GEO 6533
Midterm
Review Material

Midterm

- Use these slides for your detailed exam preparation and review
- Exam will be two parts: Theoretical and practical
- 95% of the questions will be drawn from this review ppt.
- Review the python exercises we did in class (this includes codes in BB)
- Practical part will include using IDLE to write some code.
- You will first have to finish the theoretical part (01H:00m), then start the practical (01H:30m).

- Good luck to you

ArcToolbox 

- In any **ArcGIS for Desktop** application, you open the **ArcToolbox** window with the **Show/Hide ArcToolbox Window button** found on the standard toolbar or by clicking Geoprocessing > ArcToolbox.

- **Toolboxes have a name, label, and alias property**. The name and label properties are to support different languages, and **the alias property is used in scripting to uniquely identify a tool and its toolbox**.

- **Tools have a name and label property. The tool name is used in scripting and cannot contain spaces**.

- Toolboxes can be a file (**.tbx**) in a folder or an item in a geodatabase.

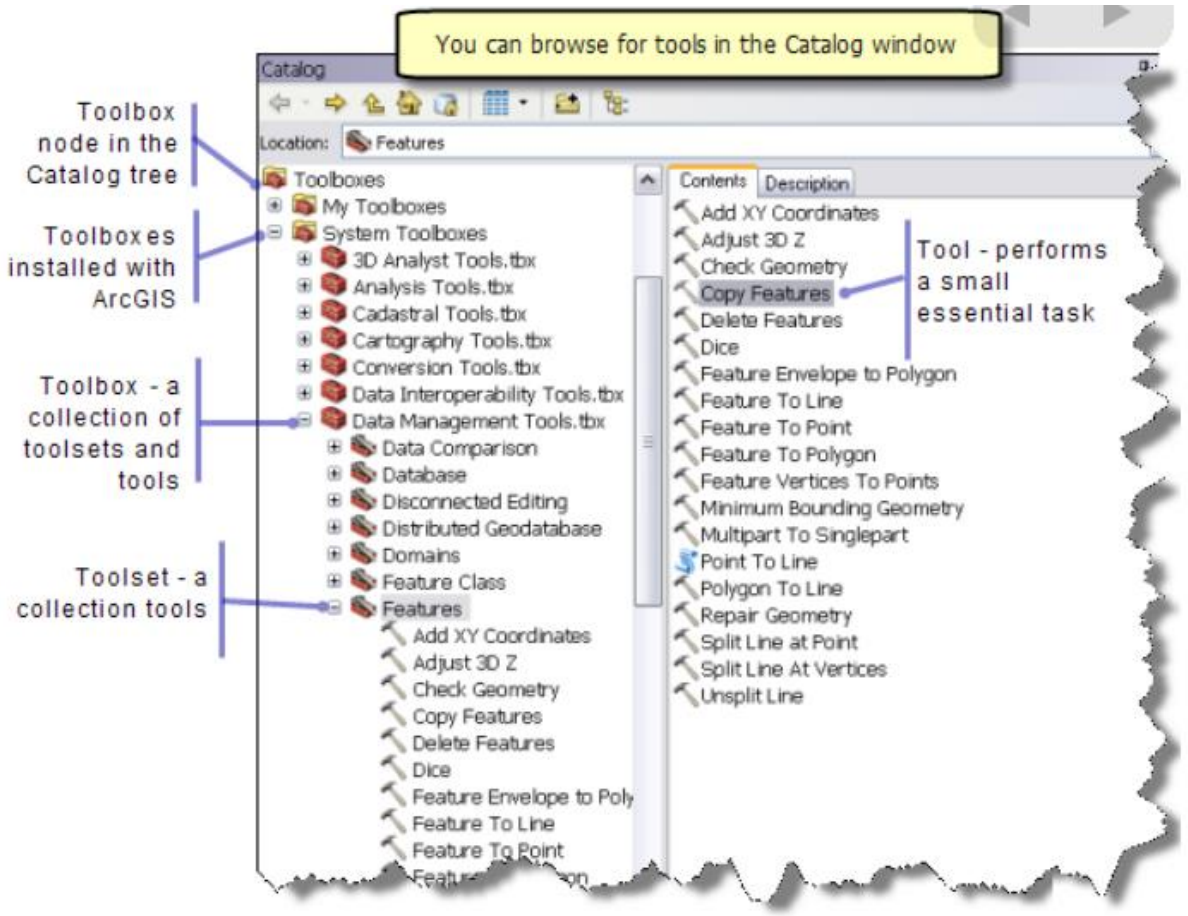
- A toolbox residing in a geodatabase has a different internal format than a toolbox residing in a system folder, and you cannot copy a toolbox from one format to another.

- **Python toolboxes are geoprocessing toolboxes that are created entirely in Python**. 

- You cannot add tools to a Python toolbox as you can with custom toolboxes.

ArcToolbox: Types of Tools in Arcgis

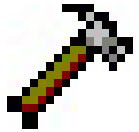
- System tool 
- Model Tool 
- Script Tool 
- Custom Tool 



Toolbox Aliases

System toolbox	Alias
3D Analyst	3d
Analysis	**analysis**
Cartography	cartography
Conversion	conversion
Coverage	arc
Data Interoperability	interop
Data Management	**management**
Editing	edit
Geocoding	geocoding
Geostatistical Analyst	**ga**
Linear Referencing	lr
Multidimension	md
Network Analyst	na
Parcel Fabric	fabric
Samples	samples
Schematics	schematics
Server	server
Spatial Analyst	**sa**
Spatial Statistics	stats
Tracking Analyst	ta

System Tools

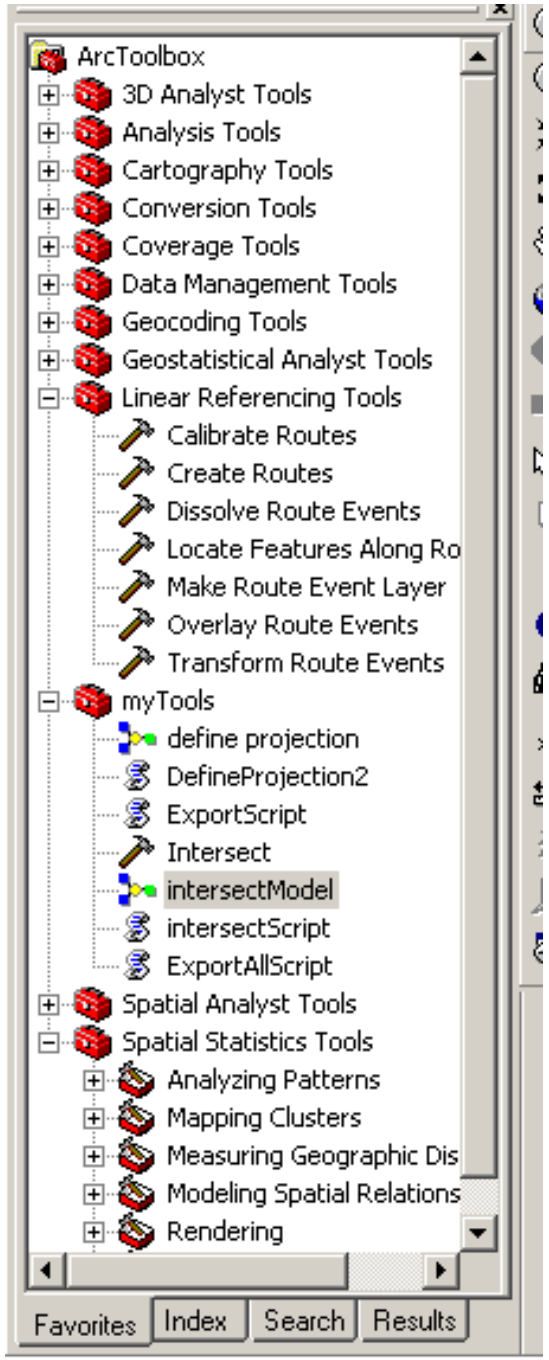


These tools are installed and registered on your system.

Usually, these tools are installed and registered when you install ArcGIS.

Although third-party developers can also create and register system tools.

System tools are sometimes called **function tools** by developers.



Geoprocesing in ArcGIS

- Geoprocessing is one of the most powerful components of a GIS. In ArcGIS® Desktop, you are provided with a framework for addressing geoprocessing tasks, which includes an extensive list of geoprocessing tools organized within a set of toolboxes.

- You can employ the tools directly or chain them together to model a particular workflow. You can put geoprocessing tools to work in custom scripts and you can create your own tools and toolboxes.

- The geoprocessing framework also provides functionality for organizing and managing your work environment, performing simple and complex analyses, and making your custom tools usable by others.

Geoprocesing in ArcGIS

- Three ways to **automate** geoprocessing tasks in ArcGIS.
- These three options differ in the amount of skill required to produce the automated solution and in the range of scenarios that each can address.
- **Model Builder: Model Builder is an interactive program that allows the user to "chain" tools together, using the output of one tool as input in another**
- **Scripts: A script is a program that executes a sequential procedure of steps.** Within a script, you can run GIS tools individually or chain them together.
- **ArcObjects: The programming building blocks used by Esri's own developers to produce the ArcGIS desktop products**.
- With ArcObjects, it is possible to customize the user interface to include specific commands and tools that either go outside the abilities of the out-of-the-box **ArcGIS tools** or modify them to work in a more focused way.

Geopreocessing Framework

- 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.

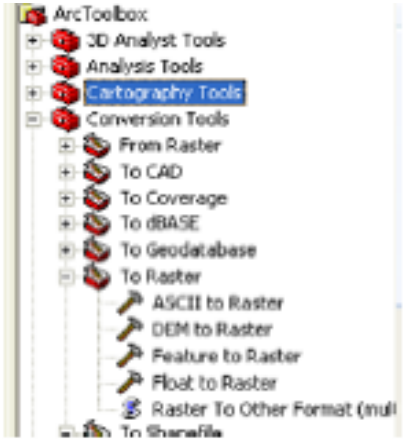
(2) ModelBuilder. You can run tools by dragging them in from ArcToolbox, and process input data sets to create outputs. Can be complex, can run scripts and other models in addition to system tools.

