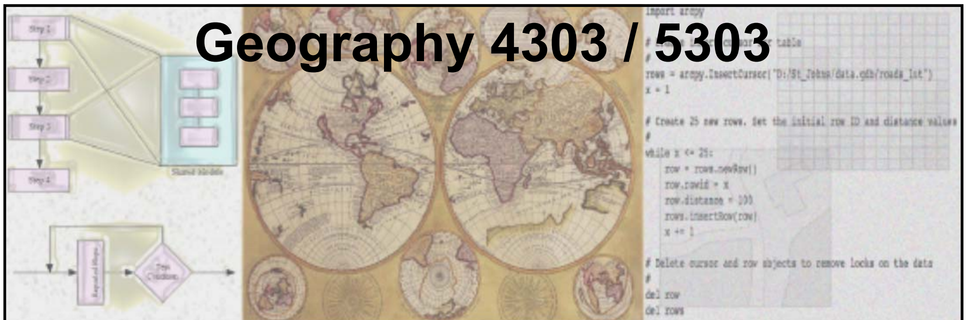
Geography 4303 / 5303



GIS PROGRAMMING FOR SPATIAL ANALYSIS

Class 1: Python & Programming Basics

Some Updates

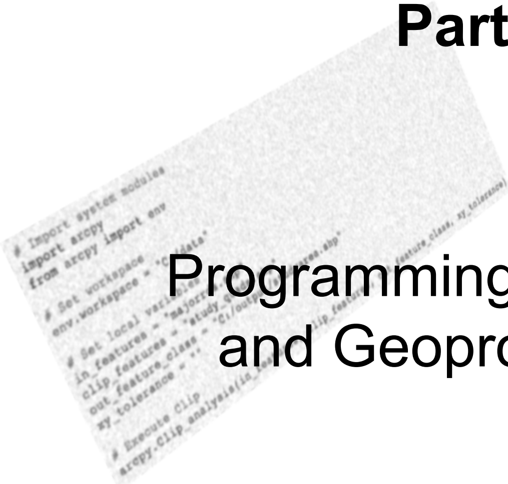
- Office hours – Kesda is very busy ...!
- Readings updated as we go
- Undergraduate lounge
- ArcGIS install and Py/Anaconda setup
- Using the instruction sheet
- Website, Z Drive and your backup
- Buff Ids

Today ‘s Outline

- **Part 1**: Programming in Python and Geoprocessing
- **Part 2**: Basic elements of (programming) logic and concepts applied
- **Part 3: Take-Home Notes** for Syntax, Python Basics & IDE

Learning Objectives

- Concepts of **execution for programming languages**
- Why **Python** is a good choice for GIS scripting and programming
- Basic **logic** elements and **structures** in programs



Part 1

Programming in Python and Geoprocessing

Let‘s start from the beginning

- What is a **program**?

  A **program** is a sequence of instructions which can be **interpreted** by the computer.

  The content of a program is called the **source code**.

  When the computer carries out the program it (the CPU) **executes** the program.
- **Programming** is the process of **writing**, **testing**, and **maintaining** the source code of computer programs which are written in a **programming language**.

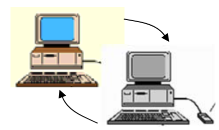

```
#The Ahoj-Svet-Prog
import string
var = “Svet”
print “Ahoj ”, var
```

Pin ex.

What's important for the "right" decision

A **programming language** should be:

-**Simple** to read / easy to learn
-**Affordable**
-**Supported**
-**Portable**: Platform independent
-Efficient / **interpreted** – just a minute ...
-**Object-oriented** – We will come back to that…
-**Embeddable/extensible** – control over & link btw. apps
-Easy to **debug** – tools for error search
-**Robust**



High-Level Languages – Three Execution Models

- **Interpreted**

  Read & executed directly (no compilation)

  Combined interpreting / compilation (compiled to **virtual machine** code & interpreted at runtime to **native** code (Py))

- **Compiled**

  Transforming into an **executable** form before running:

  (a) Compilation (**intermediate** representation such as **bytecode**) for later execution (Java)

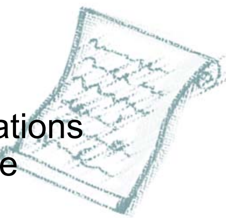
  (b) **Direct** Machine code generation (Fortran, C++, VB)

- **Translated**

  Translated into a low-level programming language using native code compilers (C)

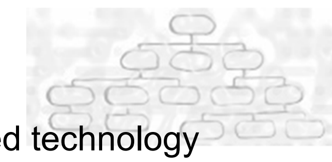
Spyder ex.

