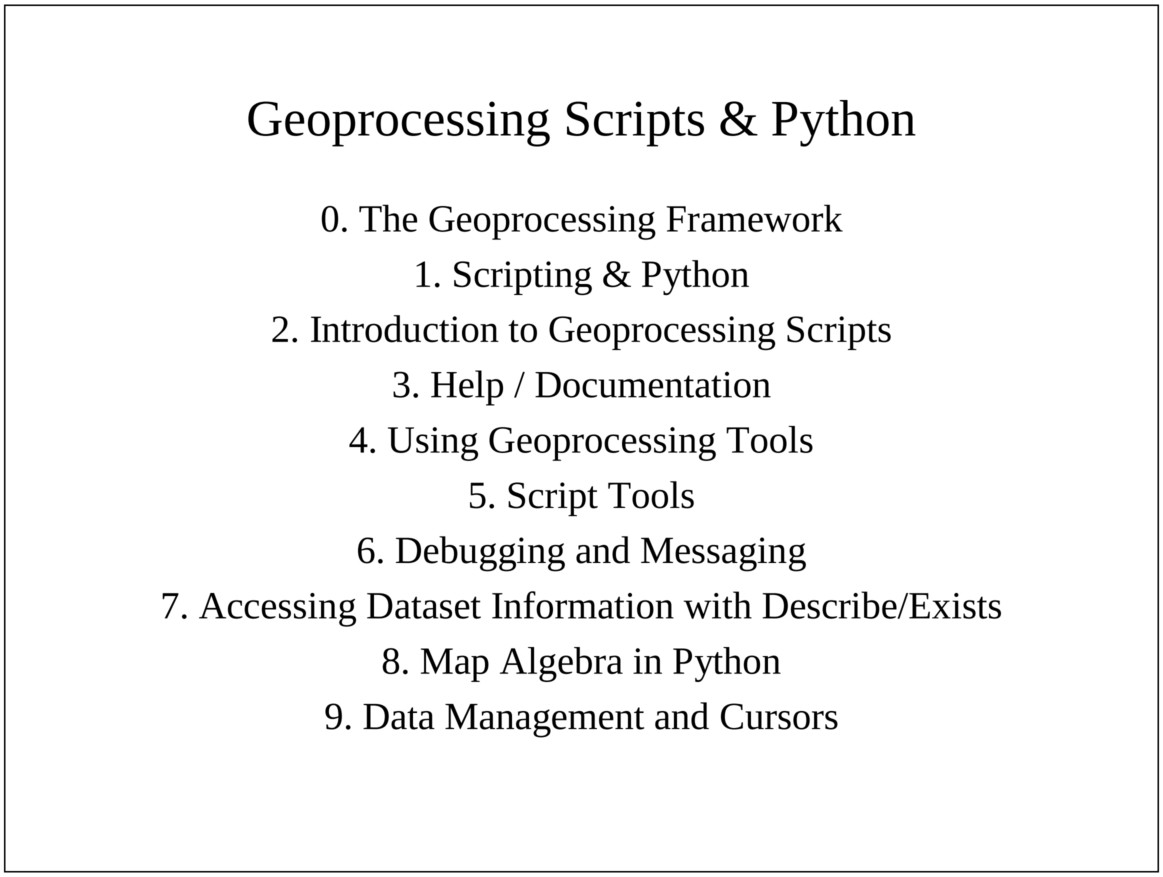
Slide 1



Geoprocessing Scripts & Python

0. The Geoprocessing Framework
1. Scripting & Python
2. Introduction to Geoprocessing Scripts
3. Help / Documentation
4. Using Geoprocessing Tools
5. Script Tools
6. Debugging and Messaging
7. Accessing Dataset Information with Describe/Exists
8. Map Algebra in Python
9. Data Management and Cursors

0. **The Geoprocessing Framework** : Python in the context of the geoprocessing framework, including the ArcToolbox and ModelBuilder.
1. **Scripting & Python** : Introduction to the Python language and programming environment as a scripting method.
2. **Introduction to Geoprocessing Scripts** : Brief introduction to geoprocessing using Python.
3. **Help / Documentation** : Help resources, including the help system and the geoprocessing model diagram.
4. **Using Geoprocessing Tools** : How tools in the geoprocessing framework are used with Python.
5. **Script Tools** : How to convert a Python script into a script tool, so you can access it from ArcToolbox and ModelBuilder.
6. **Debugging and Messaging** : Methods available for debugging scripts and sending messages to the user.
7. **Describe** : Accessing Dataset Information with Describe and Exists
8. **Map Algebra in Python** : Map Algebra, how it differs from geoprocessing, and how we can use it in a script anyway.
9. **Data Management and Cursors** : Accessing data and processing rows from our data, including accessing geometry.

Slide 2

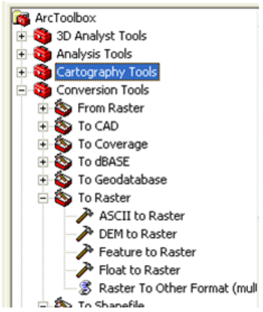
The Geoprocessing Framework

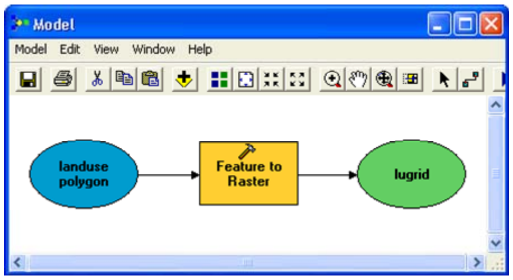
ArcToolbox

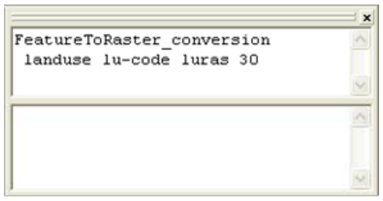
ModelBuilder

Command Line

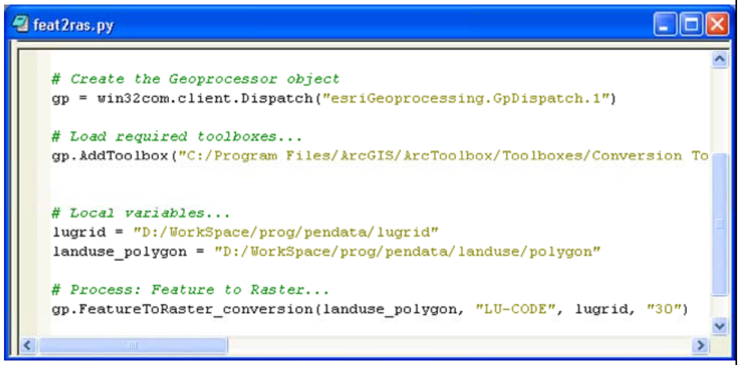
Scripting







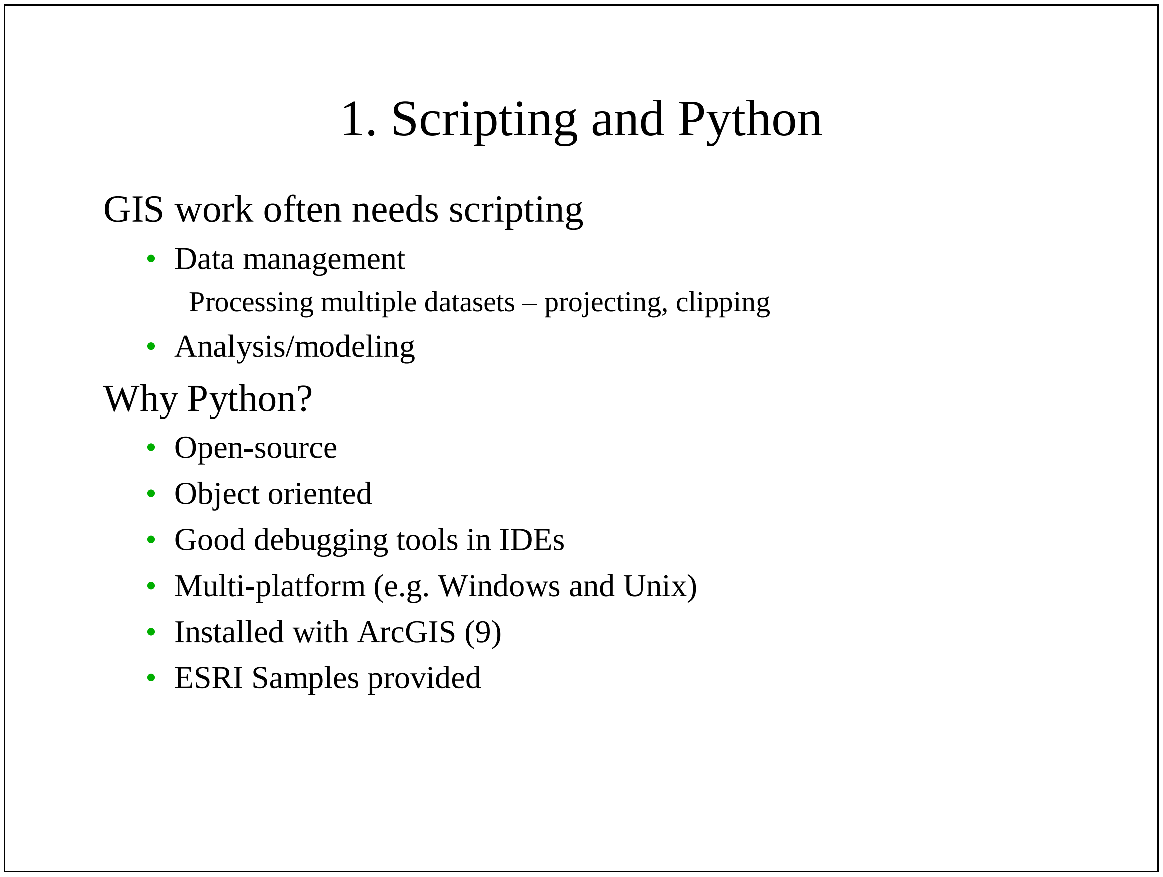
```
FeatureToRaster_conversion
 landuse lu-code luras 30
```



```
# Create the Geoprocessor object
gp = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")

# Load required toolboxes...
gp.AddToolbox("C:/Program Files/ArcGIS/ArcToolbox/Toolboxes/Conversion To


# Local variables...
lugrid = "D:/WorkSpace/prog/pendata/lugrid"
landuse_polygon = "D:/WorkSpace/prog/pendata/landuse/polygon"

# Process: Feature to Raster...
gp.FeatureToRaster_conversion(landuse_polygon, "LU-CODE", lugrid, "30")
```

The Geoprocessing Framework includes several ways of running tools:
(1) The ArcToolbox. Organized by toolboxes, with toolsets. Includes system tools, scripts, and models. Many tools come with the software, but you can add your own toolboxes with scripts and models.
(2) ModelBuilder. You can run tools by dragging them in from ArcToolbox, and process input data sets to create outputs. Can be complex, can run scripts and other models in addition to system tools.
(3) Command line. Can run tools by typing them in, using command line syntax. Note: this command window and these commands are not the same as classic ArcInfo (Arc:) command window and commands.
(4) Scripts. Primarily Python scripts, letting you use programming structures like loops and process conditionals either not available or difficult to use in ModelBuilder.

Slide 3



1. Scripting and Python

GIS work often needs scripting

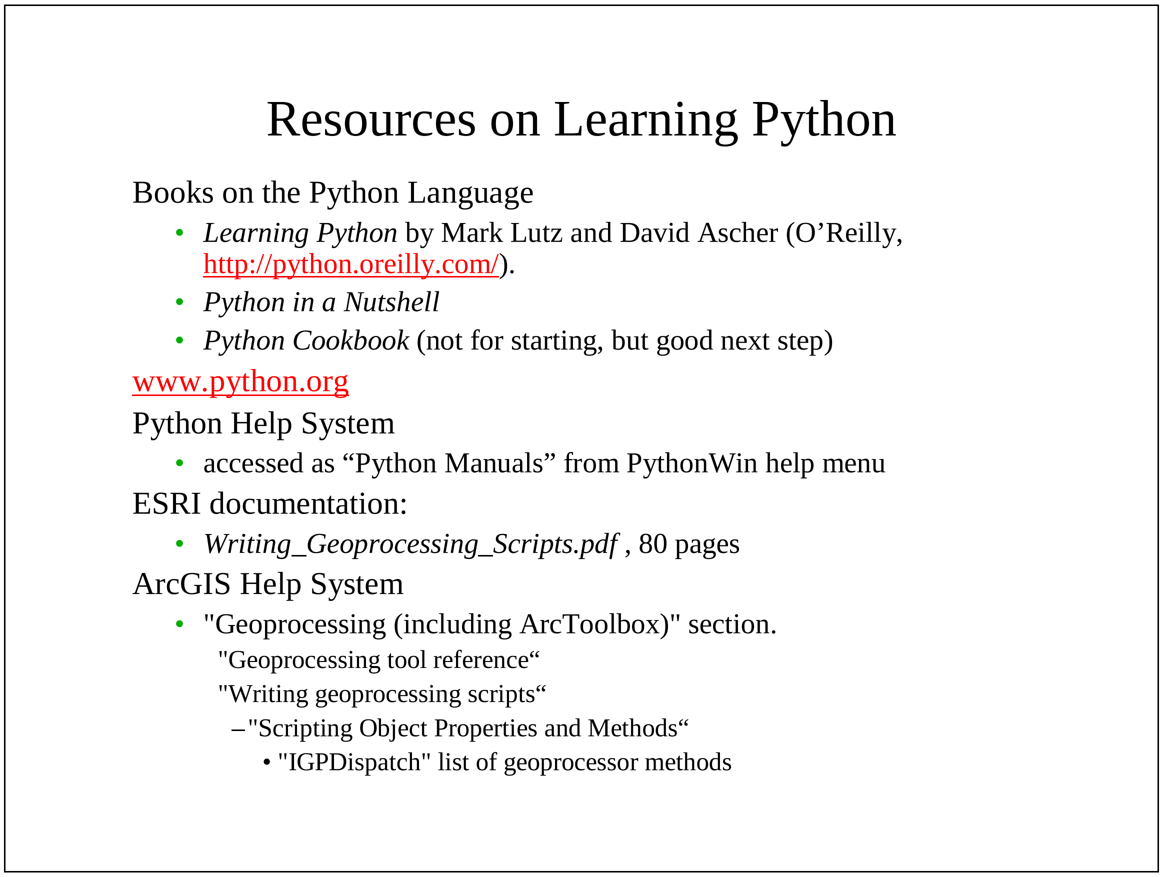
- Data management
  - Processing multiple datasets – projecting, clipping
- Analysis/modeling

Why Python?

- Open-source
- Object oriented
- Good debugging tools in IDEs
- Multi-platform (e.g. Windows and Unix)
- Installed with ArcGIS (9)
- ESRI Samples provided

Thinking of data management and analysis separately is useful in helping us understand what we use geoprocessing for. But the boundary is somewhat fuzzy. Data management operations may be a part of an analysis process. For instance if your analysis involves a step of clipping data for further analysis, we see the use of what might otherwise be a data management tools (clip) being used in analysis.

Slide 4



Resources on Learning Python

Books on the Python Language

- *Learning Python* by Mark Lutz and David Ascher (O’Reilly, http://python.oreilly.com/).
- *Python in a Nutshell*
- *Python Cookbook* (not for starting, but good next step)

www.python.org

Python Help System

- accessed as “Python Manuals” from PythonWin help menu

ESRI documentation:

- *Writing_Geoprocessing_Scripts.pdf* , 80 pages

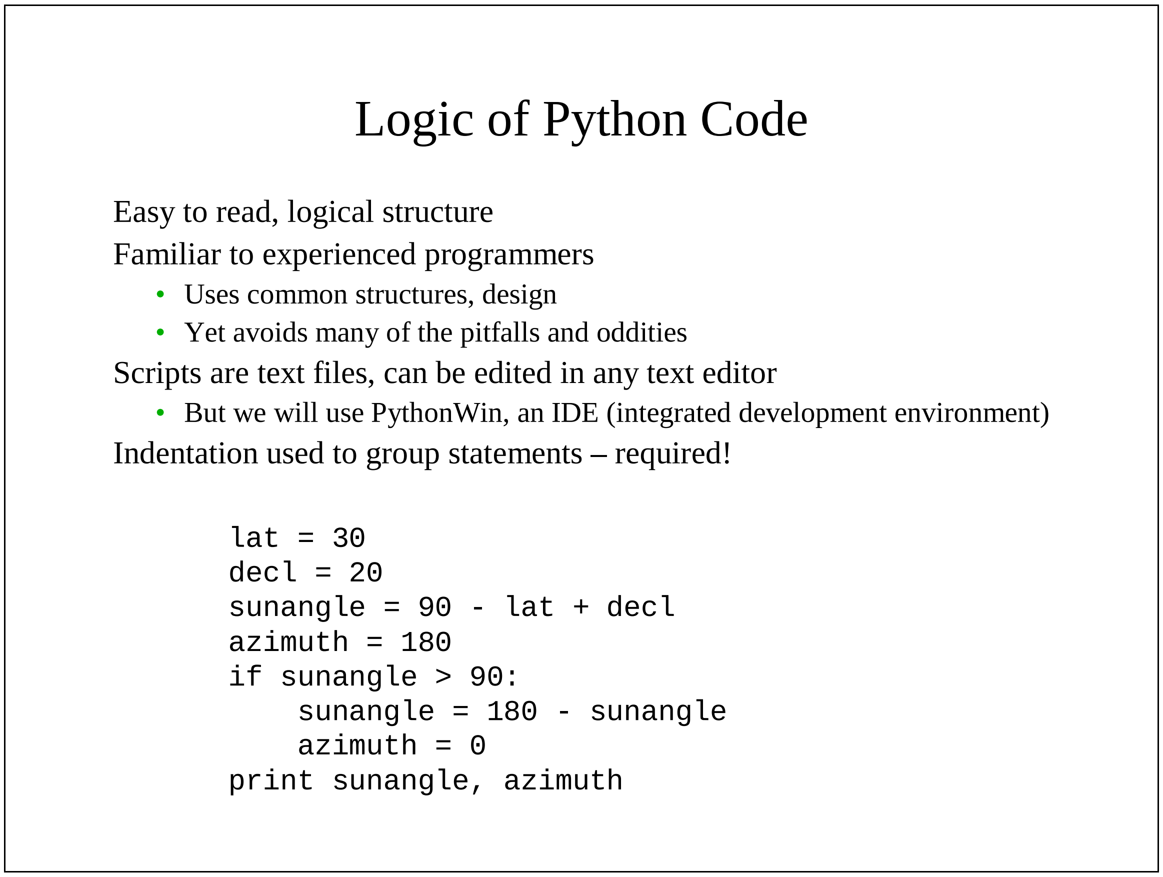
ArcGIS Help System

- "Geoprocessing (including ArcToolbox)" section.
    - "Geoprocessing tool reference“
    - "Writing geoprocessing scripts“
        - –"Scripting Object Properties and Methods“
            - • "IGPDispatch" list of geoprocessor methods

Python is a very rich language, especially as it is extended by modules developed by the Python community. In this course, we will barely scratch the surface on Python capabilities. We’ll learn the basic programming methods sufficient to getting our geoprocessing work done. You can learn a great deal more in other references. The ESRI pdf is a useful reference, but exploring the online help will give you the most current information, and sample scripts.

The Python Manuals in the help system are very detailed. See the Library Reference, and the Global Module Index. You can also get to tutorials and lots of other stuff.

Slide 5



Logic of Python Code

Easy to read, logical structure

Familiar to experienced programmers

- Uses common structures, design
- Yet avoids many of the pitfalls and oddities

Scripts are text files, can be edited in any text editor

- But we will use PythonWin, an IDE (integrated development environment)

Indentation used to group statements – required!

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

Notepad++ understands Python code structures, and uses color coding for different kinds of elements, so is useful in looking at code. But with no debugger, it’s limited.

If you have experience with other languages, such as BASIC or C++, you may have learned to indent your code to make it more readable. But since indentation is not required in these languages, you can get sloppy and create code that is difficult to read. Python forces you to indent to define code segments. This is a good thing in that our code is very readable and efficient – no need for end statements.

Slide 6

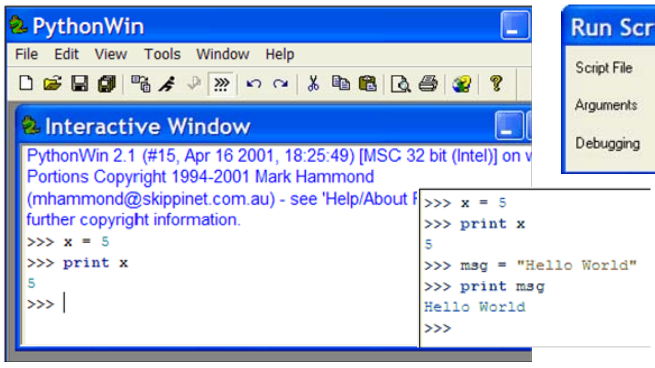
PythonWin

An IDE (Interactive Development Environment) for Python

- There are others – IDLE – that are more platform independent
- But PythonWin works ok in Windows

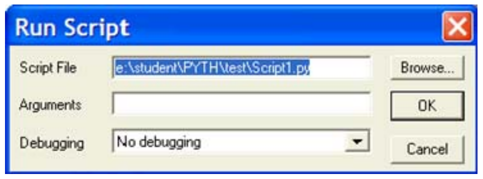
Can run Geoprocessing scripts in Python

- Or regular Python scripts (we'll start here)…. demo



```
PythonWin 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on v
Portions Copyright 1994-2001 Mark Hammond
(mhammond@skippinet.com.au) - see 'Help/About F
further copyright information.
>>> x = 5
>>> print x
5
>>> |
```

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



Script File: e:\student\PYTH\test\Script1.py
Arguments:
Debugging: No debugging

Many python programmers like to use other IDEs, like IDLE, or even just a text editor like notepad, or a more capable editor like notepad++ or emacs. But when you right-click a Python script in ArcToolbox or ModelBuilder, and select Edit, PythonWin is the editor selected by default (and I don't know how you'd change this.) So you kind of need to learn PythonWin anyway. This may change with future versions of ArcGIS – the word is that a custom IDE will be used in a future release.

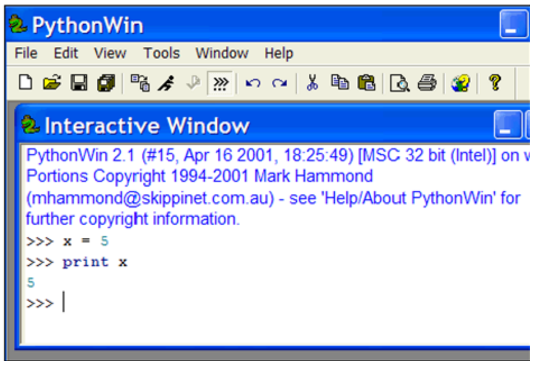
Slide 7

Using the Interactive Window

Print statements produce output here

Also errors and other messages

Good place to explore the language

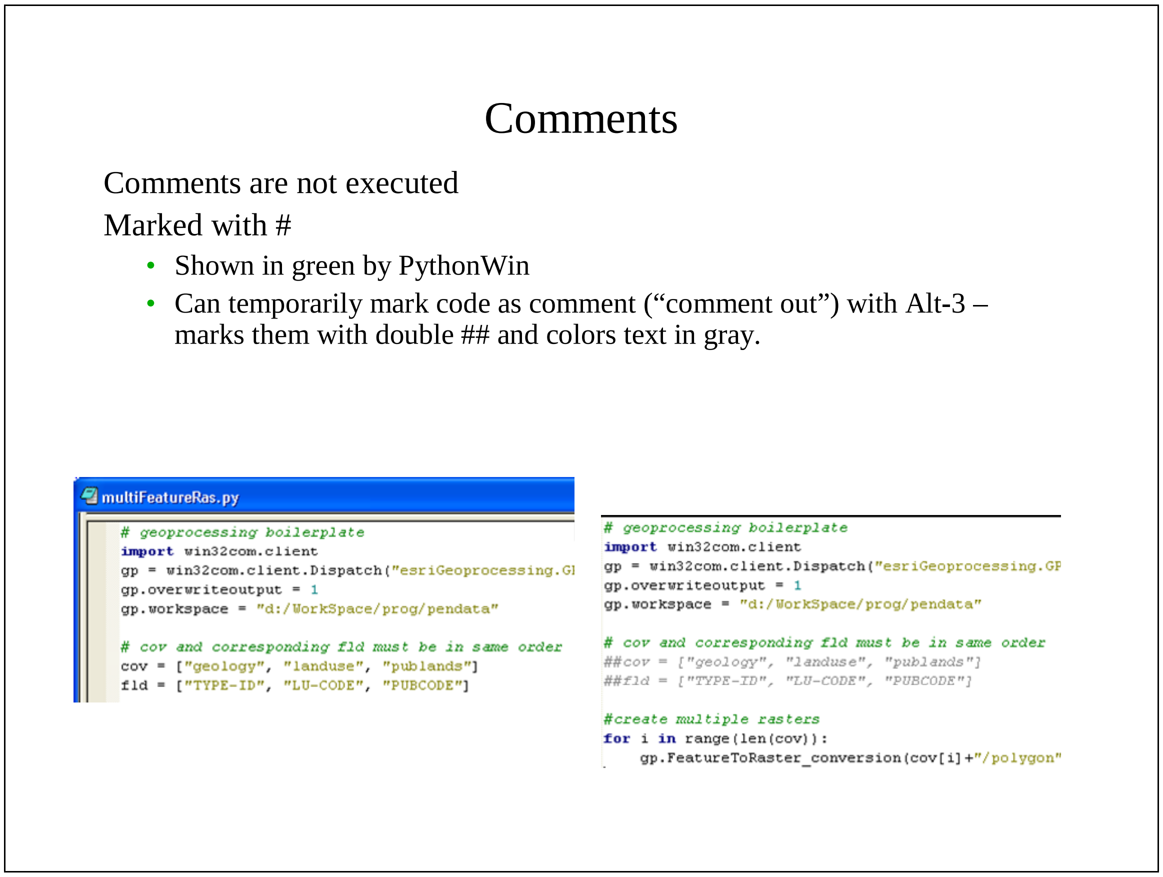


```
PythonWin 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on v
Portions Copyright 1994-2001 Mark Hammond
(mhammond@skippinet.com.au) - see 'Help/About PythonWin' for
further copyright information.
>>> x = 5
>>> print x
5
>>> |
```

The best way to learn the language or try out simple operations is to type directly into the Interactive Window.

We’ll use this in the exercises to explore the python language.

Slide 8



Comments

Comments are not executed

Marked with #

- Shown in green by PythonWin
- Can temporarily mark code as comment ("comment out") with Alt-3 – marks them with double ## and colors text in gray.

multiFeatureRas.py

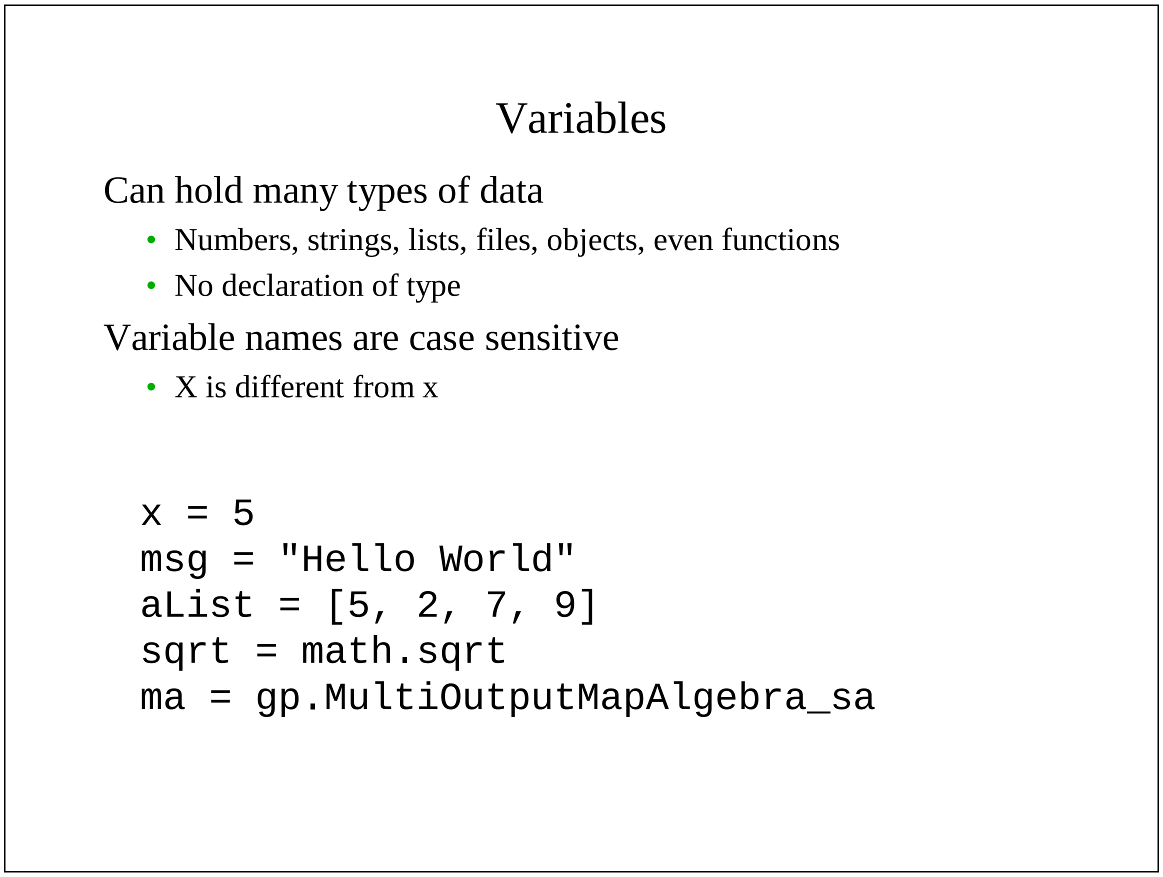
```
# geoprocessing boilerplate
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.Gl
gp.overwriteoutput = 1
gp.workspace = "d:/WorkSpace/prog/pendata"

# cov and corresponding fld must be in same order
cov = ["geology", "landuse", "publands"]
fld = ["TYPE-ID", "LU-CODE", "PUBCODE"]
```

```
# geoprocessing boilerplate
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.GP
gp.overwriteoutput = 1
gp.workspace = "d:/WorkSpace/prog/pendata"

# cov and corresponding fld must be in same order
##cov = ["geology", "landuse", "publands"]
##fld = ["TYPE-ID", "LU-CODE", "PUBCODE"]

#create multiple rasters
for i in range(len(cov)):
    gp.FeatureToRaster_conversion(cov[i]+"/polygon"
```

Good code should include comments to help explain what it's doing. This is helpful not only for other people reading your code, but also for the programmer understanding code later on.