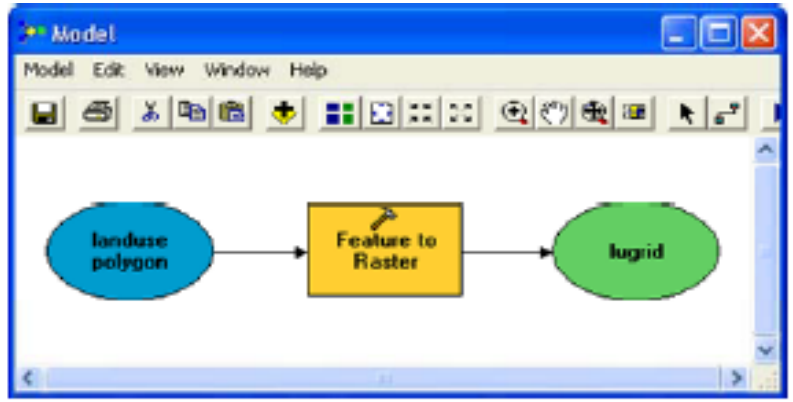
(3) Command line. Can run tools by typing them in, using command line syntax. Note: this command window and these commands are not the same as classic ArcInfo command window and commands.

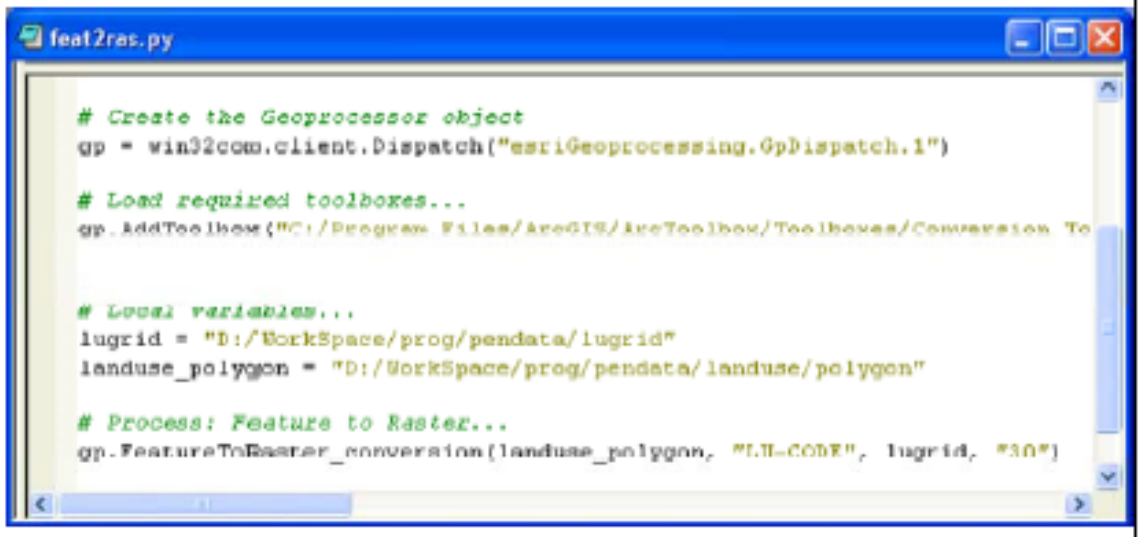
(4) Scripts. Primarily Python scripts, letting you use programming structures like loops and process conditionals either not available or difficult to use in ModelBuilder.

Geopreocessing Framework

ArcToolbox
ModelBuilder
Command Line
Scripting

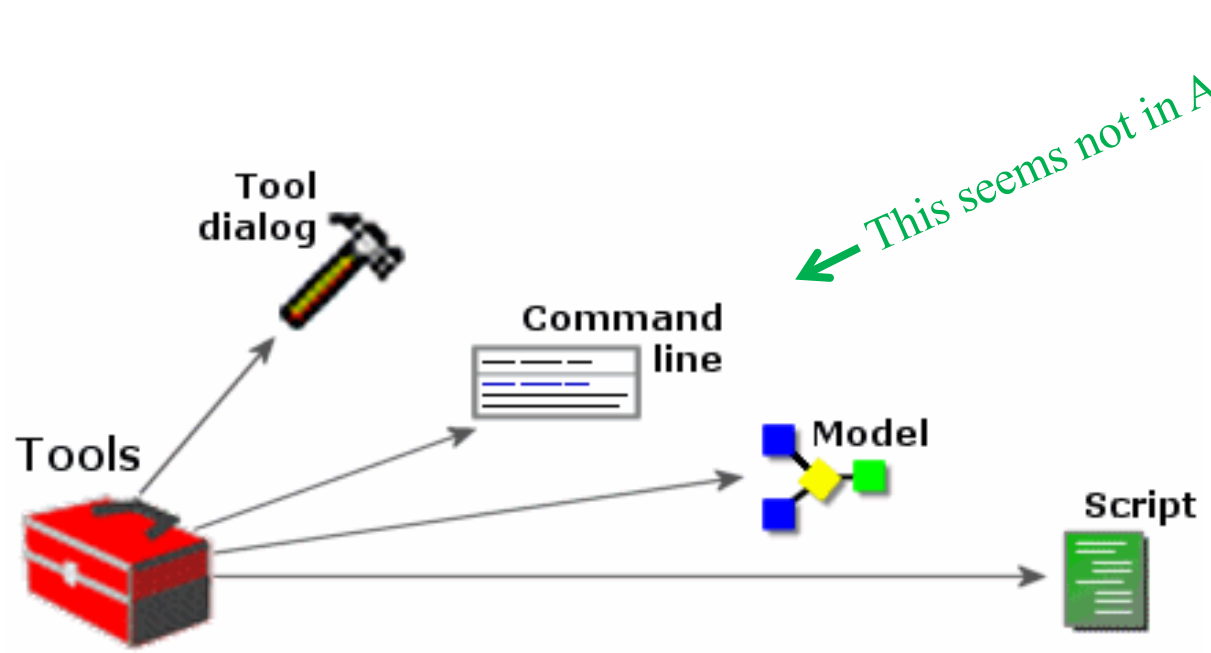






Geoprocesing in ArcGIS

- The most important thing to understand about geoprocessing in ArcGIS Desktop is that all geoprocessing operations involve the use of tools.



The ArcGIS geoprocessing framework provides multiple ways of working.

Automation Benefits

There are several major benefits to automating tasks like this:

- Automation makes work **easier**. Once you automate a process, you don't have to put in as much effort remembering which tools to use or the proper sequence in which they should be run.
- Automation makes work **faster**. A computer can open and execute tools in sequence much faster than you can accomplish the same task by pointing and clicking.
- Automation makes work **more accurate**. Any time you perform a manual task on a computer, there is a chance for error.
- The chance multiplies with the number and complexity of the steps in your analysis.
- In contrast, once an automated task is configured, a computer can be trusted to perform the same sequence of steps every time.

Model Builder: Model Elements

- **Model canvas**

The model canvas is the white empty space in a model.

- **Model diagram**

The model diagram is the appearance and layout of the tools and variables connected together in a model.

- **Model elements**

There are three main types of model elements: tools, variables, and connectors.

- **Tools:** Geoprocessing tools are the basic building blocks of workflows in a model. Tools perform various operations on geographic or tabular data. When tools are added to a model, they become model elements.
- **Variables:** Variables are elements in a model that hold a value or a reference to data stored on disk. There are two types of variables:
  - **Data:** Data variables are model elements that contain descriptive information about data stored on disk. Properties of data that are described in a data variable include field information, spatial reference, and path.
  - **Values:** Value variables are values such as strings, numbers, Booleans (true/false values), spatial references, linear units, or extents. Value variables contain anything but references to data stored on disk.
- **Connectors:** Connectors connect data and values to tools. The connector arrows show the direction of processing. There are four types of connectors:
  - **Data:** Data connectors connect data and value variables to tools.
  - **Environment:** Environment connectors connect a variable containing an environment setting (data or value) to a tool. When the tool is executed, it will use the environment setting.
  - **Precondition:** Precondition connectors connect a variable to a tool. The tool will execute only after the contents of the precondition variable are created.
  - **Feedback:** Feedback connectors connect the output of a tool back into the same tool as input.