Programming or Scripting?



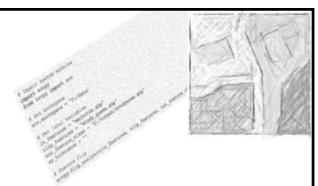
Scripting languages are linking applications using built-in higher level functions of the computer without consideration of raw resources (Python, Perl)

Object-Oriented or Not?



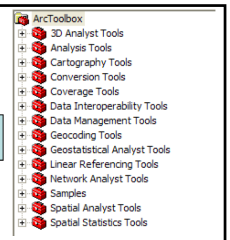
- Python supports object-oriented technology
- Python offers a full developing environment
- We will discuss object-orientation later and create and use Python objects (instances of classes)

Programming & GIS?

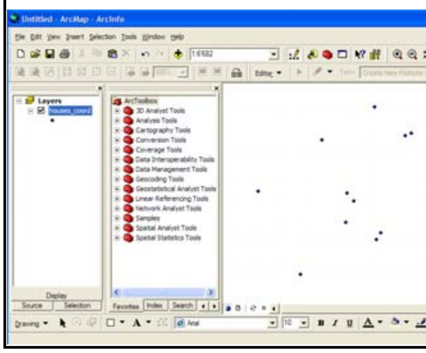


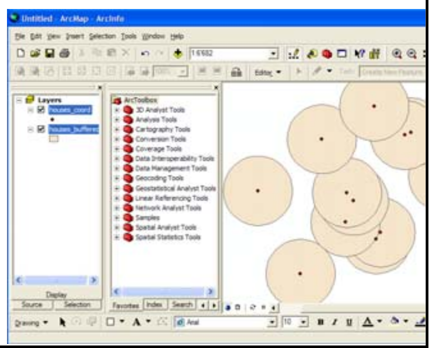
- GIS tasks: **Workflow**, **automation** & **efficiency**
- Software and method **development**
- **Problem-specific analysis** for creative solutions
- **Complexity** in combining **geography** and **programming** skills for solving spatial problems
- Interdisciplinary context, **creatively**!
- Exchange and code **distribution**

```python
import arcpy
from arcpy import env

# Set environment settings
env.workspace = "C:/data/Habitat_Analysis.gdb"

# Select suitable vegetation patches from all vegetation
veg = "vegtype"
suitableVeg = "C:/output/Output.gdb/suitable_vegetation"
whereClause = "HABITAT = 1"
arcpy.Select_analysis(veg, suitableVeg, whereClause)

# Buffer areas of impact around major roads
roads = "majorrds"
roadsBuffer = "C:/output/Output.gdb/buffer_output"
distanceField = "Distance"
sideType = "FULL"
endType = "ROUND"
dissolveType = "LIST"
dissolveField = "Distance"
arcpy.Buffer_analysis(roads, roadsBuffer, distanceField, sideType, endType, dissolveType, dissolveField)

# Erase areas of impact around major roads from the suitable vegetation patches
eraseOutput = "C:/output/Output.gdb/suitable_vegetation_minus_roads"
xyTol = "1 Meters"
arcpy.Erase_analysis(suitableVeg, roadsBuffer, eraseOutput, xyTol)
```

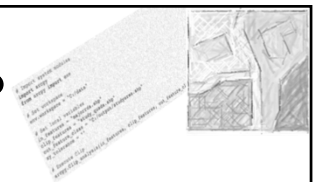


Choices & descisions for GIS programming?



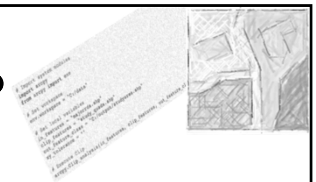


Why Python?