Temporarily commenting out code is handy during debugging, or alternating between using code at with an IDE or as a script tool (discussed later), where print statements only work from the IDE. (We'll learn alternative printing methods later on.) Alt-3 comments an entire line, using gray text. Alt-shift-3 should undo the commenting.

Slide 9



Variables

Can hold many types of data

- Numbers, strings, lists, files, objects, even functions
- No declaration of type

Variable names are case sensitive

- X is different from x

```
x = 5
msg = "Hello World"
aList = [5, 2, 7, 9]
sqrt = math.sqrt
ma = gp.MultiOutputMapAlgebra_sa
```

Variables can hold quite a lot of things, even names of commands or tools.

Geoprocessing tool names are not case sensitive, nor are paths. But variable names are case sensitive. So in the last statement above,

```
ma = gp.MultiOutputMapAlgebra_sa
```

… you could have just as well written it:

```
ma = gp.MULTIOUTPUTMAPALGEBRA_SA
```

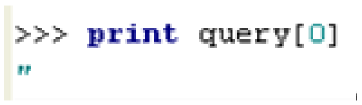
… but the variable **`ma`** cannot be accessed as **`Ma`**, **`MA`**, **`mA`** or anything other than **`ma`**. Variables are case-sensitive.

Slide 10

Strings

- Variables can be assigned strings
- Defined by enclosing in single or double quotes
  - Can include quotes by using other type:

```
mystr = "My name is 'Hieronymous'"
query = '"AREA" > 1000'
```



```
>>> print query[0]
"
```

- Strings are indexed (zero based)
- Special characters can be used with escape character ‘\’
  - \n newline
  - \t tab
  - \\ ‘\’
  - \ line continuation
- Concatenate with + msg + str(5)

Note that paths can’t use single back-slashes.
`"d:\workspace\prog\hmbarea"` doesn’t work in a program. Instead use either:
`"d:\\workspace\\prog\\hmbarea"` or
`"d:/workspace/prog/hmbarea"`

This takes some getting used to, and it limits your ability to paste in paths from windows.

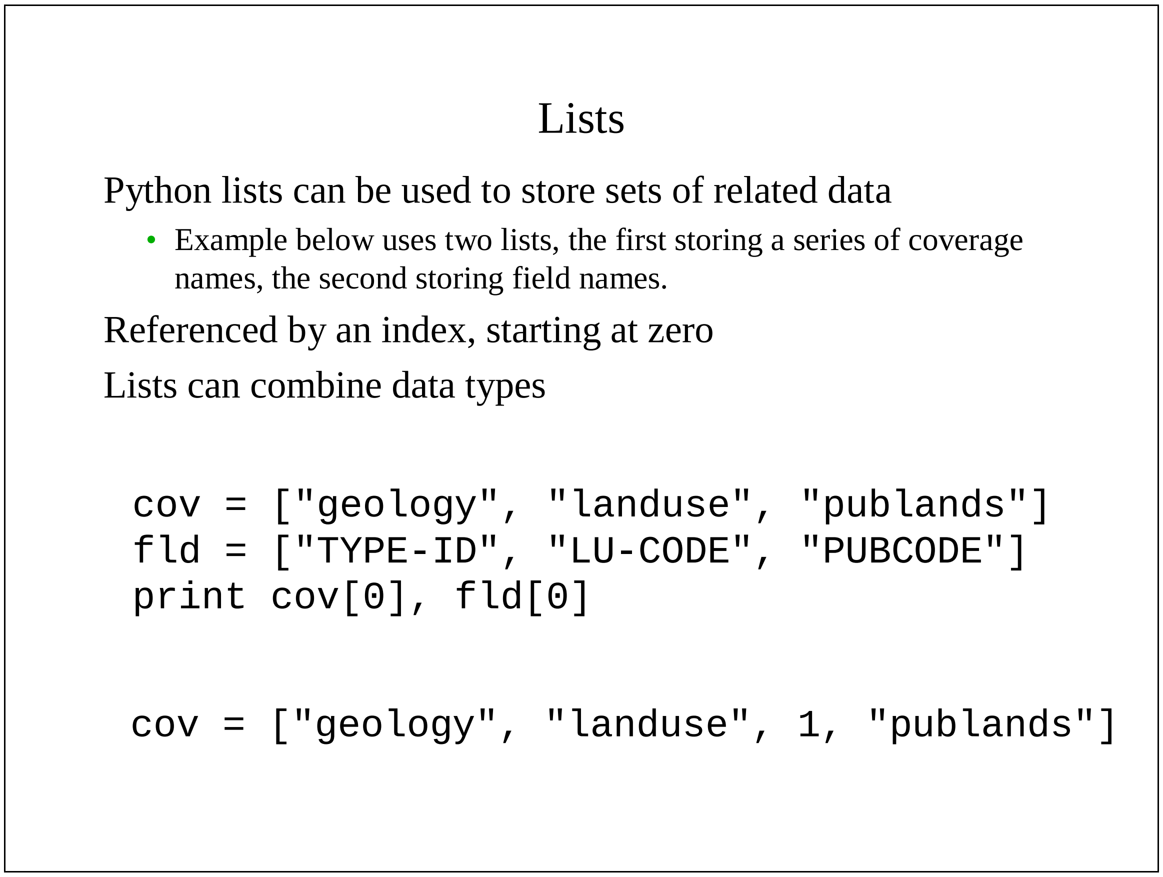
While you can print out numerical variables like:

```
x = 5
print x
```

… if you want to combine these with strings and use concatenation to do, both parts being concatenated need to be strings, so you’d need to do something like the following:

```
x = 5
print "My number is: " + str(x)
```

Slide 11



Lists

Python lists can be used to store sets of related data

- Example below uses two lists, the first storing a series of coverage names, the second storing field names.

Referenced by an index, starting at zero

Lists can combine data types

```
cov = ["geology", "landuse", "publands"]
fld = ["TYPE-ID", "LU-CODE", "PUBCODE"]
print cov[0], fld[0]
```

```
cov = ["geology", "landuse", 1, "publands"]
```

We might find these lists handy when we want to loop through a series of datasets, though we'll mainly use geoprocessing enumerations for these… more on these later.

Note that the index starts with zero. If this bothers you, sorry, but we'll discover that it makes certain coding structures easier.

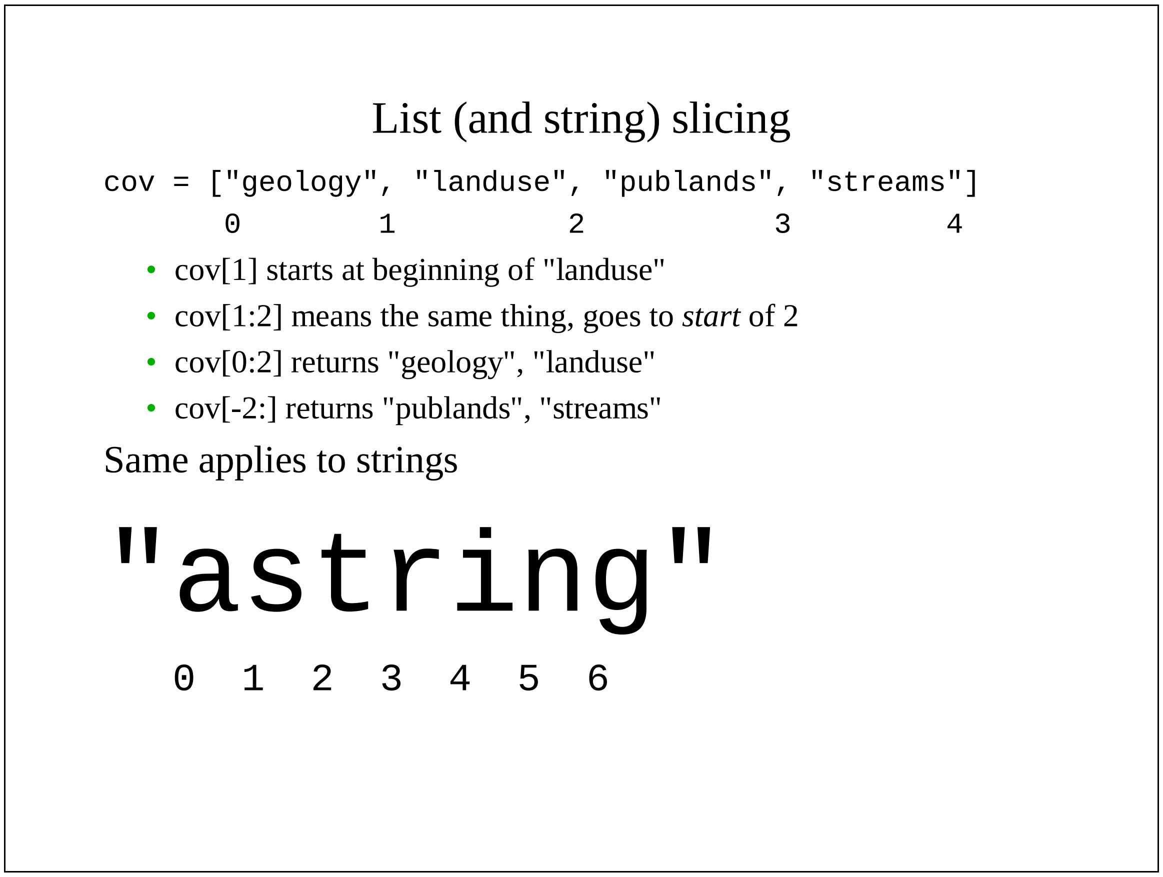
To see how long a list (or a string) is, use the len( ) function

```
>>> cov = ["geology", "landuse", "publands", "cities"]
>>> print len(cov)
4
>>> print len(cov[0])
7
```

Strings are lists of characters. Can you see what the following does?

```
>>> print cov[0][1]
e
```

Slide 12



List (and string) slicing

```
cov = ["geology", "landuse", "publands", "streams"]
       0        1          2            3         4
```

- cov[1] starts at beginning of "landuse"
- cov[1:2] means the same thing, goes to *start* of 2
- cov[0:2] returns "geology", "landuse"
- cov[-2:] returns "publands", "streams"

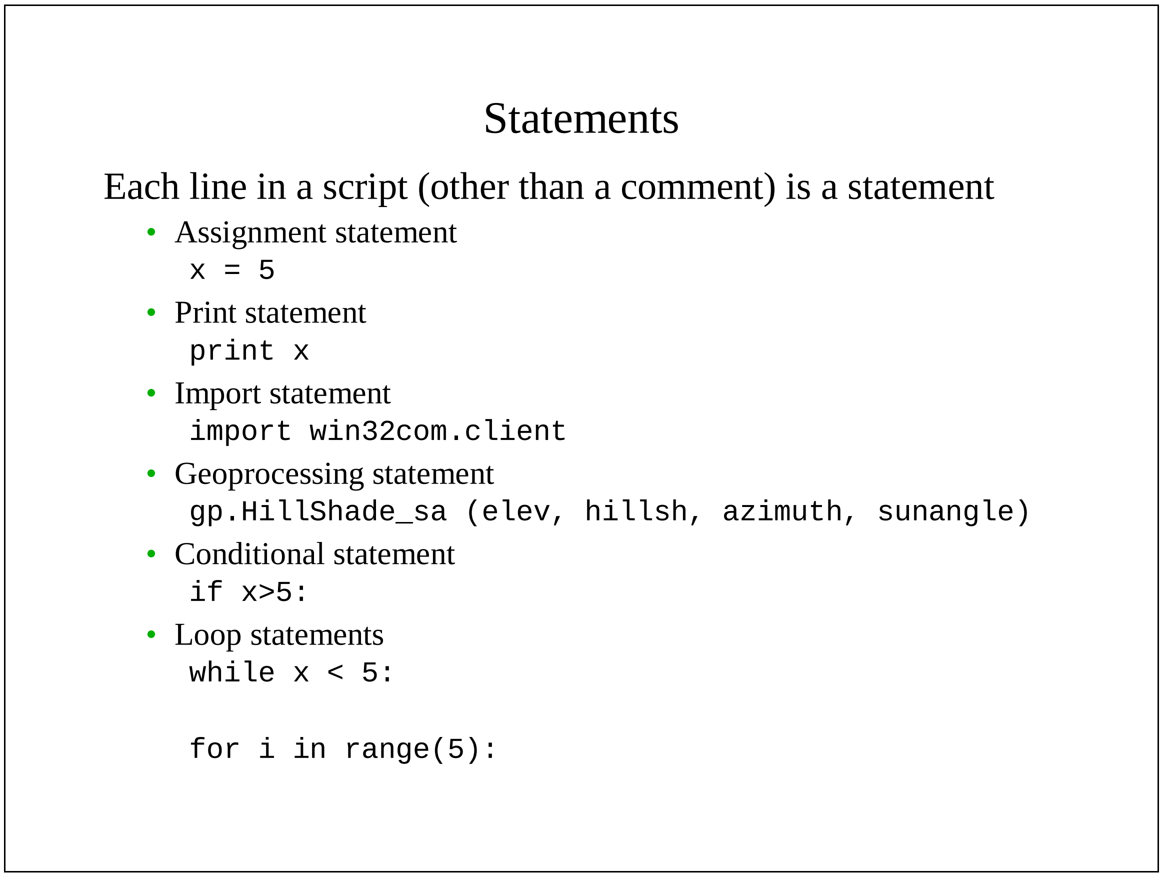
Same applies to strings

```
"astring"
 0 1 2 3 4 5 6
```

See Python textbooks for the many ways you can process strings.   You can also find methods via the Python help system, in the library reference section for strings.

In Python, text strings are almost the same thing as a list of characters.

Slide 13



Statements

Each line in a script (other than a comment) is a statement

- Assignment statement
  ```
  x = 5
  ```
- Print statement
  ```
  print x
  ```
- Import statement
  ```
  import win32com.client
  ```
- Geoprocessing statement
  ```
  gp.HillShade_sa (elev, hillsh, azimuth, sunangle)
  ```
- Conditional statement
  ```
  if x>5:
  ```
- Loop statements
  ```
  while x < 5:

  for i in range(5):
  ```

Note that some statements end in a colon. You would follow them with a series of statements that process as a result of that statement. For example, conditional statements are followed by a series of statements to process if the condition is true. One of the most common coding mistakes I make is forgetting the colon.

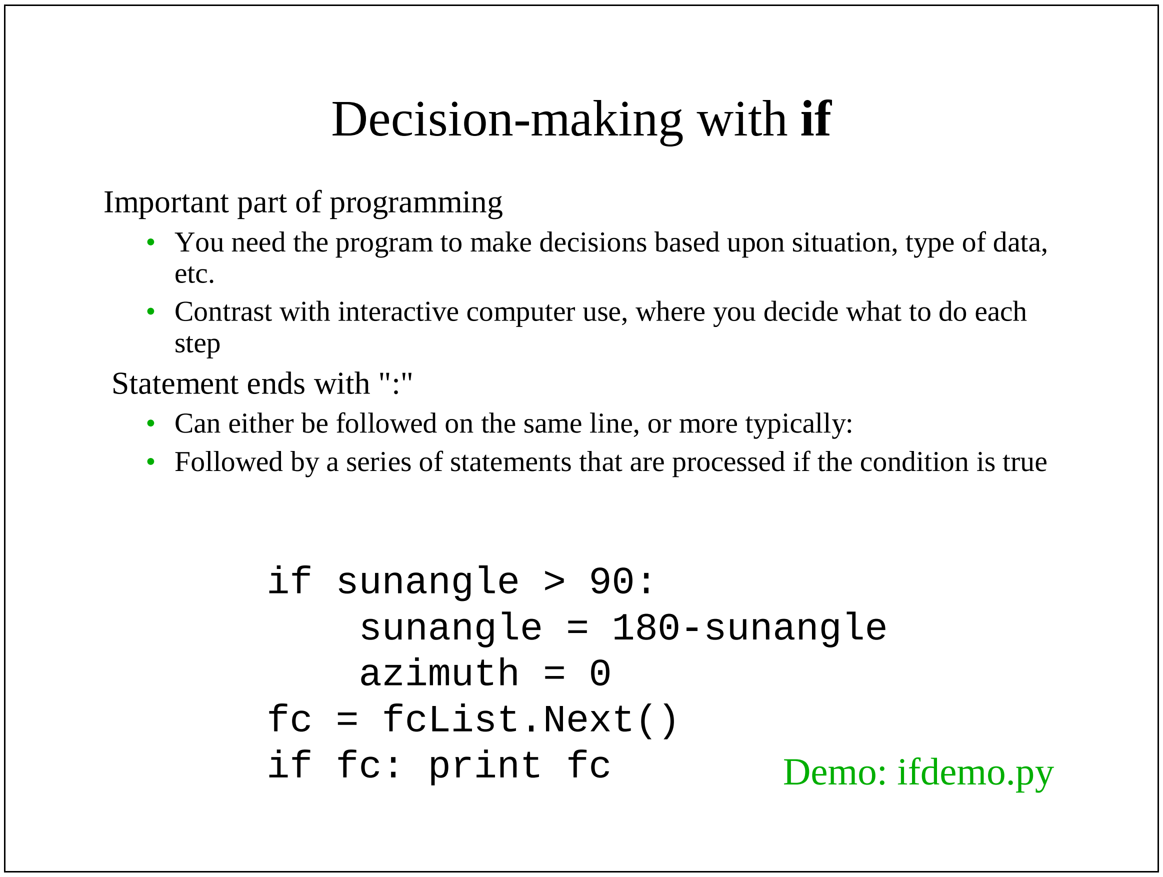
Line Continuation
Multiple ways of doing this. If a list, a comma can separate:

```
cov = ["geology", "landuse", "publands", "flood", "streams",
       "roads"]
```

Or use a single backslash to immediately break to the next line. A downside to using a slash continuation is the next line starts in column 1 on the next line, so it kind of messes up the formatting.

```
fld = ["TYPE-ID", "LU-CODE", "PUB-CODE", "FLOOD-CODE", "ST-CO\
DE", "ROAD-CODE"]
```

Slide 14



Decision-making with **if**

Important part of programming

- You need the program to make decisions based upon situation, type of data, etc.
- Contrast with interactive computer use, where you decide what to do each step

Statement ends with ":"

- Can either be followed on the same line, or more typically:
- Followed by a series of statements that are processed if the condition is true

```
if sunangle > 90:
    sunangle = 180-sunangle
    azimuth = 0
fc = fcList.Next()
if fc: print fc
```

Demo: ifdemo.py

Note the use of the colon with the if statement. The indented lines that follow are processed if the expression sunangle > 90 is true.

Note that you can just follow the colon on the same line with a short statement. If you do this, don't follow the line with an indented line – you'll get a syntax error. This short if statement is very handy for making your code more compact, yet just as readable (I think more readable since you don't have to scroll through as much text if some of the if structures only take up one line.)

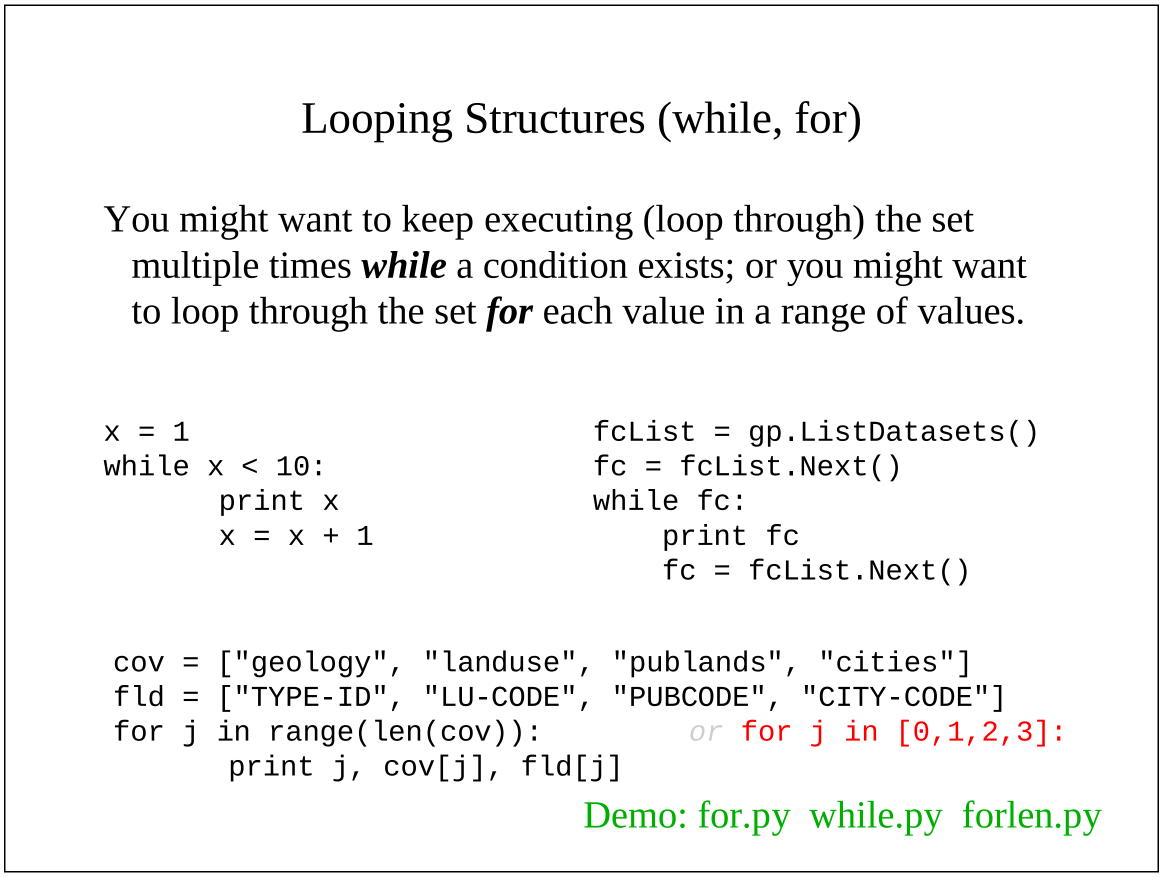
If you want to do something else if the expression is not true, use an else statement like the following:

```
if sunangle > 90:
      sunangle = 180 – sunangle
else:
      print "normal situation"
```

You can also nest another if statement within, with another "if",
Or an "else if" with:

```
if sunangle > 90:
      sunangle = 180 – sunangle
elif sunangle > 66:
      sunangle = 90 - sunangle
```

Slide 15



Looping Structures (while, for)

You might want to keep executing (loop through) the set multiple times ***while*** a condition exists; or you might want to loop through the set ***for*** each value in a range of values.

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

```
fcList = gp.ListDatasets()
fc = fcList.Next()
while fc:
    print fc
    fc = fcList.Next()
```

```
cov = ["geology", "landuse", "publands", "cities"]
fld = ["TYPE-ID", "LU-CODE", "PUBCODE", "CITY-CODE"]
for j in range(len(cov)):        or for j in [0,1,2,3]:
      print j, cov[j], fld[j]
```

Demo: for.py while.py forlen.py

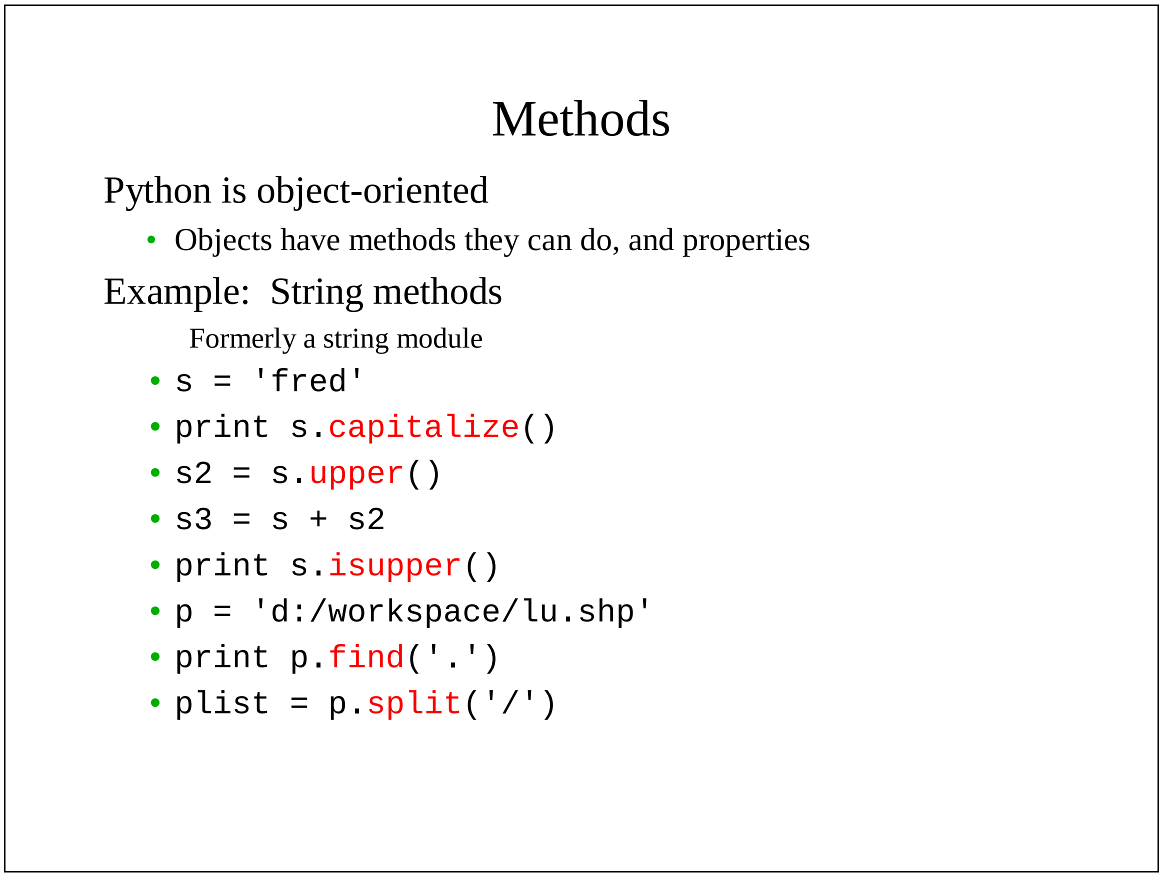
The **`for`** statement in Python is different from that of other languages I've used, and takes a bit of getting used to, but in the long run, it makes a lot of sense for the normal ways we use loops. In BASIC, we might write:

```
for i = 1 to 10
     print i
next i
```

In Python, this would be the following, but with i going from 0 to 9

```
for i in range(10):
    print i
```

Using **`while`** structures is probably more common in geoprocessing, since you often only want to process data while a changing condition remains true. It combines the conditional aspect of an **`if`** structure with looping. Be careful that you don't create an endless loop – one common way is not changing the condition in the loop, such as not using the `fc = fcList.Next()` statement within the **`with`** loop.



