Chapter 16
User-Defined Modules

Abstract User-defined functions enable code reuse within a script; they can also be called from other scripts. To amplify code reusability, functions can be defined in a supporting script. Scripts that house sets of related function definitions and other related code are referred to as modules. This chapter focuses on defining and importing user-modules. It also discusses how to structure development code within a module. Finally, the chapter concludes with a practical example for managing GIS temporal data and time attributes.

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

- Articulate the purpose of a user-defined module.
- Create a user-defined module containing related functions.
- Import distributed user-defined modules.
- Write absolute and relative versions of a path.
- Call a function in a user-defined module.
- Use a conditional construct to exclude code in a user-defined module.

16.1 Importing User-Defined Modules

Functions within a user-defined module can be called from other scripts. A *module* is a Python script containing related function definitions and Python statements. Every Python script is a module, but we refer to it as a module particularly when we are importing it to use its functions in another script. You're already familiar with importing Python built-in modules and the `arcpy` package installed with ArcGIS software. Our geoprocessing scripts often begin with an import statement like this:

```
import arcpy, os, sys
```

The name of a module is simply the Python file name without the '.py' extension. For example, the `os` module refers to the 'os.py' script. Once you've imported the `os` module, you can use the `os` functions by referring to them with dot notation.

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_16

For example, you can use `os.getcwd()` to determine the current working directory and `os.listdir(path)` to get a list of the files in the `path` directory.

Importing user-defined modules and accessing the functions therein works in a similar way to using built-in modules. The main difference is that the location of a user-defined module may need to be specified explicitly before importing it, as discussed next.

Example 16.1: A user-defined module.

```
# Excerpt from 'listManager.py' in
#    C:\gispy\sample_scripts\ch16\supportCode
# Purpose: Provide list and delimited string manipulation functions.
def list2String(delimiter, L):
    '''Take a list and return a delimited string.'''
    # Join fails for non-string elements, so use list
    # comprehension to cast each element to string.
    stringL = [str(i) for i in L]
    # Join the string elements of stringL
    s = delimiter.join(stringL)
    return s

def string2List(delimiter, s):
    '''Take a delimited string and return a list'''
    L = s.split(delimiter)
    return L
```

The user-defined module, 'listManager.py', shown in Example 16.1, resides in a `supportCode` subdirectory of Chapter 16 sample scripts and contains two function definitions for manipulating Python lists. (The script also contains some commented code that will be used later in this chapter. For now, leave those lines commented. In other words, leave `##` at the front of the lines). To use these functions outside of 'listManager.py', we need to first import the module, 'listManager'. The format for importing user-defined modules is the same as importing a built-in module. Try the following code:

```
>>> import listManager
```

Did it work? When you try this code, Python might throw the following exception:

```
Traceback (most recent call last):
File '<interactive input>', line 1, in <module>
ImportError: No module named listManager
```

If you didn't get this exception, try to generate it by restarting your IDE and entering this statement again. You should get an ImportError. Why should it work sometimes and not others? The answer lies in how Python imports modules. Python keeps a list of directory paths and searches the directories in this list for the files and packages that the user imports. If the file is located in one of the listed directories, the import succeeds. Otherwise, Python throws an `ImportError` exception. The directory path list is stored in the `sys` module as a property named `path` that gets initialized when the IDE is opened. You can see the list by importing `sys` and printing the `path` variable:

```
>>> import sys
>>> sys.path
['C:\\Windows\\SYSTEM32\\python27.zip','C:\\Python27\\ArcGIS10.3\\DLLs',...]
```

Only a few of the paths are shown here. The full list includes paths that are loaded automatically based on PythonWin installation dependent defaults, the PYTHONPATH environment variable, and the `os` current working directory. You can also modify the `sys.path` variable in your script. This way, you can define your own module and add its path to the list, so that it can be imported. Since `sys.path` is a Python list, you can just use the list `append` method. To add a path to the list. For instance, to add the directory 'C:/gispy/sample_scripts/ch16/supportCode' to the path, append this value. Then print the `sys.path` list again to see that the additional directory appears at the end of the list, once it has been appended:

```
>>> sys.path.append('C:/gispy/sample_scripts/ch16/supportCode')
>>> sys.path
['C:\\Windows\\SYSTEM32\\python27.zip', 'C:\\Python27\\ArcGIS10.3\\
DLLs',...,'C:/gispy/sample_scripts/ch16/supportCode']
```

