

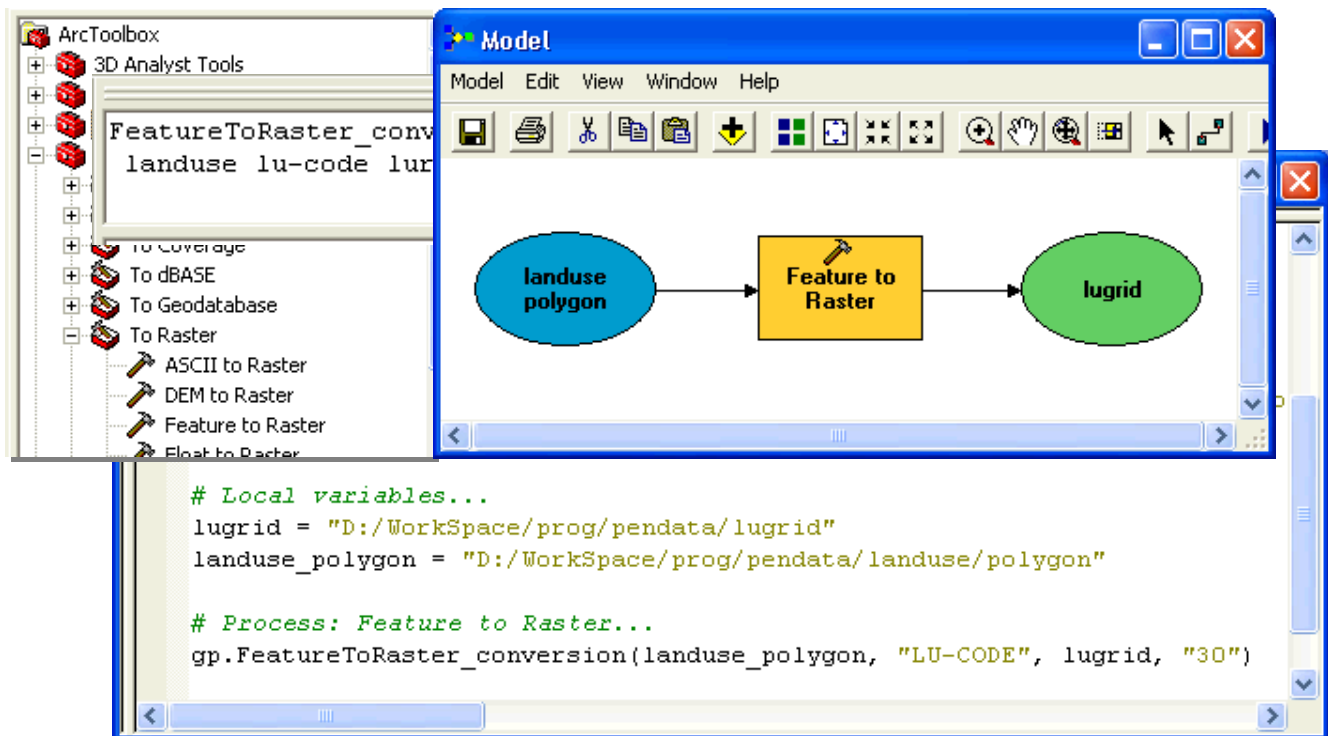
1. Scripting and Python

In GIS modeling or GIS data management, you often need to process a series of steps to get your work done. You might have a workspace full of data to reproject, clip to a study area, or combine in some way to get to an output product. We also often need to process data in different ways depending on conditions – thus we need to make *decisions*, and while high-level decisions require considered thought by users there are many low-level decisions that can be aided by a script programming model.

The main purpose of script programming is to *automate* tedious work in processing our data, and to use *logic* to direct that process. I think those two words are key: *automate* and *logic*. They distinguish this activity from the more common *interaction* we use computers for most of the time. To communicate by email, to compose a document, or to design a map, we need to *interact*; to process a lot of data, we need to *automate* and use *logic* to guide the automation.

In geoprocessing script logic, we'll make decisions that allow us to, for example, handle rasters differently from vector data, or only set map projections for unprojected data, or process datasets collected only at certain times. For any serious GIS work, scripting and other forms of programming becomes a necessity, not an option.

In this exercise, we'll explore the use of Python to create scripts that allow us to use the vast suite of geoprocessing tools in ArcGIS. All of the tools you can use from ArcToolbox or in a model can also be used in a Python script. And these scripts can be made into script tools that we can use like any other geoprocessing tools. We'll be doing this in a later section of the exercise.



1.1 Python

We'll start these exercises with a look at Python itself, primarily by accessing it through PythonWin, a Windows program that gives you access to writing Python scripts. We're going to take a very brief look at Python and PythonWin, but to learn more try the following resources:

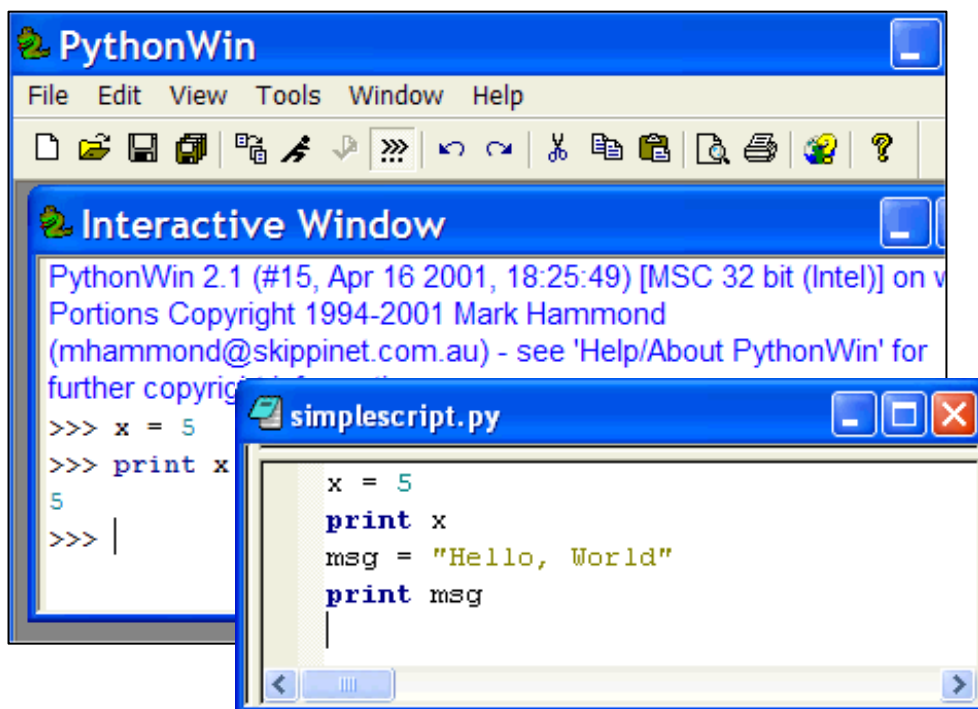
1. Books on the Python language, such as *Learning Python* by Mark Lutz and David Ascher (O'Reilly, <http://python.oreilly.com/>). You don't need to learn much about Python to use it for geoprocessing scripts, but learning more about the language will help you go beyond the basics.
2. ESRI documentation: *Writing_Geoprocessing_Scripts.pdf*, 80 pages. However, use it carefully. In at least its first edition (with the release of ArcGIS 9 in 2004), it contained a few errors in code – the latest report from ESRI is that these are fixed in the current version. If you want to use code from this, the best thing to do is to look in ArcGIS Desktop Help Online, described next:
3. ArcGIS Desktop Help **Online** (in the ArcCatalog and ArcMap Help menu). Go to Geoprocessing/Automating your work with scripts. You can also find these topics in the regular Help system

Integrated Development Environments (IDEs)

You can write Python programs using a text editor, and some text editors like Notepad++ do a pretty good job. But you can't run your program or debug it from a text editor – for this you need another program, an integrated development environment (IDE), to do this. The next section describes PythonWin, an IDE that runs when you edit a script from ArcGIS (by right-clicking the script in ArcToolbox or ModelBuilder, and choosing **edit**). PythonWin is fairly easy to use, but it only runs in Windows, and isn't bug-free. Some people like to use **IDLE** instead. It's not quite as comfortable for Windows users, but it's stable and runs in Unix and other operating systems as well as Windows.

PythonWin: Interactive Window vs. Script

We'll be running PythonWin, which includes, among other things: (1) an **interactive window**, where we can explore how variables work, and also displays all of the results of our operations; and (2) any number of **scripts** we're editing, in separate windows. If we run a script, its output will be displayed on the interactive window. We will start by using the interactive window to try things in, and you'll find it handy in the future as well.




1.2 Running PythonWin

The following brief intro assumes you have Python and PythonWin installed on your computer, which should be true if you have ArcGIS installed. If not, they are free downloads from www.python.org. (Make sure to get both Python and PythonWin.) Caution: don't try to update Python to a new version on a computer with ArcGIS installed.

- Start PythonWin from the start menu. You'll initially be presented with the main interface, with menus and tool buttons, and the Interactive Window, with a prompt looking like `>>>`.
- Assign 5 to a variable `x` by typing "`x = 5`" (no quotations) at the prompt.
- Then print the current value of `x` to the interactive window with "`print x`"

```
>>> x = 5
>>> print x
5
>>> msg = "Hello World"
>>> print msg
Hello World
>>>
```

- Now assign "Hello World" to a variable `msg`, and print it.
- Now create a script with the New button or File>New or Ctrl-N, choosing Python Script not Grep, and put the following code in it:
`x = 3`
`print x`
 - Use Windows or ArcCatalog to create a "**mypy**" folder in **prog**.
 - In PythonWin, save your script in the **mypy** folder as **script1.py**.
 - Now, after making sure you can see the Interactive Window, but with the script editing window active, run it with the run button . You should get a Run Script dialog with the file path of your script shown.

? Does this make sense? What did the Interactive Window display when you ran the script?

- Now edit your script to look like this, then run it:

```
x = 3
msg = "Hello World"
print msg
```

- Now edit it and run it as follows.

```
x = 3
msg = "Hello World"
print x * msg
```

- What happened? _____

1.3 Mathematical Computation Using Operators

Python provides a variety of common mathematical operators, typical of most programming languages, and many more through modules you import – we’ll look at these later. See one of the books recommended above, or the help system (under “numeric types”), to learn more about these. Common operators are +, -, *, /, ** (raise to a power), % (used for modulo, remainder from division).

Integer vs. Floating Point Results. One rule to remember is that if all of the inputs are integer, the result will be integer – even for division. Any input that is floating point makes the output floating point.

➤ Try each of the following expressions on the **interactive window** to see how this works. Enter expressions that Python will display the value for, and fill in the result here.

expression to enter	result	Notes
2 + 3		integer result
2. + 3		2. is floating point, so gives floating point result
2 - 3		
2 * 3		integer result
2. * 3		floating point
5 / 2		Integer
5. / 2		
5 % 2		not percent, but modulo – remainder from division
az = 270 newaz = az+180 print newaz % 360		one use for the modulo -- think azimuth (add 180 to azimuth reverses direction)
5 ** 2		
25 ** 0.5		there is no sqrt() function, unless you add the math module

See the other sources for more on these.

1.4 String manipulation

A *string* is a sequential set of characters of text, like ‘San Francisco’ or ‘d:/prog/hmbarea/landuse.shp’ or ‘94132’. It helps to think of them as a sequence of characters, a string of characters, with each character having a position starting with zero. There are many ways to work with text strings in Python, and many of these come in handy in GIS work, especially when working with datasets, fields, and text field values. Learn more about strings in a Python manual, or the help system.

String methods can be used with string objects. A string object might be a string literal (like ‘a string’ or ‘d:/prog/hmbarea/landuse.shp’) or a variable that has been assigned a string. Methods are applied by writing them in after a dot following the string object. Methods always end with parentheses, even if no parameters are needed.