Methods

Python is object-oriented

- Objects have methods they can do, and properties

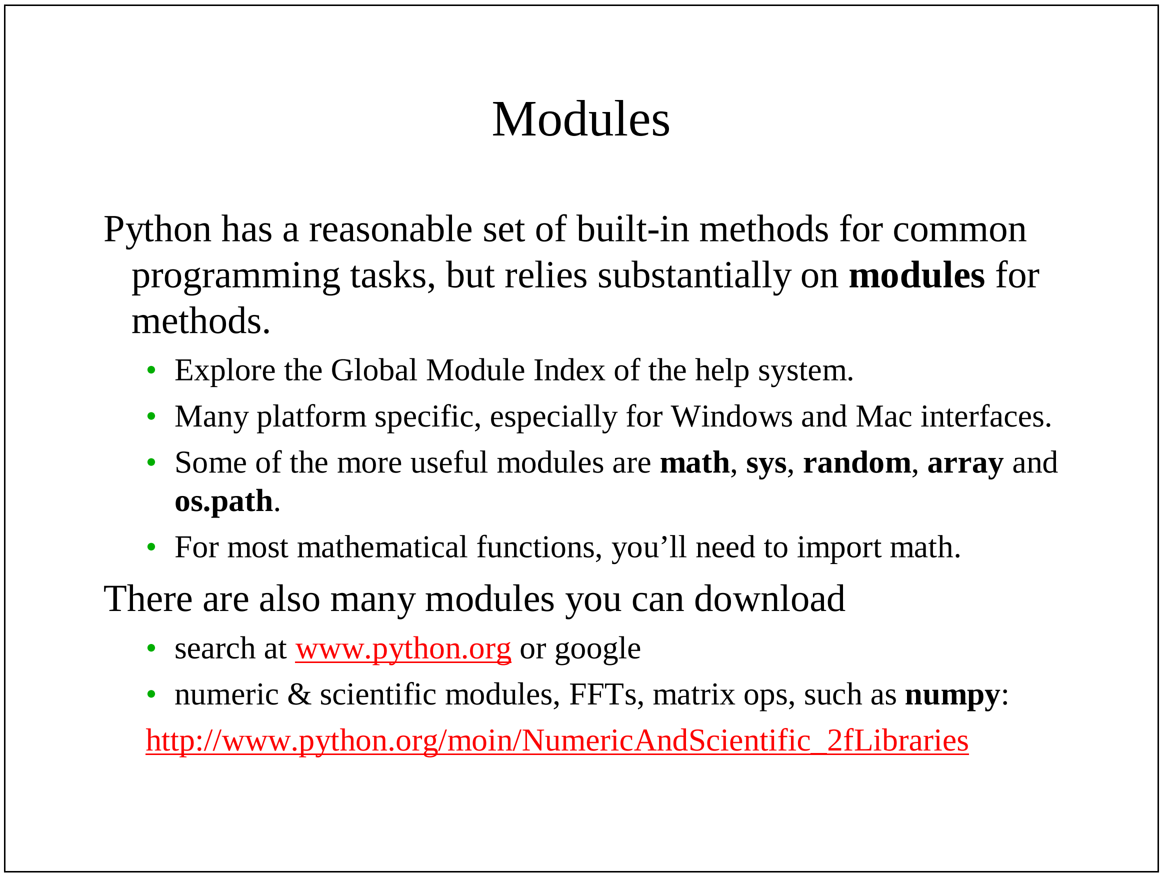
Example: String methods

Formerly a string module

- `s = 'fred'`
- `print s.capitalize()`
- `s2 = s.upper()`
- `s3 = s + s2`
- `print s.isupper()`
- `p = 'd:/workspace/lu.shp'`
- `print p.find('.')`
- `plist = p.split('/')`

With built-ins you can take a variable and apply a method to it using the variable.method() construct shown here, where the method is applied to the variable. For example, **`s.capitalize()`** returns a capitalized s. What the advantage is of doing this over using a function-like syntax **`capitalize(s)`** I don't know, but you have to write it as a method, using the method name following the object and dot.

Slide 17



Modules

Python has a reasonable set of built-in methods for common programming tasks, but relies substantially on **modules** for methods.

- Explore the Global Module Index of the help system.
- Many platform specific, especially for Windows and Mac interfaces.
- Some of the more useful modules are **math**, **sys**, **random**, **array** and **os.path**.
- For most mathematical functions, you'll need to import math.

There are also many modules you can download

- search at www.python.org or google
- numeric & scientific modules, FFTs, matrix ops, such as **numpy**:

http://www.python.org/moin/NumericAndScientific_2fLibraries

math
Many common mathematical functions are accessed via the **math** module: Trigonometric functions, Logarithms, etc.
To use complex numbers, use **cmath** which has similar functions, but allows for complex number results.
random
`random.random()` returns pseudorandom number between 0 and 1
Other variations, e.g. random.gauss

```
import random
mu = 50
s = 10
print random.gauss(mu, s)
```

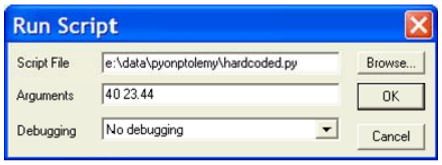
numpy
One of several available modules for matrix operations like matlab

Input

Input from run dialog

Example uses the sys module, sys.argv[1]

```
import sys
lat = float(sys.argv[1])
decl = float(sys.argv[2])
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
print "Noon sun angle = " + str(sunangle)
print "Azimuth = " + str(azimuth)
```



Script File: e:\data\pyonptolemy\hardcoded.py

Arguments: 40 23.44

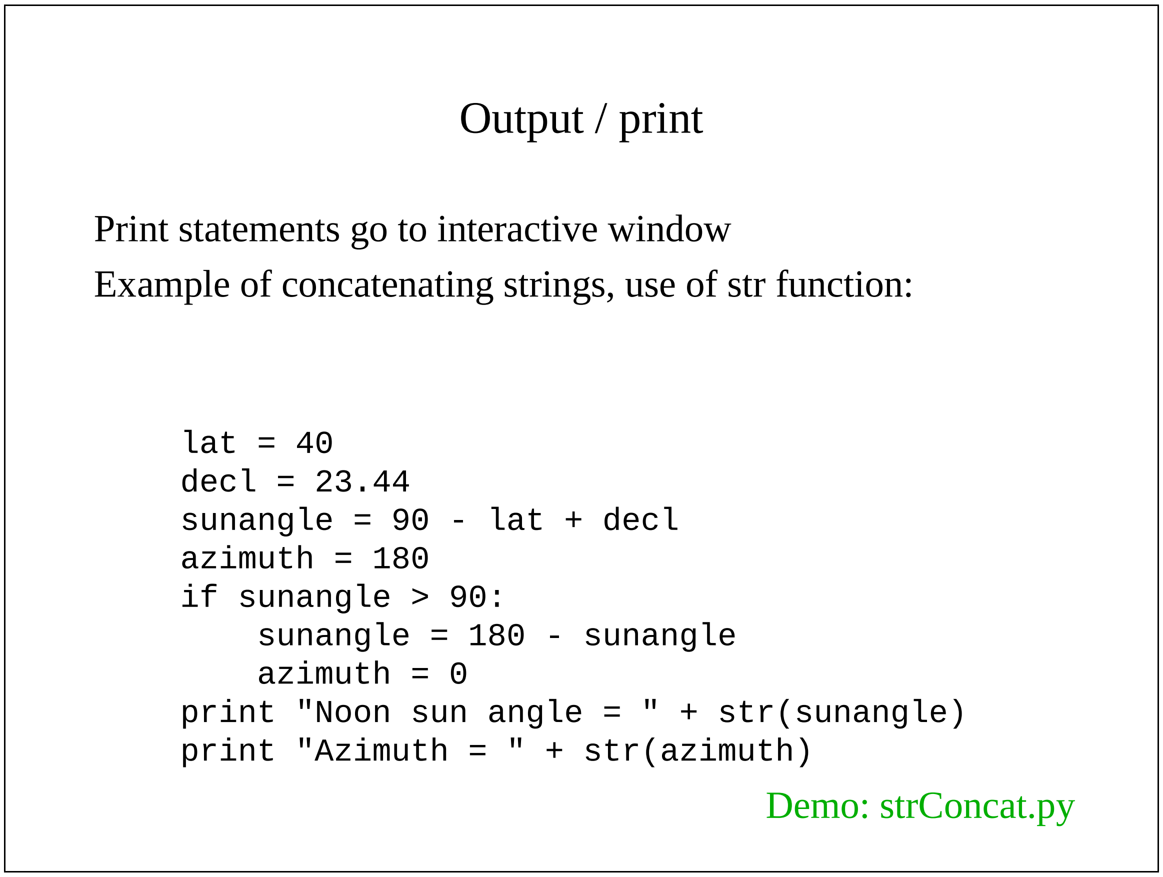
Debugging: No debugging

Useful scripts don't have "hard-coded" values but instead take input from the user. One way, shown here, is with the argv method of the sys module. We’ll also look at the GetParameterAsText geoprocessing function which does the same thing as sys.argv, but can’t be used from the IDE.

Note that sys.argv’s first index value is 1. Zero is used; sys.argv[0] refers to the script itself.

Note that in PythonWin's Run Script dialog, the arguments are provided as a single text string, with arguments separated by spaces.

Slide 19



Output / print

Print statements go to interactive window

Example of concatenating strings, use of str function:

```
lat = 40
decl = 23.44
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
print "Noon sun angle = " + str(sunangle)
print "Azimuth = " + str(azimuth)
```

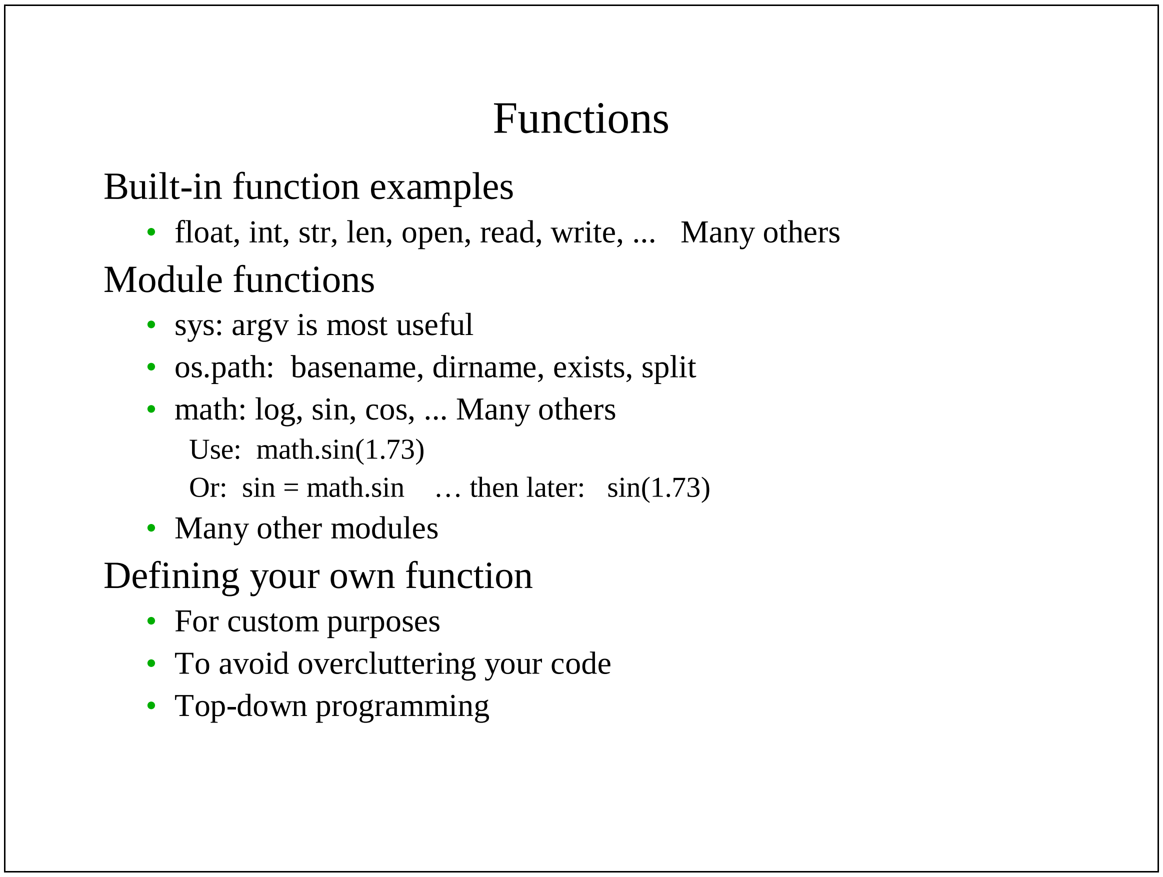
Demo: strConcat.py

While you can print out numerical variables like:

```
x = 5
print x
```

… if you want to combine these with strings and use concatenation to do, both parts being concatenated need to be strings, so you'd need to do something like the following:

```
x = 5
print "My number is: " + str(x)
```



Functions

Built-in function examples

- float, int, str, len, open, read, write, ...   Many others

Module functions

- sys: argv is most useful
- os.path: basename, dirname, exists, split
- math: log, sin, cos, ... Many others
  Use: math.sin(1.73)
  Or: sin = math.sin   … then later: sin(1.73)
- Many other modules

Defining your own function

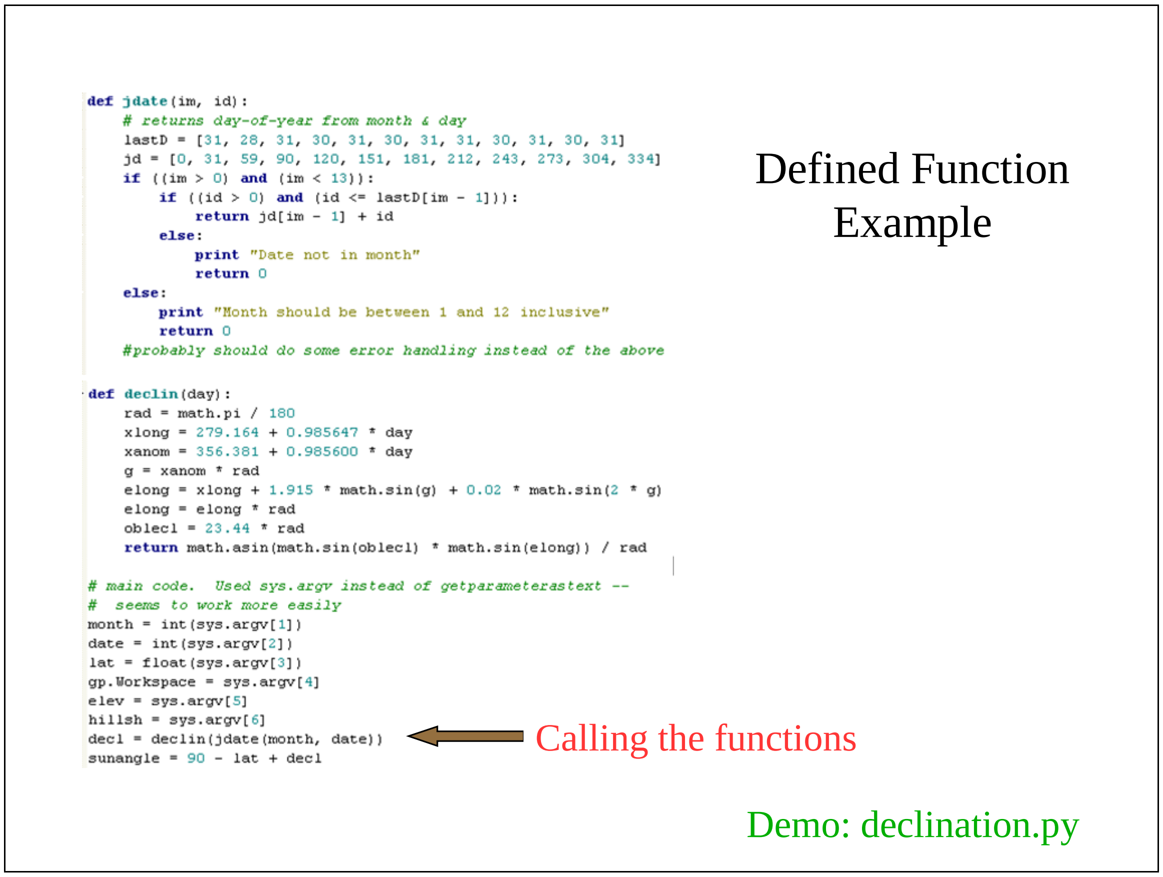
- For custom purposes
- To avoid overcluttering your code
- Top-down programming

Built-in functions
The len function is very handy for processing strings and lists. Some other handy functions are abs, int, float, str, max, min, range, open, read, write, slice. To see a list of built-in functions, enter `dir(__builtins__)` in the interactive window; the functions are at the end of what you'll see – all the ones starting with a lower-case letter (starting with 'abs').

There are many modules. See the Global Module Index in the Python Manuals section of PythonWin help. Some of the more useful ones are:

math : provides lots of things we'll need that are not included as built-in functions
random : for random numbers
string : used to be essential, but most are now built-in.
sys : various things, but we'll primarily use sys.argv.
os.path : for manipulating file names and folders, separating workspaces.

Slide 21



Defined Function Example

```
def jdate(im, id):
    # returns day-of-year from month & day
    lastD = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    jd = [0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334]
    if ((im > 0) and (im < 13)):
        if ((id > 0) and (id <= lastD[im - 1])):
            return jd[im - 1] + id
        else:
            print "Date not in month"
            return 0
    else:
        print "Month should be between 1 and 12 inclusive"
        return 0
    #probably should do some error handling instead of the above

def declin(day):
    rad = math.pi / 180
    xlong = 279.164 + 0.985647 * day
    xanom = 356.381 + 0.985600 * day
    g = xanom * rad
    elong = xlong + 1.915 * math.sin(g) + 0.02 * math.sin(2 * g)
    elong = elong * rad
    oblecl = 23.44 * rad
    return math.asin(math.sin(oblecl) * math.sin(elong)) / rad

# main code.  Used sys.argv instead of getparameterastext --
#  seems to work more easily
month = int(sys.argv[1])
date = int(sys.argv[2])
lat = float(sys.argv[3])
gp.Workspace = sys.argv[4]
elev = sys.argv[5]
hillsh = sys.argv[6]
decl = declin(jdate(month, date))
sunangle = 90 - lat + decl
```

Calling the functions

Demo: declination.py

Defining your own functions helps to keep your script readable. All part of “top-down” programming, where we define the general steps, then define the details with functions we define.

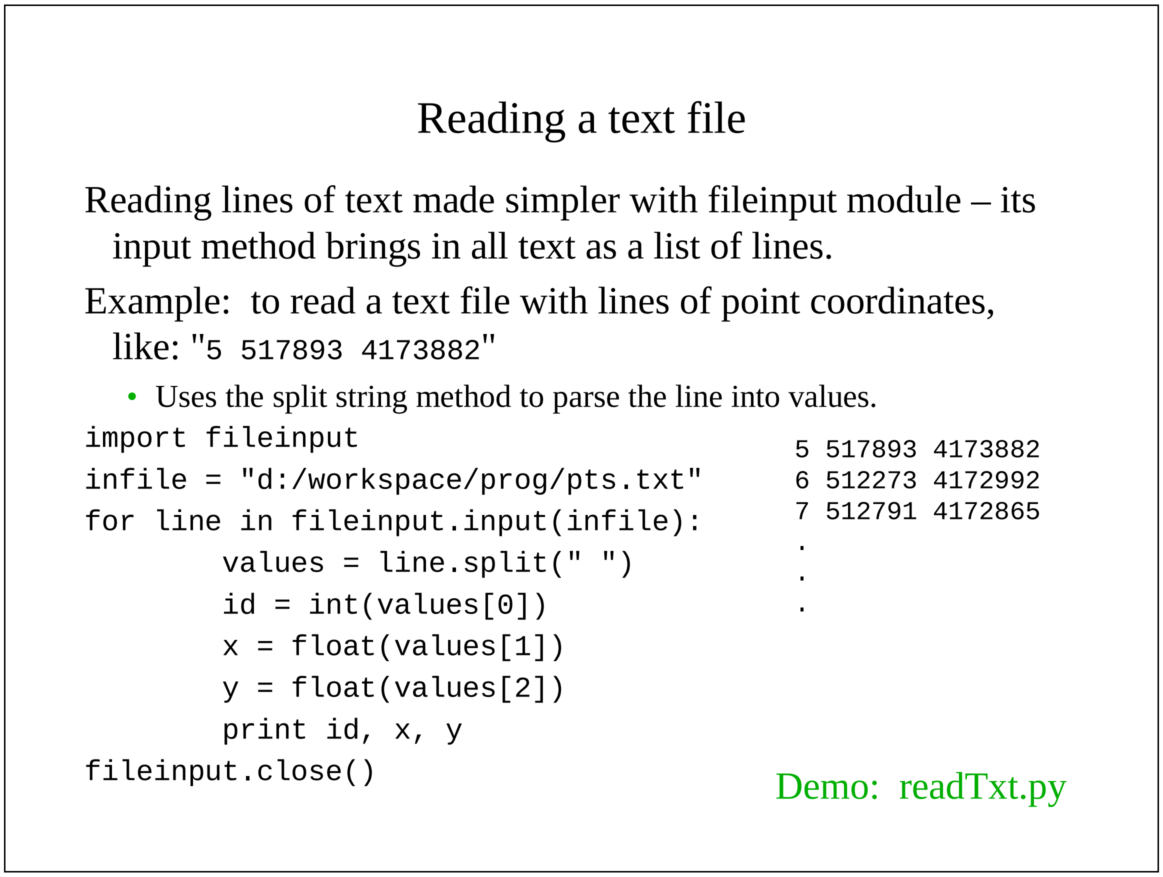
Note the use of a colon in front of the function definition. Note the use of **return** to establish what value to return. For instance, **jdate** accepts two integer inputs (month and date), and returns an integer number defining the day of the year.

The second function, **declin**, returns the solar declination in degrees. Note the conversion from degrees to radians and vice-versa.

In the main program, the following statement calls the **jdate** function using the month and date, then uses that result to derive the solar declination by calling **declin**:

```
decl = declin(jdate(month, date))
```

Slide 22



Reading a text file

Reading lines of text made simpler with fileinput module – its input method brings in all text as a list of lines.

Example: to read a text file with lines of point coordinates, like: "5 517893 4173882"

- Uses the split string method to parse the line into values.

```
import fileinput
infile = "d:/workspace/prog/pts.txt"
for line in fileinput.input(infile):
        values = line.split(" ")
        id = int(values[0])
        x = float(values[1])
        y = float(values[2])
        print id, x, y
fileinput.close()
```

```
5 512893 4173882
6 512273 4172992
7 512791 4172865
.
.
.
```

Demo: readTxt.py

There are other methods of reading text files. Refer to a Python book. Python Cookbook has several.

The most common type of text file is one with lines of text, ending in an end-line character (Python uses \n for writing these). The above script brings in lines of text as a list, then processes the list.

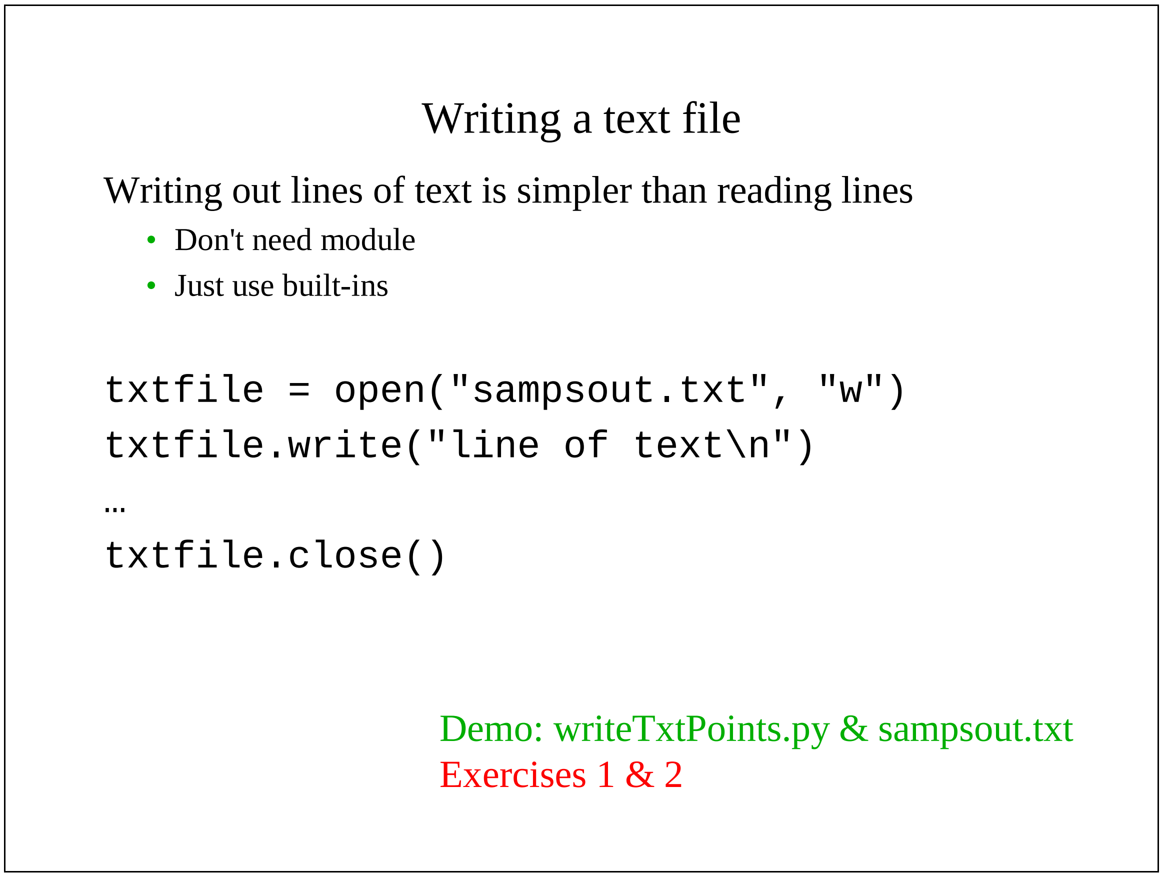
With GIS data work, we're often dealing with space-, comma-, or tab-delimited files. The above example is space-delimited. For comma delimited or tab-delimited, just replace " " with "," or "\t".

Ignoring lines of text, using the Python comment character (#):

```
if line[0] <> "#":
     ...
```

The above is handy when you want to add some metadata at the top of your text file, maybe identifying field headings, or the spatial reference for your data. Note that line[0] just looks at the first character on the line. A similar idea, also looking at the first character on the line, might be to ignore lines that don't have a number starting in the first character:

```
if line[0] in ['0','1','2','3','4','5','6','7','8','9','.','-']:
```



Writing a text file

Writing out lines of text is simpler than reading lines

- Don't need module
- Just use built-ins

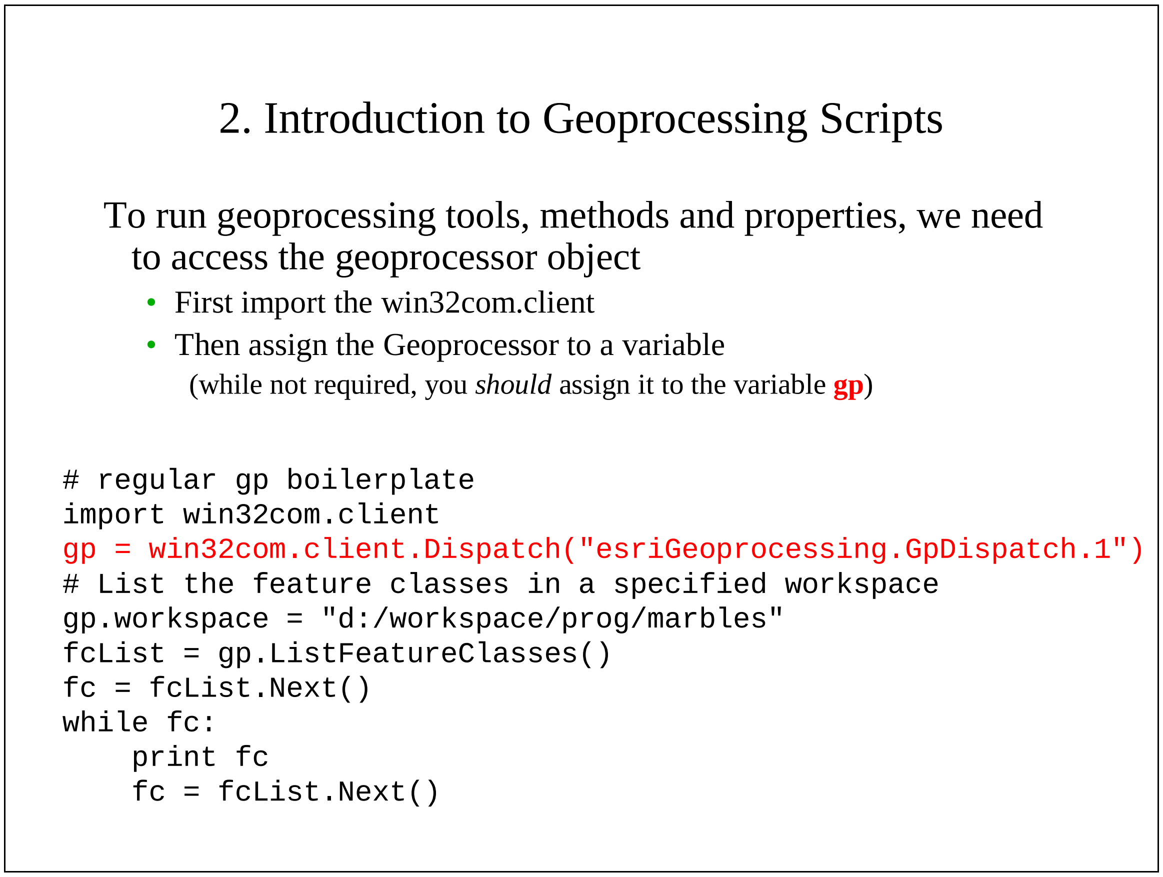
```
txtfile = open("sampsout.txt", "w")
txtfile.write("line of text\n")
…
txtfile.close()
```

Demo: writeTxtPoints.py & sampsout.txt
Exercises 1 & 2

The following code reads a point shapefile, and writes it out to a textfile as lines of id x y:

```
import win32com.client, os
gp = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")
gp.workspace = "d:/workspace/prog/marbles"
txtfile = "sampsout.txt"
fpath = gp.workspace + "/" + txtfile
if gp.Exists(fpath): os.remove(fpath)
txtfile = open(fpath, "w")
try:
    rows = gp.SearchCursor("samples.shp")
    row = rows.Next()
    while row:
        pnt = row.shape.GetPart()
        id = int(row.SAMPLES_ID)
        txtfile.write(str(id) + " " + str(pnt.x) + " " + str(pnt.y)+"\n")
        row = rows.Next()
    txtfile.close()
except:
    print gp.getmessages()
    txtfile.close()
```

Slide 24



2. Introduction to Geoprocessing Scripts

To run geoprocessing tools, methods and properties, we need to access the geoprocessor object

- First import the win32com.client
- Then assign the Geoprocessor to a variable

  (while not required, you *should* assign it to the variable **gp**)

```
# regular gp boilerplate
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")
# List the feature classes in a specified workspace
gp.workspace = "d:/workspace/prog/marbles"
fcList = gp.ListFeatureClasses()
fc = fcList.Next()
while fc:
    print fc
    fc = fcList.Next()
```

At the start of every Python programming in which you want to use geoprocessing tools, you'll need to add the regular gp boilerplate:

```
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")
```

You can just copy this from another program, so you don't have to remember it.