When you open and run a script in PythonWin, the script's directory is automatically appended to the path. So if your supporting modules are in the same directory, as the script they support, their path does not need to be appended. For example, 'script1.py' can import the `script2` module without any modifications to the system path because both scripts reside in 'C:/gispy/sample_scripts/ch16'. By contrast, 'script3.py' throws an exception when attempting to import `script4` from the 'otherCode' subdirectory. Since 'script4.py' is not within the same directory, 'script3.py' fails to determine its location. The directory where module `script4` resides must be appended to the system path first. To do so, you must use two separate import statements, one to import the `sys` module (and this must come first) and one to import the user-defined script. Appending the path comes after `sys` has been imported and before the user-defined module is imported, as in the following example:

```
import sys
sys.path.append('C:/gispy/sample_scripts/ch16/otherCode')
import script4
```

Try this by uncommenting the first two lines of code in 'script3.py' and running it again. This example hard-codes the file path to the supporting module. To make your code portable, you can dynamically specify the path. Dynamically specifying the path to append requires some understanding of absolute and relative paths. To illustrate these concepts, we'll use files from the multi-layered 'pics' directory ('C:\gispy\data\ch16\pics'). Back in Chapter 12, there is a drawing of the hierarchical structure of the same data (Figure 12.1). The *absolute path* of a file is the full path file name, starting with the name of the drive. The absolute path to 'backSeat.jpg' on the 'C:' drive is as follows:

```
filename = 'C:/gispy/data/ch16/pics/italy/backSeat.jpg'
```

The *relative path* of a file gives its position relative to another file. Hence, it doesn't need to start with a drive name. From the point of view of 'backSeat.jpg', there are several types of relative positions:

1. *Same directory*: For files in the same directory as 'backSeat.jpg', the relative path is simply the name of the other file. For example, 'bridge.jpg' is the entire relative path to 'bridge.jpg' from the point of view of the 'backSeat.jpg' file.
2. *Subdirectory*: 'backSeat.jpg' resides in the 'C:/gispy/data/ch16/pics/italy/' directory. To refer to files in subdirectories, the relative path lists the directories that fall below this one. For example, 'canal.jpg' is relatively referenced as 'venice/canal.jpg'.
3. *Up* or *up-and-over directories:* To climb upward in the hierarchy, you can use two dots. For example, 'istanbul.jpg' is relatively referenced as '../istanbul.jpg'. The double-dots take the path up out of the 'italy' directory to the next level up, which is the 'pics' directory. Relative paths have to follow the lines connecting the file in the tree. To reference 'old_city.jpg' from 'backSeat.jpg', you have to climb out of 'italy' using double-dots and then step down into the 'jerusalem' directory. So the relative path would look like this: '../jerusalem/old_city.jpg'.

You can test relative paths using Windows Explorer. For example, browse to 'C:/gispy/data/ch16/pics/italy/'. Then enter '../jerusalem/' in the window to confirm that it takes you to the directory that contains 'old_city.jpg'.

Supporting user-defined modules can be placed in another directory with a fixed position relative to the script that imports it. The goal is that you can move the entire workspace and the code won't break due to hard-coded paths. Figure 16.1 shows the relevant portion of the sample scripts directory tree. As an example, consider looking at the relative positions of 'script1.py' and 'script4.py'. The following four steps can be used to import the `script4` module dynamically from 'script1.py'.

