    arcpy.CheckInExtension("3D")
```

- Python can respond to error conditions that occur in one of 2 ways:
 - Exceptions automatically generated in your code
 - Automatic
 - Exceptions that you raise in your code
 - Manual trigger
- To raise exceptions in your code use the raise statement
 - create custom exceptions
- For example:
 - Let's say you have a script that does some geoprocessing on a layer that contains polygon features. In this script you may want to raise a custom exception in the event that a user specifies a line or point layer as an argument to this script. You could create a custom exception to handle this situation.



```
raise.py - Notepad
File Edit Format View Help
class NoFeatures(Exception):
    pass

import arcpy
import os

arcpy.env.overwriteOutput = 1
fc = arcpy.GetParameterAsText(0)

try:
    # Check that the input has features
    #
    result = arcpy.GetCount_management(fc)
    if int(result.getOutput(0)) > 0:
        arcpy.FeatureToPolygon_management(fc, os.path.dirname(fc))
    else:
        # Raise custom exception
        #
        raise NoFeatures(result)

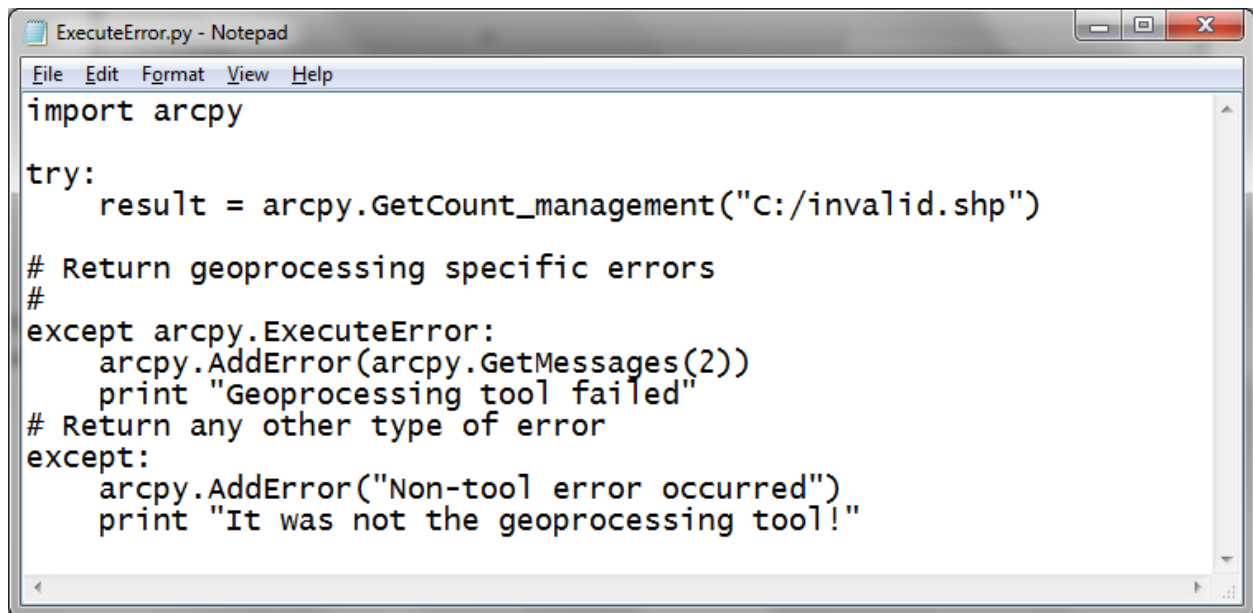
except NoFeatures:
    # The input has no features
    #
    print fc + " has no features."
except:
    # By default any other errors will be caught here
    #
    print arcpy.GetMessages(2)
```

Syntax:

- raise <name> #manual triggering
- raise <name>, <data> #manual trigger with extra data passed to the catcher

The <name> variable refers to an exception handler which you set up with the except statement. You can also pass additional data to the exception handler as noted with the data variable.

- When a geoprocessing tool fails, it throws an [ExecuteError](#) exception class.
- What this means is that you can divide errors into two groups, geoprocessing errors (those that throw the ExecuteError exception) and everything else. You can then handle the errors differently, as demonstrated in the code below:
- Example: ExecuteError.py