Variations on this would import additional modules like sys or math, so the first line might look like:

```
import win32com.client, sys, math
```

Note that the dispatch is actually *assigned* to the gp variable. You could name it anything you want, but for consistency, and so you can share code with others, we standardize on gp (some people use GP). This variable is your link to the geoprocessing object, providing access to all of the geoprocessing tools and settings. You *must* use the name you assigned, and since variables are case-sensitive, gp is not the same as GP. Since some sample scripts use gp and some use GP, a common mistake is to mix it up. You can use both by making them equivalent:

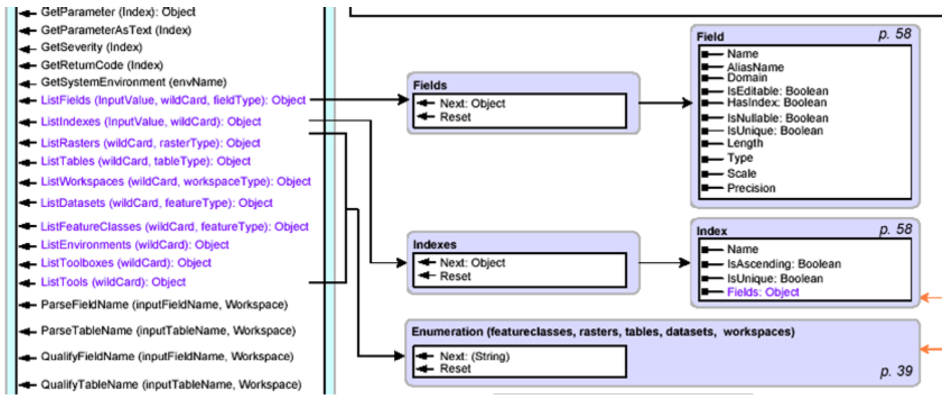
```
GP = gp = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")
```

Slide 25

Enumerations

ListFields, ListIndexes, ListRasters, ListTables, ListWorkspaces, ListDatasets, ListFeatureClasses, ListEnvironments, ListToolboxes, ListTools

Not the same thing as Python lists, or Python enumerations


GetParameter (Index): Object
GetParameterAsText (Index)
GetSeverity (Index)
GetReturnCode (Index)
GetSystemEnvironment (envName)
ListFields (InputValue, wildCard, fieldType): Object
ListIndexes (InputValue, wildCard): Object
ListRasters (wildCard, rasterType): Object
ListTables (wildCard, tableType): Object
ListWorkspaces (wildCard, workspaceType): Object
ListDatasets (wildCard, featureType): Object
ListFeatureClasses (wildCard, featureType): Object
ListEnvironments (wildCard): Object
ListToolboxes (wildCard): Object
ListTools (wildCard): Object
ParseFieldName (inputFieldName, Workspace)
ParseTableName (inputTableName, Workspace)
QualifyFieldName (inputFieldName, Workspace)
QualifyTableName (inputTableName, Workspace)
Fields
Next: Object
Reset
Field
p. 58
Name
AliasName
Domain
IsEditable: Boolean
HasIndex: Boolean
IsNullable: Boolean
IsUnique: Boolean
Length
Type
Scale
Precision
Indexes
Next: Object
Reset
Index
p. 58
Name
IsAscending: Boolean
IsUnique: Boolean
Fields: Object
Enumeration (featureclasses, rasters, tables, datasets, workspaces)
Next: (String)
Reset
p. 39

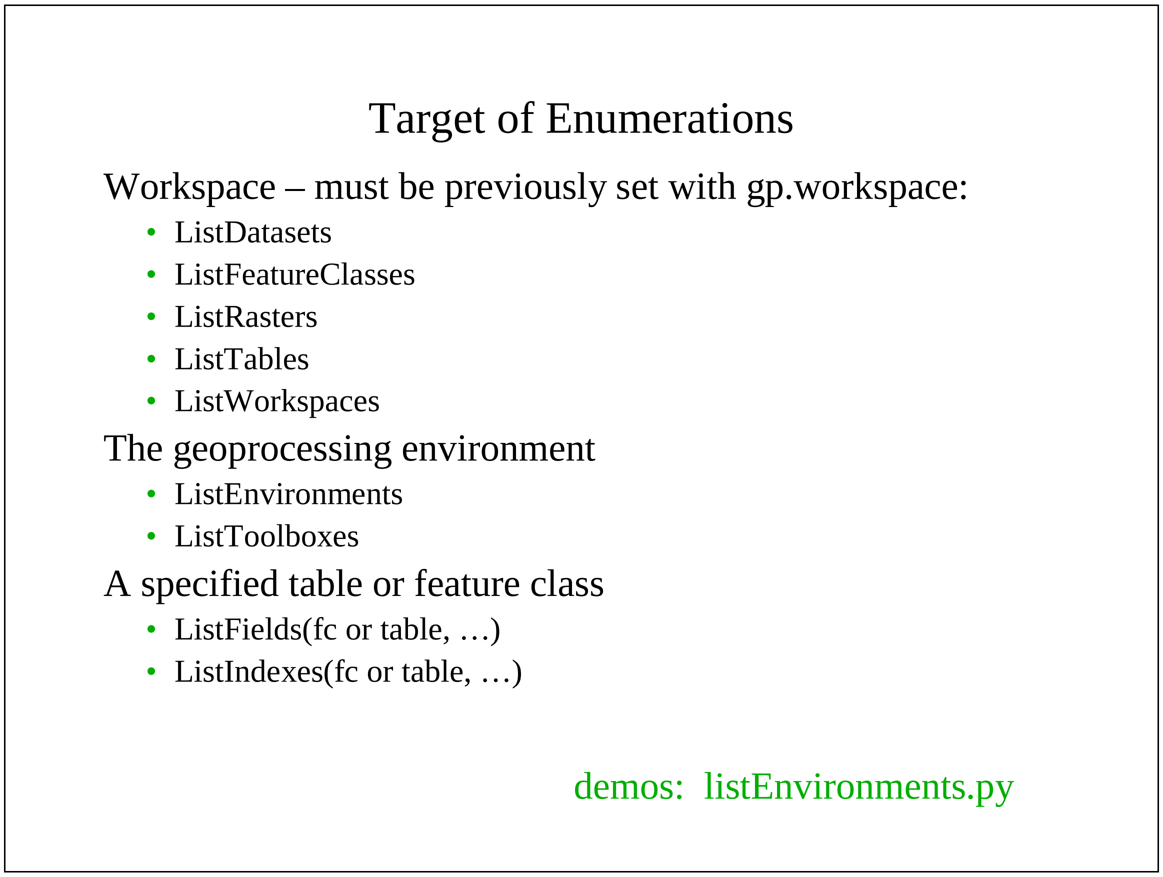
We start with enumerations because so much of what we want to do in geoprocessing requires them. Why? Because the advantage of scripting can best be seen when working with multiple datasets.

Enumerations created with the various "List…" methods of the geoprocessor are different from Python lists – for one thing, we can't see how long they are, so we use a while loop and the `Next()` method to continue through them until you get nothing after the last one. The `Reset()` method restarts at the first. These enumerations are used in many places in geoprocessing scripting, such as looping through all of the featureclasses in a workspace:

```
fcList = gp.ListFeatureClasses()
fc = fcList.Next()
while fc:
    print fc
    fc = fcList.Next()
```

With Python 2.3 there is a data structure called an enumeration. I haven't investigated to see if it is at all the same kind of thing as these geoprocessing enumerations, but I'm assuming they are different.

Slide 26



Target of Enumerations

Workspace – must be previously set with gp.workspace:

- ListDatasets
- ListFeatureClasses
- ListRasters
- ListTables
- ListWorkspaces

The geoprocessing environment

- ListEnvironments
- ListToolboxes

A specified table or feature class

- ListFields(fc or table, …)
- ListIndexes(fc or table, …)

demos: listEnvironments.py

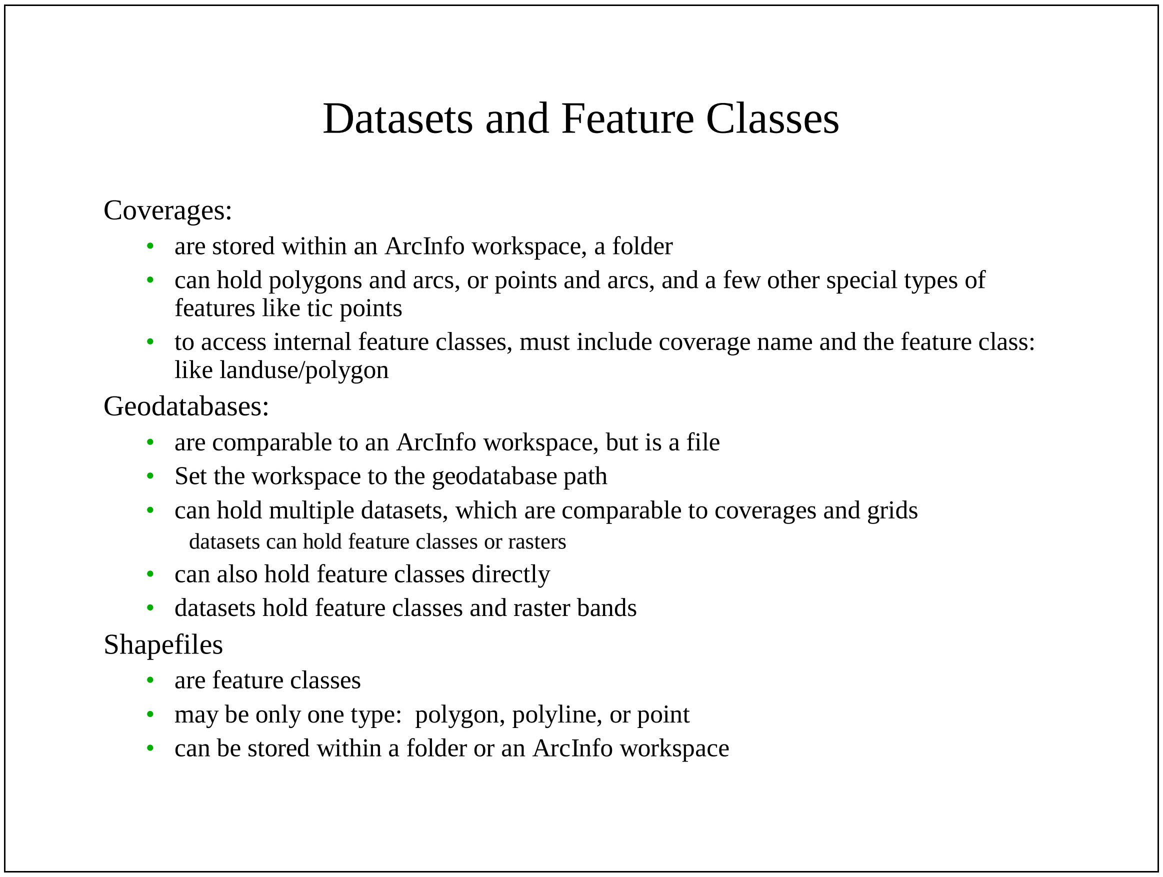
Set the workspace first.
A common mistake I make is to try to specify the workspace when using **listDatasets**, **listFeatureClasses**, etc., by trying something like
gp.listFeatureClasses("d:/workspace/prog/hmbarea").
Instead, your code must have the workspace set first:

```
gp.workspace = "d:/workspace/prog/hmbarea"
fcList = gp.listFeatureClasses()
```

Then the first parameter is a wildcard.

In contrast, with **listFields** and **listIndexes**, the first parameter is the target feature class or table, then the second parameter is the wildcard.

Slide 27



Datasets and Feature Classes

Coverages:

- are stored within an ArcInfo workspace, a folder
- can hold polygons and arcs, or points and arcs, and a few other special types of features like tic points
- to access internal feature classes, must include coverage name and the feature class: like landuse/polygon

Geodatabases:

- are comparable to an ArcInfo workspace, but is a file
- Set the workspace to the geodatabase path
- can hold multiple datasets, which are comparable to coverages and grids
  datasets can hold feature classes or rasters
- can also hold feature classes directly
- datasets hold feature classes and raster bands

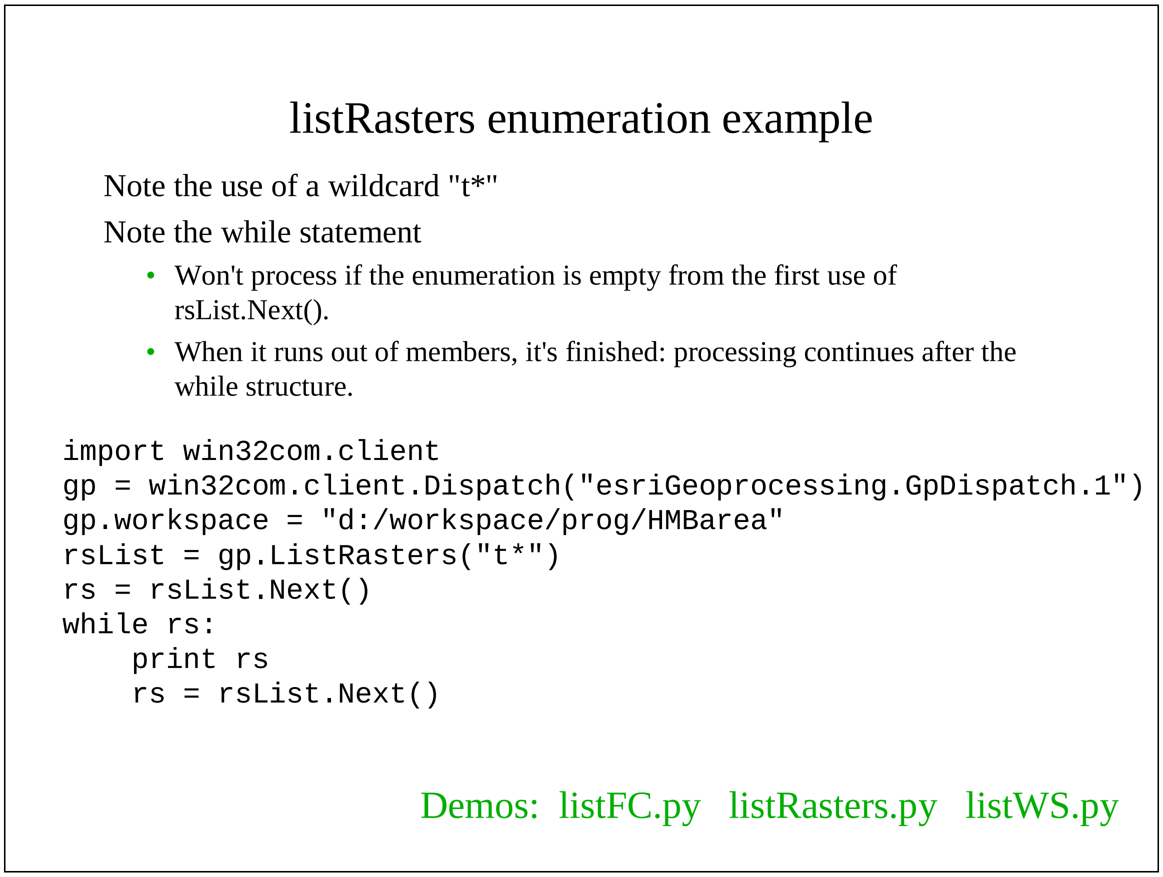
Shapefiles

- are feature classes
- may be only one type: polygon, polyline, or point
- can be stored within a folder or an ArcInfo workspace

When accessing GIS datasets, you need to know the differences in how they are accessed. There's nothing like programming to force you to learn the structure of computer datasets.

For instance, to get an enumeration list of **feature classes** *in a*:
(1) **coverage**, you set gp.workspace to the coverage path.
(2) **feature dataset** in a geodatabase, you set gp.workspace to the dataset path. You won't find this in windows explorer, since it's within a geodatabase, which is a file. From the perspective of ArcGIS, a feature dataset is comparable to a coverage, and in fact a coverage is called a feature dataset.
(3) **geodatabase**, you set the gp.workspace to the geodatabase path. Geodatabases allow you to either store feature classes directly or within feature datasets.
(4) **folder** or **ArcInfo workspace**, you set gp.workspace to the folder name. These feature classes would be shapefiles.

Slide 28



listRasters enumeration example

Note the use of a wildcard "t*"

Note the while statement

- Won't process if the enumeration is empty from the first use of rsList.Next().
- When it runs out of members, it's finished: processing continues after the while structure.

```
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")
gp.workspace = "d:/workspace/prog/HMBarea"
rsList = gp.ListRasters("t*")
rs = rsList.Next()
while rs:
    print rs
    rs = rsList.Next()
```

Demos: listFC.py listRasters.py listWS.py

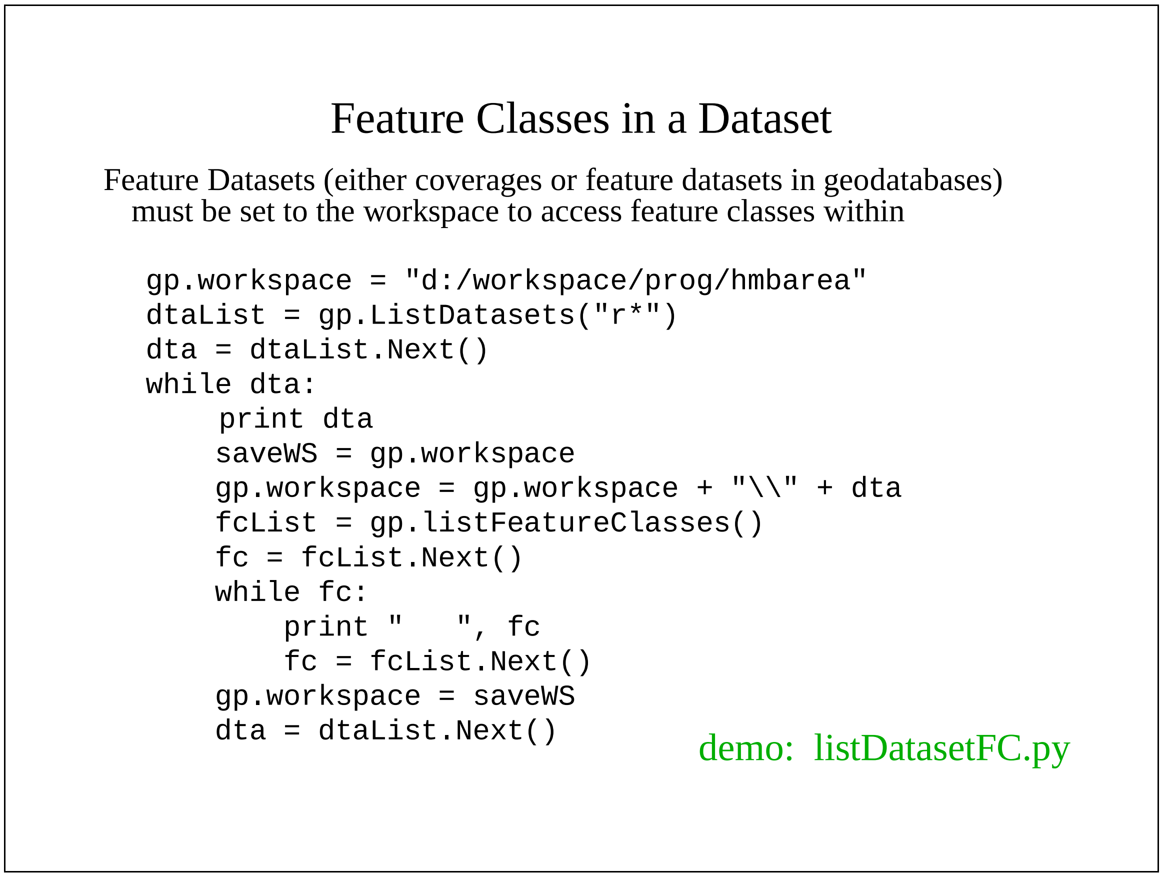
All we've done here is just print the names of the rasters to the screen. But you could also process each of those rasters with a tool, in that same while loop.

Another example lists all of the workspaces in a folder. Note that you set that folder (which might contain multiple workspaces) to be used a workspace:

```
gp.Workspace = "d:/workspace/prog"
wsList = gp.ListWorkspaces()
ws = wsList.Next()
while ws:
    print ws
    ws = wsList.Next()
```

Important: don't forget the .Next() method. When looping through *any* enumeration with a while structure, it's important to go to the next member within the while loop, as shown above. If you forget this, you create an endless loop, and you'll have a hard time getting PythonWin to get out of it. You can try to right-click PythonWin in the task bar to try to break out of running code, but it usually takes several tries, and sometimes you end up needing to use the three-finger salute (Ctrl-Alt-Delete) to stop the PythonWin task.

Slide 29



Feature Classes in a Dataset

Feature Datasets (either coverages or feature datasets in geodatabases) must be set to the workspace to access feature classes within

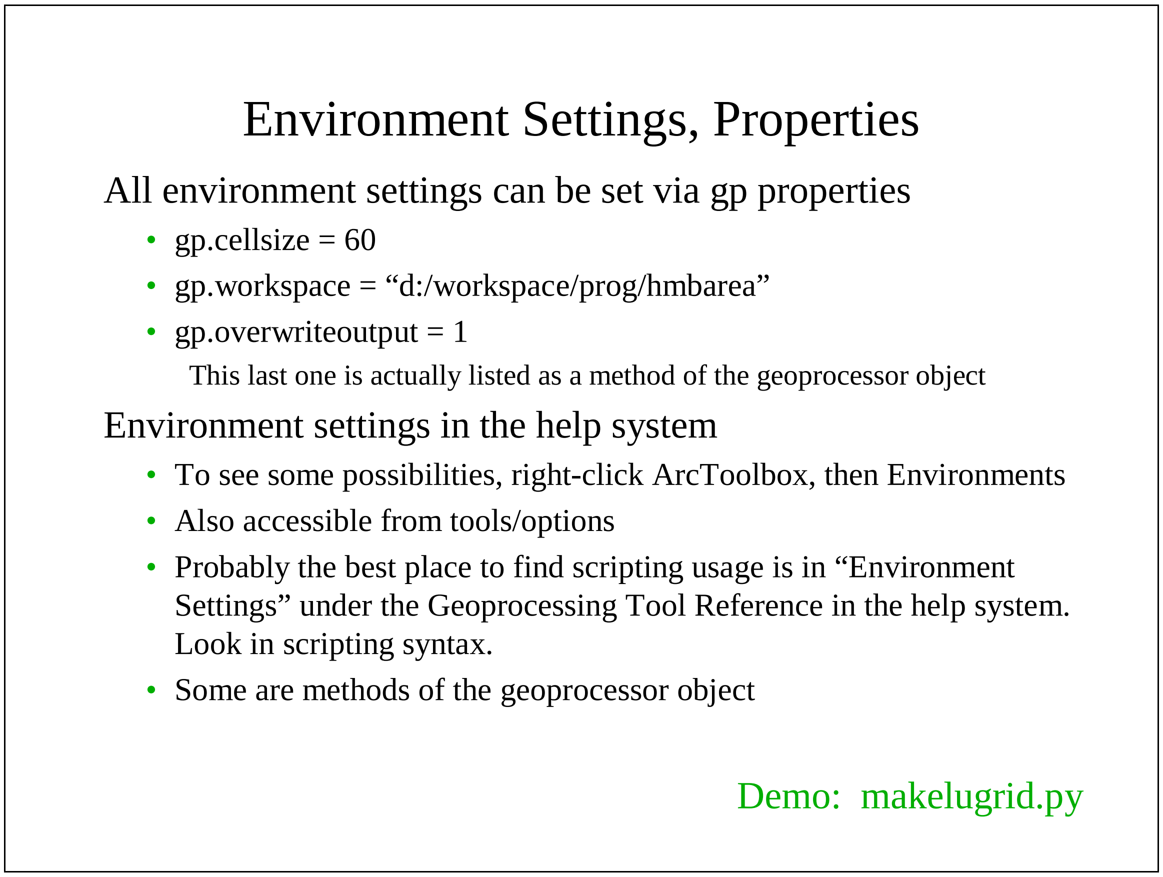
```
gp.workspace = "d:/workspace/prog/hmbarea"
dtaList = gp.ListDatasets("r*")
dta = dtaList.Next()
while dta:
    print dta
    saveWS = gp.workspace
    gp.workspace = gp.workspace + "\\" + dta
    fcList = gp.listFeatureClasses()
    fc = fcList.Next()
    while fc:
        print "   ", fc
        fc = fcList.Next()
    gp.workspace = saveWS
    dta = dtaList.Next()
```

demo: listDatasetFC.py

Since ListDatasets will include not only coverages but also raster datasets in the enumeration, this code will process all of the rasters as well. (That's why I used the "r*" wildcard – otherwise it takes too long.) All we are doing is printing the name of the dataset in that case, so there's no harm done. And fcList will just end up empty if there are no feature classes in the datasets, as is the case with raster datasets, so "while fc:" won't even run since the fcList.Next() turns up empty. But if we only want to process coverages, we might want to first detect that the dataset represents a coverage before we try to do something coverage-like to it.

This script could be improved by describing properties of the dataset before we do something with it. We'll get into that later, in the Describe method.

Slide 30



Environment Settings, Properties

All environment settings can be set via gp properties

- gp.cellsize = 60
- gp.workspace = “d:/workspace/prog/hmbarea”
- gp.overwriteoutput = 1

  This last one is actually listed as a method of the geoprocessor object

Environment settings in the help system

- To see some possibilities, right-click ArcToolbox, then Environments
- Also accessible from tools/options
- Probably the best place to find scripting usage is in “Environment Settings” under the Geoprocessing Tool Reference in the help system. Look in scripting syntax.
- Some are methods of the geoprocessor object

Demo: makelugrid.py

Note that you can use properties and environment settings, as well as setting them. We commonly use the workspace setting:

```
gp.workspace = "d:/workspace/prog"
s = gp.workspace + "/pendata"
```

Help system areas to find these:

Geoprocessing Tool Reference: Environment Settings
workspace, scratchworkspace, extent, cellsize, mask, etc.