Model Builder: Intermediate data

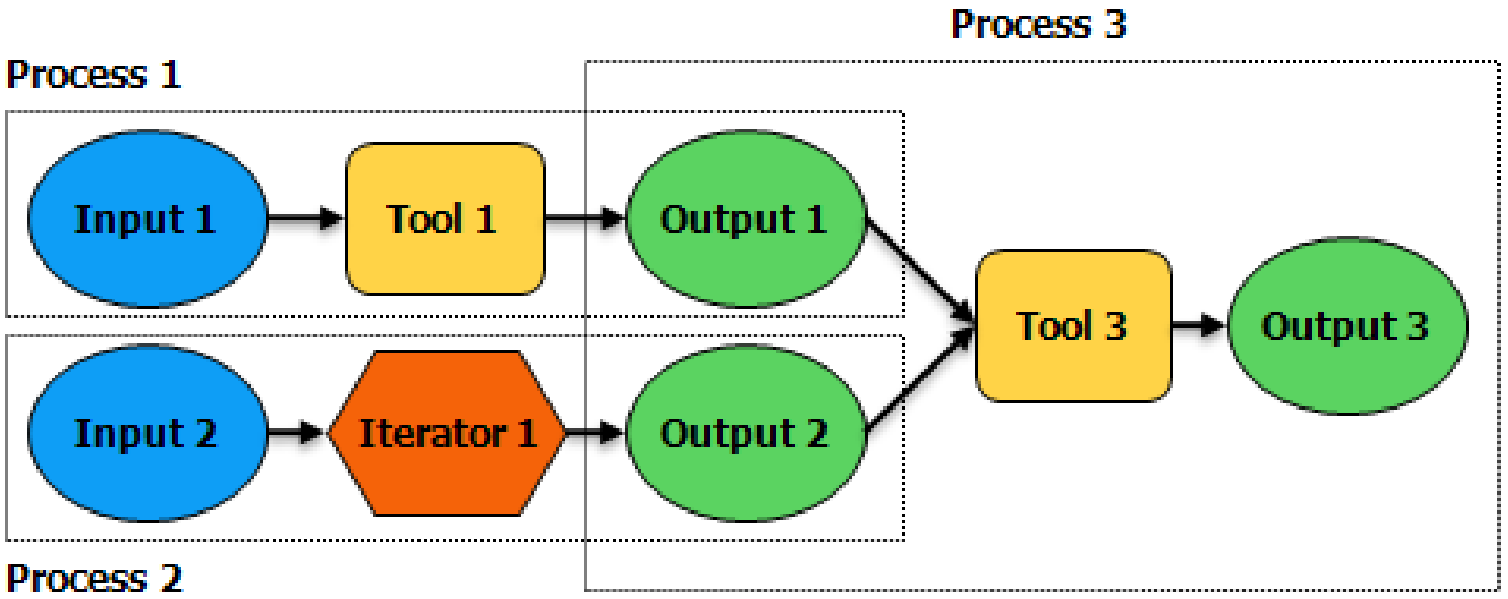
- Output data is created for each process in the model
- Some of this output data is only created as a middle step to connect to other processes that will create the final output
- By default intermediate data is not deleted

- **Model Validation: Refers to the process of making sure all model variables (data or values) are valid.**

- **It's a check process**

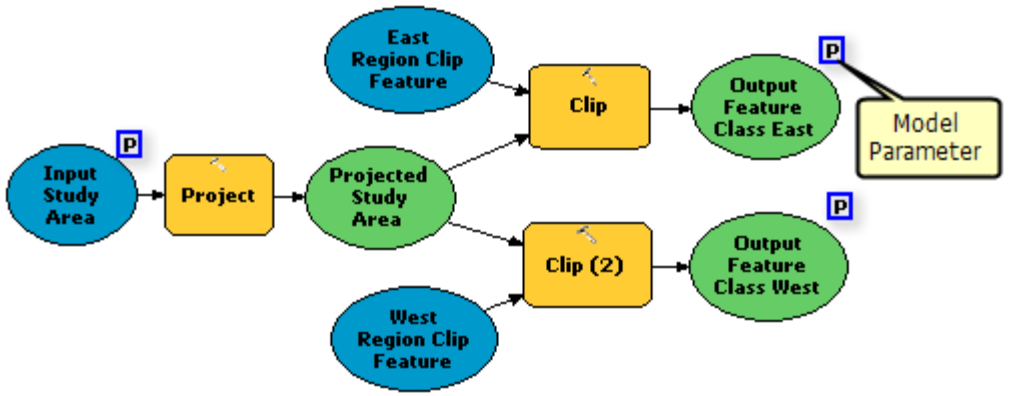
Model Builder: Model Process

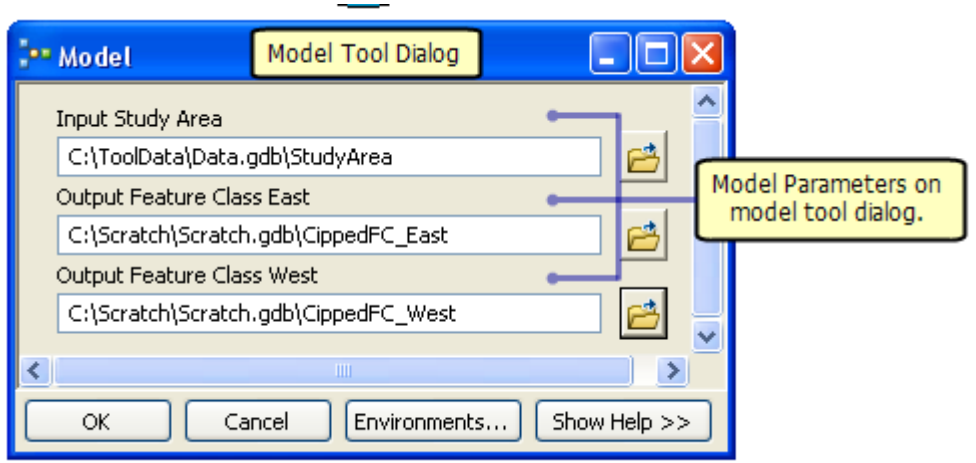
- **A model process consists of a tool and all variables connected to it.**
- **Connector lines indicate the sequence of processing.**
- Many processes can be chained together to create a larger process.



Model Builder: Model Parameters

- **Model parameters are the parameters that appear on the model tool dialog box.**
- Any variable in the model can be made a model parameter.





Model Builder: Workspace Environments

- There are three workspaces that can be used in ModelBuilder to simplify model data management:

- **Current:** Tools that honor the Current Workspace environment setting use the workspace specified as the default location for geoprocessing tool inputs and outputs.

- **Scratch:**Tools that honor the Scratch Workspace environment setting use the specified location as the default workspace for output datasets.

- The Scratch Workspace is intended for output data you do not wish to maintain.

- **In-memory:** The in-memory workspace is a temporary workspace where geoprocessing outputs can be written to the system memory.

Learn more about working with the current and scratch workspace environments in ModelBuilder
Learn more about working with the in-memory workspace in ModelBuilder

Programming or Scripting

- Programming Language :
  - Used to build sophisticated applications
- Scripting Language:
  - Used to automate tasks
- Programming: allows to build components from scratch, as well as the application that incorporates these components.
  - C++, Java, Visual Basic, .NET Platforms
- Scripting: is a programming task that allows you to connect diverse existing components to accomplish a new task.

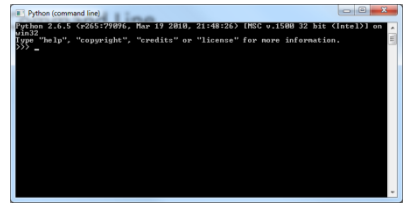
  Python , Perl, PHP, Ruby, JavaScript….

Scripting

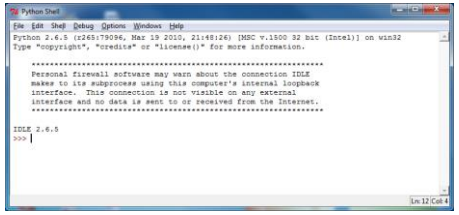
- **Scripting** is an efficient method of automating geoprocessing tasks.
- **Script is a text file that contains instructions for geoprocessing written as lines of code.**
- Scripting allows the execution of simple processes (a single tool) or complex processes (piggybacked, multi-tool tasks with validation).
- Scripts are recyclable, meaning they can be data nonspecific and used again.
- **Script: is a set of computing instructions, usually stored in a file and interpreted at run time.**

Editors

- Plain text documents with a specific file extension.
- Command Line:
  - Start>All programs> ArcGIS> Python 2.7> Python (command Line)

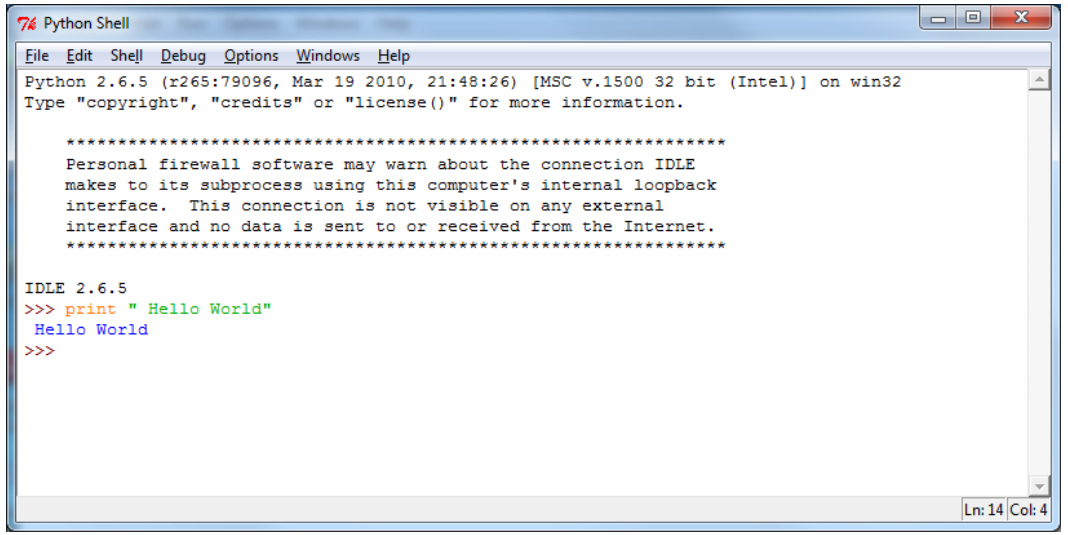


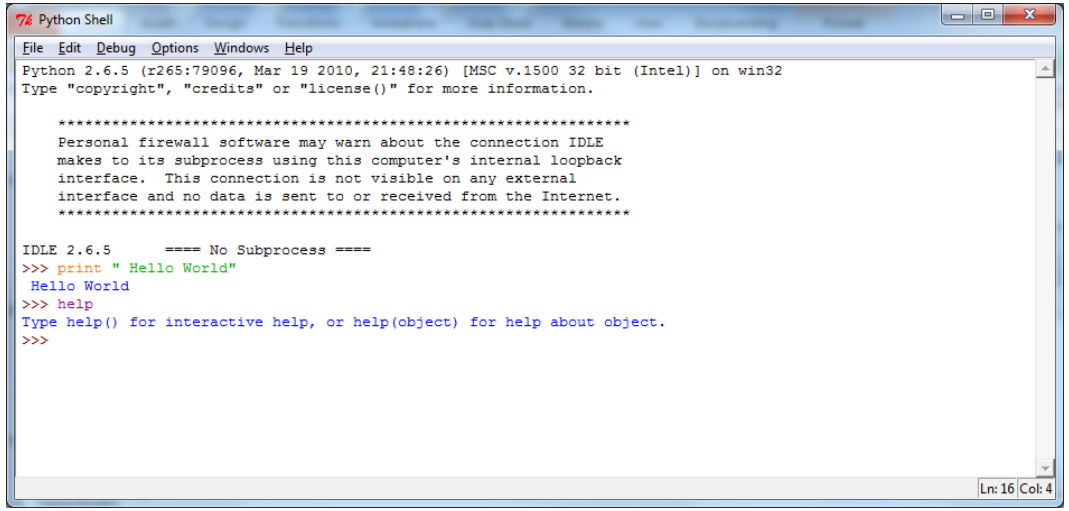
- Python Editors: with menu driven interfaces and tools for organizing and debugging code.
- Integrated Development Interfaces (IDE)
- Open source or Commercial IDEs:
- DrPython, Eclipse,NetBeans, IDLE,
- PyScripter, Pythonwin, Python Toolkit (PTK)..