- **Readability**: Clean syntax, clear concepts
- Supports **object-oriented** programming
- Working with complex data structures
- **Integration** with C++, Fortran, Java
- **Free** with a widespread **community** & support
- Programming capabilities of a **complete developer language**
- **Portable:** Platform independent; operates on (UNIX, Linux, and Windows)

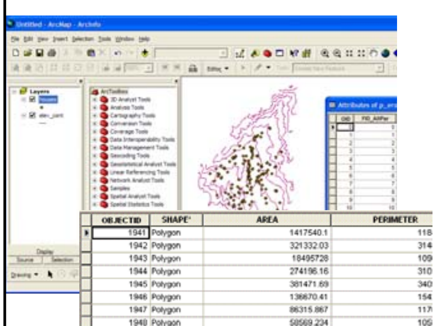
Why Python?



- **Powerful open source** tools available
- Other libraries for geospatial analysis and data management (ogr, gdal, **QGIS** Py Console)
- **ESRI supports** the use of Python scripting for Geoprocessing (ArcPy)
- Python (& Numpy) **bundled** w/ ArcGIS
- Access to the whole set of **Geoprocessing**

Clip example in Spyder

Geoprocessing framework



- Geoprocessing scripting supported by native "arcpy" module (ArcObjects component)
- GP supports the **COM** interface IDispatch: Only the GP-COM function needs to be written

COM - **C**omponent **O**bject **M**odel

```
# Import system modules
import arcpy
from arcpy import env

# Set workspace
env.workspace = "C:/data"

# Set local variables
in_features = "majorrds.shp"
clip_features = "study_quads.shp"
out_feature_class = "C:/output/studyarea.shp"
xy_tolerance = ""

# Execute Clip
arcpy.Clip_analysis(in_features, clip_features, out_feature_class, xy_tolerance)
```

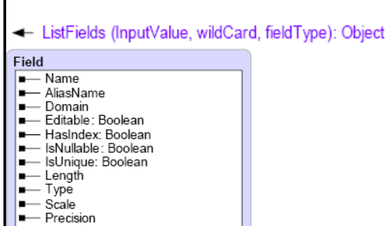
What you will learn

- How to get **access** to **spatial** & **non-spatial** data
- How to use **objects** (FieldLists, Geometry)
- … and their **methods** and **properties**
  - Creating/deleting, exploring (listing, sorting, describing) and modifying (editing)
  - Properties of geometry: # of points, coordinates, ids,…
- Working with Tables, Vector and Raster Data
- Develop complex Geoprocessing tasks

Work with Tables

- **Tables**: Collection of rows each with the same fields
- Can be linked to geographic data (**keys**)
- Attribute **data types**
- Output tables …

OBJECTID	SHAPE*	AREA	PERIMETER
1941	Polygon	1417540.1	118
1942	Polygon	321332.03	314
1943	Polygon	18495728	109
1944	Polygon	274196.16	310
1945	Polygon	381471.69	340
1946	Polygon	136670.41	154
1947	Polygon	86315.867	117
1948	Polygon	58569.234	105




```
import arcpy
from arcpy import env

# Set the current workspace
#
env.workspace = "C:/Data/MyData.gdb"

# Get the list of standalone tables in the geodatabase and
#   print to interactive window.
#
tableList = arcpy.ListTables()
for table in tableList:
    print table
```

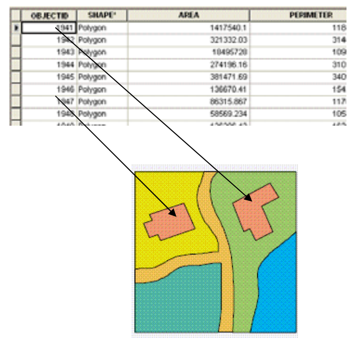
Work with Vector Data

- points, polylines, polygons
- Geometry
- Attributes

FeatureClass Properties

- FeatureType
- HasM: Boolean
- HasZ: Boolean
- HasSpatialIndex: Boolean
- RelationshipClassNames
- ShapeFieldName
- ShapeType
- TopologyName