Using Help for String Methods in Python

There are more than 30 built-in methods that can be used to process strings. They are each described in the help system. To find them from PythonWin, use the Help menu to go Python Manuals, and in the Library Reference find the section 2.1.5.1 String Methods, under 2.1.5 Sequence Types.

The best way to understand strings is to try them in Python, which we’ll do next.

- In the *interactive window*, try the following sequence of statements, which illustrate string objects and methods, one at a time. Evaluate each statement. There's a lot here.

Type in interactive window	Result	Notes
<code>print 'fred'.capitalize()</code>		s.capitalize() applies the capitalize method to the s object.
<code>s = 'fred'</code> <code>print s.capitalize()</code>		
<code>print s[0]</code>		Strings can be dealt with as a list of characters, with the first one having a zero index. A set of string characters can be defined with a format like 1:3, meaning 'from the beginning of 1 to the beginning of 3' -- so it doesn't display the character that is at index 3.
<code>s1 = s[1]</code> <code>print s1</code>		
<code>print s[2]</code>		
<code>print s[-2:]</code>		
<code>print s[2:3]</code>		
<code>print s[2:4]</code>		
<code>s2 = s.upper()</code> <code>print s2</code>		We can assign the result of a string method to a new variable.
<code>s3 = s + s2</code> <code>print s3</code>		Concatenate strings with the + operator.
<code>selstr = '"elev" > 1000'</code> <code>print selstr</code>		Strings can include either single or double quotes marks. To enter a literal, use the other type to surround the string.
<code>othersel = "'elev' > 1000"</code> <code>print othersel</code>		
<code>print s.isupper()</code>		Some methods return Boolean true (1) or false (0). Others return index values.
<code>print s2.isupper()</code>		
<code>p = 'd:/work/lu.shp'</code> <code>print p.find('.')</code>		
<code>print p.find('/')</code>		
<code>plist = p.split('/')</code> <code>print plist</code>		You can use methods to parse out different parts of a path string. Split creates a Python list , from which we can use different elements.
<code>print plist[0]</code>		
<code>print plist[1]</code>		
<code>p2 = 'd:\\work\\soil.shp'</code> <code>print p2</code>		A backslash '\' is used with a following character for many special purposes, like \n for new line. To include a single '\' in a string, preface it with another.
<code>print 'Jerry\'s Kids'</code>		
<code>print 'Jerry\'s\nKids'</code>		
<code>p3 = r'd:\work\soil.shp'</code> <code>print p3</code>		Prefacing a string with r (raw) forces backslash to be used as is. This is handy for copying and pasting file paths, since you don't have to go back in and double them.

Optional:

- ? What are some methods that return Boolean results of 1 (true) or 0 (false)?

- Try another string method, listing it here and describing what it does.

1.5 Working with modules

Python has a reasonable set of built-in methods for common programming tasks, but relies substantially on **modules** for methods. Python comes installed with a large number of modules, which you can find described in the Global Module Index of the help system. Have a look – note that many of these are platform specific, especially for Windows and Mac interfaces. Some of the more useful modules are **math**, **sys**, **random**, **array** and **os.path**.

There are also many modules you can download – for example numeric processing, such as **numpy** – but to find the latest, do searches at www.python.org or google. One page that lists several is <http://www.python.org/moin/NumericAndScientific>.

To use a given module, it must be imported first. You do a lot of importing modules in Python. Normally you would put a line **import <name of module>** at the top of your program. For instance:

```
import sys
```

would occur at the top of your program before you use any **sys** methods. It doesn't have to be the first line, just before you use it. If you are entering commands through the interactive window, then just do the import before you need to use it.

Multiple modules, separated by commas, can be entered at one time, such as:

```
import sys, string, math, os, arcgisscripting
```

You can also write your own modules for frequently needed routines. See Python texts for more information on this.

For now, we'll explore some built-in modules.

- Before we look at these, copy P:\Courses\Exted\G9021\prog to your D: drive, so you'll have some data to look at. You should end up with d:\prog\data, d:\prog\HMBarea, d:\prog\pendata, d:\prog\surf_bld, d:\prog\py and maybe some other folders.

math and random modules

Many common mathematical functions are accessed via the **math** module. For instance, you would need this for trigonometric functions or logarithms. (To use complex numbers, use **cmath** which has similar functions, but allows for complex number results.)

- As before, try the following statements in the interactive window. (Feel free to run some of these in programs as well, just remember to include the import statement at the top.)

Enter:	Results:	Notes:
<pre>import math print math.log10(100)</pre>		Enter it before you use it All module functions are prefaced with the module name. This allows for re-use of function names in different modules.
<pre>print math.log(100)</pre>		Natural log
<pre>print math.pi</pre>		pi is a constant, so it doesn't need parentheses
<pre>pi = math.pi print pi</pre>		If you don't want to type "math.pi" all the time, assign it to a variable pi .
<pre>pi</pre>		You don't have to say "print" to see the value of a variable. Print formats it – handy sometimes, but not for this.
<pre>print math.sin(radians) print math.cos(radians) print math.tan(radians)</pre>		Trig functions use radians (same as all languages and Excel). Try 0.7854 radians (same as 45 degrees)
<pre>degrad = pi /180 45 * degrad</pre>		Convert degrees to radians
<pre>sin = math.sin sin(45 * degrad) sin(90 * degrad)</pre>		Even functions (like sin) can be assigned to a variable. Try it with cos and tan too.
<pre>math.e</pre>		
<pre>math.hypot(3,4)</pre>		What's a <i>hypotenuse</i> ?
<pre>x1 = 520382; y1 = 4152373 x2 = 520475; y2 = 4152963</pre>		You can put multiple statements on a line with a semicolon.
<pre>xr = x2 - x1 yr = y2 - y1 math.hypot(xr, yr) math.sqrt(xr**2+yr**2) (xr*xr+yr*yr)**0.5</pre>		Pythagorus invented Python, right? For this example, think UTM coordinates. What do the results represent?
<pre>import random random.random() rnd = random.random rnd()</pre>		Enter the method or function multiple times to get multiple results.
<pre>mu = 50 s = 10 print random.gauss(mu, s)</pre>		Enter the print statement multiple times to get multiple results.

- *Optional:* Look up the **math** and **random** modules in the help system to see what else they can do. For matrix operations, download **numpy** or other numeric modules.

1.6 Creating functions with *def*

Somewhat similar to a module, but much simpler, is a function you define in your code to be used simply by using the function name later in your code, and providing any parameters you want to use. The function can thus be used as a variable in your program. This is easier to understand with an example. The following code defines a radians function that simply converts degrees to radians, and also defines a degrees function that converts radians to degrees. There are similarly named built-in functions in Excel.

- From here on out, we'll want to write scripts and save them into your "mypy" folder. For each script use Ctrl-n or File New to create a new script. Maximize your PythonWin screen so you can see both the script and the interactive window (where outputs go). Then after composing it, save it to your "mypy" folder, and run the script with the run button. In some cases, I'll suggest names for your scripts; otherwise make one up.
- Try the following code in a script saved into your d:/prog/py folder as **defRadians.py**.

```
import math
def radians(angdeg):
    return angdeg * math.pi / 180
def degrees(angrad):
    return angrad * 180 / math.pi
print math.sin(radians(45))
print degrees(math.acos(0.5))
```

*Optional: try the same things with a scientific calculator or use the built-in Windows Calculator program (in Accessories) with the View set to "Scientific" – you can set the calculator to work in degrees or radians. On the calculator, enter 45, make sure the Degrees option is chosen, and press the **sin** button. Write down the result to 5 or so decimal places. Then enter 0.5 and press the **Inv** option and then press the **cos** button. Compare its results to the code here.*

- ❖ Note the use of the **return** keyword to specify what value to *return* to the function when you use it. Thus the radians function returns the result of the expression `angdeg * math.pi / 180` whenever the `radians()` function is evoked, with the value in parentheses being processed as `angdeg` in the function definition. Note the colon and indentation.
- ❖ There are many other uses for function definitions, some you'll see in the later examples. In this code, we provided constants (45 and 0.5), but can you see how you could use this kind of capability in a program where your data might be read from files, etc.
- ❖ Trigonometric functions are commonly used when working with field survey measurements, such as with a compass, and since compasses all read in degrees instead of radians the above conversions are commonly needed.

? *Challenge:* How would you change the program code so that you could have the print statements read:

```
print sin(radians(45))
print degrees(acos(0.5))
```

? *Challenge:* Though this would be unconventional, you could also set up your code to *always* work in degrees. How would you write your code so that you could write the statements as:

```
print sin(45)
print acos(0.5)
```


1.7 Control Flow Structures: If, While, For

An important feature of any scripting or programming language is the ability to execute a set of statements as a group, but under controlled conditions.

Scenario: You would like to create a series of hillshade rasters to represent summer, winter and equinox conditions. The hillshade tool requires inputs of sun angle (altitude) and azimuth. You could look up values in a table, but why not have the computer derive these from what you know? We'll start with a somewhat informed situation – we know what solar declination is, and what values for solar declination are for four significant dates during the year:

Significant date	Solar declination
June solstice (June 21)	23.44
Equinox (Mar. 21, Sept 21)	0
December solstice (Dec. 21)	-23.44

The following is fairly simple code that derives sun angle and azimuth from solar declination (the latitude where the sun's rays are vertical at noon) and latitude. It starts with information to populate two variables, **lat** (latitude for the area of study, negative if south of the equator) and **decl** (solar declination), and from these derives **sunangle** and **azimuth**:

```
lat = 30
decl = 20
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
print sunangle
```

For now, we've hard-coded the inputs of **lat** and **decl**. In this case **sunangle** would be assigned the value 80 since $90 - 30 + 20$ is evaluated as 80. The next line assigns 180 to the variable **azimuth**. The last two lines assigns new values to **sunangle** and **azimuth** *if* **sunangle** ends up with value greater than 90 after the first two lines are processed. Sun azimuth is either from the south (180) or from the north (0) at solar noon.

There are three of these control flow operations:

- if** Only execute the statements *if* a particular condition is met.
- while** Keep executing (loop through) the set multiple times *while* a condition exists.
- for** Loop through the set *for* each value in a range of values.

These follow Python's clear syntax rules, similar to the **def** we just looked at:

- A colon is at the end of the initial statement.
- The block of statements to be processed is indented.

Some important commonalities to these structures:

1. The **if**, **while**, and **for** statements all end in a colon, followed by an *indented* block of statements to be used under the conditions defined by the **if**, **while** or **for**. In the script editor window, you'll note that after you enter the line with a colon, the next line will be indented. In the line following the code you want to run, backspace to get back to an unindented section.

```
if x > 0:
    print 'Within the if block, done only if expression is true...'
    print 'Still within the if block...'
print 'After the if block, done whether or not it was true...'
```

2. If you only need to do one thing, you can follow the colon with a short statement *on the same line*.

```
if x > 0: print 'x is greater than zero'
print 'Next line is not indented.'
```

- Try a simple script that demonstrates an **if** structure. (Note the use of single and double quotes):

```
print '\n\nStart of script...'
x = 5
if x > 0:
    print 'In the "if" block, since x > 0 ...'
    print 'Still in the indented "if" block ...'
print "Not indented - we're at the next step in the script."
```

- Then modify the script by changing the assignment statement to `x = 0`, thus creating a false expression in the if statement.

if (continued)

We just looked at using **if** to demonstrate running blocks of indented code using a condition.

In the following, we'll also explore another handy module, `os.path`:

- Before you start, create a "testfolder" folder in `d:/`, then create a text file "test.txt" in that folder. You might want to make sure your folder is not hiding file extensions (tools/folder options/view tab) so you don't create "test.txt.txt".
- Try the following code, in a script you name "testpath.py". Make sure to indent the print statement.

```
import os.path
if os.path.exists("d:/testfolder"):
    print "test folder exists"
```

? Did the test folder exist? If not, did you remember to create it first?

- Alter the code to read as follows. Note the use of backslashes in the path:

```
import os.path
if os.path.exists("d:\\testfolder"):
    print "test folder exists"
else:
    print "test folder doesn't exist"
```

? What happened?