IDLE

- >>> Prompt of the
- Interactive python interpreter.
- Syntax Highlighting
- Hello.py
- Extension .py





Geoprocesing with Python

- Python was introduced to the ArcGIS community at ArcGIS 9.0. Since then, it has been accepted as the scripting language of choice for geoprocessing users and continues to grow.
- Each release has furthered the Python experience, providing you with more capabilities and a richer, more Python-friendly experience.

- ESRI has fully embraced Python for ArcGIS and sees Python as the language that fulfills the needs of our user community.
- Here are just some of the advantages of Python:

  Easy to learn and excellent for beginners, yet superb for experts

  Highly scalable, suitable for large projects or small one-off programs known as scripts

  Portable, cross-platform

  Embeddable (making ArcGIS scriptable)

  Stable and mature

  A large user community

- **Python extends across ArcGIS** and becomes the language for data analysis, data conversion, data management, and map automation, helping increase productivity.

Why Python ?

www.python.org

- Simple and Easy to Learn Language
- Free and Open Source Software (**FOSS**)
- Cross Platform
- Large User Community
- Interpreted Language (no compilation)
- Object Oriented
- It is both a scripting and a programming language.

Scripting for ArcGIS

- ArcGIS 9x and up: **Python**, **VBScript**, **Jscript**..
- ArcGIS is COM compliant.
- Component Object Model (COM):
  - manages all the geoprocessing functions available within **ArcGIS**.
  - It is an object that provides a single access point and environment for the execution of any geoprocessing tool in ArcGIS, including extensions.
    - Using the native arcgis scripting module, depending on version of software (Arcgis in your computer)
- ArcGIS 10x: VBA is not installed by default anymore.
- **Python is directly embedded in many ArcGIS tools.**
- **Python is directly installed with ArcGIS, be careful of version compatibility if you install your own Python.**

Python Vocabulary in ArcGIS

- Python
- PythonWin
- ArcPy
- ArcPy Modules
- ArcPy Classes
- ArcPy Functions
- Stand-alone Python Script
- Python Script Tool
- Python Window

Geoprocessing Framework

- Collection of Tools
- Methods, Search window, Catalog window..
- Tool Dialog Boxes
- Model Builder
- Python Window
- Results Window
- Methods for creating Python scripts using existing tools

Geoprocessing and ArcObjects

- ESRI's Extensive library of basic **programming objects** created to develop ArcGIS software.
- Available through the ArcObejcts.NET Software Development KIT (SDK) and the ArcObjects Java SDK
- Can be used to create new applications or to enhance the functionality of ArcGIS applications.
- ESRI software developers use ArcObjects to create most of the tools in **ArcGIS** as well as to build the **geoprocesing framework**.
- ArcObjects can be used with system programming languages:
  - C++, VB.NET, C#.NET
- **ArcObjects is used to extend ArcGIS through new behaviors and to write stand alone applications that build on functionality of ArcGIS**
  - Creating new user interfaces and adding new behavior to feature classes
- **The Geoprocesing framework is used to run existing tools and to create new tools (models and scripts) that automate tasks withing the existing functionality of ArcGIS.**
  - ArcGIS 10 x and higher introduced the **desktop add-in model** from customizing and extending ArGIS Desktop applications
  - Python can **Author** desktop add-ins
  - Python add-ins can be used for some of the same things that were only possible using **ArcObjects**

Geoprocessing and ArcObjects

- **ArcObjects is the extensive library of low-level programming objects delivered as part of the ArcGIS Software Development Kit (SDK).**

- Developers use ArcObjects to build new applications or extend the existing functionality of ArcGIS applications. (For the record, most system tools and the entire geoprocessing framework were all built using ArcObjects.) Like geoprocessing, the ArcObjects SDK can be used to create new software.

- The ArcObjects SDK and geoprocessing are complementary.
- **As a general statement, ArcObjects is used to extend ArcGIS with new behavior, while geoprocessing is designed to automate tasks.**

- You use ArcObjects to do things like add new user interfaces, add custom behavior to feature classes, or create a special cartographic renderer. Geoprocessing is used to create software (models and scripts) that automates tasks within the confines of a well-behaved framework.

- **ArcObjects is meant to be used with a system programming language, where the programmer needs to access low-level primitives to implement complex logic and algorithms.** This is why

- **ArcObjects contains thousands of different objects and requests—to allow the programmer the fine degree of control they require**.
- Because ArcObjects is used in concert with a system programming language, it requires a good deal of programming knowledge—much more than geoprocessing, with its models and scripts.

- Conversely, geoprocessing is a universal capability that can be used and deployed by all GIS users to automate their work, build repeatable and well-defined methods and procedures, and model important geographic processes.

Paths in the Python Window

- Backslash (\) is a reserved character for Python, \n ; \t

```
import arcpy
arcpy.GetCount_management("c:/temp/streams.shp")
arcpy.GetCount_management("c:\\temp\\streams.shp")
arcpy.GetCount_management(r"c:\temp\streams.shp")
```

Python window keyboard shortcuts

F1	Shows the help for the current cursor location.
F2	Checks the syntax of the current line (or code block if in multiple line mode). Any errors will be shown in the Help pane.
SHIFT or CTRL+ENTER	Enters multiple line mode. To exit multiple line mode (execute the code block), press the ENTER key on the last line.
Up / Down	Access previously entered commands on the last line.
Right-click	Access additional options.

Python Window

Default text colors and their meanings

Color	Meaning
Black	**Normal informational messages.**
Red	**Error message. Results were not created.**
Orange	**Warning message. Results may not be what you expect.**

Tool parameters can be either **required** or **optional**.

Optional parameters are surrounded by braces { }; required parameters are not.

Parameter Type	Symbol	Meaning
Required		Required parameter. These parameters are always the first parameters in the command. You must provide a value for required parameters.
Optional	{ }	Optional parameter. These parameters always follow the required parameters. If you don't enter a value for an optional parameter, the default value is calculated and used. A parameter's default value can be found in the tool's help.

Python Window: Setting Tool Parameters

- A tool parameter can accept a single value or many values, depending on the parameter. When multiple values are acceptable, the **parameter value can be specified as a Python list**.

- The Delete Field tool accepts multiple fields for deletion.
- To delete multiple fields using Delete Field, enter the field names as strings within a Python list.

```
# Use empty strings to skip optional parameters
arcpy.DeleteField_management("c:/base/rivers.shp", ["Type", "Turbidity", "Depth"])
```

Python Window: Parameters Conventions

- **Parameter names** for all **input datasets** are prefixed with in_, and **output datasets** are prefixed with out_.

- The input dataset is usually the **first parameter**, and the output dataset is usually the **last required parameter**.

- Other required parameters are placed between the input and output datasets.

- **Optional parameters always follow the required parameters**.

Python Window: Multiple Parameters Tool

```
# Use empty strings to skip optional parameters
arcpy.AddField_management("c:/data/streets.shp", "Address",
"TEXT", "", "", 120)

# Use the # sign to skip optional arguments
arcpy.AddField_management("c:/data/streets.shp", "Address",
"TEXT", "#", "#", 120)

# Use the parameter name to bypass unused optional arguments
arcpy.AddField_management("c:/data/streets.shp", "Address",
"TEXT", field_length)
```

Python Window: Setting Environments

In ArcPy, **geoprocessing environments** are organized as properties under the ArcPy class env.

In the example below, several environment values are printed to the display, then set to new values.

```
>>> print arcpy.env.overwriteOutput
True

>>> print arcpy.env.workspace
None

>>> arcpy.env.overwriteOutput = False
>>> arcpy.env.workspace = "c:/temp"
>>> print arcpy.env.overwriteOutput
False

>>> print arcpy.env.workspace
c:/temp

>>>
```

The ArcPy function ResetEnvironments can be used to restore the default environment values.

```
>>> arcpy.ResetEnvironments()
>>>
```

The **ArcPy function** ListEnvironments can be used to create a list of all geoprocessing environments.

```
>>> environments = arcpy.ListEnvironments()
... for environment in environments:
...     envSetting = eval("arcpy.env." + environment)
...     print "%-30s: %s" % (environment, envSetting)
...
newPrecision                   : SINGLE

autoCommit                     : 1000
```

Python Keywords

```
import  keyword
print keyword.kwlist

['and', ‘as', ‘assert',
'break',
'class', ‘continue', ‘
def', 'del',
'elif', 'else', 'except', 'exec',
'finally', 'for', 'from',
 'global', 'if',
'import', 'in', 'is',
'lambda',
'not',
 'or',
 'pass', 'print',
 'raise', 'return',
'try',
'while', 'with', 'yield']
```

Scheduling a Python script

Steps:

1 The method to schedule a script depends on your system.

For Windows XP:

- Click the Windows Start menu, point to Control Panel, then double-click Scheduled Tasks.
- If the control panel is in category view, click Performance and Maintenance and click Scheduled Tasks.