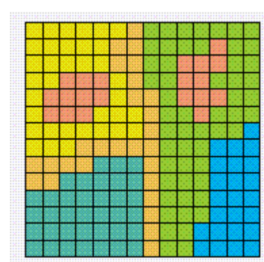
Table Properties
Dataset Properties



```
#create a feature class
arcpy.CreateFeatureclass_management(out_path, out_name, geometry_type)
```

Work with Raster Data

- Two-Dimensional **arrays** of equally-spaced cells
- Access to data structure & properties
- Single & multiple bands (RS imagery)



ListRasters (wildCard, rasterType): Object

Raster Dataset Properties

- BandCount
- CompressionType
- Format
- Permanent: Boolean
- SensorType

Raster Band Properties
Dataset Properties

```
import arcpy
from arcpy import env

# Set the current workspace
#
env.workspace = "C:/Data/DEMS"

# Get a list of ESRI GRIDs from the workspace and print
#
rasterList = arcpy.ListRasters("*", "GRID")
for raster in rasterList:
    print raster
```

Geoprocessing

- **Geoprocessing**: Carrying out operations on GIS datasets to produce a new dataset
- Geoprocessing tools in ArcGIS are accessed from ArcToolbox
- Based on a framework of **data transformation**

Tools and Scripts

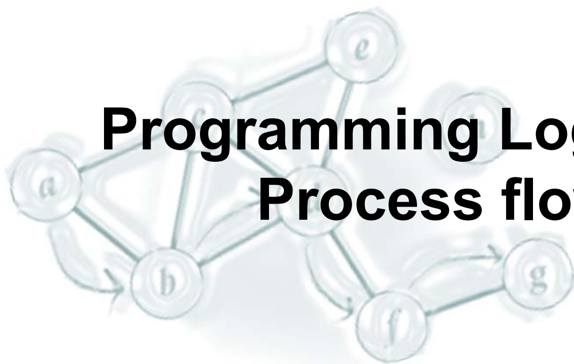
- Resource exchange between ArcToolbox and the Python environment
- **Export** tools & models into scripts
- **Import** scripts as executable tools (script tools)
- All tools in ArcToolbox are **methods** of the **Geoprocessor** object in Python!!!
- ModelBuilder for Geoprocessing workflows

Show: Export model to script, script tool, and tool export (script)

Summary Part 1

- Using Python as a scripting language for ArcGIS you can:
  - get **access** to the data properties in a programming environment
  - get access to **geoprocessing** tools
  - **objects**, their **methods** (exploring, editing or describing) and **properties** for data management
- Work with many different tools (non-ESRI) that use Python (open source tools and software e.g., QGIS)

Part 2

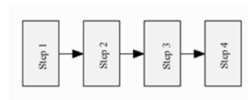


Programming Logic and Process flow

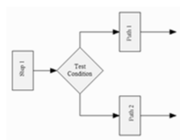
Dijkstra ‘s *Structured Programming* Concept

- “All programs could be structured in four possible ways“:

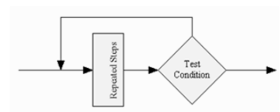
Sequences of instructions



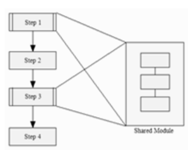
Branches



Loops



Modules



- Basis for **programming logic (controlling process flow)**

What else is needed...?

… to provide the program with contents:

- **Data** (type definition, parameterization)

```
myVar=5.0
```

- **Operations**: Actions performed on data (modify, add, multiply, compare etc.)

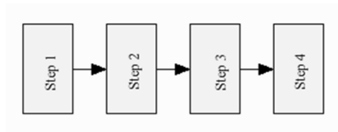
```
myResult=myVar+4.0
```

- **Dialog capabilities**: Input/Output (read, display, export)