- Edit the program to work, and meet the first condition that the test folder exists. You could just change the backslashes back to forward slashes, but how would you do it with backslashes?
- You can also have other conditions to test if the first condition isn't met, using the **elif** statement. Note the use of indentation, and we've gone with forward slashes (Windows doesn't care whether you use forward or back slashes.)

```
import os.path
if os.path.exists("d:/testfolder/test.txt"):
    print "Test folder exists."
    print "Text file exists."
elif os.path.exists("d:/testfolder"):
    print "Test folder exists"
    print "... but text file doesn't."
else:
    print "Neither exist."
```

Optional example to explore

The following code does something pretty useful for GIS, but doesn't actually use any geoprocessing code. USGS 7.5' DEMs (digital elevation models) are text files in UTM coordinates with northings and eastings in meters, but elevation is either in feet or meters. This is a problem when deriving information where both horizontal and vertical distances are used, like with slope which can be derived as the vertical/horizontal ratio. If you don't tell it to use a zfactor setting of 0.3048, it will give you incorrect results. But unfortunately, you may not know whether the DEM text file's vertical units are in feet or meters. Well, this information happens to be stored at character #539 in the file – it's a "1" if feet, a "2" if meters – so you could read through the file to find it. This is a pain.

➤ Try this script **GetDEMunits.py** : *(optional)*

```
import fileinput
infile = r"d:\prog\pendata\woodside.dem"
firstline = fileinput.input(infile)[0]
unitchar = firstline[539]
unit = "(unknown: not a 7.5' DEM?)"
if unitchar=="1": unit="feet"
if unitchar=="2": unit="meters"
print "\nElevation in " + unit
fileinput.close()
```

while

- Try the following script, which illustrates a type of loop, a **while** loop:

```
x = 1
while x < 10:
    print x
    x = x + 1
```

- ❖ The last bit of code employs a variable as a "counter" to keep track of how many times we've gone through the loop. Looking at the last statement we can clearly see that it represents an assignment, not a statement of equality – x can never be equal to x + 1, right? But we can assign a new value of x to be one greater than its previous value. This is a very basic but extremely important concept in programming; if you don't feel comfortable with this, ask for help before you continue.
- To illustrate this point, and introduce a related concept, we'll use a real "is equal to" operator, the double equal sign == which means "is equal to". Try the following code:

First try this: x = 5 z = x == 4 print z	Then try modifying it to read: x = 5 z = x == 5 print z
---	--

The equality operator == (2 equal signs) is just one of several logical operators. Others are

<	less than	>	greater than
>=	greater than or equal to	<=	less than or equal to
<>	not equal to		

(there isn't a key on your keyboard to type in something like ≠, ≤, or ≥).

The result of using a logical operator with two values is a Boolean value of true (1) or false (0). In Python 2.4, results display as 'true' or 'false'.

We just saw one use of a Boolean expression:

```
x = 1
while x < 10:
    print x
    x = x + 1
```

The expression x < 10 can be evaluated as true or false, so this allows us to process a set of code after evaluating a condition. As we'll see, there are many situations where we will want to use conditional code – a type of low-level decision that illustrates why we use computers.

One advantage of the **while** loop is it lets us skip the whole section if the condition isn't met to begin with. It's even tempting to use it instead of an **if** statement, but this is an easy way to get into an endless loop: the while loop will keep repeating until the condition is false. When looping through datasets, a common task in GIS work, the **while** structure is often useful. We'll see some examples when we get to using the geoprocessor.

for

- Try the following code, which illustrates a **for** loop:

```
for x in [1, 2, 3, 4]:  
    msg = "Hello World"  
    print str(x) + " " + msg
```
- Replace the list "[1, 2, 3, 4]" with "range(4)". This will also run it four times, but how does it differ?
- Note the use of lists and the len function in the following. What values do you get for j?

```
cov = ["geology", "landuse", "publands"]  
fld = ["TYPE-ID", "LU-CODE", "PUBCODE"]  
for j in range(len(cov)):  
    print j, cov[j], fld[j]
```

- The following script builds and then prints a Python list of shapefiles (each ends in '.shp'):

```
import os  
ws = "d:/prog/HMBarea"  
# Create list of files in the workspace folder  
ilist=os.listdir(ws)  
# Build a list of shapefiles from the file list  
fcs = [] # Start with an empty list  
for i in ilist:  
    if i.endswith(".shp"):  
        fcs.append(i)  
#Loop through the list and print to window  
for fc in fcs:  
    print fc
```

- *Optional:* Type in the following script, saving it as **twotail.py**. It loops a large number of times. If you remember your statistics, you should be able to explain the result. The variable **mu** is the arithmetic mean, **s** is standard deviation – these are used by the random.gauss method. Try changing the n value.

```
import random  
mu = 50  
s = 10  
z5 = mu - s * 1.96  
z95 = mu + s * 1.96  
count = 0  
n = 10000  
for i in range(n):  
    x = random.gauss(mu, s)  
    if x < z5 or x > z95: count = count + 1  
print float(count) / n
```

Note that when we want to concatenate a number and a string (like `x + msg`), we must convert the number to a string with `str(x)`. While **print x** works, the concatenation `x + msg` fails since it can't concatenate a number and a string. But `str(x) + msg` works, as does `str(x) + " " + msg` shown here.

1.8 Simple Input and Output

Let's take the snippet of the sunangle code and make it useful. We'll start with our example where we "hard code" data directly into it by assigning variables. We'll also have it print the results to the interactive window. Try the following (**sunangle.py**) code:

```
lat = 40
decl = 23.44
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
print sunangle
print azimuth
```

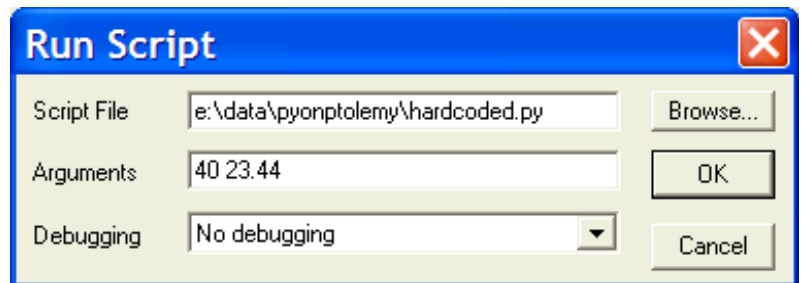
? What results do you get? Sunangle: _____ Azimuth: _____

- The print statements work, but they're kind of crude -- you just see numbers appear. To make it a bit more readable, change the last two lines to read as follows:

```
print "Noon sun angle = " + str(sunangle)
print "Azimuth = " + str(azimuth)
```

- Now let's get away from the hard-coded input. We probably don't want to have to alter our program every time we want to run it. There are a variety of ways of providing input. For now, we'll use a **sys** method. Replace the first two lines of code with the following and run it, providing inputs "40 23.44" as *arguments* in the Run Script dialog (the two inputs are separated by a space):

```
import sys
lat = float(sys.argv[1])
decl = float(sys.argv[2])
```



What we did was:

- (1) import a module – **sys** – that provides a variety of python system methods;
- (2) use the arguments (**argv**) method to derive input from the "Arguments" line of the Run Script dialog; and
- (3) assign a floating point conversion of these text string inputs to the variables **lat** and **decl**.

Try it again with different values. Valid values of latitude are -90 to 90, while valid solar declination values are -23.44 to 23.44. Try -70 for latitude and 23.44 for declination to see what the sun angle would be at 70°S on the June solstice. If you don't understand what you get, sign up for Geog 101.

Challenge: Modify **GetDEMunits.py** to provide the DEM name as an argument. The biggest challenge is getting the path to the DEM file. You could type it, but it would be better to paste it from a path bar. Unless you know other Python tricks, you'll probably want to use two separate pastes, one the folder, the other the filename.

2. Introduction to Geoprocessing scripts in PythonWin

Let's now explore some ArcGIS Geoprocessing tools that we can use as part of a script. Using PythonWin, enter the following code and save it in your **mypy** folder as **listFC.py**. It's a very simple script that lists the feature classes in a workspace. After an initial comment line, it starts with two lines of "boilerplate" code needed for all ArcGIS geoprocessing scripts. One of these imports **arcgisscripting**, that provides access to the ArcGIS geoprocessor. To clarify things, we'll also use some comment lines that python will ignore, by starting the line with #. Note how PythonWin uses color to distinguish different parts of your code.

```
# regular gp boilerplate
import arcgisscripting
gp = arcgisscripting.create(9.3)
# List the feature classes in a specified workspace
# -- Make sure the workspace path is valid -- edit if necessary
gp.workspace = "d:/prog/pendata"
fcList = gp.ListFeatureClasses()
for fc in fcList:
    print fc
```

- ❖ As with the simple scripts we just looked at, you would run this from PythonWin. As before, the print statement prints to the "interactive window." We've imported Windows COM (Component Object Model), letting us use COM-based programs like the ArcGIS geoprocessor.
- Before running this, *get out of ArcMap or ArcCatalog*. While there are situations under which you can leave one of these programs running, such subtleties are difficult to keep track of when you're learning, so for now just remember to close both programs before you run any Python script from PythonWin. In fact, the above script probably won't create any problems because you're not creating or deleting any datasets, but you should get in the habit anyway.

Some of the mysteries are geoprocessor-specific:

- gp** a variable that holds the ArcGIS geoprocessor. It's the link between python and a whole stack of methods and tools in ArcGIS. Some of these are geoprocessor-specific things, others are the 400+ GIS tools (like buffer, clip, etc.) that you normally use in the ArcToolbox. The gp variable is established in the second line of boilerplate code, then we use this variable to access geoprocessing tools.
- gp.workspace = "..."** sets the **workspace** environment variable to the path you specify. *Important:* I've hard-coded this path into the program. Use a valid path to a workspace you know about – edit this line as appropriate.
- ListFeatureClasses()** – sent to the geoprocessor (since it is prefaced with "gp."), returns a list of feature classes (shape files in this case, but could also include parts of coverages and geodatabases).
- fcList** a variable we've defined that holds a kind of list that we can loop through, in this case feature classes, so we've called it fc....
- for fc in fcList:** Loops through all of the members of fcList, assigning each in turn to **fc**.

- Run the **listFC.py** script within PythonWin. Make sure the workspace path is correct – we haven't included any error handling in this script.
- ? Did it give you what you expected? What do you see?

- After running it once and seeing the single feature class _____ that it finds, try changing the program to look in hmbarea ("d:/prog/hmbarea").
 - ? How many feature classes does it find, and what kinds of feature classes are they?

- ? What feature classes do you see when you set the workspace to a coverage "d:/prog/pendata/geology"?

- ❖ Understanding the variety of data types that ArcGIS uses is important to understanding what we are seeing with geoprocessing tools. For instance, feature classes may either be (1) individual shapefiles; or (2) feature classes within a geodatabase (where we might specify the geodatabase as the workspace); or (3) polygon, arc or point feature classes within a coverage. We might be more interested in looping through all of the coverages in a workspace, in which case we would want to know that coverages are *feature datasets*. ListFeatureClasses won't work; we'll need another enumeration, created with ListDatasets...

2.1 Lists

These geoprocessor methods starting with "List..." and (as of ArcGIS 9.3) produce *Python lists*, if you include any number ≥ 9.3 as the parameter to `arcgisscripting.create`. They are used to organize lists of objects and settings that we would like to loop through. Others are `ListFields`, `ListRasters`, `ListTables`, `ListWorkspaces`, `ListDatasets`, `ListEnvironments`, `ListTools`, and `ListToolboxes`. To learn more about how to use each, look in ArcGIS Desktop Help section `Geoprocessing>Automating your work with scripts>Scripting Object: Properties and Methods>Properties and Methods`. Starting with the same boilerplate, try this script that uses `ListRasters` (**listRas.py**):

```
import arcgisscripting
gp = arcgisscripting.create(9.3)

gp.workspace = "d:/prog/HMBarea"
rsList = gp.ListRasters("t*")
for rs in rsList:
    print rs
```

? Can you see the difference and the similarity to the other?
One difference is we use an optional wild card to only find rasters starting with "t" (try another letter). What else?