Writing Geoprocessing Scripts: Scripting Object: Properties and Methods.
overwriteoutput, toolbox, messagecount, maxseverity, productinfo, many others

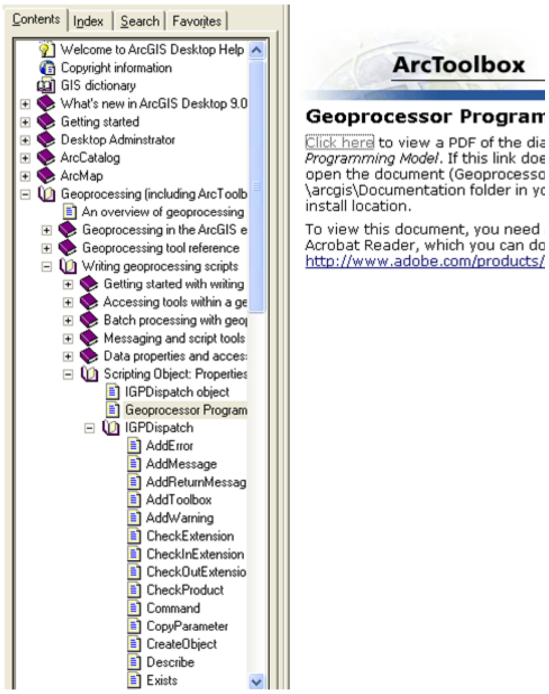
Slide 31

3. Help / Documentation

Help system
- Tool help
- IGPDispatch

Geoprocessor Programming Model

ArcToolbox
- Right-click tool, click help



Welcome to ArcGIS Desktop Help
Copyright information
GIS dictionary
What's new in ArcGIS Desktop 9.0
Getting started
Desktop Adminstrator
ArcCatalog
ArcMap
Geoprocessing (including ArcToolb
An overview of geoprocessing
Geoprocessing in the ArcGIS e
Geoprocessing tool reference
Writing geoprocessing scripts
Getting started with writing
Accessing tools within a ge
Batch processing with geo
Messaging and script tools
Data properties and acces
Scripting Object: Properties
IGPDispatch object
Geoprocessor Program
IGPDispatch
AddError
AddMessage
AddReturnMessag
AddToolbox
AddWarning
CheckExtension
CheckInExtension
CheckOutExtensio
CheckProduct
Command
CopyParameter
CreateObject
Describe
Exists

Geoprocessor Program

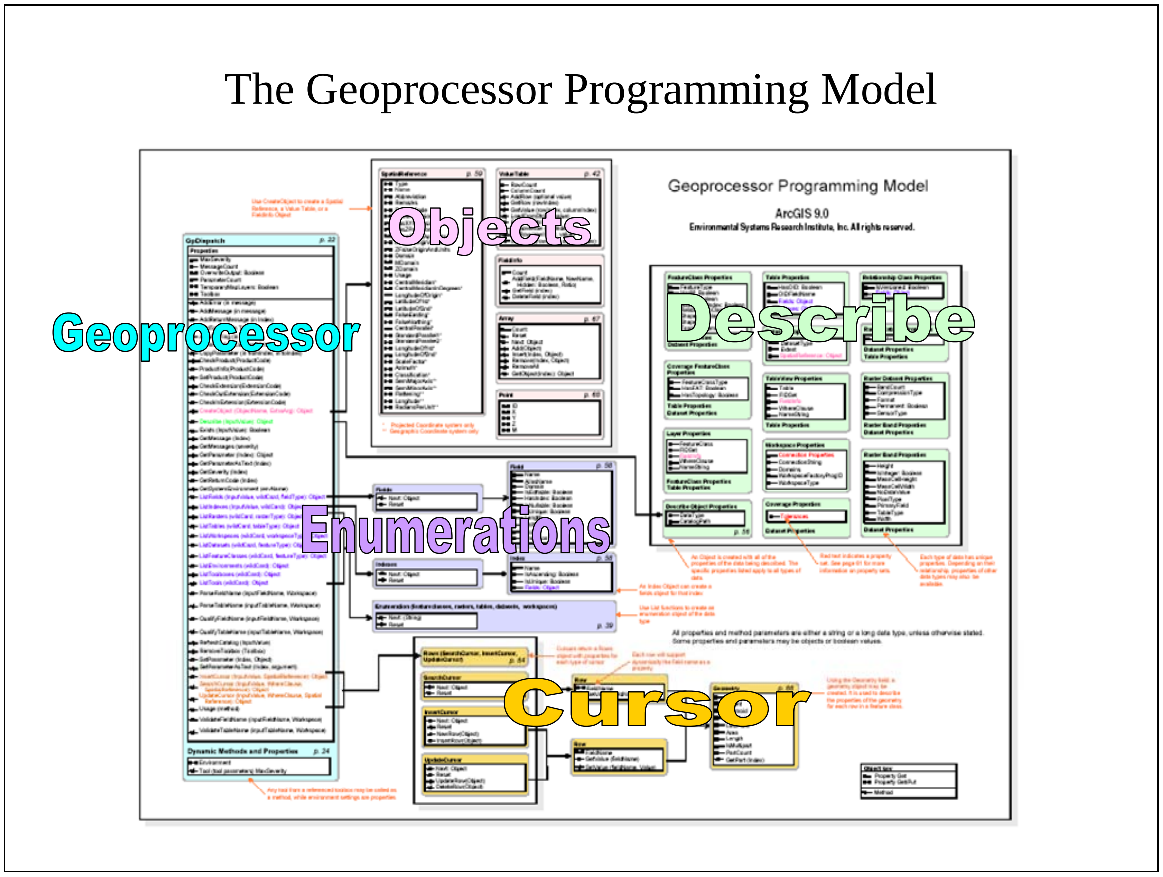
Click here to view a PDF of the dia *Programming Model*. If this link doe open the document (Geoprocesso \arcgis\Documentation folder in yo install location.

To view this document, you need Acrobat Reader, which you can do http://www.adobe.com/products/

There are multiple ways of finding help on the use of geoprocessing tools, and we'll also need help on using the various methods and properties of the geoprocessor. For tools, we can access these from ArcToolbox, or while using the tool, or we can look them up in the Geoprocessing tool reference in the help system.

The organization of the help system changes a bit in ArcGIS 9.1, with Writing Geoprocessing Scripts and the other sections at the top level of the help system.

Slide 32


The Geoprocessor Programming Model
Geoprocessor Programming Model
ArcGIS 9.0
Environmental Systems Research Institute, Inc. All rights reserved.
Objects
Geoprocessor
Describe
Enumerations
Cursor

This pdf can be plotted as a poster to decorate your wall. Or you can just copy it to your desktop for easy access, then zoom around to see different parts. The 9.1 version has some additions, but we won't be needing them.

Sections of the Model
The plot can be divided into color-coded sections. The color coding makes this big poster more manageable. Certain geoprocessing methods have certain properties, for instance the (purple) enumeration methods have (purple) settings that apply to them. Describing a dataset populates a variable with many properties (green) which you can use.

Sections
(1) Geoprocessor – all of the methods and direct properties of the geoprocessor, links to other sections
(2) Describe
(3) Enumerations
(4) Cursors
(5) Objects : various objects created by the CreateObject method (e.g. Spatial reference, array, point, value table, field info)

Slide 33

Using the Geoprocessor Programming Model

Elements

- Environment properties

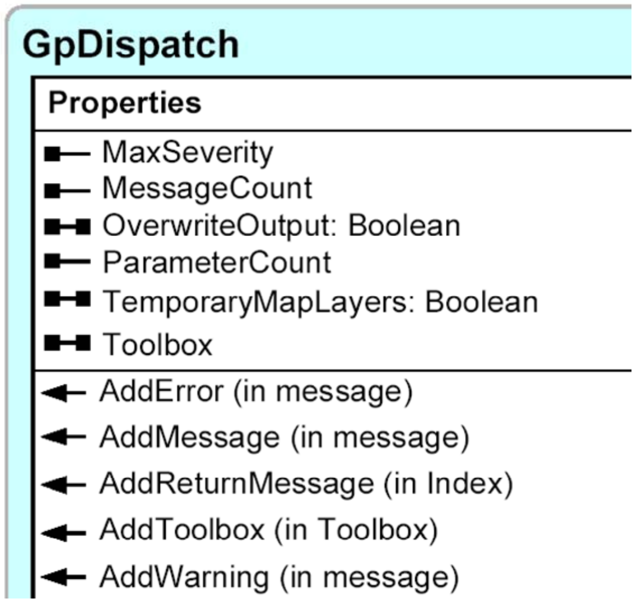
  Read/write, Read-only

- Methods

Sections

- Geoprocessor
- Describe
- Enumerations
- Cursors
- Other

  Spatial reference, array, point, value table, field info



GpDispatch

Properties

- ■— MaxSeverity
- ■— MessageCount
- ■━■ OverwriteOutput: Boolean
- ■— ParameterCount
- ■━■ TemporaryMapLayers: Boolean
- ■━■ Toolbox

- ← AddError (in message)
- ← AddMessage (in message)
- ← AddReturnMessage (in Index)
- ← AddToolbox (in Toolbox)
- ← AddWarning (in message)

Note the symbology for read-only properties (like MessageCount), read-write properties (like OverwriteOutput), and methods (arrow).

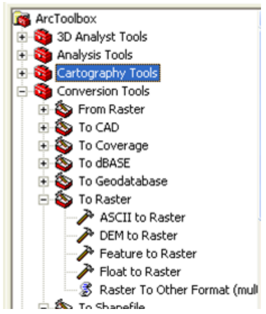
You should explore the geoprocessor programming model sections to find properties and methods.

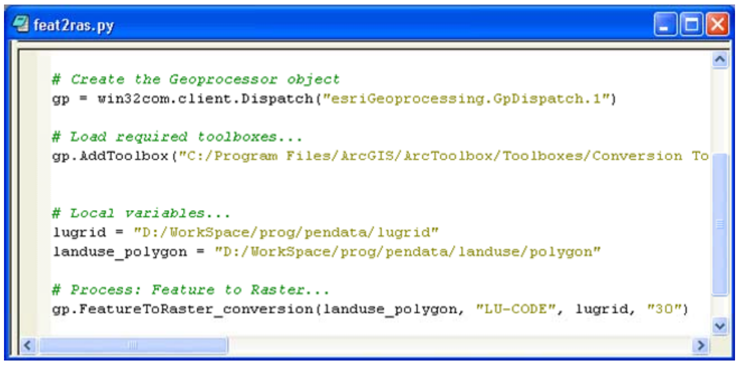
Slide 34

4. Using Geoprocessing Tools

Tools in ArcToolbox can be used in scripts

- For syntax, go to scripting in the help section for the tool





```
# Create the Geoprocessor object
gp = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")

# Load required toolboxes...
gp.AddToolbox("C:/Program Files/ArcGIS/ArcToolbox/Toolboxes/Conversion To


# Local variables...
lugrid = "D:/WorkSpace/prog/pendata/lugrid"
landuse_polygon = "D:/WorkSpace/prog/pendata/landuse/polygon"

# Process: Feature to Raster...
gp.FeatureToRaster_conversion(landuse_polygon, "LU-CODE", lugrid, "30")
```

There are hundreds of geoprocessing tools you can use. Explore these using ArcToolbox. Use these tools is the purpose of geoprocessing scripts. The advantage of these tools over ArcObjects methods is these tools are high level – they each do a substantial amount of processing, that would require a lot more coding when using ArcObjects and VBA.

Case Sensitivity
Geoprocessing tool names are **not case-sensitive**. FeatureToRaster works the same as featuretoraster, as FEATUREtoRASTER, etc. Similarly, **paths** are **not** case sensitive, following the DOS & Windows convention (not the UNIX convention). In contrast, all of Pythons functions, statements, and variables are case sensitive: if ≠If for ≠ For myVariable ≠ MyVariable

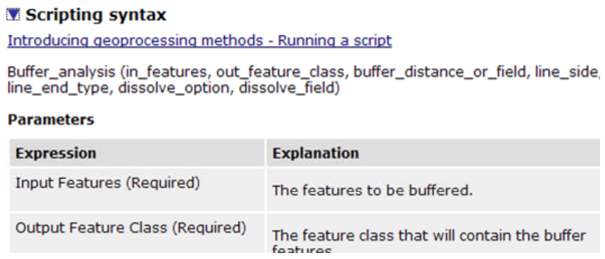
Slide 35

Scripting Syntax for Geoprocessing Tools

Scripting syntax shows

- Python syntax
- Required & optional parameters
- Example scripts

```
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")
gp.workspace = "d:/workspace/prog/HMBarea"
gp.Buffer_analysis ("roads/arc", "Buff100", 100)
```



Scripting syntax

Introducing geoprocessing methods - Running a script

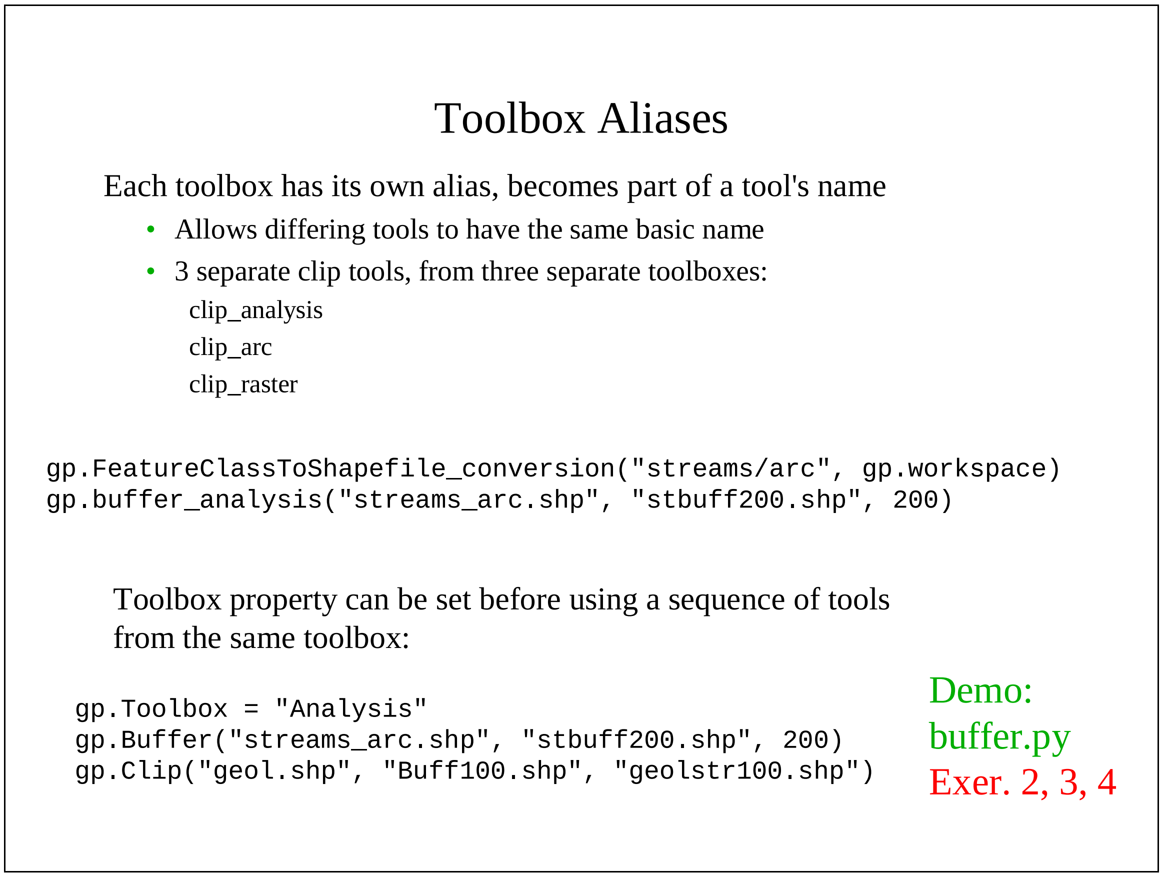
Buffer_analysis (in_features, out_feature_class, buffer_distance_or_field, line_side, line_end_type, dissolve_option, dissolve_field)

Parameters

Expression	Explanation
Input Features (Required)	The features to be buffered.
Output Feature Class (Required)	The feature class that will contain the buffer features

You can just copy script syntax or sample scripts from the help system. Or copy longer sample scripts from online sources. The ArcGIS Online help system is particularly useful, or you can find sample scripts at http://arcscripts.esri.com.

Slide 36



Toolbox Aliases

Each toolbox has its own alias, becomes part of a tool's name

- Allows differing tools to have the same basic name
- 3 separate clip tools, from three separate toolboxes:
  - clip_analysis
  - clip_arc
  - clip_raster

```
gp.FeatureClassToShapefile_conversion("streams/arc", gp.workspace)
gp.buffer_analysis("streams_arc.shp", "stbuff200.shp", 200)
```

Toolbox property can be set before using a sequence of tools from the same toolbox:

```
gp.Toolbox = "Analysis"
gp.Buffer("streams_arc.shp", "stbuff200.shp", 200)
gp.Clip("geol.shp", "Buff100.shp", "geolstr100.shp")
```

Demo:
buffer.py
Exer. 2, 3, 4

Toolbox Aliases:

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

Caution: don’t set the toolbox setting until you’re about to use it. I’ve found it causes problems, for instance setting it before the gp.workspace makes the workspace setting fail.

Slide 37

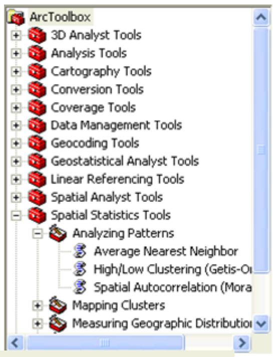
5. Script Tools & using them in ModelBuilder

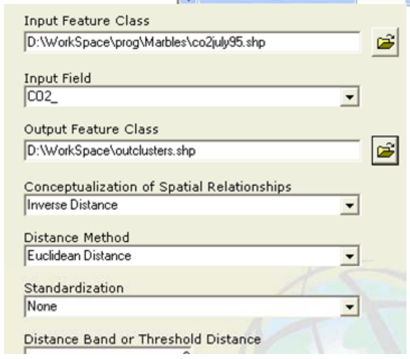
Many tools in ArcToolbox are scripts

- All of Spatial Statistics tools are scripts or models that run scripts

Advantages of creating script tools

- Can run from ArcCatalog or ArcMap
- Runs like a system tool, allowing the user to specify inputs in a familiar way
- Can use with ModelBuilder, and use the model to extend the script's functionality


ArcToolbox
3D Analyst Tools
Analysis Tools
Cartography Tools
Conversion Tools
Coverage Tools
Data Management Tools
Geocoding Tools
Geostatistical Analyst Tools
Linear Referencing Tools
Spatial Analyst Tools
Spatial Statistics Tools
Analyzing Patterns
Average Nearest Neighbor
High/Low Clustering (Getis-Or
Spatial Autocorrelation (Mora
Mapping Clusters
Measuring Geographic Distributio


Input Feature Class
D:\WorkSpace\prog\Marbles\co2july95.shp

Input Field
CO2_

Output Feature Class
D:\WorkSpace\outclusters.shp

Conceptualization of Spatial Relationships
Inverse Distance

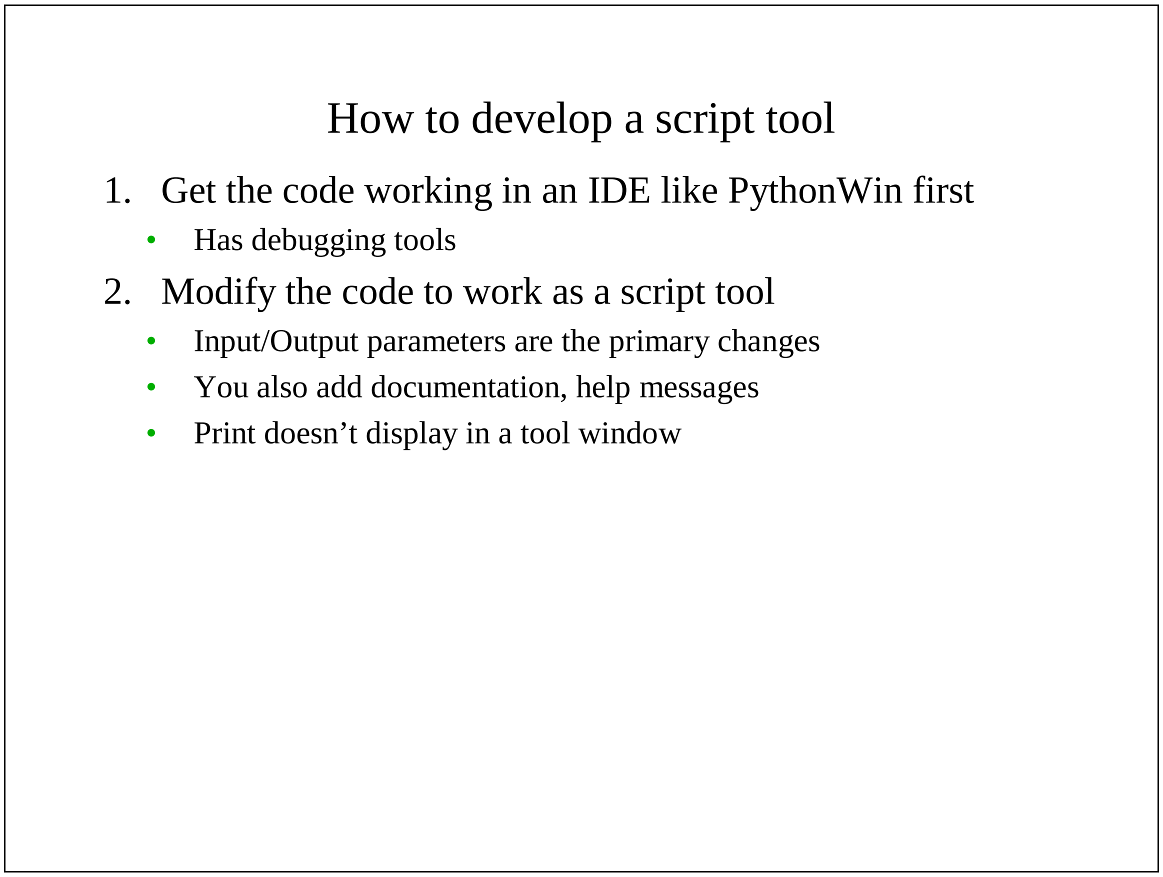
Distance Method
Euclidean Distance

Standardization
None

Distance Band or Threshold Distance

One advantage of running a script from ModelBuilder is adding the output to the ArcMap display. If you run your program multiple times (and have set overwriteoutput to 1), ModelBuilder and ArcMap will deal with redisplaying the output.

Slide 38



How to develop a script tool

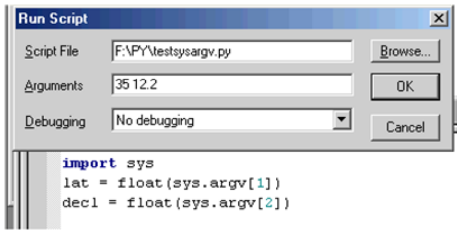
1. Get the code working in an IDE like PythonWin first
   - Has debugging tools
2. Modify the code to work as a script tool
   - Input/Output parameters are the primary changes
   - You also add documentation, help messages
   - Print doesn’t display in a tool window

Changing print statements: In the IDE, you’ll use print statements; then change these to gp.Addmessage() statements to do the same thing in when you want to display what you had been printing.

Slide 39

GetParameterAsText & sys.argv

- Parameter inputs from the user
- Sys.argv starts at 1
- Getparameterastext starts at zero
  - The examples below are equivalent



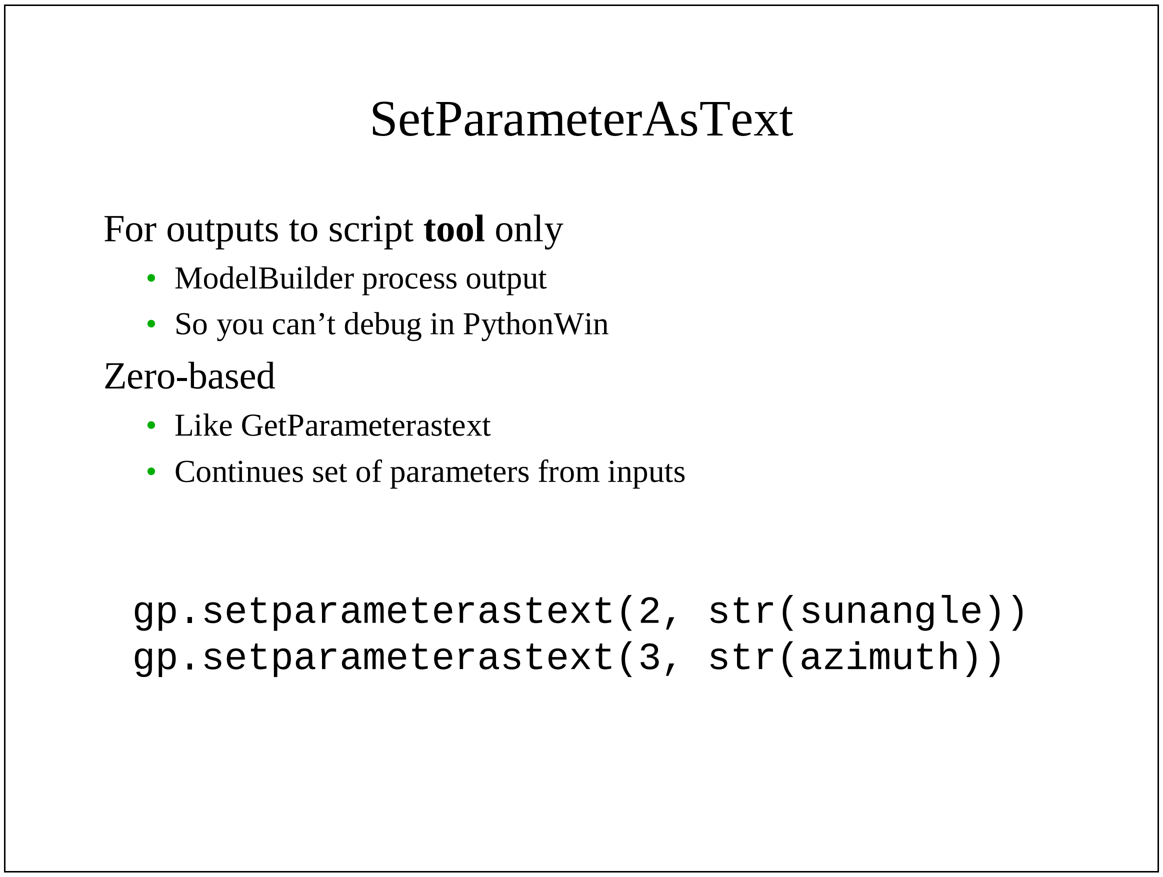
Script File: F:\PY\testsysargv.py
Arguments: 35 12.2
Debugging: No debugging

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

```
# Works either in script tool or PythonWin:
import sys
lat = float(sys.argv[1])
decl = float(sys.argv[2])

# Only if run as script tool:
lat = float(gp.getparameterastext(0))
decl = float(gp.getparameterastext(1))
```

As we just saw, the best way to code a script tool is to first get it running in PythonWin or IDLE. You can use sys.argv[n] for all of the inputs in either PythonWin or as a script tool. So it's a good idea to just stick with sys.argv for both. While you're testing it from the IDE, you can provide inputs in the Arguments section of the Run Script window, each argument separated by spaces. (Note: All inputs are taken in as text strings, so if you need a number you need to use float( ) or int( ) to convert the input to a number.) These same inputs can then be accessed as script tool parameters, established either when you set up the script tool or modify its properties.

Notes: (1) The sys.argv index starts with 1, while if you had used getparameterastext, the index starts with 0. (2) You must import sys to be able to use sys.argv.



SetParameterAsText

For outputs to script **tool** only

- ModelBuilder process output
- So you can’t debug in PythonWin

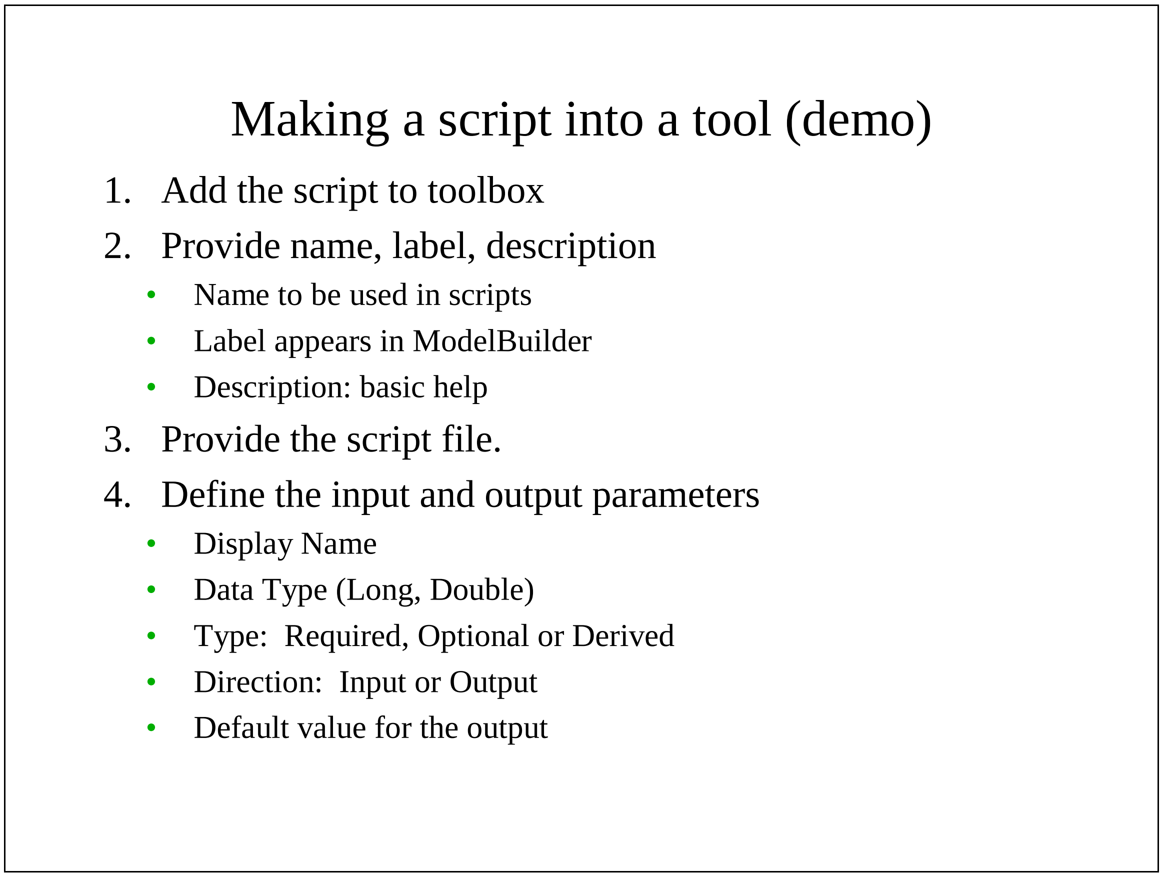
Zero-based

- Like GetParameterastext
- Continues set of parameters from inputs

```
gp.setparameterastext(2, str(sunangle))
gp.setparameterastext(3, str(azimuth))
```

For output, you use setparameterastext. There is no corresponding output parameter we’ll use from PythonWin.