```
ExecuteError.py - Notepad
File Edit Format View Help
import arcpy

try:
    result = arcpy.GetCount_management("C:/invalid.shp")
# Return geoprocessing specific errors
#
except arcpy.ExecuteError:
    arcpy.AddError(arcpy.GetMessages(2))
    print "Geoprocessing tool failed"
# Return any other type of error
except:
    arcpy.AddError("Non-tool error occurred")
    print "It was not the geoprocessing tool!"
```

- In larger, more complex scripts, it can be difficult to determine the precise location of an error.
- Python's sys and traceback modules can be used together to isolate the exact location and cause of the error, identifying the cause of an error more accurately and saving you valuable debugging time.

More on sys & traceback

```
traceback.py - Notepad
File Edit Format View Help
# Import the required modules
#
import arcpy
import sys
import traceback

arcpy.env.workspace = "C:/Data/myData.gdb"
try:
    arcpy.CreateSpatialReference_management()
    #-----
    # Your code goes here
    #
    # See the table below for examples
    #-----
except arcpy.ExecuteError:
    # Get the tool error messages
    #
    msgs = arcpy.GetMessages(2)

    # Return tool error messages for use with a script tool |
    #
    arcpy.AddError(msgs)

    # Print tool error messages for use in Python/Pythonwin
    #
    print msgs
except:
    # Get the traceback object
    #
    tb = sys.exc_info()[2]
    tbinfo = traceback.format_tb(tb)[0]

    # Concatenate information together concerning the error into a message string
    #
    pymsg = "PYTHON ERRORS:\nTraceback info:\n" + tbinfo + "\nError Info:\n" + str(sys.exc_info()[1])
    msgsg = "ArcPy ERRORS:\n" + arcpy.GetMessages(2) + "\n"

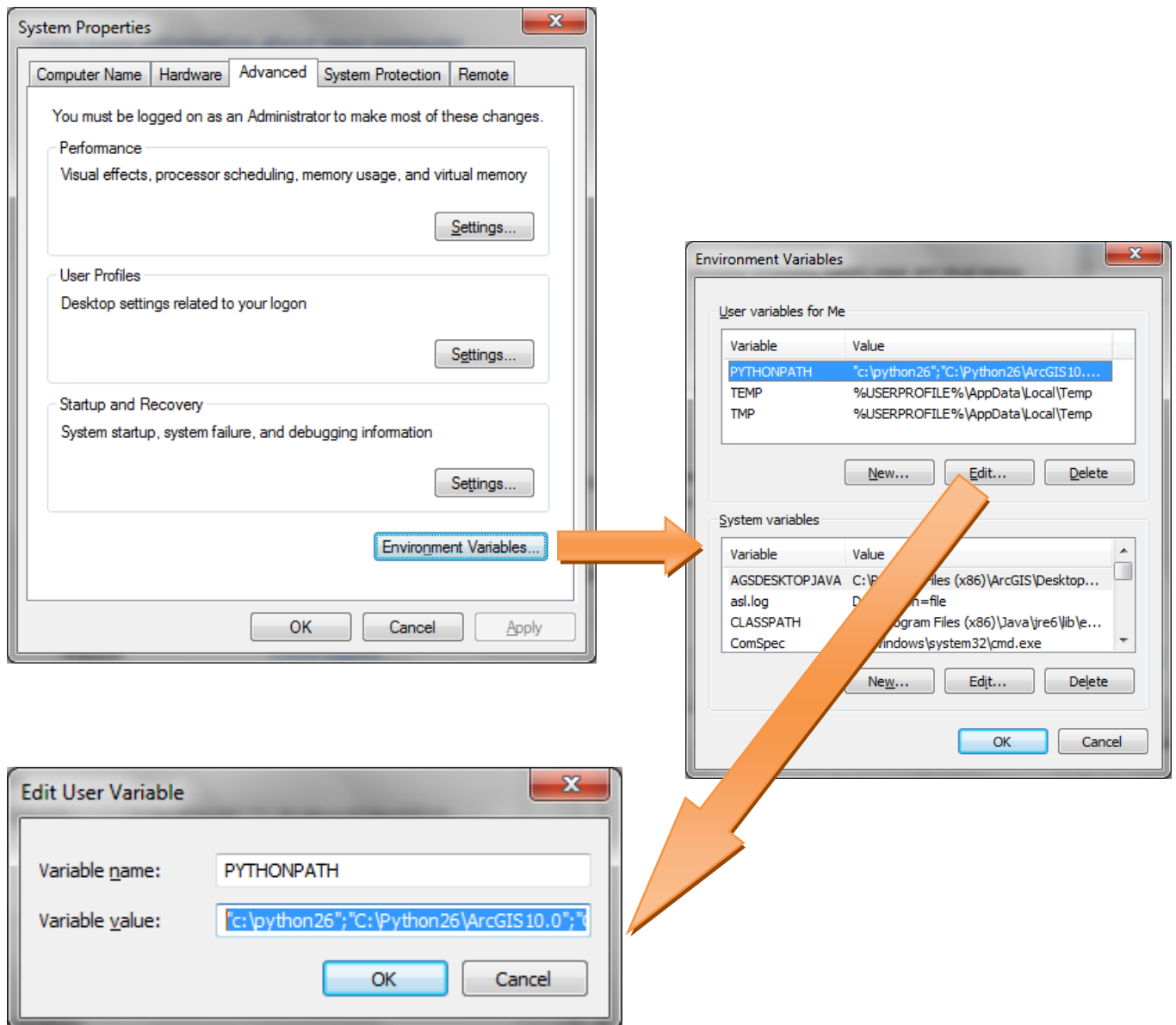
    # Return python error messages for use in script tool or Python window
    #
    arcpy.AddError(pymsg)
    arcpy.AddError(msgsg)