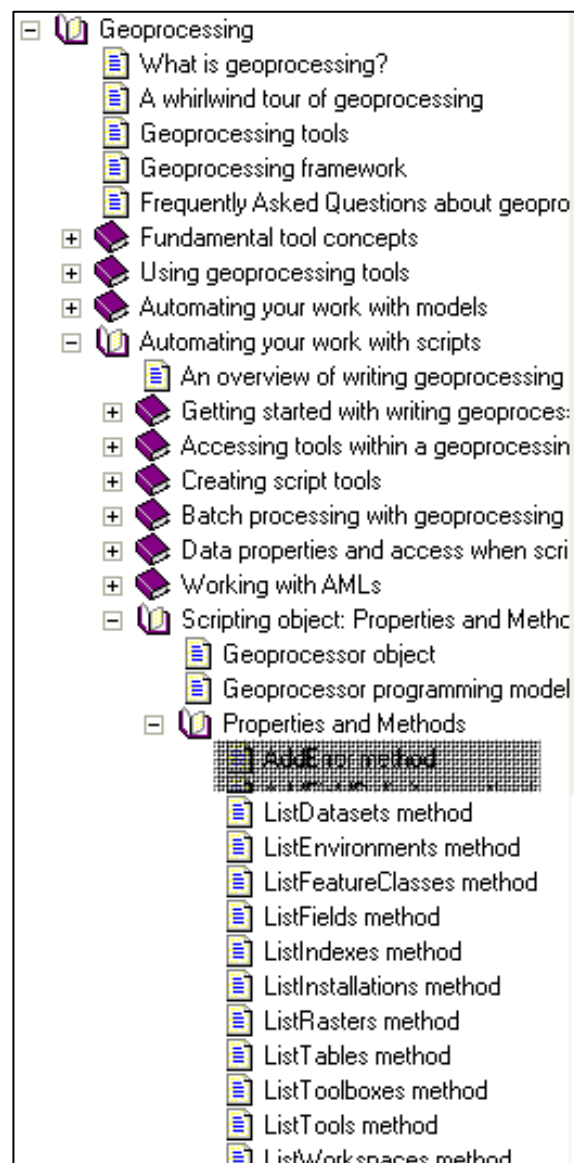
? What does the following code do?
Don't forget the boilerplate

```
gp.Workspace = "d:/prog"
for ws in gp.listWorkspaces(): print ws
```

? How about the following code?

```
envs = gp.ListEnvironments()
for env in envs:
    print gp.Usage(env)
```

? Using the previous, `listWorkspaces` code as an example, how could you rewrite this `listEnvironments` snippet into one line of code?



Workspace – many uses

Note that the first line of code, setting the workspace, must be included for **listWorkspaces** to work. The same applied to **listRasters** and **listFeatureClasses**. Workspaces can be folders, ArcInfo workspace folders, geodatabases (even though a geodatabase is actually a file), coverages or datasets to access feature classes within.

3. Help for Geoprocessing

The listing methods we just looked at are not tools, but are methods of the geoprocessor scripting object. Help for these methods can be found in two places:

(1) The help system (Contents tab). Go to

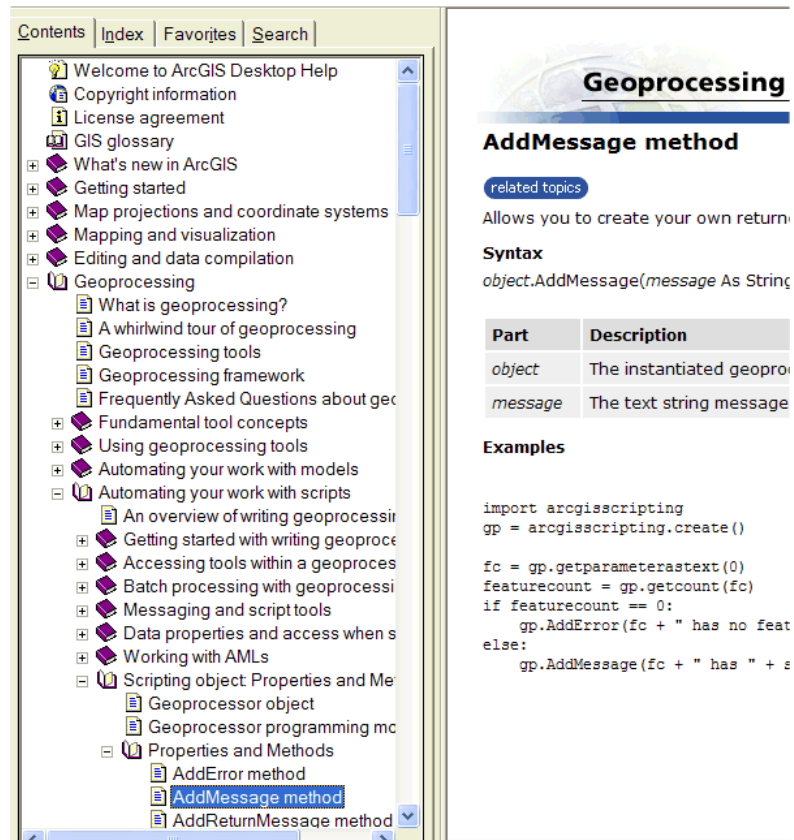
Geoprocessing/

**Automating your work with scripts/
Scripting object:**

Properties and Methods.

There you'll see the

Geoprocessor Object, which will gain you access to documentation on all of the enumerations, settings, and other operations you can use in the Geoprocessor – very handy. For example, you may want to list only certain types of features – see the `listfeatureclasses` method section to make sense of:



```
fcList = gp.ListFeatureClasses("w*", "polygon")
```

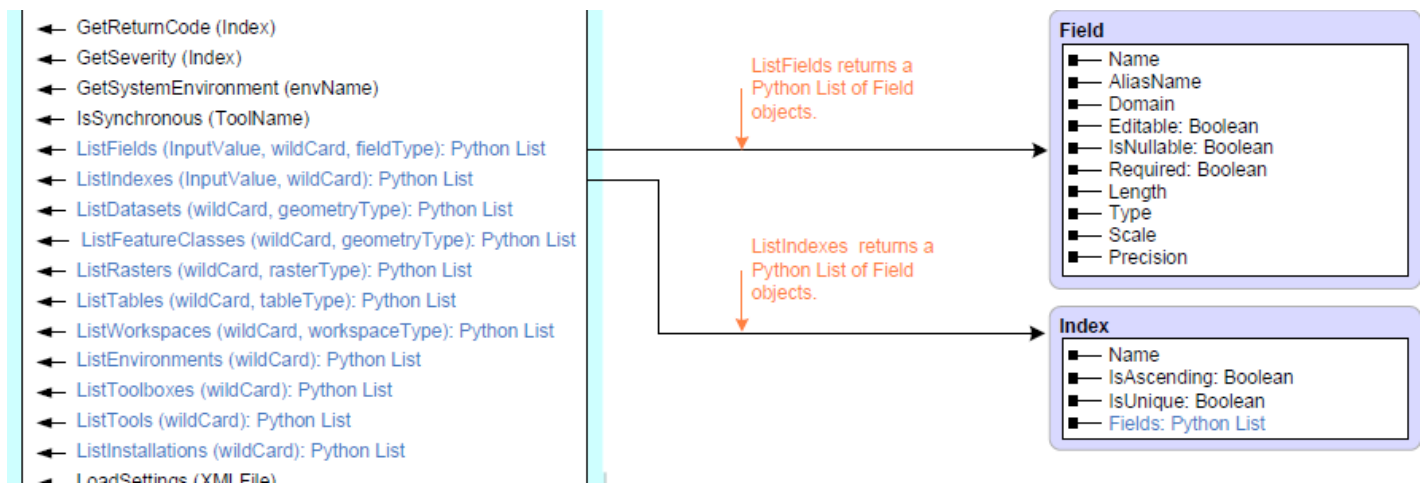
? With these two parameters, what does the above method do,?

(2) The **Geoprocessor Programming Model**, which is printed onto a poster in the classroom, but also can be accessed onscreen as a pdf. This is useful for seeing how it all ties together.

- In the Scripting Object Properties and Methods (in Geoprocessing/Automating your work with scripts), find the Geoprocessor Programming Model, and click the hyperlink to view the PDF, then save `geoprocessor.pdf` to your desktop or your workspace, somewhere that is a bit easier to get to.
- Explore the PDF. It includes methods (shown with a left arrow) and properties (shown with a barbell if both read-write, or a partial barbell for read-only).

3.1 Example: Using the Geoprocessor Programming Model for Lists

- The lists and their properties and methods are highlighted in purple on the diagram.



- Note the variety of properties for Field in ListFields, and see if you can write a script that lists all the fields in the attribute table for the landuse grid in hmbarea:

- Start with the boilerplate: _____

- Set the workspace to the location of your hmbarea folder: _____
- Use gp.listfields to assign the fields of the raster **landuse** to an enumeration variable **flds**.

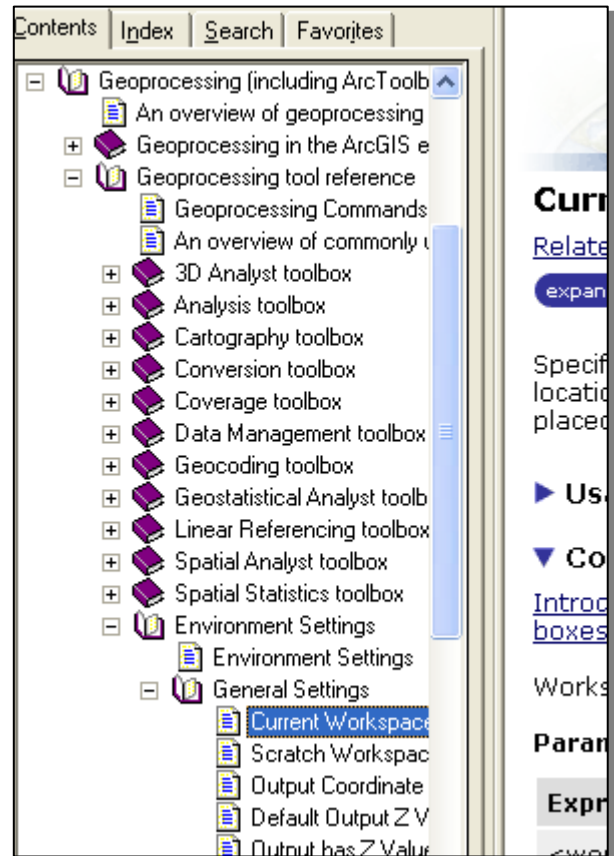
- Referring to the ListFields enumeration, note that it creates an object, not a string.
- Use a **for** loop to assign to **fld** each of the members **in flds**, print the name of the field accessed by requesting the name property with **fld.Name**.

- Then modify this script to also show the **fld.Type** by concatenating (fld.Name + fld.Type)
- Then modify the workspace path to use the landuse coverage in pendata (since the dataset name is the same, you just need to change the workspace. Note that the fields for the first feature class (arc) of the coverage is displayed.

3.2 Help for Tools and Environment Settings.

If you look in the bottom of the main entry arcgisscripting section of the pdf, so at the lower left of the pdf, you'll see two items listed under "Dynamic Methods and Properties": Environment properties and Tool methods. There are hundreds of tools -- things you've used ArcGIS for like clip, buffer, slope -- and quite a few environment settings, so they didn't include all of these on this diagram.

- Use the help system to learn about these. For tools, the quickest way is to right-click the tool and click help. For environment settings, you can also find help by going to the environment settings in ArcCatalog or ArcMap via Tools>Options>Geoprocessing tab>Environments button, and displaying the help setting; or you can get to these directly in the help system in Environment Settings under Geoprocessing>Geoprocessing tool reference.