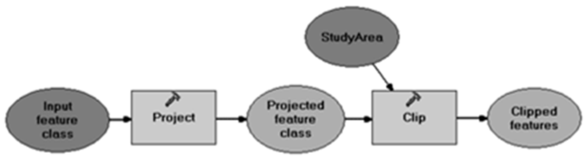
```
print “My result is”, myResult
```

Equation example

Sequence of instructions



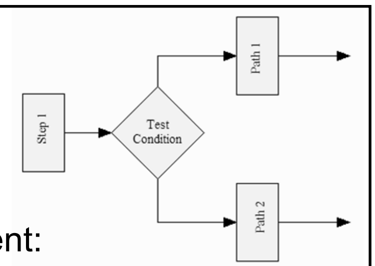
- Strict sequential flow
- Enforcement of (sub-)process **sequence**




```python
import arcpy
from arcpy import env

env.workspace = "Y://GIS3"

try:
    # Buffer roads based on a distance
    arcpy.Buffer_analysis("lyons_mrd.shp", "//results//roads_buffered.shp", "100", "", "ROUND")

    print "Buffer operation executed."

except:
    print "Try it again..."
```

Equation example + buff ex.

Branches

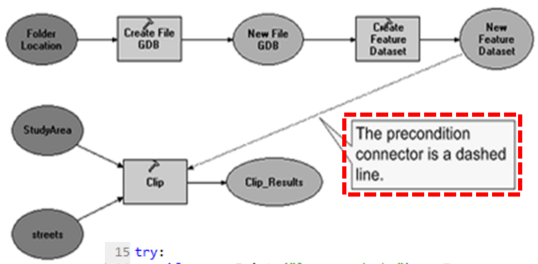


- **Conditional** construct/statement:
  - Program flow is dependent on the result of a test (**true/false**)
- Process **flow direction** can be modified, interrupted or continued based on required **preconditions**
  - count of …
  - existence of …
  - specificity of the model (e.g., distance > 0)

Branching: Testing for preconditions

if dataset exists **then**

… <execute the model>

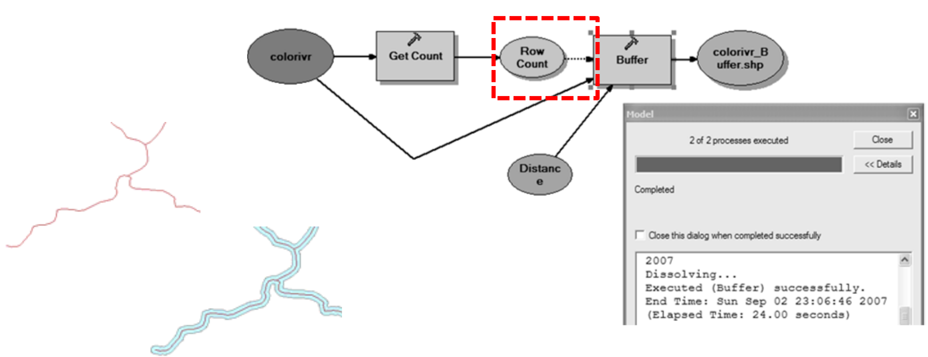



```
try:
    if arcpy.Exists("lyons_mrd.shp") == True
        # Buffer roads based on a distance
        arcpy.Buffer_analysis("lyons_mrd.shp", "//results//roads_buffered.shp", "100", "", "ROUND")

        print "Buffer operation executed."
    else:
        print "Dataset does not exist in directory."
except:
    print "Try it again..."
```

Branching: Testing for preconditions

if amount of objects > 0 **then**

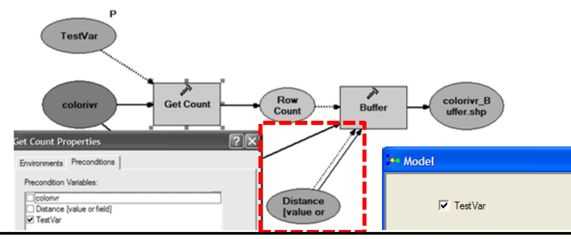
… <execute the model>



Branching: Testing for preconditions

Boolean or Integer variable “stand alone” defined as precondition

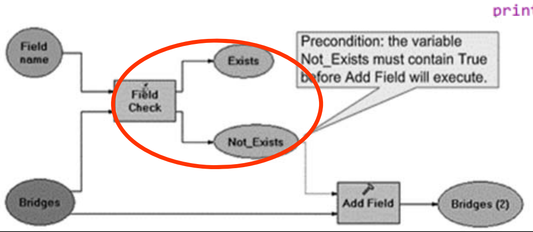
```
if var == true then
  … <execute the model>

if var == <value> then
  … <execute the model>
```