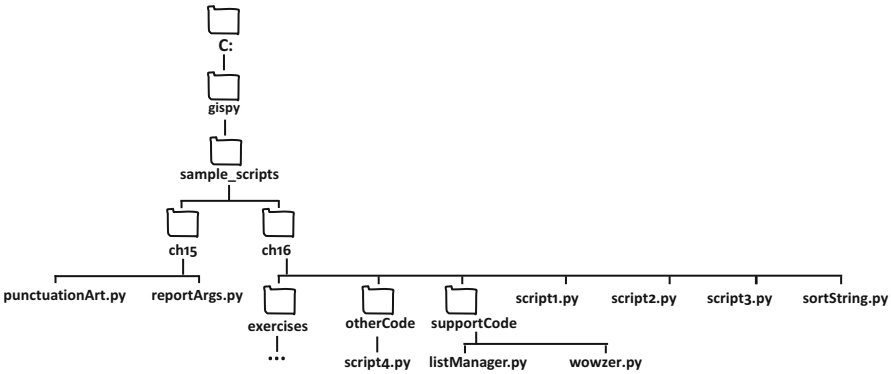

Figure 16.1 The directory tree structure for the examples discussed in this chapter.

1. Retrieve the full path of 'script1.py' using the `os` module:

```
import sys, os
scriptPath = os.path.abspath(__file__)
```

2. Combine the script path with the relative path of the user-defined module using the `os.path.join` method.

```
scriptDir = os.path.dirname(scriptPath)
relativePath = 'otherCode'
modulePath = os.path.join(scriptDir, relativePath)
```

3. Append `sys.path`.

```
sys.path.append(modulePath)
```

4. And finally, import the module.

```
import script4
```

These steps will usually be placed together near the beginning of the main script. As another example, to import the `punctuationArt` module (a Chapter 15 sample script) from 'script4.py', the following code could be placed at the beginning of 'script4.py':

```
import sys, os
scriptPath = os.path.abspath(__file__)
scriptDir = os.path.dirname(scriptPath)
relativePath = '../../ch15'
```

```
modulePath = os.path.join(scriptDir, relativePath)
sys.path.append(modulePath)
import punctuationArt
```

16.2 Using Functions in Another Module

To call a function in another module, import the module and then you can use dot notation with the following format:

```
moduleName.functionName (arg1, arg2, ...)
```

The two functions in Example 16.1 are designed for working with lists. One converts a delimited string (such as a string of comma separated values) to a list. The other converts a list to a delimited string. These functions are saved in the module 'listManager.py', so instead of just using the function names to call them, you need to use the dot notation to indicate where Python should look for them. The following code imports the module and calls the `string2List` function using dot notation:

```
>>> import listManager
>>> theString = 'z;x;y'
>>> theList = listManager.string2List(';' , theString)
>>> theList
['z', 'y', 'x']
```

This Interactive Window code works because the path to this module was appended to `sys.path` earlier in the same IDE session. The user-defined module functions can also be called from within another script, as in Example 16.2. The script 'sortString.py' resides in the directory above 'listManager.py'. This script first appends the path to 'listManager.py', then imports the `listManager` module and calls its functions. The purpose of 'sortString.py' is to sort the items in a delimited string. To do so, it converts the string to a list, sorts the list, and then converts the list to a string. To see how it works, run 'sortString.py'. The printed output should look like this:

```
>>> sortString.py results:
fireID~unit~shelter~alt~campgr -> alt~campgr~fireID~shelter~unit
```

Two of the functions it calls, `string2List` and `list2String`, are defined in 'listManager.py'. The third function called in this example is the list `sort` method discussed in Chapter 4. This is an object method, so instead of module-dot-functionName, `mylist.sort()` is called on the object, with the object-dot-method syntax. The purpose of the double hash comments, the reload statement, and the `delimStrLen` function are discussed next.