3.3 Sample Scripts

It's very useful to have examples to use, ideas to steal, etc., when learning how to use a programming language. Two good sources for Python geoprocessing scripts are:

1. ArcScripts at ESRI: <http://arcscripsts.esri.com> : put Python in the language option, and you'll find lots of examples, more all the time.
2. Script examples we provide you. In the py folder you copied from the server, which should now be on D:\prog\py, there are some python scripts. Some of these you'll be asked to open later on in the exercise. There are also the answer scripts from this exercise you can find on the course web page and other pages.

4. Using geoprocessing tools -- Toolboxes and Aliases

As we've seen, Geoprocessing tools used in a script are the same tools as are used in ArcToolbox, but we access them by providing the tool name. But remember that there may be more than one tool by a given name -- for instance, there is also a **Clip** tool in the **Extract** toolset of the **Analysis** toolset of **Coverage** Tools; and another one in the **Raster** toolset of **Data Management Tools**. Each is distinct, working with different types of data, but how do we distinguish them in a script? In ArcToolbox, we select which clip we wish to use, but in a script we have to distinguish them in some way. To do this we use aliases, which become part of the tool name.

- Go to the help for the Buffer and Clip tools of the Analysis Toolbox, and then look at their scripting syntax. Note that the syntax gives each an *alias* that identifies which toolbox it comes from, in this case **analysis**.

Clip_analysis (in_features, clip_features, out_feature_class, cluster_tolerance)

Buffer_analysis (in_features, out_feature_class, buffer_distance.....)

- To use this tool in a script, we would then write it as shown in this script segment (**clipstr200.py**). *It also needs the boilerplate at the top!* Note that this script first converts a stream coverage to a shapefile, then buffers the streams to 200 m, and uses that as the clip feature class. It also uses an environment setting to allow the system to overwrite existing files:

```
gp.workspace = "d:/prog/hmbarea"    (your path may differ)
gp.overwriteoutput = 1
# Convert streams coverage arcs to a shapefile
gp.copyfeatures_management("streams/arc", "streams.shp")
# Create 200 m around streams, using the converted streams
gp.buffer_analysis("streams.shp", "stbuff200.shp", 200)
# Clip
gp.Clip_analysis("geol.shp", "stbuff200.shp", "geolstr200.shp")
```

- ❖ Each toolbox has its own alias, which you can look up by right-clicking the toolboxes and going to properties:


Alias	Toolbox		"conversion"	Conversion
"3d"	3D Analyst		"geocoding"	Geocoding
"analysis"	Analysis		"ga"	Geostatistical Analyst
"arc"	Coverage		"lr"	Linear Referencing
"management"	Data Management		"sa"	Spatial Analyst
"cartography"	Cartography		"stats"	Spatial Statistics

- ? *Challenge:* Assuming we have a geology coverage (there is one in pendata), how would we change this program to use coverage tools (only works with full ArcInfo level)?

- If you're going to run several tools from a toolbox, you can save a bit of typing by specifying the toolbox as an environment setting. The following code is equivalent to what you did above. Modify your **clipstr200.py** code to use this.

```
gp.Toolbox = "Analysis"
gp.Buffer("streams.shp", "stbuff200.shp", 200)
gp.Clip("geol.shp", "stbuff200.shp", "geolstr200.shp")
```

5. Scripts in ModelBuilder

First, it's important to remember that quite a few tools in ArcToolbox are actually scripts. Scripts have an  icon. For instance, the Spatial Statistics tools are almost all Python scripts (some are models that call scripts). You can actually copy these scripts and edit them to make them do other things.

To use a script in ModelBuilder or even to use it as a tool in ArcToolbox, we need to consider how to give it inputs and let it set outputs. As an example, let's look at our first sun angle code – it needed a bit more to make it a useful script. If we wrap the sun angle code we already looked at with a few more statements, two of which are boilerplate, the other four providing input and output, we can use this as a step in Model Builder. Note that the indentation ends where the conditionally run section ends. All code after that is always run.

- Enter the following code as **getnoonsun.py** (or edit the **sunangle.py** script you entered earlier, and save it as **getnoonsun.py**) but don't try to run it from PythonWin – the `getparameterastext` and `setparameterastext` can only be used in script *tools* – i.e. in ArcToolbox or ModelBuilder.

```
# geoprocessing boilerplate
import arcgisscripting
gp = arcgisscripting.create(9.3)

# Get lat and decl as input variables
lat = float(gp.getparameterastext(0))
decl = float(gp.getparameterastext(1))

# Noon sun angle & azimuth code
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0

# Send sunangle and azimuth to be used as output in a model
gp.setparameterastext(2, str(sunangle))
gp.setparameterastext(3, str(azimuth))
```

Some new things in this code:

- getparameterastext(?)** The `getparameterastext(?)` part is a method sent by Python to the geoprocessor, which we've called `gp`. And what it does is allows tools to provide input. You can see this happening every time you use a tool and you get a dialog box. The input boxes you've used lots of times by now are all parameters, and this method lets you then use them in your program. The index 0 and 1 refer to the first and second parameter – it starts at zero.
- setparameterastext(?, ?)** The opposite of `getparameterastext`, it sends the second item (like the value of `str(sunangle)`) to the specified output parameter. Since the first two parameters were 0 and 1, these two parameters follow as 2 and 3. We'll need to distinguish these as output derived values. Note that both inputs and outputs are text (string) variables, though we'll actually define them to be used as floating point.
- float** a python function that turns the number it has received as text into a floating point number.
- str** converts a number into a text string.

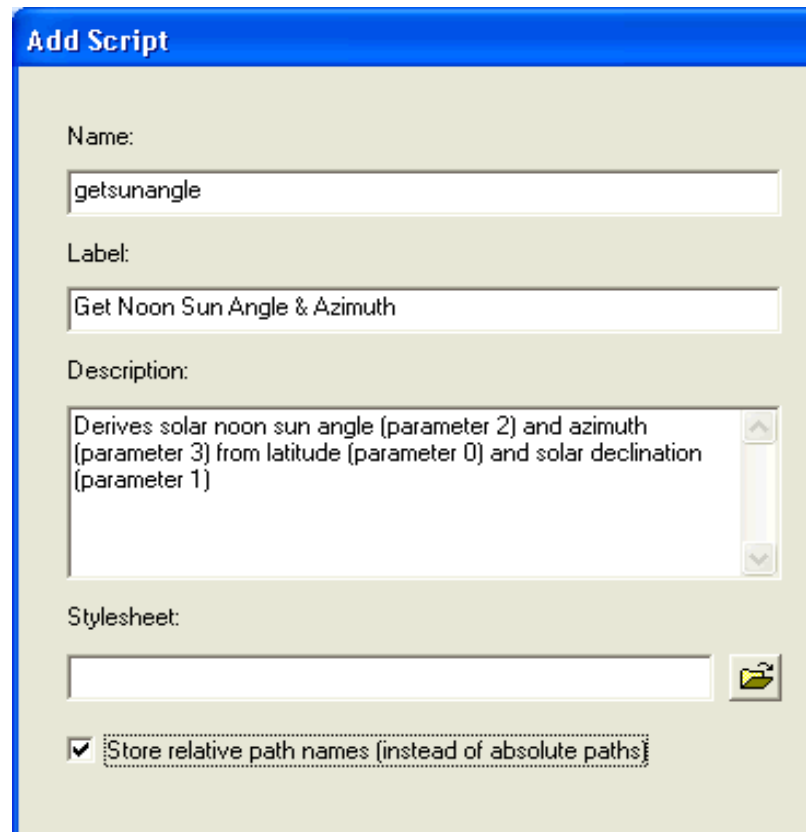
Making a script into a tool

The next step is to make a tool out of our script, so we can use it in ArcToolbox or ModelBuilder (or the command line). As mentioned above, this script can only be used as a tool – it includes input/output methods that PythonWin won't handle, but that are nearly always necessary for tools. (So debugging a script like this requires some tricks like temporarily using alternative methods of input and output, but we won't cover this here.)

- In ArcCatalog, go to the folder where your python scripts are stored, preferably on the same drive as your data, then right-click the contents area and create a new toolbox called *mytools*. You can also use a toolbox you've already created.
- In ArcToolbox, from ArcMap, in your toolbox, right-click and *add* a script. Give it:
 - Name (no spaces): **getsunangle**
 - Label (spaces are ok): **Get Noon Sun Angle and Azimuth** – this will be displayed when you use the tool in ModelBuilder.
 - Description: displayed when you open the tool in ArcToolbox – a bit of help in using the tool can go here, or just a longer description. Identifying inputs and outputs are handy here. For the above script, use the following:

"Derives solar noon sun angle (parameter 2) and azimuth (parameter 3) from latitude (parameter 0) and solar declination (parameter 1)." Or use more verbose English, if you want.

- Ignore the style sheet, and check the box that says to use relative path names.
- Press the Next button to go to the dialog to specify the script file name.



Remember: A script tool *references* a script

It's important to remember that when you make a script into a tool, the script itself is *not saved with the tool, but only references it*. A script tool is stored as part of a toolbox, which ends up being a file like *mytools.tbx*, which can store script tools and models you've created. You can also edit your script by right-clicking the script tool, and it will bring up PythonWin or whichever IDE you have specified as the editor; this makes it look like the script is in the toolbox.

- Why is this important? You need to remember to copy both the script tool and the script to put it somewhere else.

Next is a parameter page where you specify input and output parameters. Under Display name, first type latitude, then choose Double as the data type, then while this row is selected (shown with an @ character in the left column) set the Type to Required, and the Direction to Input. Then do the other three parameters, according to the following table.

Display name	Data type	Type	Direction	Default
Latitude	Double	Required	Input	0
Declination	Double	Required	Input	0
Sun Angle	Double	Derived	Output	35
Azimuth	Double	Derived	Output	300

Add Script

Display Name	Data Type
Latitude	Double
Declination	Double
Sun Angle	Double
@ Azimuth	Double
	Feature Layer

Click any parameter above to see its properties below.

Parameter Properties

Property	Value
Type	Derived
Direction	Output
MultiValue	No
Default	300
Environment	
Domain	
Dependency	

To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

< Back Finish Cancel

- Now run the tool from ArcToolbox. You should be prompted for a latitude and declination – valid values need to be floating point (Double means "double-precision floating point") though you don't need to use a decimal place if you want to use 40 as your latitude. Try the following values: latitude 40, declination 23.44.
- ? Does the tool run correctly? If so you should have gotten a Completed message in a dialog – you may see a black screen display temporarily, but don't worry about it. So what does it do?

Remember that the results were to set two output parameters to numbers. Where do the numbers go? Good question, but this just illustrates that you can create a tool that doesn't really work as an ArcToolbox tool. One that would work would probably create an output dataset like a raster, feature class or some such.

- But it does work as a ModelBuilder tool – we'll use the outputs to create a hillshade.
- Create a new model in your toolbox, and drag your script tool into it.
- ? What do you get? Specifically, what happens with your output parameters?

- Double-click the tool in Model Builder to initialize it. Enter values for latitude and declination as before.
- Open the sun angle and azimuth objects to see what values they received. Note that they are the same as the default values. What this is telling us is the tool hasn't actually run yet.
- To run it, now right-click the tool in modelbuilder and select "Run". After it's done check the values in sun angle and azimuth. Valid values of latitude are -90 to 90, while valid solar declination values are -23.44 to 23.44.
- ? Try latitude -70, declination 23.5 – and what do you get, and why? But since the result won't be valid for hillshade, change the latitude or declination to give a valid (positive) sun angle before you proceed to the next step.

- Making sure first that you have valid outputs, with shadowed symbols in model builder and positive values in the right ranges for sun angle and azimuth, use the outputs as inputs to hillshade. First add a hillshade tool (you'll need to first add the Spatial Analyst extension using the Tools/Extension menu in ArcMap or ArcCatalog – whichever one you're running the model from), then double-click it to set up inputs such as an elevation raster – use one from the marbles or hmb – and use the pull-down on **azimuth** and **altitude** to select the **azimuth** and **sun angle** outputs you have. Check the Add to Display button for the output and see what you get in ArcMap.