Branching

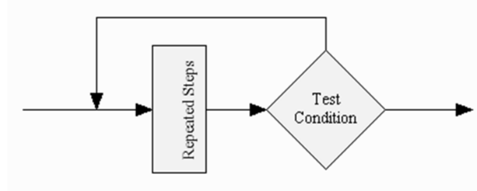
- Follows full `if <> then <> else` logic
- Extends the precondition function
- Alternative branches (“else”) to execute the model though first test omitted (false)

```
if firstTree == "Pine":
    print "A ", firstTree, " has been recorded."
    countConiferous+=1
elif firstTree == "Oak":
    print "A ", firstTree, " has been recorded."
    countDeciduous+=1
else:
    print "An unknown species has been encountered."
```



buff – branch ex.

Looping

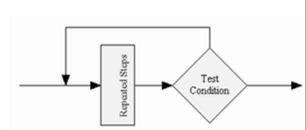


- Continuously repeat tests until some condition is reached to go on with program flow (batch processing)
- Repeating execution processes (**Looping**)
- Each execution of a process done with different data (some condition has to change)
- Sequences as “containers” (e.g., Lists)

Looping: Iteration using Lists

- **List variables** (any kind: datasets, values)
- All **subsequent processes** will execute once for each list value

```
myTrees = ["Pine","Oak","Pine","Oak"]

for tree in myTrees:
    if tree == "Pine":
        print "A ", tree, " has been recorded."
        countConiferous += 1
    elif tree == "Oak":
        print "A ", tree, " has been recorded."
        countDeciduous += 1
    else:
        print "No idea what tree this is."

```



Show list real time

Understanding Counted Loops

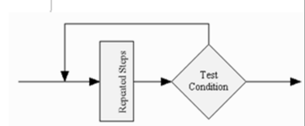
- **for** <initial element> **to** <final element>

  ... do something

  go to next element

- "**Counted loops**" have a known number of elements to work on:
  - "For all apples in my basket, I will remove the sticker"

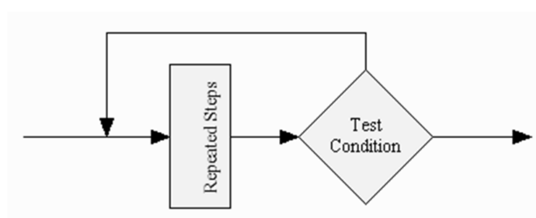
```
try:
    for a in range(100,200,20):
        arcpy.Buffer_analysis("lyons_mrd.shp", "//results//roads_buffered"+str(a)+".shp", str(a), "", "ROUND")

    print "Buffer operation executed."

except:
    print "Try it again..."
```



Iteration/Looping using Boolean Tests

while <something is true>

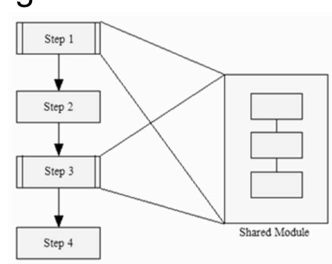
do <…some action>




```
try:
    b = 100

    while b < 200:
        arcpy.Buffer_analysis("lyons_mrd.shp", "//results//roads_buffered"+str(b)+".shp", str(b), "", "ROUND")
        b = b + 20

    print "Buffer operation executed."

except:
    print "Try it again..."
```

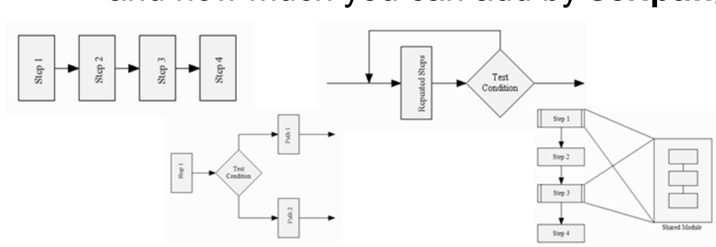
Modularity?

- Performing identical sequences of actions several times
- Actions are placed in a **module** (**sub-routine**, **procedure** or **function**)
- Exectued from within the main program
- We use modules from the beginning
- … and we will create new ones!