Example 16.2: Using a user-defined module from a script.

```
# excerpt from sortString.py
# Purpose: Sort the items in a delimited string

import sys, os
scriptPath = os.path.abspath(__file__)
scriptDir = os.path.dirname(scriptPath)
relativePath = 'supportCode'
modulePath = os.path.join(scriptDir, relativePath)
sys.path.append(modulePath)
import listManager

delimiter = '~'# set delimiter
# Set delimited string.
theString = 'fireID~unit~shelter~alt~campgr'

# Get a list from the string.
# module.function(arg1, arg2,...) format.
theList = listManager.string2List(delimiter, theString)

# Sort the list. Sort is a native list method
# (not a method defined in listManager.
# Notice the difference in how it is called.
# It's called with object.method format,
# where the object is mylist.
theList.sort()

# Get a string from the list.
# module.function(arg1, arg2,...) format.
newString = listManager.list2String(delimiter, theList)

print '\nsortString.py results:'
print '{0} -> {1}'.format(theString,newString)
```

16.3 Modifying User-Defined Modules (Reload!)

When you import a user-defined module, it loads the module and stores the name in an internal list. As you step through a script in the debugger, you may notice that the first time you step past an import statement for a large package, such as `arcpy`, the import statement takes a few seconds; whereas, if you run the same script again within the same PythonWin session, you can step past this import instantly. This is because during the second run, Python quickly looks up the module in its internal list of loaded modules, finds that module name, and does not reload the module contents. This makes Python more efficient, but it can be baffling when you first work with user-defined modules, because it can seem as if your updates to a user-defined module are not being heeded. As an example, run sample script 'sortString.py' (if you haven't already done so). Next, make a change to 'listManager.py' by adding the `delimStrLen` function in 'listManager.py'. To do so, you can select the following lines of code which are commented out in the script and use the block uncomment shortcut (Alt+4):

```
def delimStrLen(delimiter, s):
    '''Return the number of items in a delimited string.'''
    theList = string2List(delimiter, s)
    return len(theList)
```

Save 'listManager.py'. Then uncomment the following code in 'sortString.py', to call the new function:

```
num = listManager.delimStrLen(delimiter,theString)
print 'Number of items: {0}'.format(num)
```

Save and run 'sortString.py' again. Despite the fact that you've just added the function to listManager, Python throws the following traceback exception:

```
AttributeError: 'module' object has no attribute 'delimStrLen'
```

As far as Python is concerned, there is no such function in the `listManager` module, because when the module was loaded, that function was not there. When you modify a supporting module, the changes are not automatically recognized by the file that imports it. To force new content in a module to be loaded, you can reload it with the following steps:

1. Shift the focus to the user-defined module, by clicking on that module's window. In our case, click on 'listManager.py'.
2. Then, click the 'Import/Reload' button  on the PythonWin standard toolbar next to the 'Run' button,

The module has now been reloaded. To see the results, run 'sortString.py' again. The results should appear as follows:

```
>>> sortString.py results:
fireID~unit~shelter~alt~campgr -> alt~campgr~fireID~shelter~unit
Number of items: 5
```

The `5` is returned by the new function call, which counts the number of items in the delimited string.

There is also a built-in `reload` function which has the same effect as clicking the 'Import/Reload' button. This function reloads a previously imported module passed to it as an argument. The argument must be a `module` object, so it must be successfully imported before it can be reloaded. In other words, you must put an import statement before a reload statement, as in the following example:

```
import listManager
reload(listManager)
```

Temporarily placing a reload statement inside the main script can be useful while making repeated modifications to the code in user-defined modules. However, usually the reload statement should not be left in code to be shared, as it can slow the performance. You may have noticed that the first time a module is imported, Python creates a corresponding file with a '.pyc' extension. For example, look in the sample scripts Chapter 16 'supportCode' directory to see the 'listManager.pyc' file. This is an intermediate file that encodes the module in a format called 'byte code'. That file gets created based on the user-defined 'listManager.py' module the first time you import `listManager` and it gets rebuilt each time that module is reloaded. Once it has been created, this file is executed when the current IDE session uses `listManager`.