Challenge: What if you wanted to derive solar declination first from a calendar date? Load **getnoonsunfromdate.py**. There are five parameters now, so set them up correctly. (Long is a type of integer).

Display name	Data type	Type	Direction	Default
Month	Long	Required	Input	1
Date	Long	Required	Input	1
Latitude	Double	Required	Input	0
Sun Angle	Double	Derived	Output	35
Azimuth	Double	Derived	Output	300

Or even further: What about deriving sun angle and azimuth for any given time during the day, not just noon. I have an algorithm and code I'll demonstrate in class.

Challenge: Running this script from PythonWin

We set up our script to run as a tool, but would it also run directly from PythonWin? There are a few issues. For one thing, we'll want to use the hillshade tool as a step in the script, and we'll need to provide its inputs. Another issue is the use of `GetParameterAsText` – this only works when used as a tool.

What about running the hillshade tool within a script? It will need an elevation raster as an input. See **noonhillshadefromdate.py**. These might be the script tool parameter settings:

Display name	Data type	Type	Direction	Default
Month	Long	Required	Input	
Date	Long	Required	Input	
Latitude	Double	Required	Input	
Workspace	Workspace	Required	Input	
Elevation	Raster Dataset	Required	Input	
Hillshade	String	Required	Input	

Note that the first inputs doesn't use `getparameterastext(0)`, but uses `sys.argv[1]`. The `getparameterastext` method can only be used from a tool, in ArcToolbox or ModelBuilder. The `sys.argv` method works just as well, but starts at 1 instead 0, and requires importing the `sys` module. Since we're writing out a hillshade to disk and not making it a specific output, as we would need in ModelBuilder, there is no `SetParameterastext`, so we don't need to worry about this method, also not suitable for PythonWin.

6. Debugging Python Geoprocessing Scripts

Now that we're seriously getting into running geoprocessing tools from Python, we need to consider how to debug our programs. While there's always the need to debug programs, communication needs between Python and the geoprocessing system adds to this. If a geoprocessing tool fails, we need to get a message from PythonWin more informative than "Unspecified Error."

6.1. Debugging options & Messages

One of the advantages of Python over AML is much better debugging methods. There are many options to debugging our programs; we won't explore them all.

Print statements

The tried and true method used by programmers since the beginning of time: simply printing the current value of variables, or printing information about the progress of the script .

For example, in the script you were just editing, we can add a print statement to tell the user where the program has gotten to:

```
gp.Toolbox = "Analysis"
gp.Buffer("streams.shp", "stbuff200.shp", 200)
print "Finished Buffer.  Now clipping..."
gp.Clip("geol.shp", "stbuff200.shp", "geolstr200.shp")
```

- For one of the programs you've already run, add a print statement to tell the user what the program is doing.

What about if you are running your script as a script tool? Print statements don't display from a script tool; they don't create an error, but they don't display anything. For this, use `gp.AddMessage(...)` to print what you put in the parentheses, such as:

```
gp.addmessage("Finished Buffer.  Now clipping...")
```

What if you want to print messages whether your program is running from the IDE or as a script tool? Since `gp.addmessage()` also has no effect in PythonWin or other IDEs, you can leave them in the program and just provide both `print` and `gp.addmessage` statements. What I like to do is define a 'sendmsg' function to send a message whenever I want to with a simple 'sendmsg' command:

```
def sendmsg(msg):
    print msg
    gp.addmessage(msg)
.
.
.
sendmsg("Finished Buffer.  Now clipping...")
```

Getting Tool Messages & try:... except:.

While the above print methods are useful, one major problem in debugging a script running geoprocessing tools is we don't get much information when they don't run. If there's a problem with the Buffer in the script – maybe the input "streams_arc.shp" doesn't exist, when we're running the script from PythonWin, we may only see "unspecified error" displayed in the interactive window. This is where **GetMessages** comes in handy.

We'll explore two debugging tricks here – one the Getmessages geoprocessor method, the other a Python error-handling routine try...except. Let's first try the Python try...except method with a simple script that needs error handling.

- Try it two ways, one with a non-zero value assigned to x, the other as shown with zero assigned:

```
x = 0.  
y = 15.  
try:  
    print y/x  
except:  
    print "Error encountered. Divide by zero?"
```

Now let's add GetMessages to display informative errors.

- Edit clipStr200.py to purposely include an error. Take out the print statement, and change the first parameter in the Buffer tool from "streams.shp" to "stream.shp", as follows, then run it to see what is displayed.

```
gp.Toolbox = "Analysis"  
gp.Buffer("stream.shp", "stbuff200.shp", 200)  
gp.Clip("geol.shp", "stbuff200.shp", "geolstr200.shp")
```

? What can you use in the Interactive Window to tell you where the problem lies?

- Add a try... except structure to your code. You can actually add the try: statement to the top of the program if all of your code is indented, followed at the end with the except: clause. This is probably the best practice. The following illustrates the use in just a short section of code:

```
try:  
    gp.Toolbox = "Analysis"  
    gp.Buffer("stream.shp", "stbuff200.shp", 200)  
    gp.Clip("geol.shp", "stbuff200.shp", "geolstr200.shp")  
except:  
    print gp.GetMessages()
```



- Try the above and see what message is displayed, then fix the error.

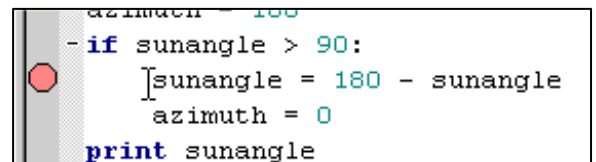
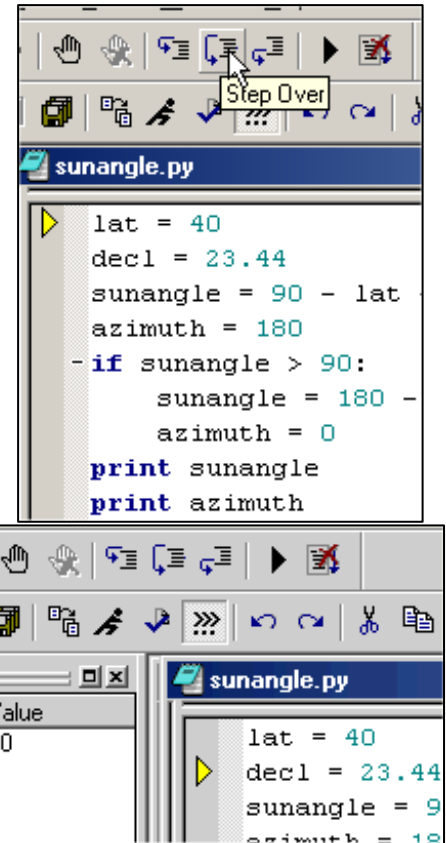
Types of GetMessages

GetMessages () : all messages
GetMessages (0) : messages only
GetMessages (1) : warnings only
GetMessages (2) : errors only

6.2 PythonWin Debugging Tools

PythonWin provides tools for (1) stepping through code, (2) inserting break points, (3) observing variables, and other things. To learn more about these, you might try a tutorial you can find in the PythonWin (not Python) help system. We'll just try the step-through method.

- With the simple `sunangle.py` program, start to run the program, but when the Run Script dialog appears, pull down the Debugging options, which includes:
 - No debugging
 - Step-through in the debugger
 - Run in the debugger
 - Post-mortem of unhandled exceptions
- Pick "Step-through in the debugger." Note that a set of debugger tools appear, and a yellow triangle points to the first line of code. Press the Step Over button to progress to the next line of code.
- Click the Watch button (eyeglasses icon). Under the Expression, enter **lat** to display the value of this variable. Then add **sunangle**.
- Progress through your code one step at a time. You may want to experiment with other buttons – try the plain Step button and you may find other routines appearing, for instance when you go to the print statement – the Step Over only uses your script.
- When you're done exploring the program, exit it with the close button.
- Try inserting a break point by clicking on a line of code and pressing  -- use the statement just after the if statement ("sunangle = 180 – sunangle"), and then run the script using the "Run in the debugger" option – not the step-through. The program should run without any special effect.
- Now change the code to make it go to this statement, by changing the first line to `lat = 10`. You should see the watch display the value of `sunangle` that made the program get to this breakpoint. Then just press the Go  button to continue the program.



6.3. Geoprocessing Tool Use Example

... now that you know enough to be dangerous, but maybe can debug your way back...

conversion script

Now that we know how to look up things in the help system, and we have some debugging methods at hand, let's build a script from scratch.

- From PythonWin, enter the code to do the following (makewoodfolder.py):

1. Put the geoprocessor boilerplate in. Feel free to copy this from another script.

2. Create a sendmsg function (copy it from a few pages ago).
3. Set gp.OverwriteOutput to 1 (true).

4. Create a folder called "woodside" in "d:/prog" using a tool from the Data Management toolbox, Workspace toolset.

5. Set the gp.workspace to this woodside folder.

6. From the conversion toolbox, in the To Raster toolset, use the DEMtoRaster tool to convert the DEM "d:/prog/pendata/woodside.dem" into a "woodelev" raster in this workspace. Don't provide any of the optional parameters.

7. Select the entire program, press tab to indent it, then press the left arrow key to unmark the text and position the cursor at the top of the code and press enter to insert a new line at the top. Insert the line **try:** as the first line of code, then go to the end of the code and insert an **except:** statement followed by one line of code (indented)
sendmsg(gp.GetMessages())

Using the help system for tool help:

You will find it handy to have the ArcGIS help system available – you can start it independently from the Windows start menu, in the ArcGIS folder, avoiding data lock conflicts you may encounter if you have ArcCatalog or ArcMap running. Use the Geoprocessing tool reference in the help system.

feature to raster conversion

- Now look up the feature to raster tool in the conversion toolbox, and do the following (makelugrid.py). As before include this in a try...except structure and send messages.
 1. The normal boilerplate, including setting overwriteoutput to 1.
 2. Set the workspace to "d:/prog/pendata"

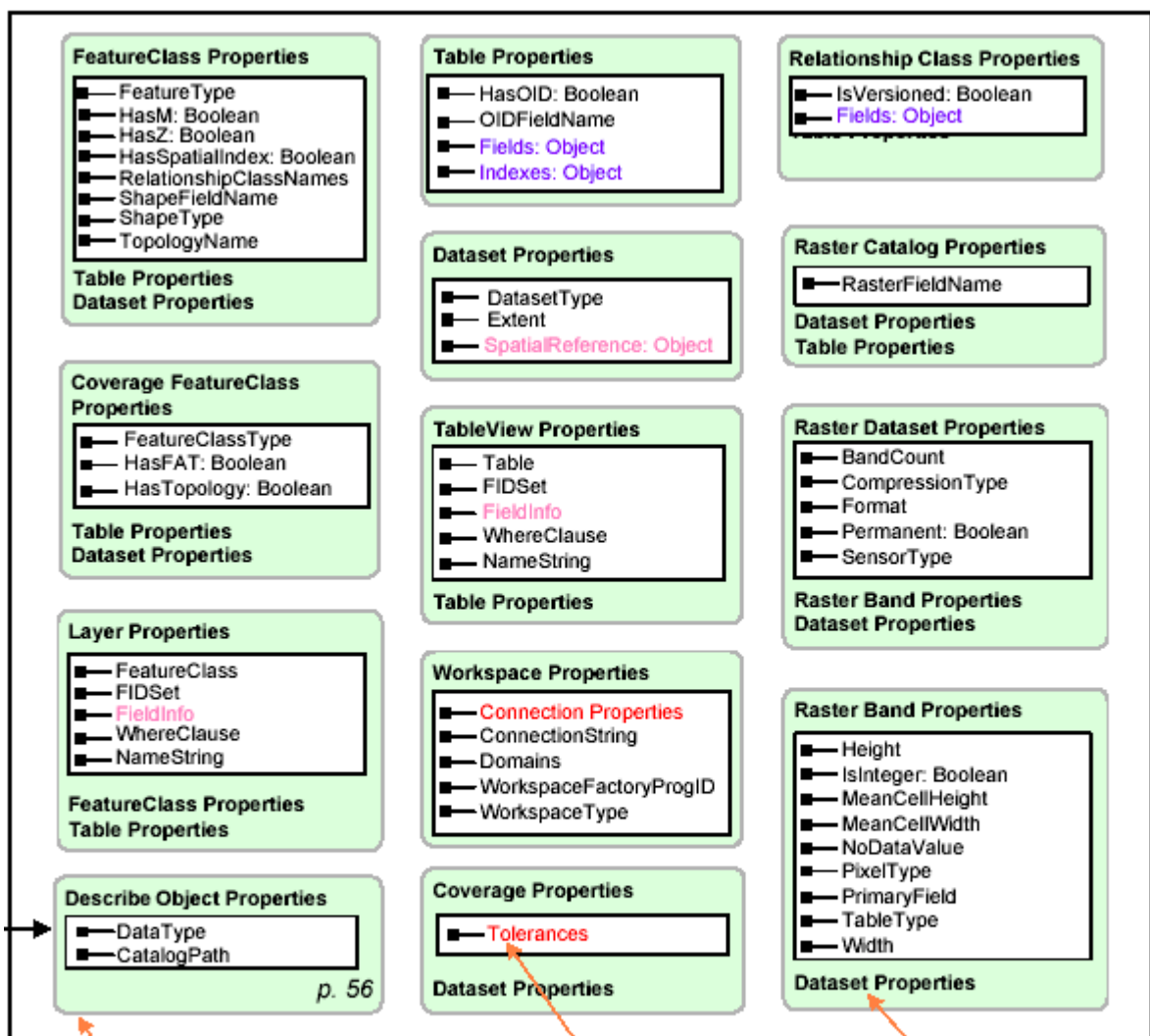
 - 3. Use the feature to raster conversion tool to convert the coverage feature class "landuse/polygon", with the field "LU-CODE", to an output grid "lugrid", with cell size of "30"
-