For Windows 2000 and NT:

- Click the Windows Start menu, point to Settings, point to Control Panel, then click Scheduled Tasks.

For Windows Vista:

- Click the Windows Start menu, click Settings, point to Control Panel, then click System and Maintenance. Click Administrative Tools and click Schedule tasks.

For Windows 7

- Click the Windows Start menu, click Control Panel > Administrative Tools and click Task Scheduler.

2 Double-click Add Scheduled Task

3 Complete the options on the wizard

a- When asked to click the program you want Windows to run, click the Browse button and the Python script.

Python Functions

- **A function is a defined bit of functionality that does a specific task and can be incorporated into a larger program.**
- In ArcPy, all geoprocessing tools are provided as functions, but not all functions are geoprocessing tools.
- Functions can be used to list certain datasets, retrieve a dataset's properties, validate a table name before adding it to a geodatabase, or perform many other useful geoprocessing tasks.
- The general form of a function is similar to that of tool; it takes arguments, which may or may not be required, and returns something.
- The returned value of a non-tool function can be varied—anything from strings to geoprocessing objects.
- Tool functions will always return a Result object and provide geoprocessing messages support.

```
import arcpy
input = arcpy.GetParameterAsText(0)

if arcpy.Exists(input):
        print "Data exists"
else:
        print "Data does not exist"
```

Python Functions

```
import os, sys, arcpy
from arcpy
import env
# The workspace environment needs to be set before ListFeatureClasses
# to identify which workspace the list will be based on
env.workspace = "c:/data"
out_workspace = "c:/data/results/"
clip_features = "c:/data/testarea/boundary.shp"
# Loop through a list of feature classes in the workspace
for fc in arcpy.ListFeatureClasses():
    # Set the output name to be the same as the input name, and
    # locate in the 'out_workspace' workspace
    output = os.path.join(out_workspace, fc)

    # Clip each input feature class in the list
    arcpy.Clip_analysis(fc, clip_features, output, 0.1)
```

Python Classes

- **In object-oriented programming , a class is a template definition of the method (s) and variable (s) in a particular kind of object . Thus, an object is a specific instance of a class; it contains real values instead of variables.**
- The class is one of the defining ideas of object-oriented programming. Among the important ideas about classes are:
- **A class can have subclasses that can inherit all or some of the characteristics of the class. In relation to each subclass, the class becomes the superclass.**
- **Subclasses can also define their own methods and variables that are not part of their superclass.**
- **The structure of a class and its subclasses is called the class hierarchy.**

Classes can be used to create objects, often referred to as an instance.

ArcPy classes, such as the SpatialReference and Extent classes, are often used as shortcuts to complete geoprocessing tool parameters that would otherwise have a more complicated string equivalent.

ArcPy includes several classes, including SpatialReference, ValueTable, and Point.

Once instantiated, its properties and methods may be used.

Classes have one or more methods called constructors.

A constructor is a method for initializing a new instance of a class.

In the example below, SpatialReference(prjFile) is the class constructor—it creates the spatialRef object by reading a projection file.

Python Classes

In the example below, SpatialReference(prjFile) is the class constructor—it creates the spatialRef object by reading a **projection file**.

```
import arcpy
prjFile = "c:/projections/North America Equidistant Conic.prj"
spatialRef = arcpy.SpatialReference(prjFile)
```

```
import arcpy
prjFile = "c:/projections/North America Equidistant Conic.prj"
spatialRef = arcpy.SpatialReference(prjFile)
#Print the SpatialReference's name, and type
print spatialRef.name
print spatialRef.type
```

Classes can be used repeatedly

```
Import arcpy
pointA =arcpy.Point (2.0,4.5)
pointB =arcpy.Point (3.0,7.0)
```

Using classes with geoprocessing tools

- **Tool parameters** are usually defined using simple text strings.
- Dataset names, paths, keywords, field names, tolerances, and domain names can be specified using a **quoted string**.
- Some parameters are harder to define using simple strings; they are more complex parameters that require many properties.
- Instead of using long, complicated text strings to define these parameters, you can use classes (for example, SpatialReference, ValueTable, and Point classes).

```
import arcpy
inputWorkspace = "c:/temp"
outputName = "rivers.shp"
prjFile = "c:/projections/North America Equidistant Conic.prj"
spatialRef = arcpy.SpatialReference(prjFile)
# Run CreateFeatureclass using the spatial reference object  arcpy.CreateFeatureclass_management(inputWorkspace, outputName,
    "POLYLINE", "", "", "", spatialRef)
```

Using classes with geoprocessing tools

- The string equivalent for this parameter looks something like this:

PROJCS['North_America_Equidistant_Conic',GEOGCS['GCS_North_American_1983',DATUM['D_North_American_1983',SPHEROID['GRS_1980',6378137.0,298.257222101]],PRIMEM['Greenwich',0.0],UNIT['Degree',0.0174532925199433]],PROJECTION['Equidistant_Conic'],PARAMETER['False_Easting',0.0],PARAMETER['False_Northing',0.0],PARAMETER['Central_Meridian',-96.0],PARAMETER['Standard_Parallel_1',20.0],PARAMETER['Standard_Parallel_2',60.0],PARAMETER['Latitude_Of_Origin',40.0],UNIT['Meter',1.0]];IsHighPrecision

ArcPy Modules

- Mapping module (arcpy.mapping)
  - Manipulate contents of .mxd and .lyr files
  - Functions to automate exporting and printing
- Spatial Analyst module (arcpy.sa)
  - Perform map algebra
- Geostatistical Analysis module (arcpy.ga)
  - Set up complex neighborhood searches
- Network Analyst module (arcpy.na)
  - Automate network analysis workflows
- Data access module (arcpy.da)
  - Module for working with data: edit sessions, improved cursors, NumPy arrays

Importing Python Modules

- **Python Modules** includes many **built-in functions**.
- Math module stores specific functions related to processing numeric values and the R module provides statistical analysis functions.
- Modules are imported through the use of the **import statement**.
- When writing geoprocessing scripts with ArcGIS, you will always need to import the ArcPy module, which is the Python package for accessing GIS tools and functions provided by ArcGIS.
- **Import statements will be the first lines of code** (not including comments) in your scripts:

import os, sys,

import arcpy, numpy

import math

import calendar

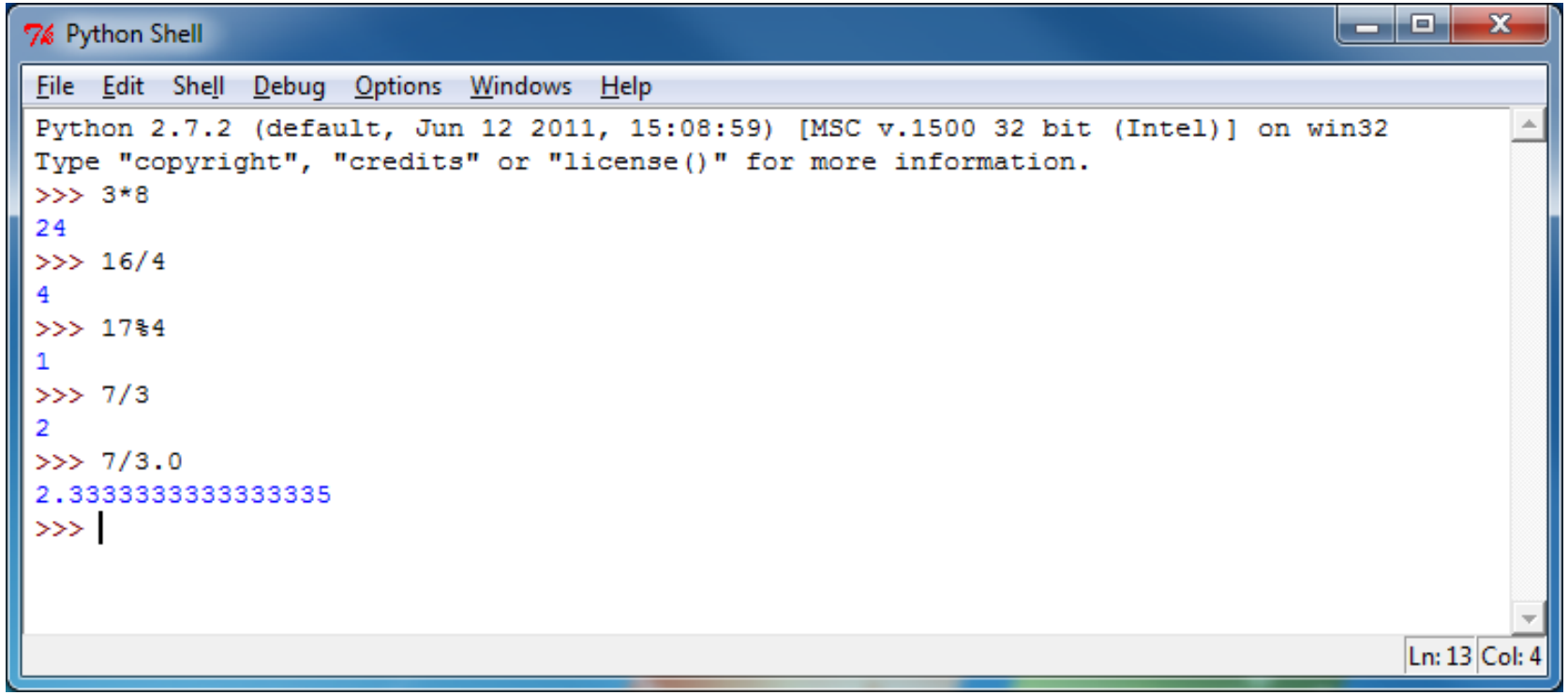
Source: Programming ArcGIS 10.1 with Python Cookbook, Packt Publishing 2013

Data Types and Structures

- The first best Python documentation is the one already installed on your PC: http://docs.python.org
- **The data type of an object determines what type of values it can have and what operations can be performed on the object**.
  - **String: one or more characters, including letters, numbers, other**
  - **Numeric: integers, floats**
  - **Lists: consist of a collection of data elements**
  - **Tuples: consist of a collection of data elements**
  - **Dictionaries: consist of a collection of data elements**
  - ……
- **Data Structure: Collection of data elements structured in some way, for examples elements that are numbered in some way. (Days, Months..)**
- **Sequence: most basic data structure, where elements are assigned a number or an index (Strings, Lists, Tuples…)**
- **Strings, Numbers and Tuples are *immutable***
- **Lists and Dictionaries are *mutable***

Working With Numbers

- **Integers**
- **Floats** (Doubles)
- In python you **do not need to declare variables** , just assign values to them.