    # Print Python error messages for use in Python / Python window
    #
    print pymsg + "\n"
    print msgsg
```

- If the above code was used and a geoprocessing tool error occurred, such as an invalid input, this would raise [ExecuteError](#), and the first except statement would be used. This statement would print out the error messages using the [GetMessages](#) function. If the same code was used, but a different type of error occurred, the second except statement would be used. Instead of printing geoprocessing messages, it would get a traceback object and print out the appropriate system error messages.
- The table below shows the expected errors that result from three different lines of codes that could be substituted into the code above. The first is a geoprocessing tool error, which prints out the traceback information and the geoprocessing error messages. The second and third examples are not specifically caught and print out only the traceback information.
- Follow the link:

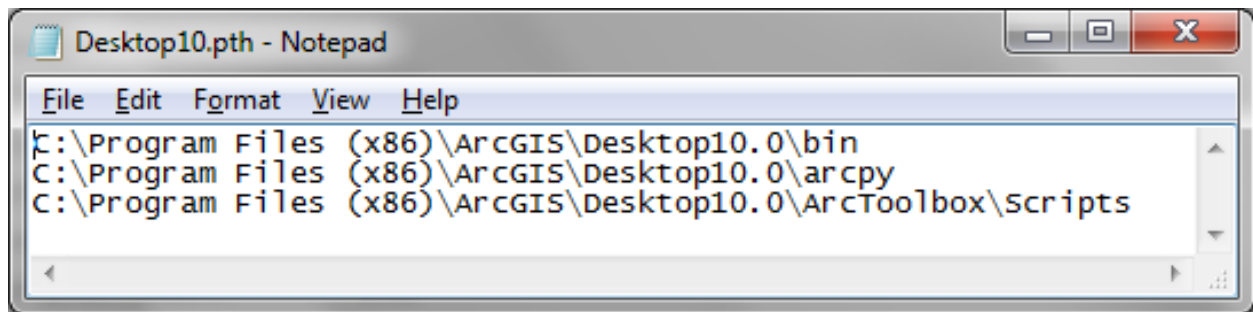
7.8 Paths and import

- When using an import statement, Python looks for a module matching that name in the following locations (and in the following order):
 1. Paths specified in the PYTHONPATH system environment variable
 2. A set of standard Python folders (the current folder, c:\python2x\lib, c:\python2x\Lib\site-packages, and so on)



3. Paths specified inside any .pth file found in 1 and 2 (Located at C:\Python26\Lib\site-packages)

For more information on this, see the following: <http://docs.python.org/install/index.html#modifying-python-s-search-path>.

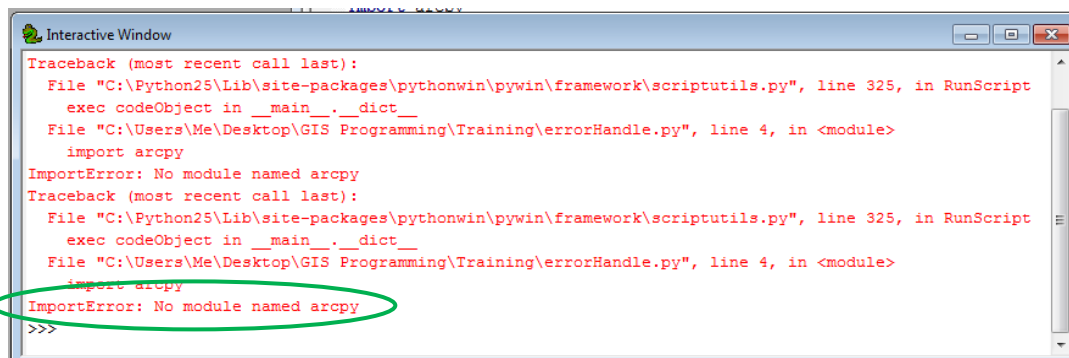


- The installation of ArcGIS 10.0 products will install Python 2.6 if it isn't already installed.