7. Using Describe= & Exists to access information from data

There are many situations in which we want to use characteristics of GIS datasets to process other data. For instance, in raster operations we may want to use the extent of one dataset to delimit another dataset, much like a clip operation, or to detect the type of topology of a feature class.

7.1 Describe

The gp.Describe method applied to a dataset generates a variety of properties about that dataset. As shown in the green area of the Geoprocessor Programming Model, there are many properties, and they are hierarchical: all datasets have a DataType and CatalogPath. FeatureClass properties include all table properties and dataset properties, and Raster Dataset properties include Raster Band properties and Dataset properties, so the Dataset property extent is available to Feature Classes, Rasters, Coverages.



An Object is created with all of the properties of the data being described. The specific properties listed apply to all types of data.

Red text indicates a property set. See page 61 for more information on property sets.

Each type of data has unique properties. Depending on their relationship, properties of other data types may also be

- Use the Describe system to clip a raster based upon the extent of another dataset. We'll put a clipped landuse map into the woodside folder (clipwoodlanduse.py):

1. The boilerplate (now including try...except).
2. Set the workspace to "d:/prog/woodside"
3. Assign `gp.Describe("woodelev")` to a variable `dscRas`.

4. Since `dscRas` now stores information on the woodelev raster, you can use `dscRas.Extent` to provide a rectangle for use with the clip tool in the raster toolset of the data management toolbox. Use this to clip "d:/prog/pendata/lugrid" and create "luwood" as the output (placed in the working directory).

- Modify your **showFields.py** script to display the name and type of the data set, using the code here, with `dta` already assigned "landuse", just before you loop through the fields. Save the script as **showFieldsDesc.py**:

```
dsc = gp.Describe(dta)
print dta + " " + dsc.DataType + ":"
```

- Write a script that displays the band count in "d:/prog/hmbarea/tmcomp.bil", naming it **countBands.py**. Use a similar assignment to `dsc` of the Describe result. Look in the Raster Dataset properties to find the method.

Challenge: Write a script that loops through the coverage datasets in a workspace and display all that are of type polygon. Use hmbarea or other workspaces. Remember that you'll need to set the workspace to the coverage to list its featureclasses.

7.2 Exists

An even more basic piece of information about a dataset is whether it exists or not. Testing for the existence of a dataset can also help us avoid errors. A very useful technique is to detect the existence of a particular dataset using `gp.Exists`.

`gp.Exists` can be used to detect any type of data. For instance if you wanted to detect whether an input dataset existed, you could modify code a statement to read something like this:

```
if gp.Exists("streams.shp"):
    gp.Buffer("streams.shp", "stbuff200.shp", 200)
```

In the next script, we'll use this to detect the existence of datasets before we create them. This could be used as an alternative to the overwriteoutput setting we've used.

- ❖ **Detecting if a field exists.** You can't use the above method to test whether a field exists, but the following use of the `listFields` enumeration does the trick.

```
if not gp.listFields("streams.shp", "st-code"):
```

We'll use it later in the *Data Management and Cursors* section to first check whether a field exists before we try to create it.

7.3. Using Exists and Describe in a Loop

This will give you a taste of how scripts can be very useful for processing multiple datasets. We'll clip a whole series of rasters and put them into a new workspace. In the process we'll also use a method for checking to see if an output exists before we create a new one, with the very handy Exists tool. We'll also look at using an if ... else structure – note that both are followed by colons and then an indented section. In this example, we're using an ArcInfo workspace, and we'll store grid rasters there, but you could also put these in a geodatabase. Script name: **clipRasters.py**

1. Start with the normal boilerplate.
2. Check out the Spatial Analyst extension with
`gp.CheckOutExtension("spatial")`
3. Set the gp.workspace to "d:/prog/HMBarea"

-
4. Use the following code to do a raster to polygon conversion if it doesn't already exist. Note that we use the workspace setting in addition to setting it:

```
if not gp.Exists(gp.Workspace + "/citypoly.shp"):
    gp.RasterToPolygon_conversion("city", "citypoly")
else:
    print gp.Workspace + "/citypoly.shp exists"
```

5. Using a similar set of code, check to see if the "hmbcity" workspace exists within the existing workspace, and if not, use gp.CreateArcInfoWorkspace to create it with
`gp.CreateArcInfoWorkspace(gp.workspace, "hmbcity")`

(don't forget the rest of the code – I've just given you one line)

6. Use the describe method to define the gp.Extent setting to match that of "citypoly.shp" and the gp.Mask setting to "city".

7. Create a list **raslist** of rasters.

8. Use a for loop to loop through each raster, and store the result in the hmbcity folder. I'll give you this:

```
for ras in rasList:
    outputname = gp.Workspace + "/hmbcity/" + ras
    print outputname
    if not gp.Exists(outputname):
```

```
gp.ExtractByMask_sa (ras, gp.Mask, outputname)
```

? Note the use of outputname. What does it represent?

9. Check the extension back in with

```
gp.CheckInExtension("spatial")
```

Note: You should put the checkin code in both the try: and except: sections.

Note: it may take a while to run. Look at the interactive window to see which dataset is processing.

Create a Smaller Workspace

Use coverages from **pendata** to create a smaller workspace called **penshapes** including the following coverages converted to shapefiles (makePenShapes.py):

cities/polygon geology/polygon landuse/polygon streams/arc roads/arc

Challenge: selSanMateoSel.py

Use Select_Analysis to create a city of San Mateo shapefile, create a smaller workspace called SanMateo, then populate that workspace with shapefiles clipped from polygon coverages in pendata that start with 'g' ("g*"). Hints: you'll need to use gp.listdatasets("g*") instead of listfeatureclasses, since you need to find coverages, and as you're processing the list members and have assigned a member as the variable f, use gp.exists(f + "/polygon") to find the polygon feature class in the coverage. You'll also find gp.CreateFolder_management useful. You could also clip all of the dataset by leaving the wildcard set to "*" or not using the wildcard.

Challenge: multiFeatureRas.py

Create a two Python lists, to hold the dataset pairs "geology", "type-id", "landuse", "lu-code", "publands", "pubcode", and "vegetation", "veg-code". Convert each coverage to a raster, with the corresponding field, and use "60" as the cell size.

```
cov = ["geology", "landuse", "publands"]  
fld = ["TYPE-ID", "LU-CODE", "PUBCODE"]
```

8. Using Map Algebra in Python scripts

The only hesitation I had in moving to Python from AML was that it didn't appear that you could code Map Algebra statements in Python as you could in AML running Grid operations. But there is a way, and it actually works just fine, with a little polishing with a defined function. The MultiOutputMapAlgebra tool of the Spatial Analyst toolbox lets you send a complete Map Algebra assignment statement, including nested functions (!). The only minor catch is the tool name is so long, so I've embedded it in a defined function with a very short name "ma" which also includes starting and stopping the "Spatial" extension. The following illustrates its use with just a few statements sent, but it includes a couple that are awkward to do in multiple tool steps.

usingMapAlgebra.py

```
# Using Map Algebra in Python
#
import arcgisscripting
gp = arcgisscripting.create()

def ma(expression):
    gp.CheckOutExtension("spatial")
    gp.MultiOutputMapAlgebra_sa(expression)
    gp.CheckInExtension("spatial")

gp.workspace = "d:/prog/hmbarea"
gp.OverwriteOutput = 1

gp.FeatureToRaster_conversion("streams/arc", "st-code", "streamsg", "60")
gp.CellSize = 60

ma("slopeg = slope(elev, 0.3048)")
ma("streams2 = con(isnull(streamsg), 0, streamsg)")
ma("hstreams = (elev > 1000) and streams2")
```

Another way to do this:

Since Python lets you assign methods to variables, you could also do the same thing with the function by first assigning the following:

```
ma = gp.MultiOutputMapAlgebra_sa
```

The use of ma would work the same, though you'd need to make sure to check out the extension first, *including when you are initially assigning the **ma** variable*, as shown below. Safe programming would add gp.CheckInExtension("spatial") to an Except: clause – just to avoid the error you'd see if you try to check out an extension that's already checked out.

```
gp.CheckOutExtension("spatial")
ma = gp.MultiOutputMapAlgebra_sa
ma("slopeg = slope(elev, 0.3048)")
ma("streams2 = con(isnull(streamsg), 0, streamsg)")
ma("hstreams = (elev > 1000) and streams2")
gp.CheckInExtension("spatial")
```

9. Data Management and Cursors

Processing data in tables, as well as creating fields to store those data, and other data operations are important in GIS work. Using a script is a good choice when you need to perform a sequence of data management and analysis steps involving data tables. In some cases we need to process data fields, and there is an array of tools we can use to, for example, add fields, calculate values for fields, delete fields, and join tables to bring in additional data fields via a relate field. Some of these tools also create new summary tables, where input data fields are summarized using various statistics. In other cases, we need to work with rows of data, which might be individual features with vector data or values for raster data; we'll learn about using **cursors** to process rows, one at a time.

First, we'll look at some of the tools we can use to process data tables, primarily involving fields. A few tools also select records (by attributes), or copy or delete selected records.

Toolbox	Tool	What it does	Output
Analysis	frequency	Calculates frequency statistics for field(s) in the input table.	Table
	statistics	Calculates summary statistics for field(s) in the input table.	Table
	select	Uses a where clause to selects features from an input to create an output	new feature class
	table_select	Extracts selected attributes from an input table, using an SQL Where clause	Table
Data Management Alias: <i>management</i>	addField	Adds a field to an data table	Field in existing table
	calculateField	Calculates a value for a field using an expression	Values in an existing field
	deleteField	Removes a field from a table	
	addXY	Adds an x & y fields to a point feature class	X & Y Fields with values
	selectLayer ByAttribute	Creates, updates, or removes a selection on the layer or table view using an attribute query.	
	copyFeatures	Copies selected features to a new feature class	new feature class
	deleteFeatures	Deletes selected features	
	addJoin	Joins a table to a layer (or a table to a table) based on a common field	join relationship
	removeJoin	removes an existing join	
Coverage Alias: <i>Arc</i>	select (reselect)	Extracts map features from the input coverage based on logical expressions	Output coverage (and attribute table)
	additem	Adds a field to an info table	Field in existing table
	dropitem	Removes a field from an info table	
	joinitem	Merges data tables based on a relate field	Table
	addxy	Adds x and y fields to point, label or node tables	Fields in existing table, with calculated values

- Create a script (**addCalcField.py**) that adds the field "elevm" to contours.shp in surf_bld, and calculate its values as [elev] * 0.3048. First set the gp.toolbox to "management". Use the code displayed here to detect if the field already exists, then add the field if it doesn't. The next step is to use the calculatefield tool to assign the value "[elev] * 0.3048" to "elevm".

```
if not gp.listFields("contour.shp", "elevm"):  
    gp.addField ("contour.shp", "elevm", "float")
```

- Create a script (**addXY.py**) that uses the addxy tool in the management toolbox (features toolset) to add x & y values to "samples.shp" in the Marbles workspace.