16.4 Am I the Main Module? What's My Name?

When you reload a module or import it for the first time, it not only stores the names of the functions, but it also executes any statements in that script that are outside of the functions. As an example, uncomment the reload statement in 'sortString.py' and then uncomment the following code at the end of 'listManager.py':

```
print '\nIn listManager.py, test string2List: '
theString = 'z;x;y'
theList = string2List(';', theString)
print '{0} -> {1}'.format(theString,theList)
```

Save both scripts and run ‘sortString.py’. You’ll see the following output:

```
In listManager.py, test string2List:
z;x;y -> ['z', 'x', 'y']

sortString.py results:
fireID~unit~shelter~alt~campgr -> alt~campgr~fireID~shelter~unit
Number of items: 5
```

The first two lines are output from the ‘listManager.py’ file. Usually, you don’t want side-effects like this when you import a file. However, as you develop functions for user-defined modules, you need to test them. To include test code in your module and still avoid this undesired artifact, you can programmatically determine whether you’re running the module directly or importing it from another module. The built-in variable, `__name__`, gets assigned the value `'__main__'` when the module is loaded as the main module. When the module is imported, the value of `__name__` is set to the name of the module. To see this in action, we’ll take a brief detour to another script, since you can only see this print the first time a script is imported during a PythonWin session. The script named ‘wowzer.py’ contains a line of code to print the value of `__name__`:

```
print '**** Name = ', __name__
```

Run ‘wowzer.py’ with the ‘Run’ button and it prints the following:

```
>>> **** Name = __main__
```

But import ‘wowzer.py’ in the Interactive Window and it prints `wowzer`:

```
>>> import wowzer
**** Name = wowzer
```

Now returning to ‘listManager.py’, you use this difference in the value of `__name__` to restrict code in a user-defined module from being run when it is imported, by using a conditional expression that compares the value of `__name__` to `'__main__'`. To try this, insert the following line of code into ‘listManager.py’ just after the last function definition:

```
if __name__== '__main__':
```

Then indent the four lines of code that you uncommented at the start of Section 6.4 and add a dedented print statement to the end of the script, so that it appears as follows:

```
if __name__== '__main__':
    print '\nIn listManager.py, test string2List: '
    theString = 'z;x;y'
```

```
    theList = string2List(';' , theString)
    print '{0} -> {1}'.format(theString,theList)
print 'Zowee!'
```

Save and run 'listManager.py' once more to see that it prints output. Then restart PythonWin, open 'sortString.py', and run it again to confirm that the code inside the conditional construct is not executed, and so it only activates the dedented print statement (Zowee!).

16.5 Time Handling Example

User-defined modules group related functions for convenient re-use. Python has two built-in time related modules, `time` and `datatime`. The user-defined `reportTime` module in Example 16.3 (located in 'C:\gispy\solutions\ch16\exercises\exerSupportCode') contains some convenient functions that leverage these built-in modules. It uses the `ctime` and `sleep` methods from the `time` module. The `time` module can access the current time and convert time across time zones. The `ctime` function returns a string representing the current date and time. The following example shows what it would have printed if called just before midnight on New Year's Eve back in 1999:

```
>>> import time
>>> time.ctime()
'Fri Dec 31 23:59:59 1999'
```

The `sleep` function suspends processing for the given amount of seconds. This is used in 'timeReport.py' to demonstrate a time lapse. If you try the following code, you'll have to wait 10 seconds before the next command prompt appears:

```
>>> time.sleep(10)
```

The `datetime` module has data types for dealing with dates, times, and time intervals. The 'datetime' type stores both date and time information. The following code creates a `datetime` object for that moment before the start of the twenty-first century and then prints the hour, year, and weekday of the object:

```
>>> import datetime
>>> dt = datetime.datetime(1999, 12 ,31, 23, 59)
>>> dt
datetime.datetime(1999, 12, 31, 23, 59)
>>> dt.hour
23
```

```
>>> dt.year
1999
>>> dt.weekday()
4
```

The `datetime` type has a `year`, `month`, `day`, `hour`, `minutes`, and `seconds` properties. The `weekday` function returns an index, 0 for Monday, 1 for Tuesday, and so forth. Example 16.3 uses the `weekday` function and a dictionary to return a weekday name instead of an index.