- The installation will also add the file Desktop10.pth (or Engine10.pth or Server10.pth) into python26\Lib\site-packages. The contents of this file are two lines containing the path to your system's ArcGIS installation's arcpy and bin folders. These two paths are required to import ArcPy successfully in Python version 2.6.
- When using an import statement, Python refers to your system's PYTHONPATH environment variable to locate module files. This variable is set to a list of directories.

IMPORTANT

If importing ArcPy produces either of the following errors, the required modules could not be found:

- **ImportError: No module named arcpy**
- **ImportError: No module named arcgisscripting**

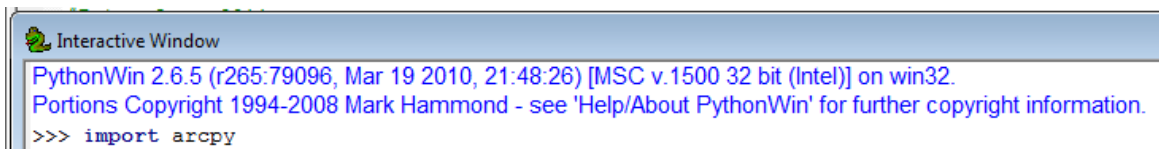


To address this, browse using Windows Explorer to the python26\Lib\site-packages folder and add or edit the Desktop10.pth file. The file should contain the two lines shown below (corrected to your system's path if they do not match):

c:\Program Files\ArcGIS\Desktop10.0\arcpy

c:\Program Files\ArcGIS\Desktop10.0\bin

- Additional information on ImportError:
 - <http://forums.arcgis.com/threads/23230-quot-ImportError-No-module-named-arcpy-quot-PYTHON?highlight=pythonwin>
 - <http://forums.arcgis.com/threads/2158-Importing-arcpy-error>
- Make sure the version on PythonWin matches the version of Python installed with ArcGIS.
 - ArcGIS 10 uses Python 2.6 therefore PythonWin 2.6 should be installed.
- Check to see that PythonWin works with ArcGIS
 - Import Arcpy into PythonWin and check that no errors are raised.



- If you can't locate PythonWin from START→ALL PROGRAMS→PYTHON26→PYTHONWIN, check the following location and execute Pythonwin.exe:
 - C:\Python26\ArcGIS10.0\Lib\site-packages\pythonwin

7.9 Exercise 6