Cursors

Cursors give you access to values in your data fields, and allow you to loop through records in your data tables. Since each record (row) might be a vector feature or a raster value, this gives you considerable power to process your data.

There are three types of cursors:

- **SearchCursor** : read values in a row
 - **InsertCursor** : insert new rows
 - **UpdateCursor** : to change values in rows and delete rows
- Try the following code, which demonstrates the use of the SearchCursor to display the results of your addCalcField.py script you ran above. Use it with all necessary boilerplate and a try...except structure. Note the use of field names as properties of the row.

```
gp.workspace = "D:/prog/surf_bld"  
cur = gp.SearchCursor("contour.shp")  
row = cur.Next()  
while row:  
    print row.elev, row.elevm  
    row = cur.Next()
```

- Try the following code, which demonstrates reading and displaying all of the vertices from stream.shp in surf_bld. It also demonstrates the use of feature geometry. *Don't forget the .Next() in the while loop, or you'll get an infinite loop!!!*

```
# Create search cursor
rows = gp.SearchCursor("D:/prog/surf_bld/stream.shp")
row = rows.Next()
# Print the coordinates of each road line feature
while row:
# Create the geometry object
    feat = row.shape
    a = 0
    while a < feat.PartCount:
        # Get each part of the geometry
        stArray = feat.GetPart(a)
        stArray.Reset
        # Get the first point object for the feature
        pnt = stArray.Next()
        while pnt:
            print str(pnt.id) + "; " + str(pnt.x) + "; " + str(pnt.y)
            pnt = stArray.Next()
        a = a + 1
    row = rows.Next()
```

- Try the following insert cursor, which demonstrates inserting a new record into a file with hard-coded data. The same basic method could be used to read data from an external source like a text file – see the "Even bigger challenge" on the next page. It also uses feature geometry.

```
gp.workspace = "d:/prog/Marbles"
try:
    cur = gp.InsertCursor("mvalley_pts.shp")
    feat = cur.NewRow()
    feat.id = 12
    feat.Name = "Trail Junction Cave"
    pnt = gp.CreateObject("Point")
    pnt.x = 483986
    pnt.y = 4600852
    feat.shape = pnt
    cur.InsertRow(feat)
    del cur
except:
    print gp.getmessages()
    if cur: del cur
```

Challenge:

Using the following snippets you can use to open, write to, and close a text file, modify the above code to export the vertices values to a text file (**exportLines.py**).

```
import os
.
.
wspath = "d:/prog/surf_bld"
txtfile = "strout.txt"
fpath = wspath + "/" + txtfile
if gp.Exists(fpath): os.remove(fpath)
txtfile = open(fpath, "w")
.
.
fid = row.GetValue("ID")
.
.
roadArray = feat.GetPart(a)
pnt = roadArray.Next()
while pnt:
    txtfile.write(str(fid) + "; " + str(pnt.x) + "; " + str(pnt.y)+"\n")
.
.
txtfile.close()
```

Even bigger challenge:

Write a script that is able to read in the text file you just created, and create a new shape file from it. This is included as **importLines.py**. Be careful with this -- while getting it running I managed to hose PythonWin a couple of times. I got the idea from a sample in *Writing Geoprocessing Scripts*, but don't try it – there are errors, now fixed in ArcGIS Desktop Help Online (in the ArcCatalog and ArcMap Help menu), by going to “Writing Geoprocessing Scripts”, then to “Data access using cursors in a geoprocessing script”. This is a good resource, since it's more updated for sample scripts.

What's next?

Where do we go from here? We've done a lot, but most of our effort has been building sections of code which do specific things, like loop through a series of datasets. We've done very little in those loops so far, but the possibilities are nearly endless.

Another thing we haven't spent much time on is getting script tools working well. Inputs and outputs are the challenging part of this process, so working on that would benefit you. I've found this the most confusing part, and so I've put together the following Appendix, which is a work in progress, but attempts to list illustrations of as many input/output scenarios I can find.

One last recommendation is to create models that illustrate your script tools. Running script tools in models is the most rewarding and useful way to work.

Appendix 1: **Guide to Input & Output Parameters for Geoprocessing Scripts**

With examples from Installed Scripts.

1. Input Datasets

- Script: Use `GetParameterAsText` or `sys.argv`.
- Tool parameters: (Feature Class, Raster Dataset, or maybe layer etc.) Required Input

2. Output Datasets (e.g. *Mean Center* in Spatial Statistics/Measuring Geographic Distributions)

- Script: Use `GetParameterAsText` or `sys.argv`.
- Tool parameters: (Feature Class, Raster Dataset, etc.) Required Output

3. Output Values and Fields (*Calculate Areas* in Spatial Statistics/Utilities)

- Script: Input and Output datasets; copies input to output
- Tool parameters: Input feature class, Output feature class.

4. Output single value. (*Spatial Autocorrelation* in Spatial Statistics/Analyzing Patterns)

- Script: `SetParameterAsText`
- Tool parameters: (Long, Double, etc.): Derived Output

5. Input and Output Tables (or Feature Class) (e.g. *Transpose Time fields* in Management/Fields)

- Script: `GetParameterAsText` for both (also see code for creating the table)
- Tool parameters: Input: Table View, Required Input; Output: Table, Required Output

6. Modifying an Input to an Output: no simple examples found, but *Workspace to Mosaic* (Samples/Conversion/Raster) modifies an output that must already exist:

- Script: Output in `GetParameterAsText`; same as `SetParameterAsText`
- Tool parameters: Output as Required Input; then as Derived Output, and set Obtained From to original

7. Batch operations in a workspace (e.g. *Batch Project* in Management/Projections and Transformations/ Feature)

- Script: Input and Output as `GetParameterAsText`; Derived Output isn't needed – see example has 5 parameters in the script, but 6 in the script tool)
- Tool parameters: Input Geodataset, Required, Input, Multivalue Yes;
Output Workspace, Workspace or Feature Dataset, Required Input;
also as Derived Output with dependency on Output Workspace

8. True if completed successfully (e.g. *Batch Calculate Statistics* in Management/Raster)

- Script: nothing
- Tool parameters: Boolean parameter, Derived Output with Default true value.

When to use 'derived'? "if the tool updates the input to the tool, or if you need an output for use in a model but you don't want the output parameter to show in the script's dialog box."

Appendix 2: **Other Comments/Issues**

1. Workspace and ScratchWorkspace Settings:

While you can use the `gp.ScratchWorkspace` setting in a script, when a script tool is added to a model, there doesn't appear to be a way to get the script tool interface to operate the way a system tool does when a scratch workspace setting is in force – that is, to initially populate the output by automatic naming in the scratch workspace. Maybe there's a way to use the environment settings of the script tool parameter to do this, but it's not clear. The same applies to the main workspace setting – it isn't automatically populated as an output as it would normally do with a system tool if the scratch workspace setting isn't in force.

2. Python and PythonWin versions:

Don't be tempted to update Python or PythonWin on your computer. It's best to use the version that comes with ArcGIS. Also, PythonWin isn't automatically installed with ArcGIS, but the installation files are on the disk; you have to search for them. ESRI has talked about having their own IDE, but on their user discussions, it sounds like people use various IDEs.

INDEX

\, 5
≠, 13
;, 7
<, 13
≤, 13
<>, 13
==, 13
>, 13
≥, 13
3d, 22
addField, 37, 38
additem info, 37
addJoin, 37
addmessage, 28
addXY, 37
addxy coverage, 37
Aliases, 22
analysis toolbox, 22
Analysis toolbox, 37
append(), 14
arcgisscripting, 16
arcgisscripting.create(9.3), 16
arguments, 15
automate, 1
Books, 2
Boolean, 13
break point, 30
buffer_analysis, 22
calculateField, 37
capitalize(), 5
CheckInExtension, 35
CheckOutExtension, 34
Clip, 28
Clip_analysis, 22
Concatenate strings, 5
copyFeatures, 37
coverage, 17
CreateArcInfoWorkspace, 34
CreateObject, 39
Cursors, 38
Data Management, 37
Data Management toolbox, 22, 37
dataset properties, 32
debugger, 30
Debugging, 28
Debugging Tools, 30
def, 8
deleteFeatures, 37
deleteField, 37
DEMs, 12
DEMtoRaster, 31
Derived, 25
'derived', 41
Describe, 32
digital elevation models, 12
Direction, 25
Double, 25
dropitem info, 37
endswith(), 14
Environment Settings, 21
Exists, 33, 40
ExtractByMask_sa, 35
feature classes, 17
feature datasets, 17
feature to raster conversion, 31
FeatureClass properties, 32
field exists, 33
field.Name, 20
field.Type, 20
fileinput.close(), 12
fileinput.input(), 12
float, 23
for, 9, 14
frequency, 37
gauss, 14
geodatabase, 17
geometry object, 39
Geoprocessing scripts, 16
geoprocessing tools, 1
Geoprocessor Object, 19
Geoprocessor Programming Model, 19, 32
GetMessages, 29, 31
GetParameterAsText, 23, 27
GetPart, 39, 40
GetValue, 40
gp, 16
hypot, 7
IDLE, 2
if, 9, 10
import, 6
Input, 15, 25
InsertCursor, 38
InsertRow, 39
Integer vs. Floating Point, 4
Integrated Development Environments (IDEs), 2
interactive window, 2
joinitem info, 37
Label of script tool, 24
Learning Python, 2
ListDatasets, 17, 18
ListEnvironments, 18
ListFeatureClasses, 16, 19
listFields, 33, 38
ListRasters, 18
ListTables, 18
ListToolboxes, 18
ListTools, 18
listWorkspaces, 18
log, 7
logic, 1
Long, 27
Loop, 34
Map Algebra, 36
math, 7
Mathematical Computation, 4
Methods, 4
ModelBuilder, 23
modules, 6
MultiOutputMapAlgebra_sa, 36
n, 5

- Name**, 20
- Name of script tool, 24
- new line, 5
- NewRow**, 39
- Next, 40
- `os.listdir(ws)`, 14
- `os.path`, 11
- `os.remove`, 40
- Output, 15, 25
- `overwriteoutput`, 22
- Parameters**, 41
- parameters of script tool, 25
- print, 15
- Print statements**, 28
- PythonWin, 2
- `r (raw)`, 5
- radians, 8
- random, 7, 14
- `random.gauss()`, 14
- `RasterToPolygon`, 34
- removeJoin**, 37
- Required, 25
- Sample Scripts**, 21
- ScratchWorkspace**, 42
- script tool, 24
- SearchCursor**, 38

- select**, 37
- select coverage**, 37
- selectLayer**, 37
- `SetParameterAsText`, 23
- shapefiles, 17
- Spatial Analyst toolbox, 22
- Spatial Statistics toolbox, 22
- Split, 5
- `sqrt`, 7
- statistics**, 37
- `str()`, 14, 23
- string*, 4
- String manipulation, 4
- `sys`, 15
- `sys.argv[]`, 15
- table properties, 32
- table_select**, 37
- Toolbox, 29
- Toolboxes, 22
- Trig functions, 7
- `try...except`, 29
- UpdateCursor**, 38
- while**, 9, 13
- workspace**, 16
- Workspace**, 35, 42
- `write`, 40