Dates can be differenced. When a subtraction sign is used between two `datetime` objects, a `timedelta` type object is returned. The following code creates a `datetime` object for noon on the last day of 1999 and demonstrates ways to use the `timedelta` object:

```
>>> dt2 = datetime.datetime(1999,12,31, 11, 59)
>>> timeDiff = dt - dt2
>>> timeDiff
datetime.timedelta(0, 43200)
>>> timeDiff.days
0
>>> timeDiff.total_seconds()
43200.0
>>> hrs = timeDiff.total_seconds()/3600
>>> hrs
12.0
```

The `timeDiff` object stores the time difference. The `total_seconds` function returns the total time difference in seconds. These moments are 12 h apart (at noon and midnight of the same day). `datetime` objects can also be compared for temporal ordering, by using Python comparison operators.

```
>>> dt2 < dt
True
>>> datetime.datetime.now() < dt
False
```

The `time.ctime()` function call returns a string describing the current date and time; Whereas, the `datetime.datetime.now()` function returns a `datetime` object for the current moment—which is greater than `dt`. Example 16.3 uses `datetime.datetime.now()` in the `getTime` function to get a `datetime` object. In the test code at the bottom of the script this is called before and after a 5 second pause in execution. Then these two objects are passed into the `reportDiffTime` method to print how much time elapsed between the two.

Example 16.3: A user-defined module.

```
# timeReport.py
import datetime, time

def reportTime(message='The current date and time is'):
    '''Print the current time'''
    now = time.ctime()
    print '{0}: {1}'.format(message, now)

def getDay(theDate):
    '''Given a date, return the day of the week'''
    index = theDate.weekday()
    wDict = { 0 : 'Monday', 1 : 'Tuesday', 2 : 'Wednesday',
              3 : 'Thursday', 4 : 'Friday', 5 : 'Saturday',
              6 : 'Sunday'}
    return wDict[index]

def getTime():
    '''Report the current time'''
    t = datetime.datetime.now()
    return t

def reportDiffTime(start, end, message= 'Time elapsed'):
    '''Print the number of seconds that passed
    between 'start' and 'end'.'''
    difference = end - start
    seconds = difference.total_seconds()
    print '{0}: {1} seconds.'.format(message, seconds)

if __name__ == '__main__':
    reportTime('Script began running at')
    # Get current time.
    beforeSleep = getTime()
    time.sleep(5)
    # Get current time.
    afterSleep = getTime()
    message = 'Time elapsed for sleeping'
    # Print the time difference.
    reportDiffTime(beforeSleep, afterSleep, message)
    reportTime('Script completed at')
    print 'Hurray! I like {0}s.'.format(getDay(afterSleep))
```

16.6 Summary

We'll be discussing a Python structure called a `class` in an upcoming chapter. User-defined modules are often used as containers for classes, so several of these concepts will come up again when we reach that topic. The following list summarizes the key concepts in this chapter:

- When calling a function from within the module where it is defined, you just need to use the name. When calling it from another module, you need to start with the name of the module and use dot notation, so that Python knows where to find the function.
- Importing a user-defined module in the same directory is the simplest scenario. The system path does not need to be appended. In all other cases, the system path needs to be appended.
- You must use a separate statement to import `sys` and a user-defined module when the system path needs to be appended. First import `sys` before appending the path, then use a second import statement for the user-defined module.
- You can append a hard-coded path, but in most applications this approach hinders portability. Generally, scripts dynamically determine their own location and then append a relative path to imported modules.
- When you make modifications to a module, remember to reload it so that the changes are recognized by the script that imports the module.
- You can use a conditional expression that checks if you're in the main script to exclude select portions of user-defined module code from imports. This allows you to use the script both as a main script and as a supporting module.