Python Arithmetic Operators

Operator	Description	Example
+	**Addition** - Adds values on either side of the operator	a + b will give 30
-	**Subtraction** - Subtracts right hand operand from left hand operand	a - b will give -10
*	**Multiplication** - Multiplies values on either side of the operator	a * b will give 200
/	**Division** - Divides left hand operand by right hand operand	b / a will give 2
%	**Modulus** - Divides left hand operand by right hand operand and returns remainder	b % a will give 0
**	**Exponent** - Performs exponential (power) calculation on operators	a**b will give 10 to the power 20
//	**Floor Division** - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

Python Comparison Operators

- The first best Python documentation is the one already installed on your PC

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assigne value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	**c += a** is equivalent to **c = c + a**
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	**c -= a** is equivalent to **c = c - a**
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	**c *= a** is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	**c /= a** is equivalent to c = c / a
%=	**Modulus** AND assignment operator, It takes modulus using two operands and assign the result to left operand	**c %= a** is equivalent to c = c % a
****=**	**Exponent** AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	**c **= a** is equivalent to c = c ** a
//=	**Floor Division** and assigns a value, Performs floor division on operators and assign value to the left operand	**c //= a** is equivalent to c = c // a

Python Logical Operators

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(a or b) is true.
not	Called Logical NOT Operator. Use to **reverses the logical state** of its operand. If a condition is true then Logical NOT operator will make false.	not(a and b) is false.

PythonOperators.py

Assume variable **a** holds **10** and variable **b** holds **20**

Python Membership Operators

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ]
```

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

PythonOperators.py

Source: http://www.tutorialspoint.com/python

Python Identity Operators

Operator	Description	Example
is	**Evaluates to true** if the variables on **either side** of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	**Evaluates to false** if the variables on **either side** of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Variables

- **Python uses variables to store information**
- **A variable is basically a name that represents or refers to a value.**
- **Assignment Statement**

```
>>> x =17
>>> x*2
34
>>>
```

- **Need to assign a value to any variable before you can use it**
- **Variable names can consist of letters, digits and underscore**
- **Variable name cannot begin with a digit**
- **Python keywords cannot be uses as variable names (*print, import..*)**
- **Use Descriptive names for variables (*count*, instead of *c*)**
- Follow Conventional variable naming (Style Guide for Python Code)
- Keep Variable names as short as possible

```
>>> x,y,z = 1,2,3
>>> x
1
>>> y
2
>>> z
3
>>>
```

Variable Assignment
Dynamic assignment

Python Expressions and Statements

- **Expression: is a value, can also contain variables**

  *>>> 2*17*

  34

- **Statement: it is a python instruction, to do something**

  Print, import, assignment

  *>>> x = 2 * 17* *Assignment Statement*

  >>> print x *Statement*

  34

  >>>

Strings

- **A set of characters surrounded by quotation marks String Literal**

```
>>> print "hello world"
hello world
>>> print ' Hello Class'
 Hello Class
```

- **Single or Double Quotation marks**

```
>>> print " I said: 'Let's go!'"
 I said: 'Let's go!'
```

- In Geoprocesing File and Folders paths are stored as strings
- String Concatenation

```
>>> x = "G"
>>> y = "I"
>>> z = "S"
>>> print x + y + z
GIS
>>> x = "Geographic"
>>> y = "Information"
>>> z = "System"
>>> print x + " " y +  " " + z
Geographic Information System
```

- ***Combining Strings with numbers: first convert numbers to strings***

```
>>> temp = 100
>>> print " The temperature is: " + str(temp) + " degrees"
 The temperature is: 100 degrees
```

- Casting: converting a value of a variable from one type to another

Lists

- **Python Lists are surrounded by Brackets [ ] and list items are separated by (,)**
- **Items can consist of numbers, strings or other type of data**

  mylist = [1,2,3,4,5] mywords =["jpg", "bmp", "tiff", "img"]

- **Lists are an ordered sets of items**

Working with Objects

```
>>> name = 'paul'
>>> name
'paul'
>>> id(name)
44859328
```

- This Object has a String type
- Variables in Python are dynamic: the object type of the variables is determined by the nature of the value assigned to it

- **Casting: is the conversion of object type from one data type to another**

```
>>> var1 = 100
>>> type(var1)
<type 'int'>
>>> var2 = 2.0
>>> type(var2)
<type 'float'>
```

```
>>> var = 100
>>> newvar = str(var)
>>> type(newvar)
<type 'str'>
>>> newvar
'100'
>>> var
100
>>>
```

Using Functions in Python

- Python expressions and statements use variables and functions.
- **Function: A procedure that is used to carry out certain actions**
- *Python already have a set of core functions referred to as **built-in-functions**.*
- *Using a function is referred to as **calling** the function.*
- ***Functions have parameters or arguments***
- ***Functions returns a value***
- ***A function Call is a type of Expression***

  Always review function syntax and description

```
>>> pow(2,3)
8
>>>
```

```
>>> print dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
llipsis', 'EnvironmentError', 'Exception', 'False', 'Fl
tWarning', 'IndentationError', 'IndexError', 'KeyError',
nted', 'NotImplementedError', 'OSError', 'OverflowError
StandardError', 'StopIteration', 'SyntaxError', 'Syntax
Error', 'UnicodeDecodeError', 'UnicodeEncodeError', 'Un
'Warning', 'WindowsError', 'ZeroDivisionError', '_', '_
```

```
>>> print pow.__doc__
pow(x, y[, z]) -> number

With two arguments, equivalent to x**y.  With three arguments,
equivalent to (x**y) % z, but may be more efficient (e.g. for longs).
>>>
```

Create your own FUNCTIONS

- You can define functions to provide the required functionality. Here are simple rules to define a function in Python.
- **Function blocks begin with the keyword def followed by the function name and parentheses (( )).**
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- **The code block within every function starts with a colon (:) and is indented.**
- **The statement return [expression] exits a function**, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return None.
- By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

```
def functionname( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

```
def printme( str ):
        "This prints a passed string into this function"
        print str
        return

Call printme.py
```

Calling a Function

- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.
- Following is the example to call printme() function:

```
#Function definition is here
def printme( str ):
        "This prints a passed string into this function"
        print str;
        return;
# Now you can call printme function
printme("I'm the first call to user defined function!");
printme("Again second call to the same function");
```

```
I'm the first call to user defined function!
Again second call to the same function
```

Pass by Reference vs. Value

- **All parameters (arguments) in the Python language are passed by reference.**
- **It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.**

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

```
# Function definition is here
mylist =[10,20,30]
def changeme( mylist ):
  "This changes a passed list into this function"
  mylist = [1,2,3,4]; # This would assign new reference in mylist
  print "Values inside the function: ", mylist
  return

# Values inside the function:  [1, 2, 3, 4]
print “Values inside the function:  “, changeme( mylist )
print "Values outside the function: ", mylist
Values outside the function:  [10, 20, 30]
```

Python: Working with Strings

- **Find Method:** ***find(str, beg=0 end=len(string))***
- **In operator**
- **Join Method:** ***join(seq)***
- **Split Method** **split(str="", num=string.count(str))**

Strip Method : to remove any combination of characters in any order from the ends of an existing string (this is true for the generic strip method)

- **lStrip Method** ***lstrip()***
- **rStrip Method** ***rstrip()***

Calling one of the strip methods without arguments removes spaces (whitespaces)

Replace Method: ***replace(old, new [, max])***

```
mytext = "GEOLOGIC Sciences"
mytext.find(“Sci")
9
>>> mytext.find(“pie")
-1
----------------------------------------
x ="Geograohic Information
Systems"
>>> "Info" in x
True
```

```
list_gis = ["Geographic",
"Information", "Systems"]
string_gis = " "
string_gis.join(list_gis)
'Geographic Information Systems'
```

```
myfile = streams.shp '
myfile.replace(“.shp“,””)
‘streams‘
z.rstrip()
'Commenting scripts is good
practice'
```

```
y = 'geoprocesing using python scripts'
y.split(" ")
['geoprocesing', 'using', 'python', 'scripts']
```

```
y = Commenting scripts is good  practice '
z.split("Cdo")
'mmenting scripts is good  practice‘
z.rstrip()
'Commenting scripts is good  practice'
```

Python: Working with Strings

- **The format method is commonly used for string formatting. Its most basic usage is to insert a value into a string using single placeholder**

```
>>> temp = 100
>>> print 'The temperature is {0} degrees'.format(temp)
The temperature is 100 degrees
```

- In this example {0} is a replacement field, which replaced by the argument of the format method

```
>>> username ='Paul'
>>> password = 'surf$&9'
>>> print "{0} 's password is {1}".format(username, password)
Paul 's password is surf$&9
```

- String formatting can also be accomplished using the % opearator. However, the format method is the recommended approach to string formatting

Multiple Substitution Values

```
>>> s1 = "cats"
>>> s2 = "dogs"
>>> s3 = " %s and %s living together" % (s1, s2)
>>> print s3
 cats and dogs living together
```

Formating a floating point number:

```
>>> pi = 3.14159
>>> print(" pi = %1.2f " % pi)
pi = 3.14
>>> print(" pi = %0.2f " % pi)
 pi = 3.14
>>> print(" pi = %1f " % pi)
 pi = 3.141590
>>> print(" pi = %9f " % pi)
 pi =  3.141590
```