Summary Part 2

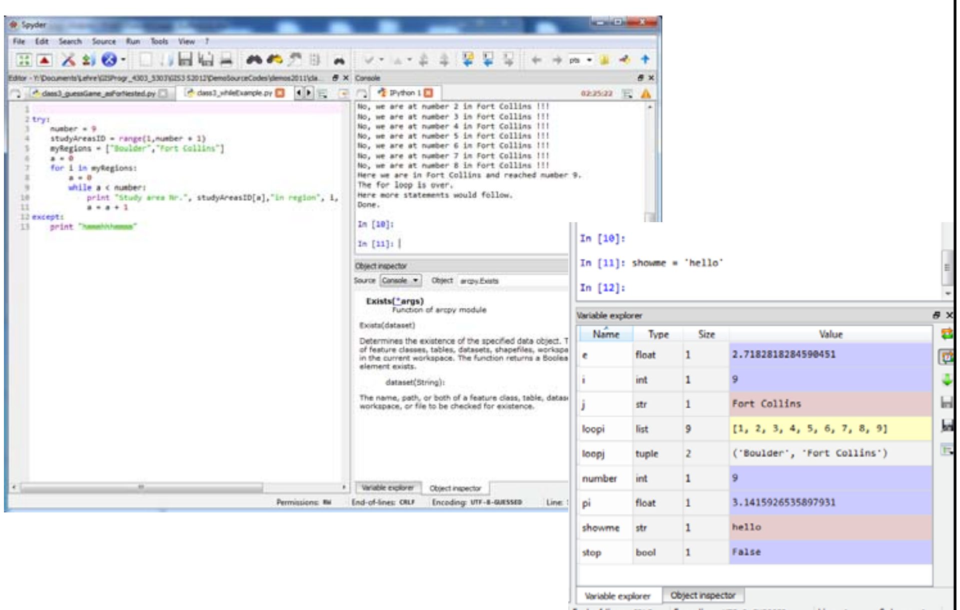
- We talked about **process flow** in much more detail and related to spatial modeling
- The basic **logic of geoprocessing** can be best explained by starting at graphical modeling
- You have a better idea of the **GP framework** and how much you can add by **scripting**



Part 3 – Notes for Take Home

IDE, Syntax and Python Basics

Spyder



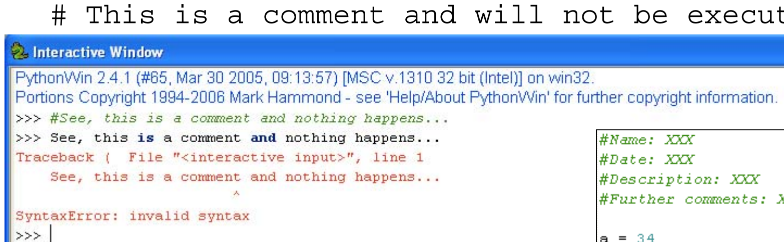
Integrated development environment (IDE)

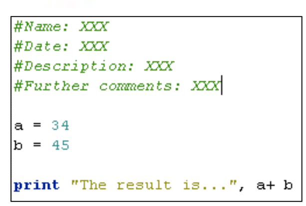
- An **IDE** offers you all components you need for development (syntax highlighting, debugging, browsing, **Intellisense - show this**)
- **Write** & **Save** code in Script Window
- **Testing** and Reporting in Interactive Window **(Ipython in Spyder)**
- **Running** and **Debugging** source code
- Menus and toolbars
- Execution by the Python **Interpreter (IPython)**

Python Basics: Comments

- Indicate **non-executable** lines of code
- Use comment flags (#) to document your work - this is important to understand later what you have been doing

```
# This is a comment and will not be executed
```





Python Basics: Line continuation

- Instructions continue in the next line if one of the following characters is used:

  Explicit Line Joining

  Back slash \

  Implicit Line Joining

  Parentheses ()

  brackets []

  braces { }

```
>>> print "This is a \
... Hello World Program"
This is a Hello World Program
>>> myList = ["cold", "warm",
... "warmer", "hot"]
>>> myList
['cold', 'warm', 'warmer', 'hot']
>>>
```