16.7 Key Terms

User-defined module
`sys.path` variable
Built-in `__file__`variable
Absolute vs. relative paths
Built-in `reload` function
Built-in `__name__` variable
Built-in `time` and `datetime` modules

16.8 Exercises

1. **owlCall.py** Write a script which imports sample script 'owl.py' and calls `printOwl`. Assume 'owlCall.py' and 'owl.py' reside in the same directory.
2. **wazzup.py** Sample script 'wazzup.py' (in 'C:\gispy\sample_scripts\ch16\exercises\favScripts') already imports scriptA. Modify the script so that it also

imports scriptB, scriptC, scriptD, and scriptE. These modules are scattered throughout 'C:/gispy/sample_scripts/ch16/exercises'. Append their relative paths to the system path variable, so that they can be imported. Check that the imports are working three ways and note your thoughts as comments within the script:

(a) First check the output. Each of the A-E scripts contains print statements to help you confirm successful import. The printed output should look like this:

```
>>> I am in scriptA.py
I am in scriptB.py
I am in scriptC.py
I am in scriptD.py
I am in scriptE.py
```

Can you explain why your output might only print the last line ('I am in scriptE.py'), when you finally have it working?

(b) Next, restart PythonWin, copy 'wazzup.py' to the 'C:/gispy/sample_scripts/ch16/exercises' directory, and rename it to 'wazzup2.py' and run it. When placed in this directory, only the first statement ('I am in scriptA.py') is printed. Can you say why this does print but the second statement doesn't? If the script used absolute paths, instead of relative paths would all the imports have worked? Why is it necessary to restart PythonWin before running this test?

(c) Finally, restart PythonWin and temporarily rename the 'ch16' directory containing these scripts to 'ch16_temp'. Open and run the original 'wazzup.py' script, again. If relative paths were used correctly, why should the imports still succeed? Rename the directory to ch16 when this test is complete.

3. **abc.py** Create a script named 'abc.py' and place it in the Chapter 16 exercises directory, 'C:/gispy/sample_scripts/ch16/exercises'. Add code to 'abc.py' that imports Chapter 15 sample script 'reportArgs.py' and calls `printArgs`. Make the script portable, assuming the scripts remain in the same relative positions. (For example, if the 'gispy' directory is renamed as 'gypsy', the import should still work.) When you run your initial solution, the argument list should be printed twice. Can you explain why?

Example input:
a b c

Example output:

```
Argument 0: C:\gispy\sample_scripts\ch16\exercises\abc.py
Argument 1: a
Argument 2: b
Argument 3: c
Argument 0: C:\gispy\sample_scripts\ch16\exercises\abc.py
Argument 1: a
Argument 2: b
Argument 3: c
```

Modify 'reportArgs.py' so that it only executes `printArgs` if it is run as the main script. Then run 'abc.py' again and confirm that the argument list is only printed once.

4. **mathMod.py** Write a user-defined module that contains three functions. One named `add` adds two arguments and returns the result, one named `mult` multiplies two arguments and returns the results, and one named `sub` subtracts two arguments and returns the results. Add a conditional expression that checks if `__name__` is equal to `'__main__'` and add the following test code inside this conditional block:

```
A = float(sys.argv[1])
B = float(sys.argv[2])
sumV = add(A, B)
product = mult(A, B)
difference = sub(A, B)
print sumV, product, difference
```

Test your code against the following sample input and output:

Example input:
5 2

Example output:
```
7.0 10.0 3.0
```

5. **listUniqueValues.py** The sample script named 'listManager2.py' which resides in the 'listManage' subdirectory is similar to 'listManager.py' in Example 16.1, but it has an additional function named 'uniqueList'. Run listManager2.py to see what the additional function does. Then create a script, 'listUniqueValues.py', in the 'C:/gispy/sample_scripts/ch16/exercises' directory, which imports the `listManager2` module (using a relative path) and uses this additional function. The script should use a search cursor to get a list of the values for a given field. Then it should get a list of unique field values. Next, it should sort the unique field value list. Finally, it should print the list. The script should take two arguments: a full path file name of a shapefile and the name of a text field.