Working With Lists

- **List are versatile Python type and can be manipulated in many different ways.**
- **Lists Objects have Functions**
- **Python Lists are indexed just like strings**.
- **List can also be sliced or fetched**
- **Membership test available**
- **Can delete, elements or append other elements**
- Count Method for lists: determines the number of times an element occurs in a list
- Extend method allows you to append several values at once.
- Index Method: used to find the index of the first occurrence of an element.
- Insert Method: to insert a new element in a particular location
- pop Method: removes an element from alist at particular location and returns the value of this element
- Remove Method: to remove the first occurrence of a value or an element

```
>>> cart = [ "Apples", "Kiwis", "Bananas", "Grapes", "Blueberries"]
>>> print len(cart)
5
>>> print cart[0:2]
['Apples', 'Kiwis']
>>> cart.sort(reverse = True)
>>> print cart
['Kiwis', 'Grapes', 'Blueberries', 'Bananas', 'Apples']
>>> cart.sort(reverse = False)
>>> print cart
['Apples', 'Bananas', 'Blueberries', 'Grapes', 'Kiwis']
>>> cart.sort(reverse = False)
>>> print cart
['Apples', 'Bananas', 'Blueberries', 'Grapes', 'Kiwis']
>>>print  cart[2]
'Blueberries‘
>>>print  cart[-2]
'Grapes‘
>>> cart[2:]
['Blueberries', 'Grapes', 'Kiwis']
>>> cart[:2]
['Blueberries', 'Grapes',]
>>> "Strawberies" in cart
False
>>> del cart[0]
>>> print cart
['Bananas', 'Blueberries', 'Grapes', 'Kiwis']
>>> cart2 = ["figues", " Watermelon"]
>>> cart.append(cart2)
>>> print cart
['Bananas', 'Blueberries', 'Grapes', 'Kiwis', ['figues', ' Watermelon']]
```

Working With Paths

- **The Backslash is a reserved character in Python (\n, \t)**

- **Path: a list of folder names separated by a backslash followed by a file name.**

C:\EsriPress\Python\Data\Exercise04\rivers.shp

1 - Use forward Slash:

"C:/EsriPress/Python/Data/Exercise04/rivers.shp"

2 - Use double Backslash:

"C:\\EsriPress\\Python\\Data\E\xercise04\\rivers.shp"

3 - Use s string literal:

r"C:\EsriPress\Python\Data\Exercise04\rivers.shp"

- It is recommended to use one single style for working with paths
- In Python paths are stored as strings

Working With Modules

- **A module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.**
- **Use the "import" statement for loading any needed python modules**

```
import os, arcpy, time, math, shutil
from arcpy import *
```

>>> dir(math)

To get the list of all functions in the math module

```
>>> import math
>>> math.cos(1)
0.5403023058681398
>>> print math.cos.__doc__
cos(x)

Return the cosine of x (measured in radians).
>>>
```

- Create Your Own Module Example

```
def print_func(par):
print "Hello : ", par
return
```

- Import and Use your Module

```
# Import module support
import support
# Now you can call defined function and
module as follows support.print_func("Zara")
```

Controlling Workflow With Conditional Statements

Branching Structures

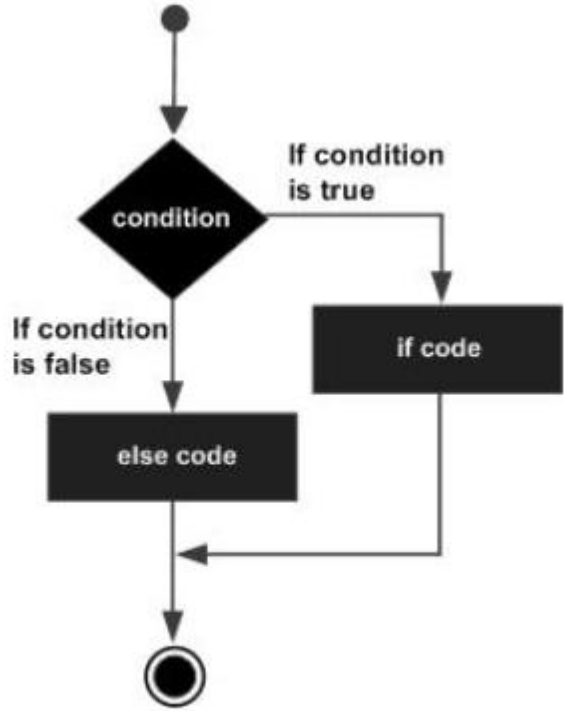
- Branching: a way to control the workflow in a script.
- Branching: making a decision to take one path or another
- Branching uses the if structure and its variants
- Conditions are mostly created by using comparison operators (next slide)
- In Python, the use of a single equal sign (=) is reserved for assigning a value to a variable
- Indentation is required in python for defining a block of code
- Under the if structure, only certain parts of the code are executed and others are skipped.

```
if expression:
        statement(s)
else:
        statement(s)
```

```
>>> import random
>>> x = random.randint(0,6)
>>> print x
4
>>> if x --6:
        print "You win!"
You win!
```

```
if Condition (True):
        code line(s)
elif Condition (False):
        code line(s)
else Condition (False):
        code line(s)
```

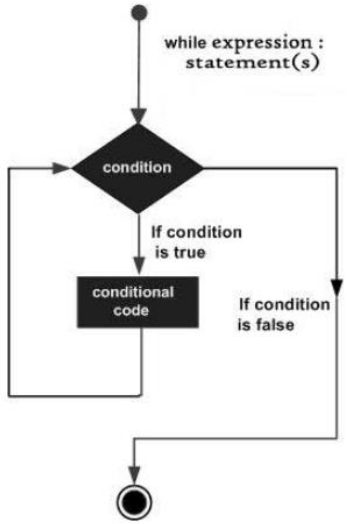
Block



Controlling Workflow With Loop Structures

- **Loop: allows to repeat a part of the code until a particular condition is reached or until all possible inputs are used.**
  - While loop
  - For Loop
- **While statement requires an exit condition,**
- The variable used in the exit condition is called **sentry variable**
- The For Loop is based on a sequence
- **For Loop: repeats the block of code for each element of the sequence, it ends once it reaches the end of the sequence.**

```
While expression:
        statement(s)
```




```
i = 0
While i ≤ 10:
        print I
        i = i + 1
```

```
i = 0
While i ≤ 10:
        print i
```

```
i = 0
While i ≤ 10:
        print i
        i +=
```

```
i = 12
While i ≤ 10:
        print i
        i = i + 1
```

```
mylist = [ “a”, “b”, “c”, “d”]
for letter in mylist:
        print  letter
```

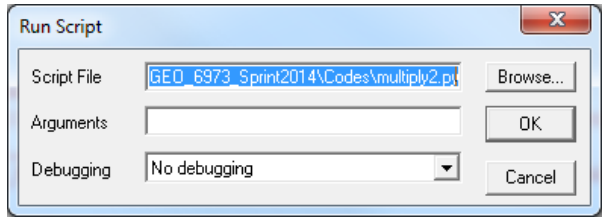
Python: Getting User **Input**

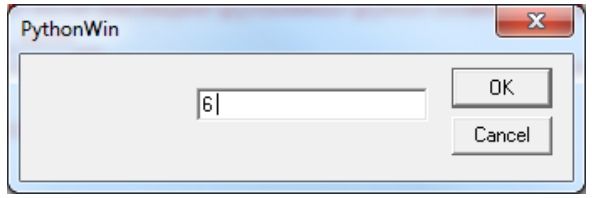
- System Argument (Necessary Modules): *import sys*
- *Second Method, use input function: >>> x – input (“ ”)*

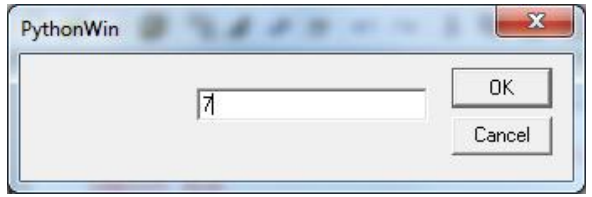
```
#   Python 2.x

import sys
var1 = raw_input("Enter something: ")
var2 = raw_input("Enter something: ")
print "you entered ", var1
print "you entered ", var2
```

```
#   Python 3.x

import sys
var1 = input("Enter something: ")
var2 = input("Enter something: ")
print "you entered ", var1
print "you entered ", var2
```

```
import sys
x = int(sys.argv[0])
y = int(sys.argv[1])
print x * y
```








```
>>> 6
>>> 7
42
```

sys.argv The list of command line arguments passed to a Python script. argv[0] is the script name (it is operating system dependent whether this is a full pathname or not).
If the command was executed using the -c command line option to the interpreter, argv[0] is set to the string '-c'. If no script name was passed to the Python interpreter, argv[0] is the empty string.

Exercises: Python Strings

- The "print" operator prints out one or more python items followed by a newline (leave a trailing comma at the end of the items to inhibit the newline).
- A "raw" string literal is prefixed by an 'r' and passes all the chars through without special treatment of backslashes, so r'x\nx' evaluates to the length-4 string 'x\nx'.
- A 'u' prefix allows you to write a unicode string literal

- **Exercises 2 :** Print the lines below:

  this\t\n and that

  """It was the best of times.

  It was the worst of times."""

```
raw = r'this\t\n and that'
 print raw     ## this\t\n and that

 multi = """It was the best of times.
 It was the worst of times."""
```

String Methods

- **s.lower(), s.upper()** -- returns the lowercase or uppercase version of the string
- **s.strip()** -- returns a string with whitespace removed from the start and end
- s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
- s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
- **s.find**('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- **s.replace**('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
- **s.split**('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['**aaa**', '**bbb**', '**ccc**']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- **s.join(**list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

Exercises 3 and 4: **If Statement**

- Python does not use { } to enclose blocks of code for if/loops/function etc.. Instead, Python uses the colon (:) and indentation/whitespace to group statements. The boolean test for an if does not need to be in parenthesis (big difference from C++/Java), and it can have *elif* and *else* clauses (mnemonic: the word "elif" is the same length as the word "else").

```
if speed >= 80:
  print 'License and registration please'
  if mood == 'terrible' or speed >= 100:
    print 'You have the right to remain silent.'
  elif mood == 'bad' or speed >= 90:
    print "I'm going to have to write you a ticket."
    write_ticket()
  else:
    print "Let's try to keep it under 80 ok?"
```

Exercises 3: Write a python code that prints the first 100 odd numbers (from 0 to 99), each number should be printed on a new line.