Python Basics: Logical and Physical Lines

- Normally, one **logical line** corresponds to one **physical line** in a program
- If more than one physical lines are needed to write a logical line we use **line continuation**
- Theoretically, **two logical lines** can be written in **one physical** line using semicolons

```
var = 5
print var
```

  can be written as

```
var = 5; print var
```

Python Basics: Indentation

- **Automatic indentation ("white space")**
  Indentation is enforced by Python
  Loops and decisions are structured based on **indentation** and **blocks** of code

```
def AddMsgAndPrint(msg, severity=0):
    # Adds a Message to the geoprocessor (in case this is run as a tool)
    # and also prints the message to the screen
    print msg

    #Split the message on \n first, so that if it's multiple lines, a GPMessage
    try:
        for string in msg.split('\n'):
            #Add a geoprocessing message (in case this is run as a tool)
            if severity == 0:
                GP.AddMessage(string)
            elif severity == 1:
                GP.AddWarning(string)
            elif severity == 2:
                GP.AddError(string)
    except:
        pass
```

Case sensitivity

- The Python interpreter is **case sensitive**:
  `var` is not the same as **`Var`**

```
>>> a = "c:\\TempDir"
>>> A
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
NameError: name 'A' is not defined
>>> a
'c:\\TempDir'
```

- In contrast, **path names** are **not**
  `"C:\\TEMP" = "c:\\temp"`
- And, **properties** and **methods** of the **Geoprocessing** object are case sensitive
  `arcpy.Buffer() != arcpy.buffer()`

Variables

- **Direct** (internal) assignment of variable types – **no declaration** or data type definition required

  `varInt = 23        varStr = “Name”      varDouble = 23.456`
- **Case sensitivity**

  `var = 23` and `Var = “Cat”` are different variables
- Types that variables can hold:
  strings, numbers, lists, files, objects

  You can create your own types using **classes …**

  We will talk about variable types in more detail
- Type **conversion** functions:
  `int(), float(), str())`

Naming Conventions for Variables

- **Descriptive** variable names
- Avoidance of **special** characters (%, $)
- Start with **lower case** and use **capitals** for each successive word
- **Acronyms** for meaning of a variable

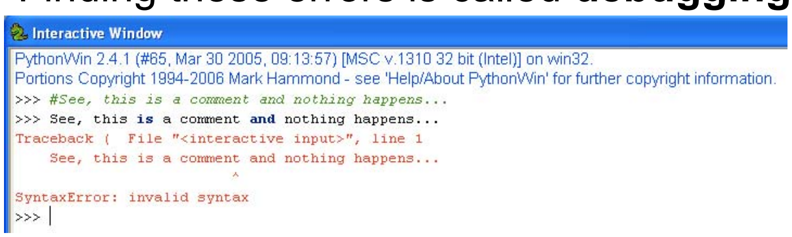
```
fcForest = “c:\TempDir\forest.shp”
outputFC = “c:\TempDir\Demography.shp”
```

How to Run now a Program

- Write the parts of the program in the **script window** of PythonWin, successively
- **Test** critical parts in your **interactive** window
- Save it to your hard disk with the extension ***.py** (for all Python programs)
- Run the **interpreter** or use the IDLE/ Python Win/ Spyder to run your program

Error Messages

- **Syntax** errors (writing errors)
- **Runtime** errors (illegal operations)
- **Semantic** errors (working properly but wrong output)
- Finding these errors is called **debugging**



To Get Help

- For quick information about functions or statements you can use the **help functionality** in the interactive window

  help(string)

  help(int)
- Python Online Documentation

Summary Part 3

- Several IDEs are available; you'll work with the Spyder environment
- You have seen some first **examples** and you will use this environment in your labs
- This class covered the **basic rules** you have to be aware of for your programming work
- These rules are important for starting with any **language** or any **environment** (but they can vary)

Next time …

- We will talk about **data types** in Python in general to shed light on some central properties
- We will discuss primitive data types, and complex data structures such as **collections** and **sequences**
- We will see some examples of how to use **Lists**, **Tuples**, **Strings** and **Dictionaries** as well as their methods
- We will touch base on **Python** and **Geoprocessing**