Example input: C:/gispy/data/ch16/park.shp COVER

Example output:
```
[u'orch', u'other', u'woods']
```

6. **csvStrings.py** The sample script, 'csvString.py', which can be found in the 'C:/gispy/sample_scripts/ch16/exercises' directory, contains a list of annual wheat yields between 2000 and 2007 for five farms. These are stored as comma separated value (csv) strings. Add code to 'csvString.py' to import the `listManager3` module, using a relative path to 'listManager3.py' which resides in

the 'listManage' subdirectory. Then add code to 'csvString.py' to use functions from the listManager3 module to analyze the wheat yield data: First, use `delimStrLen` to find and print the number of wheat yield sample time steps recorded for each farm. Run the script and print the output as follows:

```
Site 1 has 8 samples: 1,4.07,4.21,4.15,4.64,4.03,3.74,4.56
Site 2 has 8 samples: 2,,4.29,4.4,4.69,3.77,4.46,4.76
Site 3 has 8 samples: 3,3.9,4.64,4.05,4.04,3.49,3.91,4.52
Site 4 has 8 samples: 4,3.63,,4.92,4.64,3.75,4.1,4.4
Site 5 has 8 samples: 5,3.15,,4.08,4.73,3.61,3.66,
```

Second, uncomment `delimStrLen2` in 'listManager3.py' and modify 'csvStrings.py' to call this function instead of `delimStrLen`. You'll see that the results are more accurate. In fact, wheat yield readings are not available for some farms during some years; Therefore, some of the entries are empty strings. The simpler function `delimStrLen` counted these, but `delimStrLen2` avoids counting empty strings:

```
Site 1 has 8 samples: 1,4.07,4.21,4.15,4.64,4.03,3.74,4.56
Site 2 has 7 samples: 2,,4.29,4.4,4.69,3.77,4.46,4.76
Site 3 has 8 samples: 3,3.9,4.64,4.05,4.04,3.49,3.91,4.52
Site 4 has 7 samples: 4,3.63,,4.92,4.64,3.75,4.1,4.4
Site 5 has 6 samples: 5,3.15,,4.08,4.73,3.61,3.66,
```

7. **timeTrig.py** Create and save a script named 'timeTrig.py' in the directory named 'C:/gispy/sample_scripts/ch16/exercises/timeTrigDir'. The 'timeTrig.py' script will import the user-defined module `timeReport` using a relative path. Find 'timeReport.py' in the 'exerSupportCode' subdirectory. Run 'timeReport.py' as the main script to observe how the functions work. Then use the time functions in your 'timeTrig.py' script to report how long it takes to calculate the sine of the cells of each raster in the given workspace using the Sine (Spatial Analyst) tool. Be sure to check out the spatial analyst extension in the script before calling the Sine tool. The script should take two input arguments using `sys.argv` to set an input geodatabase workspace and an output directory. To name the output files, append `'.tif'` to each raster name. Print the name of each successful sine output and the overall time elapsed while performing the sine calculations.

   Example input: C:/gispy/data/ch16/rastTester.gdb C:/gispy/data/ch16

   Example output:

```
>>> C:/gispy/scratch\elev.tif created.
C:/gispy/scratch\landcov.tif created.
C:/gispy/scratch\soilsid.tif created.
```

   `...` (and so forth)

```
Calculating sine took: 11.695 seconds.
```

8. **bufferArgs.py** Write a portable script named 'bufferArgs.py' in the 'C:/gispy/sample_scripts/ch16/exercises' directory that performs a buffer based on two user input arguments: the full path file name to a shapefile and a numerical distance measure (use the default units). Use the functions from the user-defined module, 'argHandler.py' in the `exerSupportCode` subdirectory to check that the first user argument is a shapefile and the second user argument is a number. If either fails, exit the script, else create 'buffOut.shp' in the 'C:/gispy/scratch/' directory. The code can assume this directory already exists.

   Example input 1: C:/gispy/data/ch16/park.shp 5

   Example output 1:

   ```
   >>> buffOut.shp created in C:/gispy/scratch
   ```

   Example input 2: C:/gispy/data/ch16/jack.jpg 4.5

   Example output 2:

   ```
   >>> Expecting a shapefile for arg 1. Got RasterDataset instead.
   No buffer can be computed.
   ```

   Example input 3: C:/gispy/data/ch16/park.shp ba

   Example output 3:

   ```
   >>> Input ba is not a floating point number.
   No buffer can be computed.
   ```