Note that it continues the numbering system from the inputs, so if you have 2 inputs, and these have indices 1 and 2 with sys.argv, the first output would be 3, but since setparameterastext is zero-based, this becomes 2. That's kind of weird, but it works, and you'll get used to it.

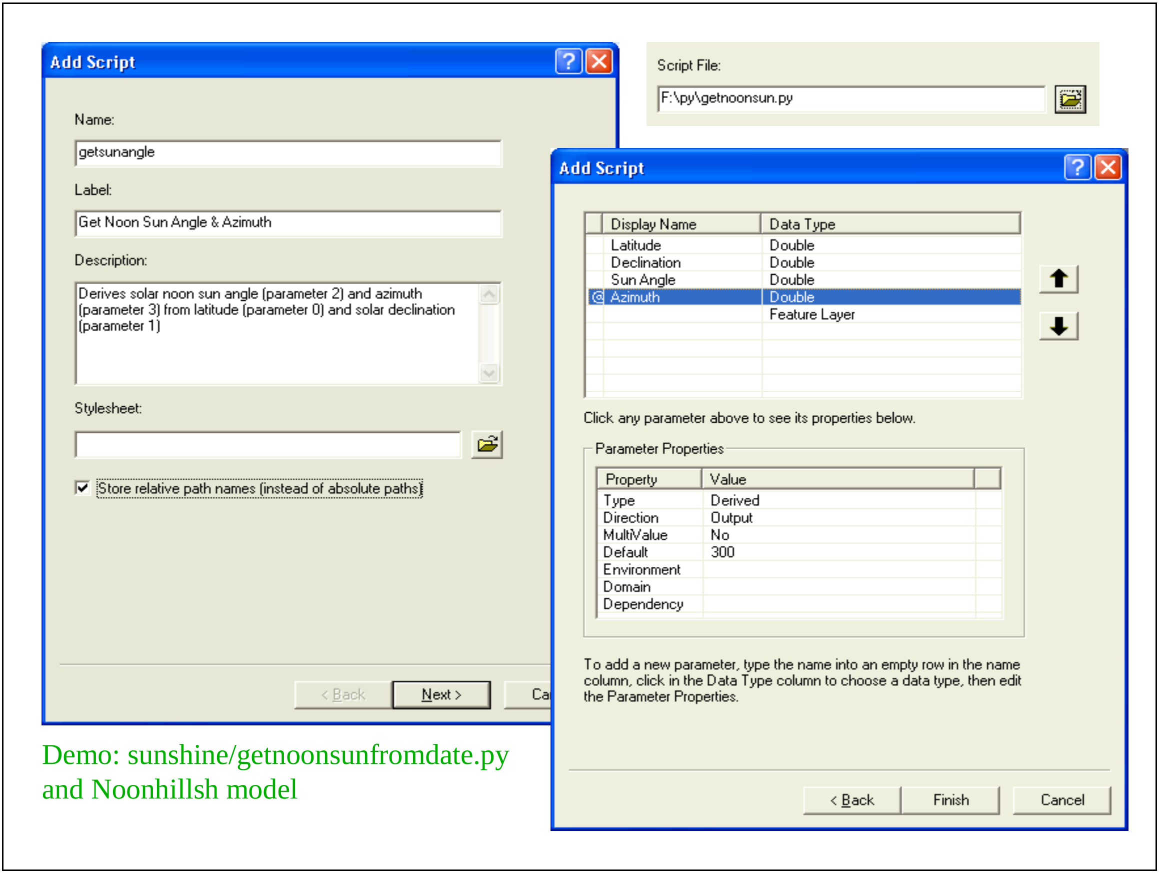


Making a script into a tool (demo)

1. Add the script to toolbox
2. Provide name, label, description
   - Name to be used in scripts
   - Label appears in ModelBuilder
   - Description: basic help
3. Provide the script file.
4. Define the input and output parameters
   - Display Name
   - Data Type (Long, Double)
   - Type: Required, Optional or Derived
   - Direction: Input or Output
   - Default value for the output

Details:
(1) Add the script to toolbox. Right-click the toolbox, and add script.
(2) Provide the name (with no spaces), label, description
   Name is used when used at the command line or in scripts
   Label appears on the tool process in ModelBuilder
   Description is basic help for the user
(3) Provide the script file.
(4) Define the input and output parameters
   Display Name: the parameter name that will be displayed in the dialog or in modelbuilder.
   Data Type: Even though all are transferred as text, must be characterized as integer (long), floating point (double), string, or other type to limit how it is used. Long is a double-precision integer; Double is a double-precision floating point.
   Type: Required, Optional or Derived
   Direction: Input or Output
   Default value for the output – important when used with some tools (e.g. hillshade needs a default value for sunangle)

Slide 42


Add Script

Name: getsunangle

Label: Get Noon Sun Angle & Azimuth

Description: Derives solar noon sun angle (parameter 2) and azimuth (parameter 3) from latitude (parameter 0) and solar declination (parameter 1)

Stylesheet:

☑ Store relative path names (instead of absolute paths)

Script File: F:\py\getnoonsun.py

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	

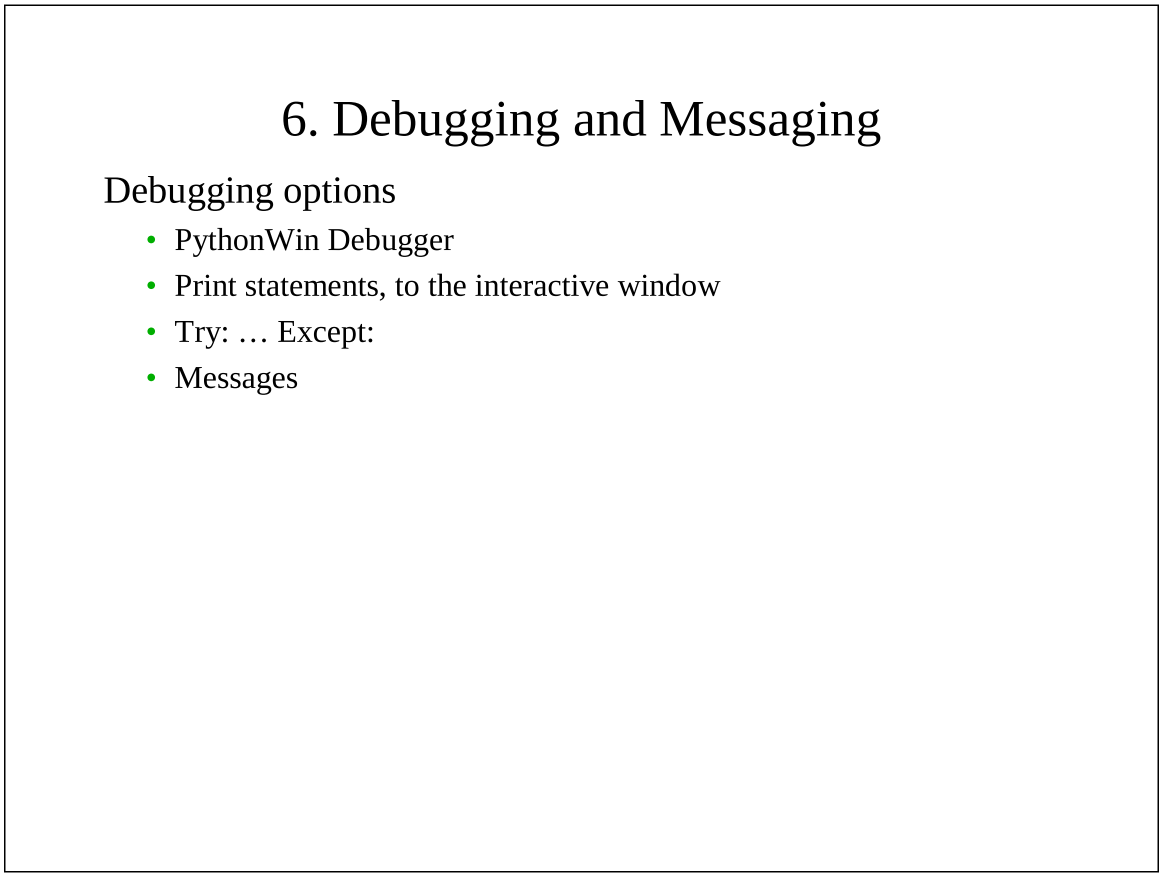
To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

Demo: sunshine/getnoonsunfromdate.py and Noonhillsh model

Setting up the script name, label, description, and defining the parameters.

Note that Latitude, Declination and Sun Angle are all given as Double precision floating point, yet in the script we must convert this input from a text string into a floating point number with `float(sys.argv[1])`, etc. So why do we have to specify the type here? This allows the script tool to evaluate first whether our input is legal as a type, or access data in particular ways. Ffor instance, if a feature dataset is the type, we'd see a browser button allowing us to find the feature class on the computer; it will still pass a text string to Python, where your code should interpret it as a path to a dataset.

Note the use of a default value for Azimuth – it won't work without it… found that out the hard way.



6. Debugging and Messaging

Debugging options

- PythonWin Debugger
- Print statements, to the interactive window
- Try: … Except:
- Messages

There are many ways to debug our programs. We can use the debugging tools of the IDE, which provides an array of debugging methods. Unfortunately, parts of IDLE's debugger doesn't completely work in Windows, and PythonWin can mess with your registry occasionally. This has to improve.

Print statements have been used since the dawn of time to debug computer programs. While the program is running, you print the value of variables to the output. Print statements only work however from an IDE – they have no effect when running a script as a script tool. We'll learn how to deal with this, using messages.

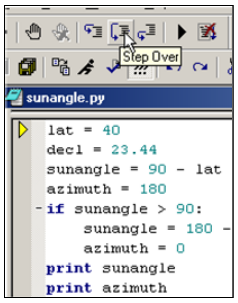
PythonWin Debugging System

PythonWin provides tools for

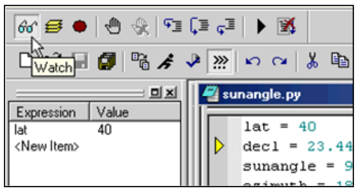
- stepping through code,
- inserting break points,
- observing variables, etc.

Can run script in debugger

- Or step through in debugger
- (try with sunangle.py)

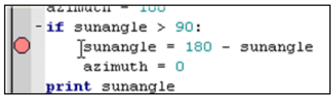


```
lat = 40
decl = 23.44
sunangle = 90 - lat
azimuth = 180
if sunangle > 90:
    sunangle = 180 -
    azimuth = 0
print sunangle
print azimuth
```



Expression	Value
lat	40
<New Item>	

```
lat = 40
decl = 23.44
sunangle = 9
```



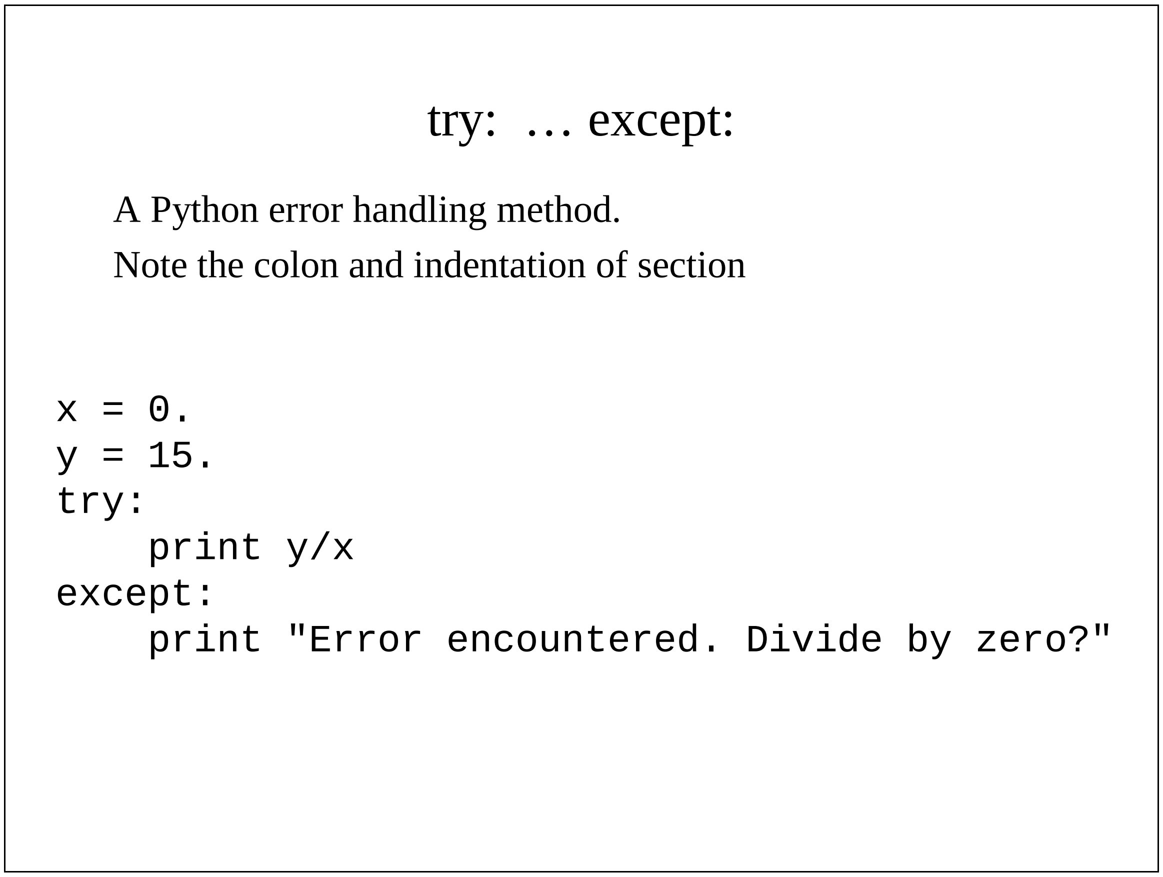
```
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
print sunangle
```

Break point

Run vs. Go: Note the difference between the Run button (on the main toolbar – looks like a little man running) and the triangular Go button on the debugger toolbar. If you set a break point, and you want to proceed to that point, use the Go button. If you just run it, it will run all the way through your program, unless you use the debugger pull-down on the run dialog to Run in the Debugger. Go also continues from a break point.
Step: To step through code, it's a good idea to use Step Over instead of the simple Step button. The Step button will go into code called by any of the steps you use, and you'll find yourself in lots of Python support programs you're not trying to debug.
Break Point: handy when you know the problem is somewhere deep in your code, and you don't want to step your way to that point manually.
Checking the value of Expressions: (or variable) – you can see what values populate the variables at various stages of your script. Much easier than putting in a lot of print statements.
Close: looks like a red mark on a script, just to the right of the Go button

PythonWin's debugger generally works pretty well. It does have an occasional bug itself however, where it creates thousands of debugging toolbars in your registry. If this happens, the only way to deal with it that I know of is to use regedit to delete the "Python for win32" section of the "Python 2.1" section of HKEY_CURRENT_USER. Be careful to do this only in HKEY_CURRENT_USER. The next time you start up PythonWin, it will initialize this section of the registry. …. or you can just use IDLE, though its debugging tools are incomplete in Windows.

Slide 45



try: … except:

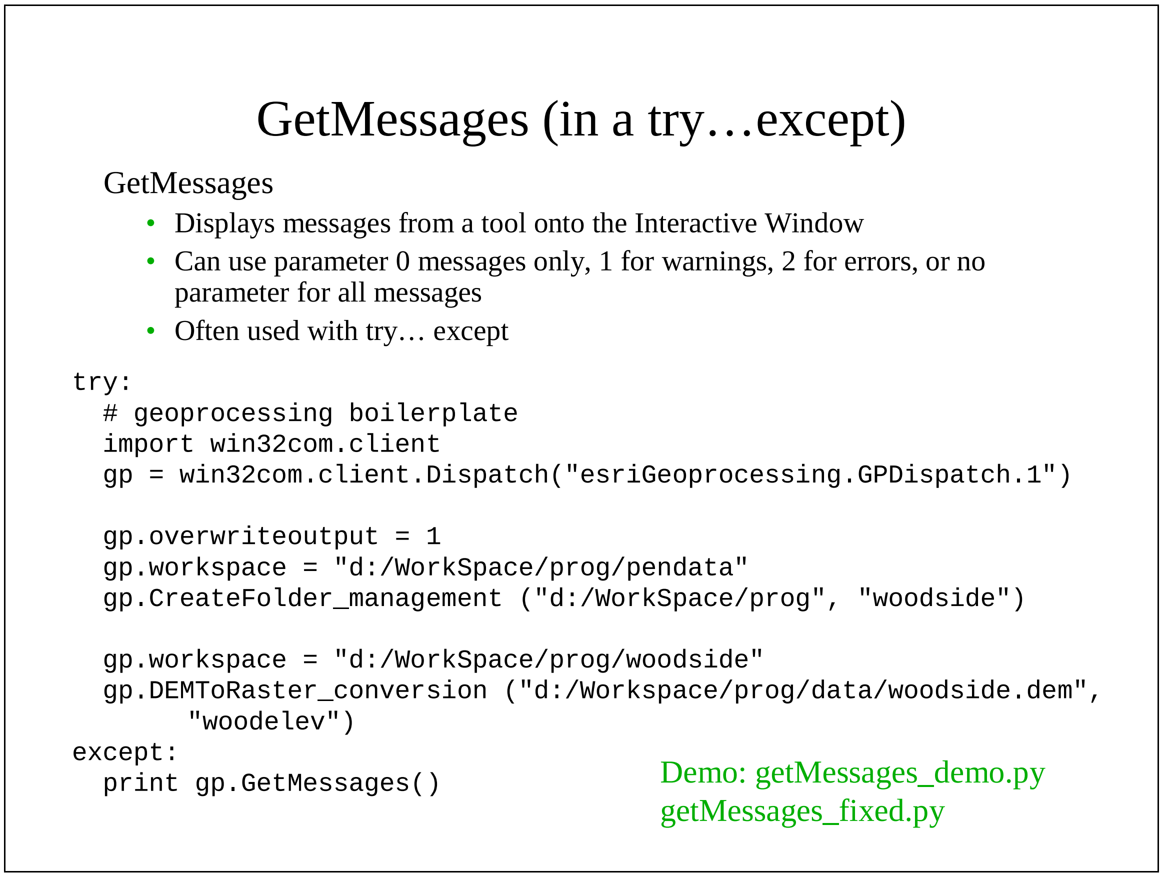
A Python error handling method.

Note the colon and indentation of section

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

Basically what happens is that python tries to execute the try section, then if an "exception" is raised, it goes to the except section and does that.

We're going to find the try…except structure very useful, really essential for our geoprocessing scripts. The example here is an example of an obvious error we've coded in simple python syntax, to demonstrate the processing of exceptions.



GetMessages (in a try…except)

GetMessages

- Displays messages from a tool onto the Interactive Window
- Can use parameter 0 messages only, 1 for warnings, 2 for errors, or no parameter for all messages
- Often used with try… except

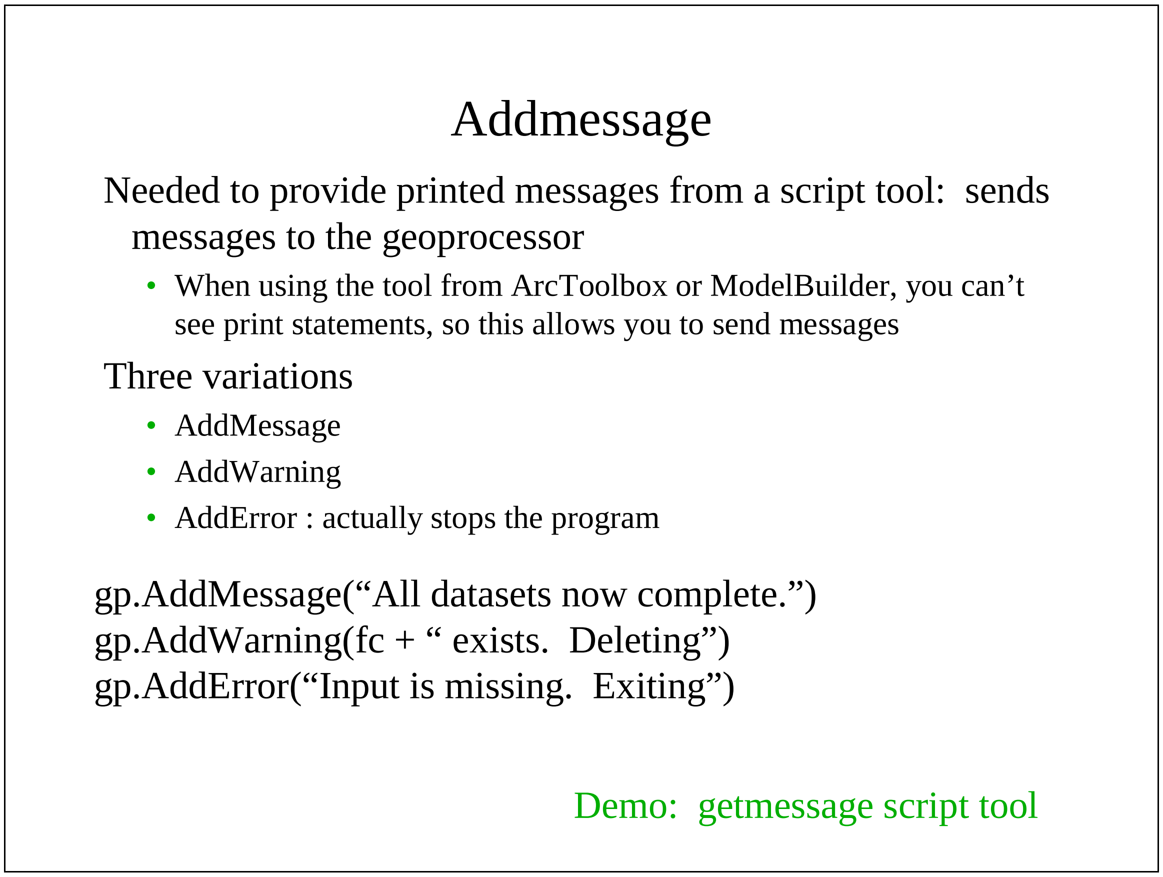
```
try:
  # geoprocessing boilerplate
  import win32com.client
  gp = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

  gp.overwriteoutput = 1
  gp.workspace = "d:/WorkSpace/prog/pendata"
  gp.CreateFolder_management ("d:/WorkSpace/prog", "woodside")

  gp.workspace = "d:/WorkSpace/prog/woodside"
  gp.DEMToRaster_conversion ("d:/Workspace/prog/data/woodside.dem",
       "woodelev")
except:
  print gp.GetMessages()
```

Demo: getMessages_demo.py
getMessages_fixed.py

GetMessages gets the multi-line message from the last geoprocessing tool.
GetMessage gets each line separately. MessageCount returns the total number of messages. Since the message includes the start and end time, one use for GetMessage is to just display the start and end time with

```
print gp.Getmessage(1)
print gp.Getmessage(gp.messagecount – 1)

try:
  # geoprocessing boilerplate
  import win32com.client
  gp = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

  gp.overwriteoutput = 1
  gp.workspace = "d:/WorkSpace/prog/pendata"
  gp.CreateFolder_management ("d:/WorkSpace/prog", "woodside")

  gp.workspace = "d:/WorkSpace/prog/woodside"
  gp.DEMToRaster_conversion ("d:/Workspace/prog/data/woodside.dem",
      "woodelev")
except:
  print gp.GetMessages()
```

Slide 47



Addmessage

Needed to provide printed messages from a script tool: sends messages to the geoprocessor

- When using the tool from ArcToolbox or ModelBuilder, you can't see print statements, so this allows you to send messages

Three variations

- AddMessage
- AddWarning
- AddError : actually stops the program

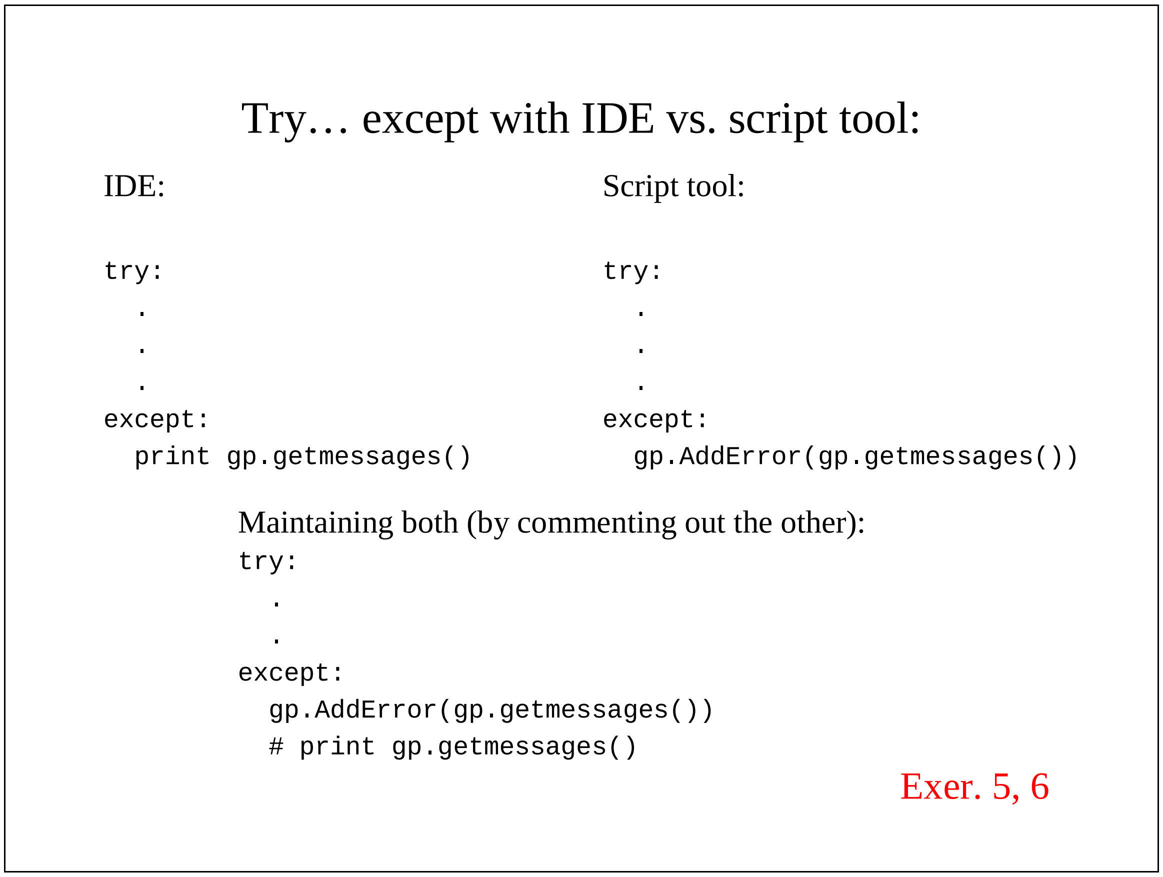
gp.AddMessage("All datasets now complete.")
gp.AddWarning(fc + " exists. Deleting")
gp.AddError("Input is missing. Exiting")

Demo: getmessage script tool

If you want, you can just duplicate your print and addmessage statements, and use them in either an IDE or a script tool. The following script, which duplicates print and addmessage statements, works in either place:

```
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")
lat = 30
decl = 25
sunangle = 90 - lat + decl
azimuth = 180
if sunangle > 90:
    sunangle = 180 - sunangle
    azimuth = 0
print sunangle
print azimuth
gp.addmessage(sunangle)
gp.addmessage(azimuth)
```

Slide 48



Try… except with IDE vs. script tool:

IDE:

```
try:
  .
  .
  .
except:
  print gp.getmessages()
```

Script tool:

```
try:
  .
  .
  .
except:
  gp.AddError(gp.getmessages())
```

Maintaining both (by commenting out the other):

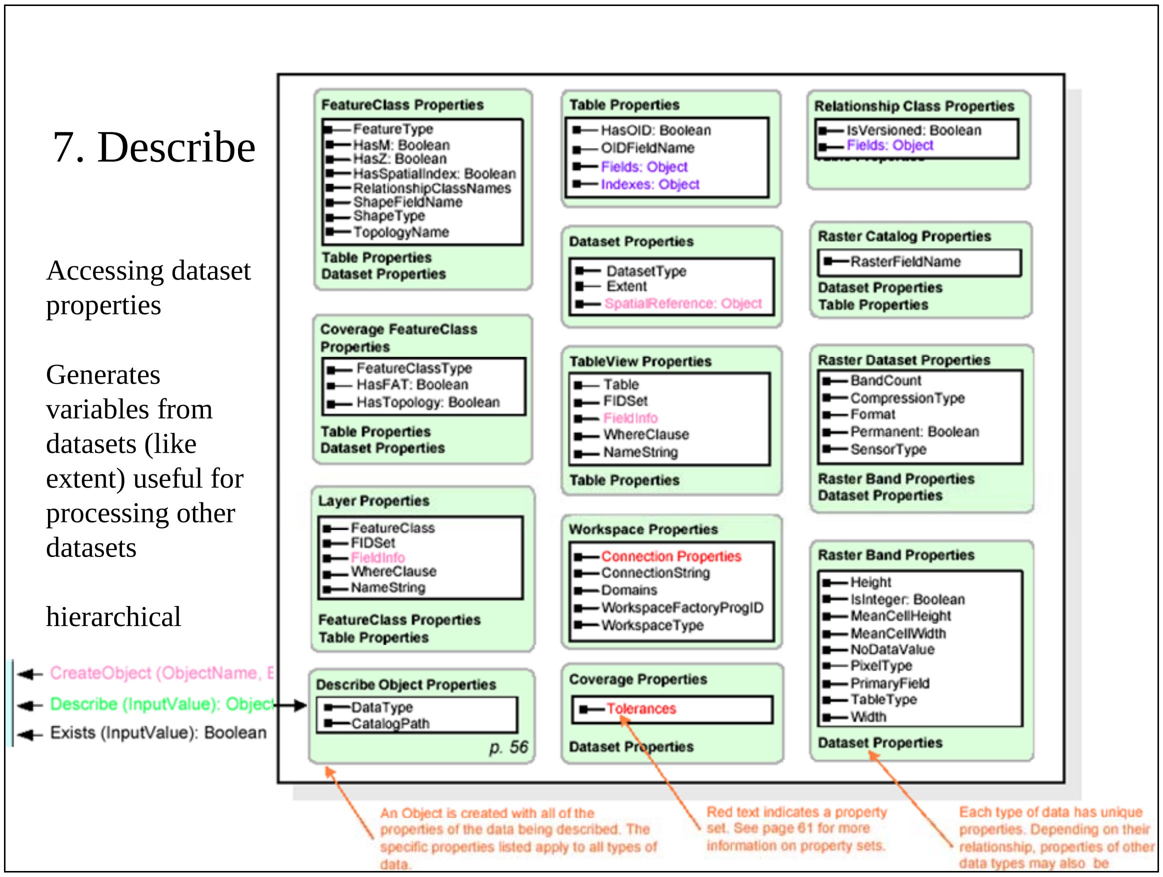
```
try:
  .
  .
except:
  gp.AddError(gp.getmessages())
  # print gp.getmessages()
```

Exer. 5, 6

As before, we could just leave the print message in the script tool without commenting it out. Print statements seem to have no effect in a script tool, so the except section works in either environment:

```
try:
  import win32com.client
  gp = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")
  gp.overwriteoutput = 1
  gp.workspace = "d:/WorkSpace/prog/pendata"
  gp.CreateFolder_management ("d:/WorkSpace/prog", "woodside")
  gp.workspace = "d:/WorkSpace/prog/woodside"
  gp.DEMToRaster_conversion ("d:/Workspace/prog/data/woodside.dem",
      "woodelev")
except:
  print gp.GetMessages()
  gp.AddError(gp.getmessages())
```

Slide 49



7. Describe

Accessing dataset properties

Generates variables from datasets (like extent) useful for processing other datasets

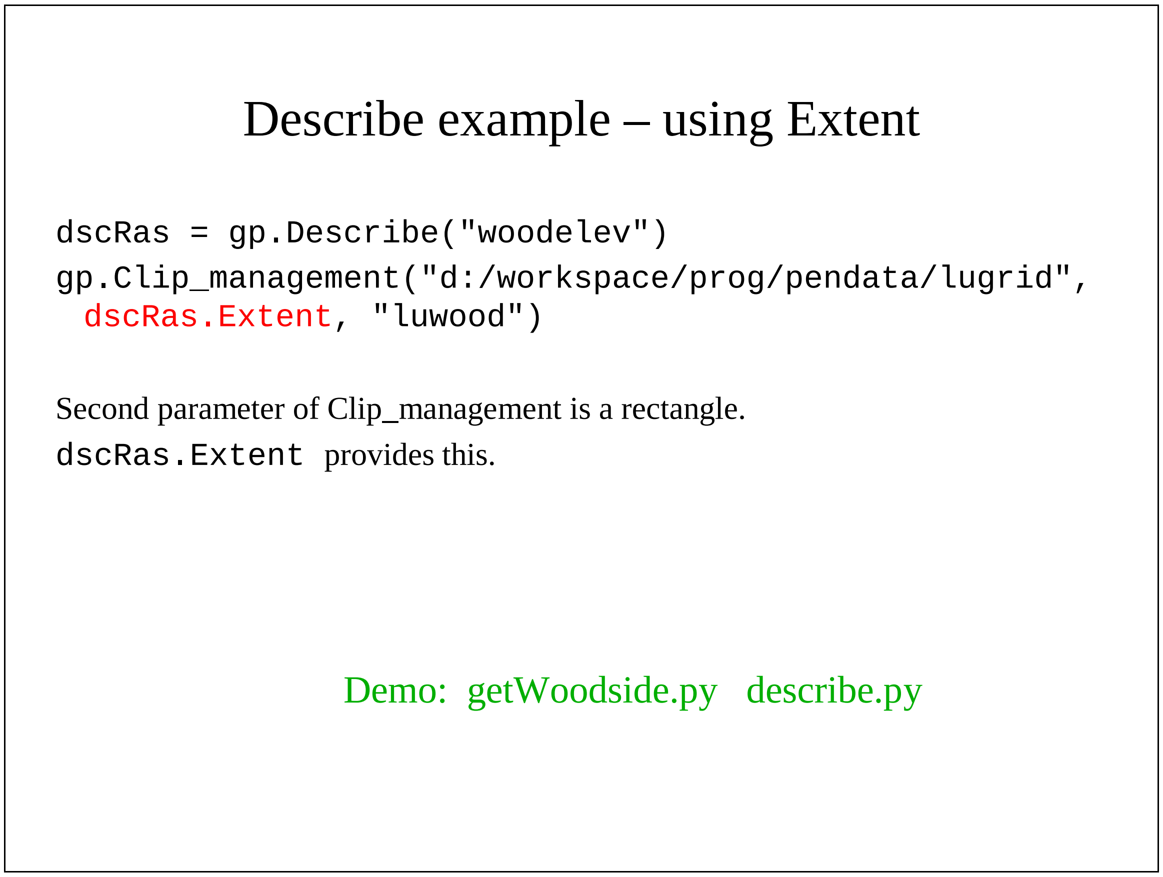
hierarchical

Need: to use characteristics of GIS datasets to process other data. For example, to use the extent of one dataset to clip another dataset. Or to detect the type of topology of a feature class, etc.

gp.Describe method generates a variety of properties about a dataset
- **Extent** : for all spatial datasets
- **Datatype**, **CatalogPath** : for all data
- **BandCount** for imagery
- **FeatureType**, **HasZ** for feature classes
- Field info, etc.

Note the hierarchical nature of the properties. For instance:
FeatureClass properties include all table properties and dataset properties
Raster Dataset properties include Raster Band properties and Dataset properties
So the Dataset property extent is available to Feature Classes, Rasters, Coverages.

Slide 50



Describe example – using Extent

```
dscRas = gp.Describe("woodelev")
gp.Clip_management("d:/workspace/prog/pendata/lugrid",
  dscRas.Extent, "luwood")
```

Second parameter of Clip_management is a rectangle.

`dscRas.Extent` provides this.

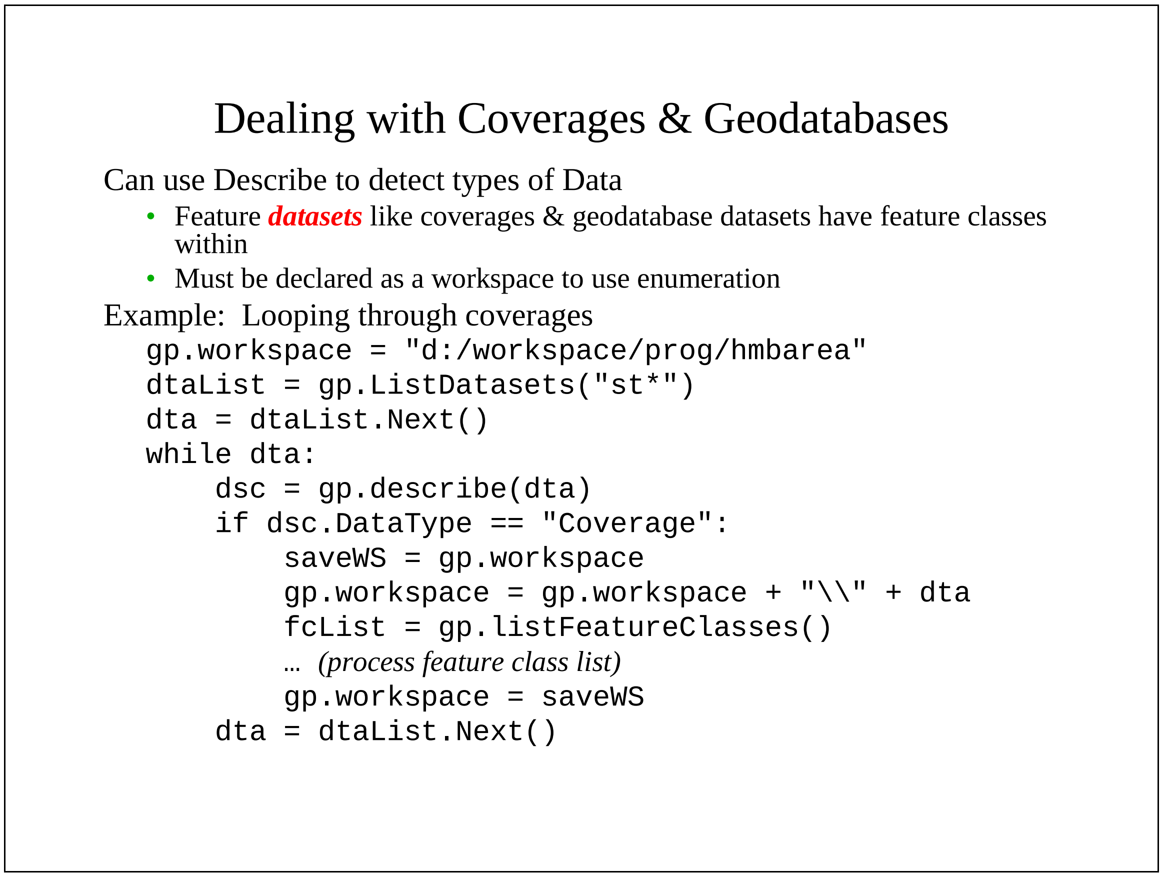
Demo: getWoodside.py describe.py

Using the extent of *any* ***dataset*** as a rectangle for a clip is very handy, especially for rasters which always need a rectangle. There are many other things that result from a describe – see the geoprocessor model diagram.

There are many useful properties provided by the Describe system. The best way to see the range of properties is to peruse the Geoprocessor Model Diagram. Only you won't find the possible values these properties have. To find what values these properties have, access the **Describe** Method in the help system. It's in the list of properties and methods (called IGPDispatch in ArcGIS 9.0) in **Scripting Object: Properties and Methods**, which is in **Writing Geoprocessing Scripts**. Some important ones are:

DataSetType : returns type of dataset, which can be a featuredataset, a rasterdataset, a table, a toolbox, or fourteen other things. To make sure your script works right, you may want to know what type of dataset you're looking at.
MeanCellHeight and **MeanCellWidth** : are the cell size for raster bands.
FeatureType : applied to feature classes
ShapeType : also applied to feature classes

Slide 51



Dealing with Coverages & Geodatabases

Can use Describe to detect types of Data

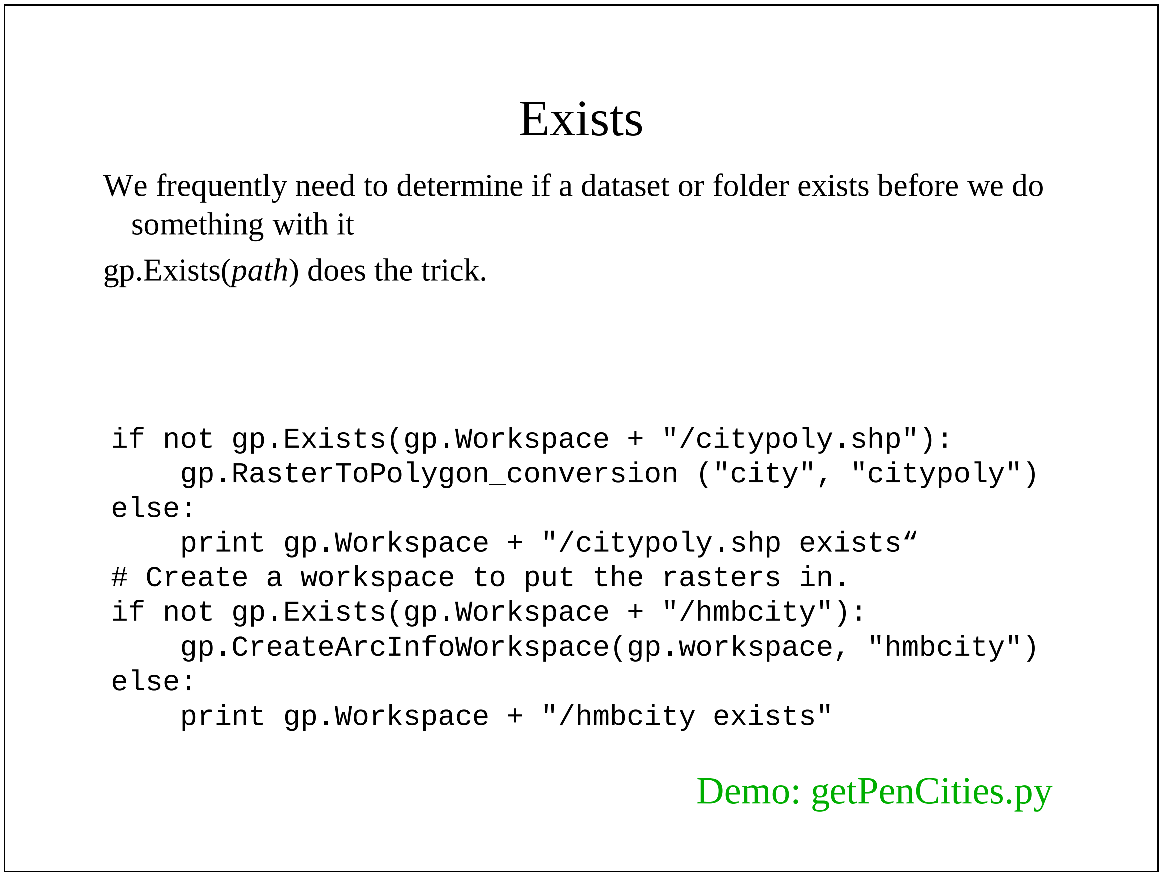
- Feature ***datasets*** like coverages & geodatabase datasets have feature classes within
- Must be declared as a workspace to use enumeration

Example: Looping through coverages

```
gp.workspace = "d:/workspace/prog/hmbarea"
dtaList = gp.ListDatasets("st*")
dta = dtaList.Next()
while dta:
    dsc = gp.describe(dta)
    if dsc.DataType == "Coverage":
        saveWS = gp.workspace
        gp.workspace = gp.workspace + "\\" + dta
        fcList = gp.listFeatureClasses()
        … (process feature class list)
        gp.workspace = saveWS
    dta = dtaList.Next()
```

```
gp.workspace = "d:/workspace/prog/hmbarea"
dtaList = gp.ListDatasets("st*")
dta = dtaList.Next()
while dta:
    dsc = gp.describe(dta)
    print dta, dsc.DataType, dsc.DatasetType
    if dsc.DataType == "Coverage":
        saveWS = gp.workspace
        gp.workspace = gp.workspace + "\\" + dta
        fcList = gp.listFeatureClasses()
        fc = fcList.Next()
        while fc:
            fdsc = gp.describe(fc)
            print fdsc.featureclasstype
            fc = fcList.Next()
        gp.workspace = saveWS
    dta = dtaList.Next()
```

Slide 52



Exists

We frequently need to determine if a dataset or folder exists before we do something with it

gp.Exists(*path*) does the trick.

```
if not gp.Exists(gp.Workspace + "/citypoly.shp"):
    gp.RasterToPolygon_conversion ("city", "citypoly")
else:
    print gp.Workspace + "/citypoly.shp exists“
# Create a workspace to put the rasters in.
if not gp.Exists(gp.Workspace + "/hmbcity"):
    gp.CreateArcInfoWorkspace(gp.workspace, "hmbcity")
else:
    print gp.Workspace + "/hmbcity exists"
```

Demo: getPenCities.py

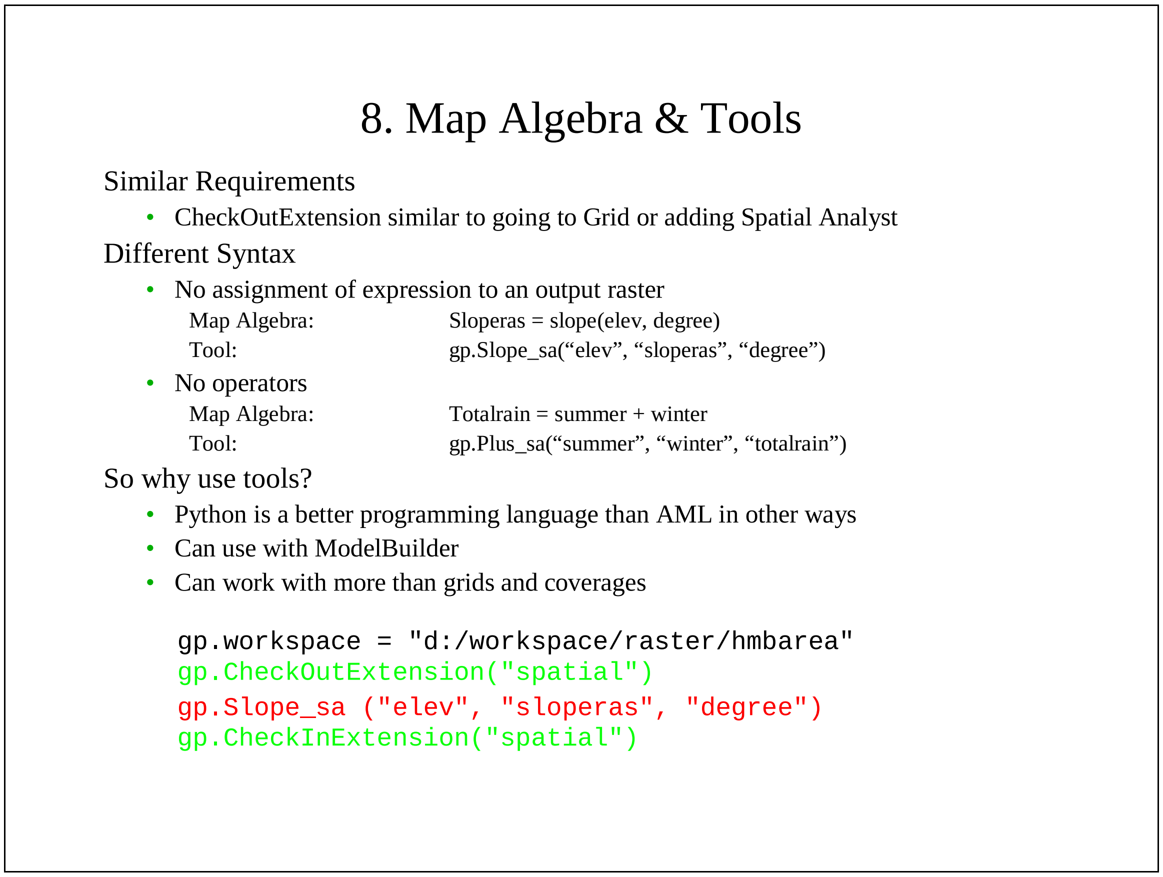
Doesn’t work for fields. The first line of the following code snippet shows how this is done (before it tries to add the field), though it's not obvious:

```
if not gp.ListFields("contour.shp","elevm").Next():
     gp.addfield ("contour.shp", "elevm", "float")
```

The above example also demonstrates how enumerations in general can be tested to see if they contain anything. The `.Next()` method will return the next member of the enumeration; when run for the first time and returns nothing, the enumeration is empty. Note that we used this earlier when we processed through all of the feature classes in a workspace. The **while** statement won't run if the enumeration is empty, and the expression `fcList.Next()` returns nothing:

```
gp.workspace = "d:/workspace/prog/HMBarea"
fcList = gp.ListFeatureClasses()
fc = fcList.Next()
while fc:
    print fc
    fc = fcList.Next()
```

Slide 53



8. Map Algebra & Tools

Similar Requirements

- CheckOutExtension similar to going to Grid or adding Spatial Analyst

Different Syntax

- No assignment of expression to an output raster

Map Algebra:	Sloperas = slope(elev, degree)
Tool:	gp.Slope_sa(“elev”, “sloperas”, “degree”)

- No operators

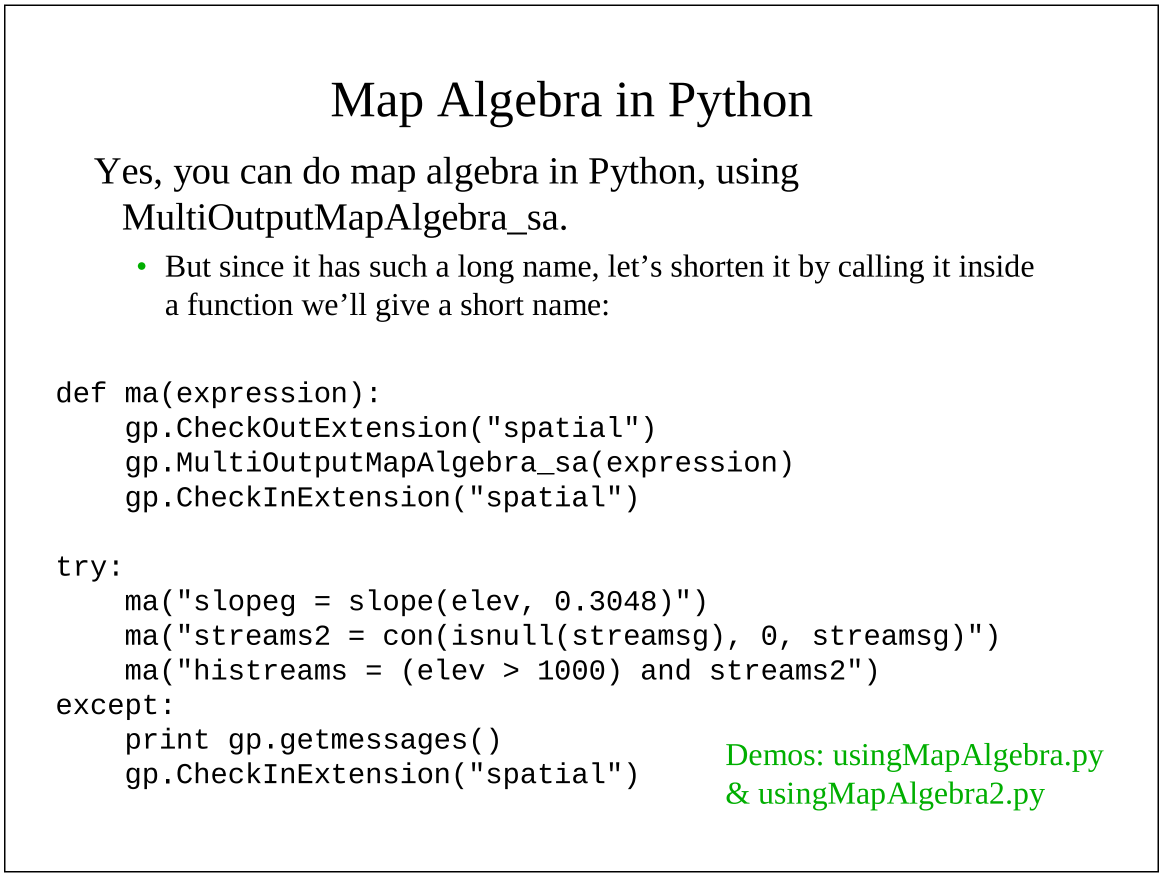
Map Algebra:	Totalrain = summer + winter
Tool:	gp.Plus_sa(“summer”, “winter”, “totalrain”)

So why use tools?

- Python is a better programming language than AML in other ways
- Can use with ModelBuilder
- Can work with more than grids and coverages

```
gp.workspace = "d:/workspace/raster/hmbarea"
gp.CheckOutExtension("spatial")
gp.Slope_sa ("elev", "sloperas", "degree")
gp.CheckInExtension("spatial")
```

Map Algebra is a very elegant way of processing raster gis data, and was the primary tool for analysis in the Arc/Info Grid system. It is very readable, and has been used in many AML programs over the years. In many ways, the geoprocessing tool model is clumsier – forcing you to use a "plus" tool to add two rasters together, for instance, instead of an expression like grid1 + grid2. Coding a more complex equation gets very messy trying to do this with tools. So many of us would prefer to use map algebra. Fortunately, there is a way (see the next slide).



Map Algebra in Python

Yes, you can do map algebra in Python, using MultiOutputMapAlgebra_sa.

- But since it has such a long name, let’s shorten it by calling it inside a function we’ll give a short name:

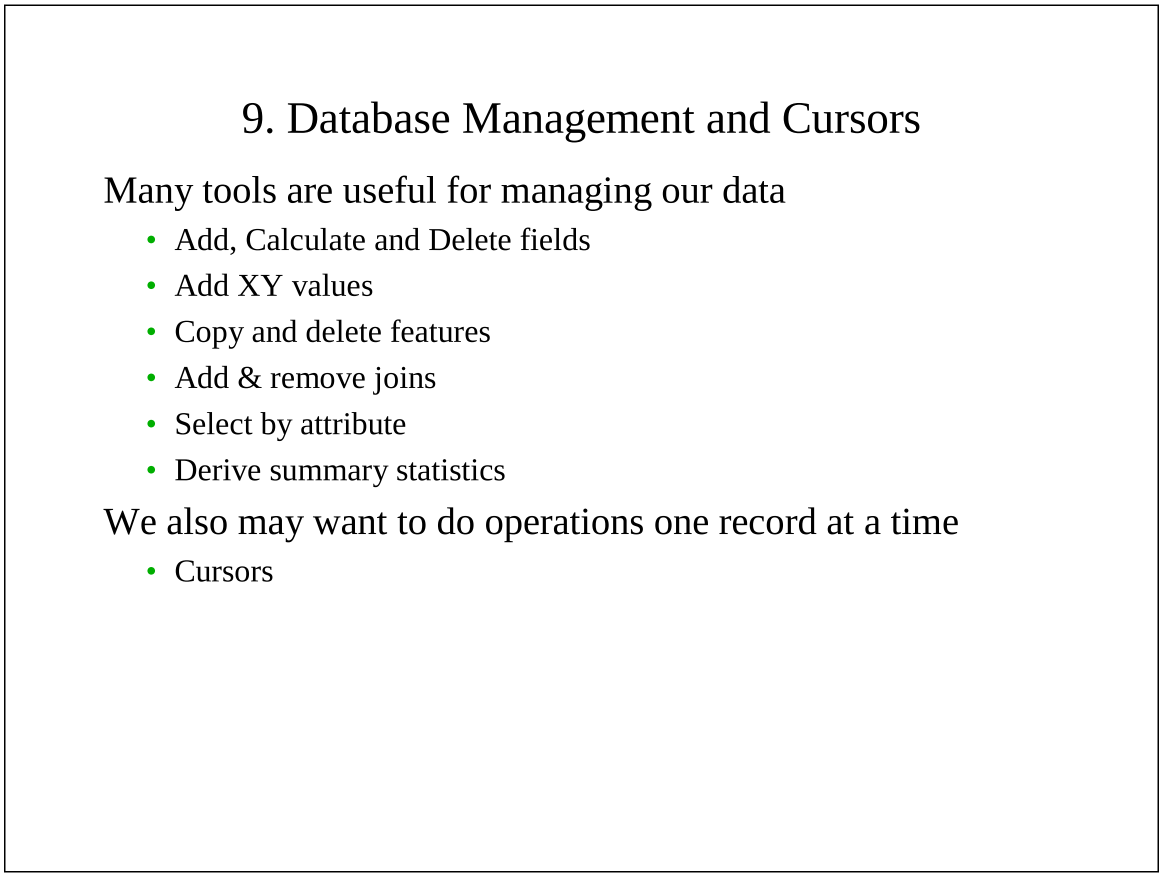
```
def ma(expression):
    gp.CheckOutExtension("spatial")
    gp.MultiOutputMapAlgebra_sa(expression)
    gp.CheckInExtension("spatial")

try:
    ma("slopeg = slope(elev, 0.3048)")
    ma("streams2 = con(isnull(streamsg), 0, streamsg)")
    ma("histreams = (elev > 1000) and streams2")
except:
    print gp.getmessages()
    gp.CheckInExtension("spatial")
```

Demos: usingMapAlgebra.py & usingMapAlgebra2.py

The MultiOutputMapAlgebra_sa tool lets you process a map algebra assignment statement, so just might be the answer for processing mathematical formulae using raster inputs. Only "MultiOutputMapAlgebra_sa" is kind of a long tool name, so the above code let's us shorten the name. Another way is to just assign the long tool name, along with gp, to that short name “ma” – works the same as the function we just defined:

```
try:
     gp.CheckOutExtension("spatial")
     ma = gp.MultiOutputMapAlgebra_sa
     ma("slopeg = slope(elev, 0.3048)")
     ma("streams2 = con(isnull(streamsg), 0, streamsg)")
     ma("histreams = (elev > 1000) and streams2")
     gp.CheckInExtension("spatial")
except:
     print gp.getmessages(2)
     gp.CheckInExtension("spatial")
```

Slide 55



9. Database Management and Cursors

Many tools are useful for managing our data

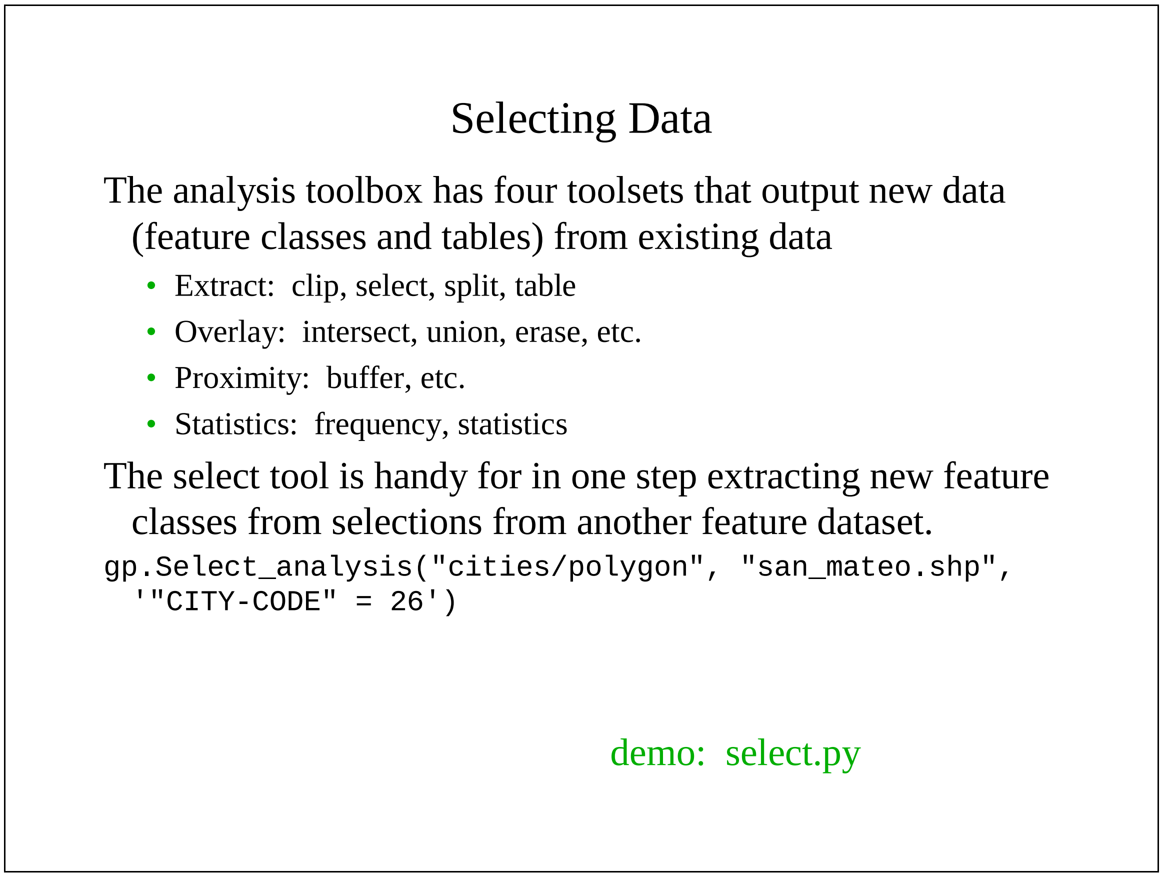
- Add, Calculate and Delete fields
- Add XY values
- Copy and delete features
- Add & remove joins
- Select by attribute
- Derive summary statistics

We also may want to do operations one record at a time

- 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.

Slide 56



Selecting Data

The analysis toolbox has four toolsets that output new data (feature classes and tables) from existing data

- Extract: clip, select, split, table
- Overlay: intersect, union, erase, etc.
- Proximity: buffer, etc.
- Statistics: frequency, statistics

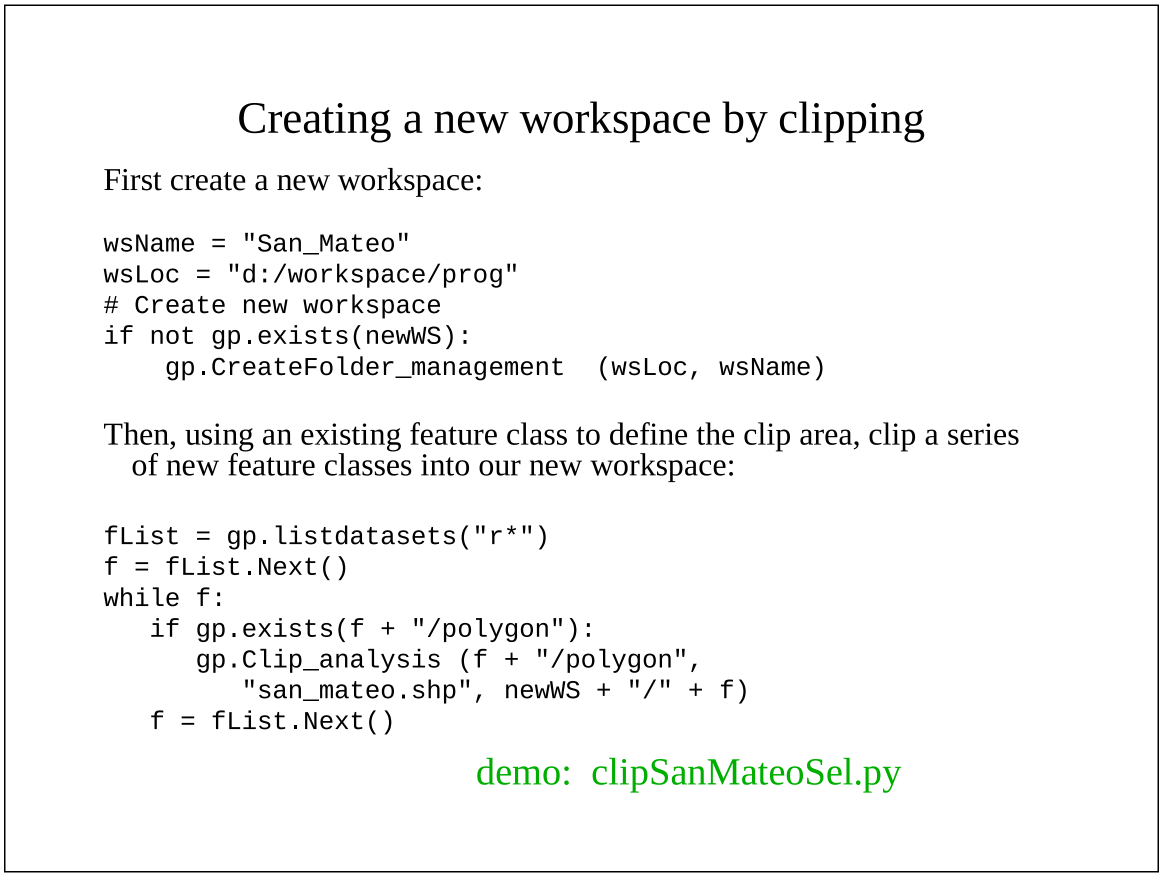
The select tool is handy for in one step extracting new feature classes from selections from another feature dataset.

```
gp.Select_analysis("cities/polygon", "san_mateo.shp",
  '"CITY-CODE" = 26')
```

demo: select.py

```
# Clips San Mateo from a city coverage
import win32com.client
gp = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")
gp.overwriteoutput = 1
gp.workspace = "d:/workspace/prog/pendata"
try:
    # Select San Mateo (city_code = 26)
    gp.Select_analysis("cities/polygon", "san_mateo.shp",
             '"CITY-CODE" = 26')
except:
    print gp.getmessages()
```

Slide 57



Creating a new workspace by clipping

First create a new workspace:

```
wsName = "San_Mateo"
wsLoc = "d:/workspace/prog"
# Create new workspace
if not gp.exists(newWS):
    gp.CreateFolder_management  (wsLoc, wsName)
```

Then, using an existing feature class to define the clip area, clip a series of new feature classes into our new workspace:

```
fList = gp.listdatasets("r*")
f = fList.Next()
while f:
   if gp.exists(f + "/polygon"):
      gp.Clip_analysis (f + "/polygon",
         "san_mateo.shp", newWS + "/" + f)
   f = fList.Next()
```

demo: clipSanMateoSel.py

Added to the previous selection code, we use the following to create a workspace and clip datasets from coverages to create San Mateo city area shapefiles in our new workspace:

```
wsName = "San_Mateo"
newWS = workspaceLoc + "/" + wsName
# Create new workspace
if not gp.exists(newWS):
    gp.CreateFolder_management (workspaceLoc, wsName)
# For each polygon coverage write out a shapefile clipped to San
Mateo
fList = gp.listdatasets("r*")
f = fList.Next()
while f:
    if gp.exists(f + "/polygon"):
        gp.Clip_analysis (f + "/polygon", "san_mateo.shp",
            newWS + "/" + f)
    f = fList.Next()
```

Slide 58

Example: Adding, Calculating, Deleting a Field

To add and calc a new field, we need to (1) see if a field exists; (2) if not, add it; (3) calculate a value for it.

```
gp.toolbox = "management"
if not gp.ListFields("contour.shp","elevm").Next():
  gp.addfield ("contour.shp", "elevm", "float")
gp.calculatefield ("contour.shp", "elevm", "[elev] * 0.3048")
```

To delete a field is even easier:

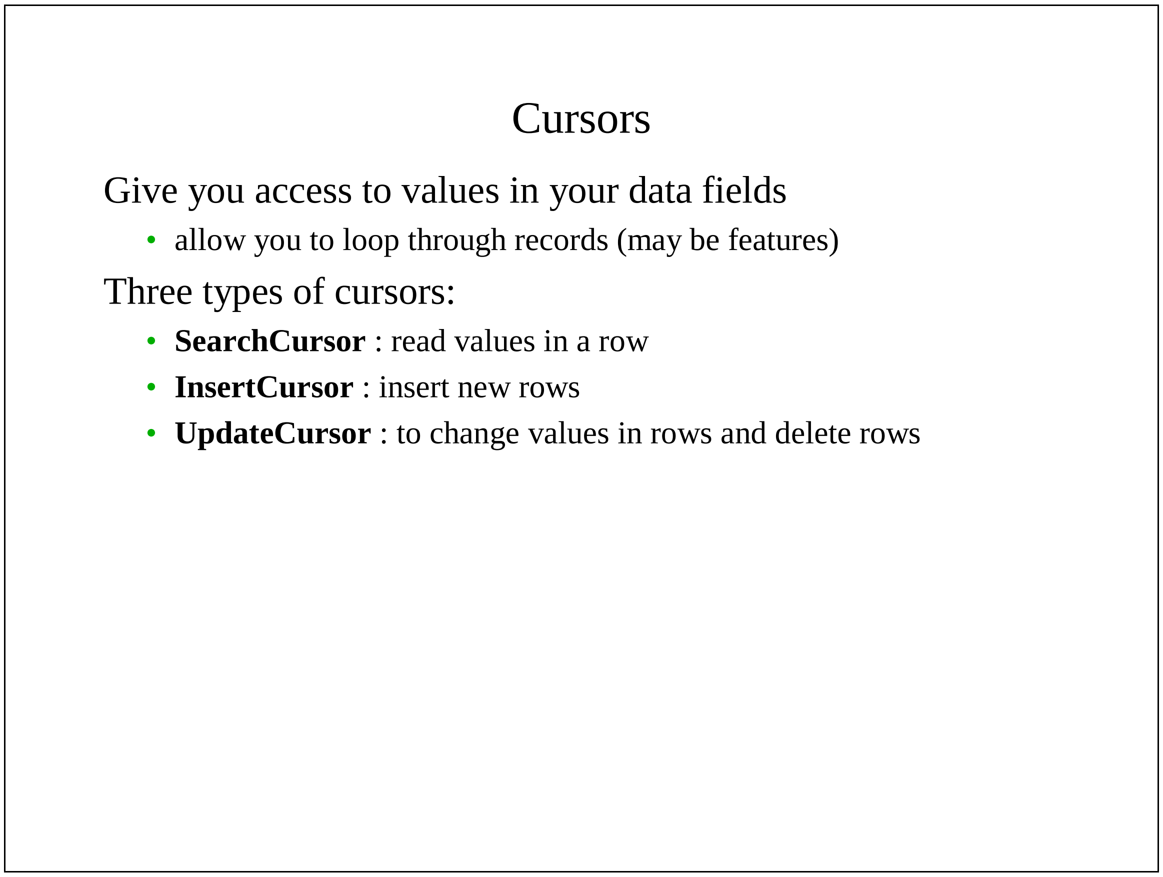
```
  gp.deletefield ("contour.shp", "elevm")
```

Demos: addCalcField.py
deleteField.py

More complete coding:

```
gp.workspace = "D:/WorkSpace/prog/surf_bld"
gp.toolbox = "management"
try:
      # add a field to this shapefile & calc its value
      if not gp.ListFields("contour.shp","elevm").Next():
            gp.addfield ("contour.shp", "elevm", "float")
      gp.calculatefield ("contour.shp", "elevm", "[elev] * 0.3048")
except:
      print gp.GetMessages(2)
```

Slide 59



Cursors

Give you access to values in your data fields

- allow you to loop through records (may be features)

Three types of cursors:

- **SearchCursor** : read values in a row
- **InsertCursor** : insert new rows
- **UpdateCursor** : to change values in rows and delete rows

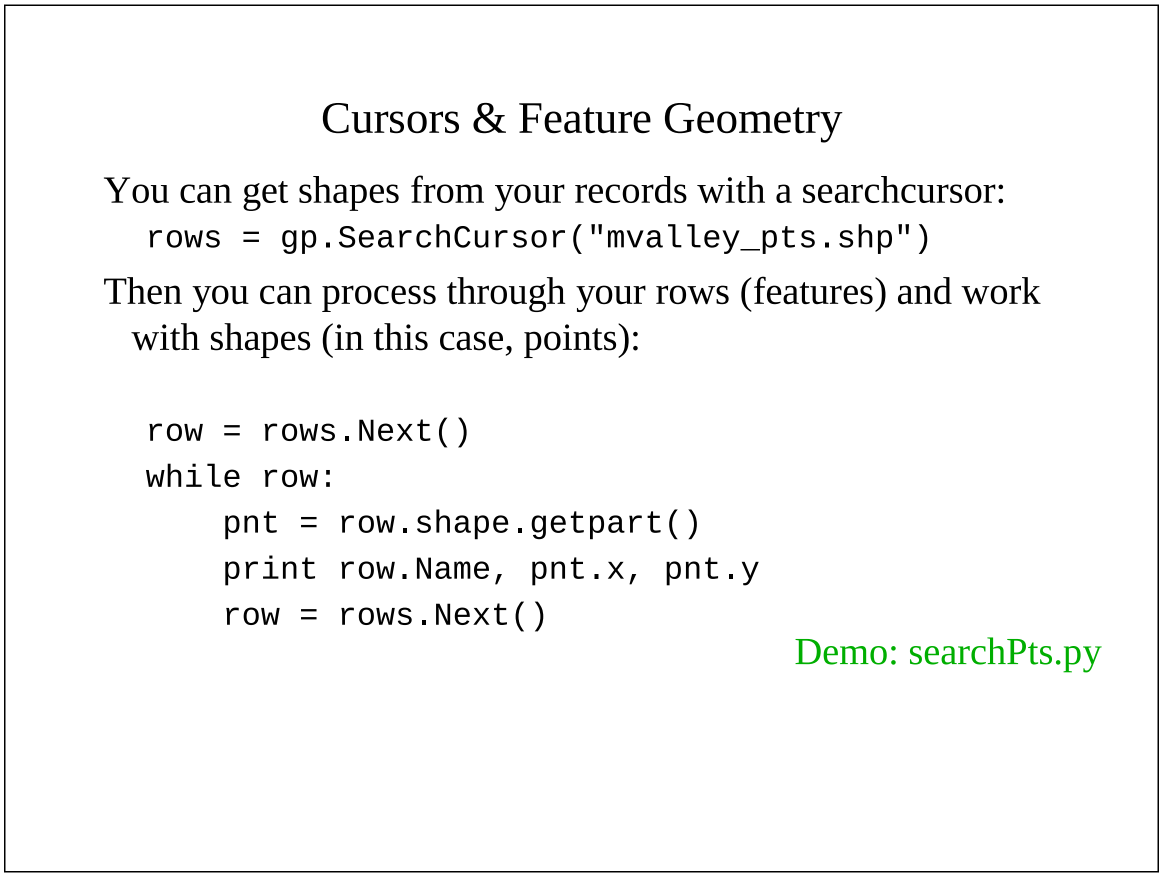
Cursors give you access to values in your data fields, and allow you to loop through records in your data tables. Since each record (row) might be a vector feature or a raster value, this gives you considerable power to process your data. The following searchcursor displays values from fields:

```
    cur = gp.SearchCursor("contour.shp")
    row = cur.Next()
    while row:
        print row.elev, row.elevm
        row = cur.Next()
```

UpdateCursor example:

```
rows = GP.UpdateCursor("roads.shp")
row = rows.Next()
while row:
     row.buffer_distance = row.road_type * 100
     rows.UpdateRow(row)
     row = rows.Next()
del row, rows
```

Slide 60



Cursors & Feature Geometry

You can get shapes from your records with a searchcursor:

```
rows = gp.SearchCursor("mvalley_pts.shp")
```

Then you can process through your rows (features) and work with shapes (in this case, points):

```
row = rows.Next()
while row:
    pnt = row.shape.getpart()
    print row.Name, pnt.x, pnt.y
    row = rows.Next()
```

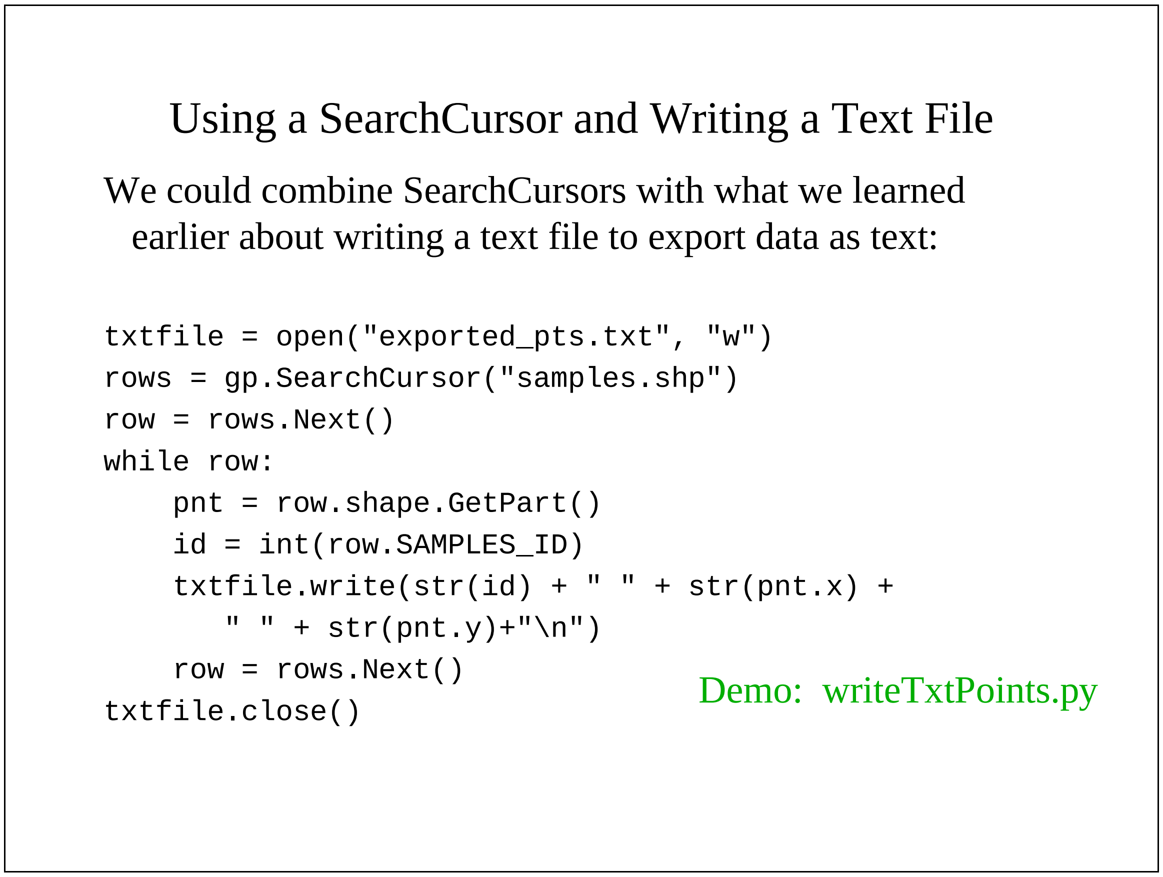
Demo: searchPts.py

The script above works with point features. (We should probably also include describe code to determine if a shapefile consists of points.) Point features have only one point per feature, but we need to use the **getpart** method to pull that geometry out of the shape, thus we use `pnt = row.shape.getpart()`.

If we have polyline features, it gets a bit more complicated in that we will have arrays of points. We can also have multi-part lines. The following code uses a search cursor to print out the coordinates of polylines in a shapefile:

```
rows = gp.SearchCursor("stream.shp")
row = rows.Next()
feat = row.shape
a = 0
while a < feat.PartCount:
  stArray = feat.GetPart(a)
  stArray.Reset
  pnt = stArray.Next()
  while pnt:
    print pnt.x, pnt.y
    pnt = stArray.Next()
  a = a + 1
row = rows.next()
```

Slide 61



Using a SearchCursor and Writing a Text File

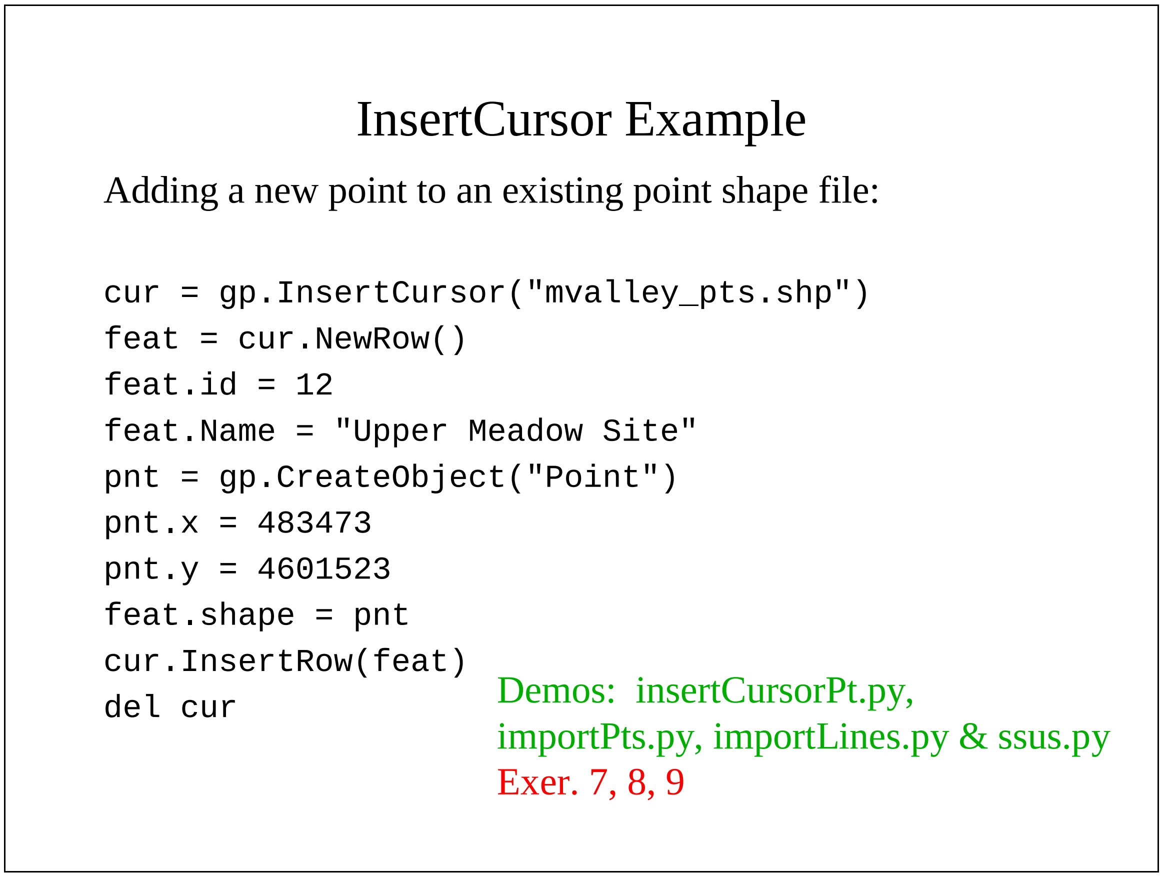
We could combine SearchCursors with what we learned earlier about writing a text file to export data as text:

```
txtfile = open("exported_pts.txt", "w")
rows = gp.SearchCursor("samples.shp")
row = rows.Next()
while row:
    pnt = row.shape.GetPart()
    id = int(row.SAMPLES_ID)
    txtfile.write(str(id) + " " + str(pnt.x) +
       " " + str(pnt.y)+"\n")
    row = rows.Next()
txtfile.close()
```

Demo: writeTxtPoints.py

The following code reads a point shapefile, and writes it out to a textfile as lines of id x y:

```
import win32com.client, os
gp = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")
gp.workspace = "d:/workspace/prog/marbles"
txtfile = "sampsout.txt"
fpath = gp.workspace + "/" + txtfile
if gp.Exists(fpath): os.remove(fpath)
txtfile = open(fpath, "w")
try:
    rows = gp.SearchCursor("samples.shp")
    row = rows.Next()
    while row:
        pnt = row.shape.GetPart()
        id = int(row.SAMPLES_ID)
        txtfile.write(str(id) + " " + str(pnt.x) + " " + str(pnt.y)+"\n")
        row = rows.Next()
    txtfile.close()
except:
    print gp.getmessages()
    txtfile.close()
```



InsertCursor Example

Adding a new point to an existing point shape file:

```
cur = gp.InsertCursor("mvalley_pts.shp")
feat = cur.NewRow()
feat.id = 12
feat.Name = "Upper Meadow Site"
pnt = gp.CreateObject("Point")
pnt.x = 483473
pnt.y = 4601523
feat.shape = pnt
cur.InsertRow(feat)
del cur
```

Demos: insertCursorPt.py, importPts.py, importLines.py & ssus.py
Exer. 7, 8, 9

For the above code to work, "mvalley_pts" would have to already exist as a point shapefile, and would need to have the text field "Name" in addition to the shape and id fields it has by default. It can already have records; this just adds another one. On its own, the above code doesn't do much; it just adds one point to a shapefile. But you may be able to see that adding in code to import a text file of coordinates might also work. Or you could do something like the following, which creates a new shapefile of 10 random points:

```
gp.CreateFeatureClass(gp.workspace, "randpts.shp", "Point", "pointemp.shp")
cur = gp.InsertCursor("randpts.shp")
for i in range(10):
        pnt = gp.CreateObject("Point")
        pnt.x = random.random() * 100
        pnt.y = random.random() * 100
        feat = cur.NewRow()
        feat.shape = pnt
        feat.id = i
        cur.InsertRow(feat)
del cur
```

Warning: InsertCursors can be very tricky, especially with the PythonWin debugger.