Exercises 4: Write a python code that prints the first 100 odd numbers (from 0 to 99), all numbers should be printed on the same line

Exercise 6 and 7: **Python Strings**

- Exercise 6
- Store your first name, in lowercase, in a variable.

- Using that one variable, print your name in lowercase, Titlecase, and UPPERCASE.

- Exercise 7
- Store your first name and last name in separate variables, and then combine them to print out your full name

Exercises 8 and 9: **Python Strings**

Exercise 8

- Choose a person you look up to. Store their first and last names in separate variables.
- Use concatenation to make a sentence about this person, and store that sentence in a variable.-
- Print the sentence.

Exercise 9

- Store your first name in a variable, but include at least two kinds of whitespace on each side of your name.
- Print your name as it is stored.
- Print your name with whitespace stripped from the left side, then from the right side, then from both sides.

Exercises 10 and 11: **Numbers**

Exercises 10

- Write a program that prints out the results of at least one calculation for each of the basic operations: addition, subtraction, multiplication, division, and exponents

Exercises 11

Find a calculation whose result depends on the order of operations.

Print the result of this calculation using the standard order of operations.

Use parentheses to force a nonstandard order of operations.

Print the result of this calculation

Challenge: Exercise 12

Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5,

between 2000 and 3200 (both included).

The numbers obtained should be printed in a comma-separated sequence on a single line.

Hints:

Consider use range(#begin, #end) method

Solution:

Challenge: Exercises 13

- Write a program which can compute the factorial of a given numbers.
- The results should be printed in a comma-separated sequence on a single line.
- Suppose the following input is supplied to the program: 8
- Then, the output should be: 40320

Hints:

- In case of input data being supplied to the question, it should be assumed to be a console input.

Solution:

```
def fact(x):
        if x == 0:
                return 1
        return x * fact(x - 1)
x=int(raw_input())
print fact(x)
#---------------------------------------#
```

Exercise 1: Python Strings

- The **str()** function converts values to a string form so they can be combined with other strings.

- **Exercises 1:** Print the line below using Python;

  The value of pi is 3.14

```
pi = 3.14
#text = 'The value of pi is ' + pi                ## NO, does not work
#print text
text = 'The value of pi is ' , pi                 ## NO, does not work
print text
text = 'The value of pi is '  + str(pi)  ## yes
print text
```

Python Functions

Defining a Function

- You can define functions to provide the required functionality. Here are simple rules to define a function in Python.
- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return None.

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Python Functions

- Write a function that takes a string from the user and print it.

```
# Create a function printme


def printme( str ):
   "This prints a passed string into this function"
   print str
   return
printme("I'm first call to user defined function!");
printme("Again second call to the same function");


# Call printme function which resides in a python file called MyFunctions.py
from MyFunctions import printme
printme(" Function was called from a different Python file!!!!")
```

Python Logical (Boolean) Operators

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
x or y	if *x* is false, then *y*, else *x*	(1)
x and y	if *x* is false, then *x*, else *y*	(2)
not x	if *x* is false, then True, else False	(3)

Notes:

This is a short-circuit operator, so it only evaluates the second argument if the first one is False.

This is a short-circuit operator, so it only evaluates the second argument if the first one is True.

not has a lower priority than non-Boolean operators,

so not **a == b** is interpreted as not (**a == b**), and **a == not b** is a syntax error.

Python Logical (Boolean) Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is **true**.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is **true**.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is **false**.

Python Logical Operators

```
print " Python Logical Operators"
a = 10
b = 20
c = 0

if ( a and b ):
   print "Line 1 - a and b are true"
else:
   print "Line 1 - Either a is not true or b is not true"
if ( a or b ):
   print "Line 2 - Either a is true or b is true or both are true"
else:
   print "Line 2 - Neither a is true nor b is true"
```

```
a = 0
if ( a and b ):
   print "Line 3 - a and b are true"
else:
   print "Line 3 - Either a is not true or b is not true"
if ( a or b ):
   print "Line 4 - Either a is true or b is true or both are true"
else:
   print "Line 4 - Neither a is true nor b is true"
if not( a and b ):
   print "Line 5 - a and b are true"
else:
   print "Line 5 - Either a is not true or b is not true"
```

Python Membership Operators

```
# Python Membership Operators

print " Python Membership Operators"
a = 10, b = 20

list = [1, 2, 3, 4, 5 ]

if ( a in list ):

   print "Line 1 - a is available in the given list“

else:

   print "Line 1 - a is not available in the given list"
```

```
# Python Membership Operators

if ( b not in list ):

   print "Line 2 - b is not available in the given list“

else:
   print "Line 2 - b is available in the given list"

a = 2
if ( a in list ):

   print "Line 3 - a is available in the given list“
else:
   print "Line 3 - a is not available in the given list"
```

Exercise 2: Python Strings

- The "**print**" operator prints out one or more python items followed by a newline (leave a trailing comma at the end of the items to inhibit the newline).
- A **"raw" string literal** is prefixed by an **'r'** and passes all the chars through without special treatment of backslashes, so **r'x\nx'** evaluates to the length-4 string **'x\nx'**.
- A **'u'** prefix allows you to write a unicode string literal

- **Exercises 2 :** Print the lines below:

  this\t\n and that

  """It was the best of times.

  It was the worst of times."""

```
raw = r'this\t\n and that'
 print raw     ## this\t\n and that


 multi = """It was the best of times.
 It was the worst of times."""
```

Python List Functions

- A list object has a number of member methods. These can be grouped arbitrarily into transformations, which change the list, and information, which returns a fact about a list. In all of the following method functions, we'll assume a list object named l .
- The following list transformation functions update a list object. In the case of the pop method, it both returns information as well as updates the list.

- **l. append (object)** Update list l by appending object to end of the list.
- **l. extend (list)** Extend list l by appending list elements. Note the difference from append(object), which treats the argument as a single list object.
- **l. insert (index , object)** Update list l by inserting object before position index . If index is greater than len(list), the object is simply appended. If index is less than zero, the object is prepended.
- **l. pop ([index])** → item Remove and return item at index (default last, -1) in list l . An exception is raised if the list is already empty.
- **l. remove (value)** → item Remove first occurrence of value from list l . An exception is raised if the value is not in the list.
- **l. reverse** Reverse the items of the list l . This is done "in place", it does not create a new list.
- **l. sort ([cmpfunc])** Sort the items of the list l . This is done "in place", it does not create a new list.

Exercises 3 and 4

Exercises 3: Write a python code that prints the first 100 odd numbers (from 0 to 99), each number should be printed on a new line.

Create a list "OddNumbers" where you will put all the odd numbers

Exercises 4: Write a python code that prints the first 100 odd numbers (from 0 to 99), all numbers should be printed on the same line

```
for x in range(0,100):
    if x%2<>0:
            print x
```

```
OddNumbers=[]
for x in range(0,100):

    if x%2<>0:

        OddNumbers.append(x)

print OddNumbers
```

Exercises

Figure out a compact way to get Python to make the string, YesYesYesYesYes, and try it.

How about MaybeMaybeMaybeYesYesYesYesYes?

```
text=''
for x in range(0,5):
    text += 'yes'
print text
```

```
text=''
for x in range(0,5):
    text += 'yes'
print text
```

Exercises 6 and 7: Python Strings

- Exercise 6
- Store your first name, in lowercase, in a variable.
- Using that one variable, print your name in lowercase, Titlecase, and UPPERCASE.

- Exercise 7
- Store your first name and last name in separate variables, and then combine them to print out your full name

```
# Exercise 6
name = ‘sponge'
a = name[0:1]
b = name[1:]
print 'lowercase: ', name
print 'Titlecase: ', a.upper()+ b
print ‘UPPERCASE: ‘, name.upper()
```

```
# Exercise 7
firstname = ‘sponge'
lastname = ‘bob'
print firstname, lastname
```

Challenge: Exercise 12

Write a program which will find all such numbers which are **divisible by 7** but are **not a multiple of 5**, **between 2000 and 3200** (both included).

The numbers obtained should be printed in a comma-separated sequence on a single line.

Hints:

Consider use **range(#begin, #end)** method

```
list=[]
for x in range(2000,3001):
    if x%7 == 0 and x%5 <> 0:
        list.append(x)
print list
```

Exercise 14: Sum and Multiply Functions

- Define a function sum() and a function multiply() that **sums** and **multiplies** (respectively) all the numbers in a list of numbers.
- For example, **sum([1, 2, 3, 4])** should **return 10**, and **multiply([1, 2, 3, 4])** should **return 24**.

```
# Exercise 14 Sum and Multiply Functions


def SumList(mylist):
    total = 0
    for x in mylist:
        total = total + x
    print " the Sum of the list is: ", str(total)
    return
mylist = [1, 2, 3, 4]
SumList(mylist
```

```
# Exercise 14 Sum and Multiply Functions


def MultiplyList(mylist):
    multiply = 1
    for x in mylist:
        multiply = multiply * x
    print " the Sum of the list is: ", str(multiply)
    return
mylist = [1, 2, 3, 4]
MultiplyList (mylist)
```

Python Lists

- A Python list can hold any number of things in a linear collection (similar to the "array" in other languages).
- Use the len() function to check the length of a list and the square bracekts [ ] to access individual elements (in this way, lists work just like strings):

```
a = ['hi', 'there', '!']  # a list with 3 elements
 len(a)      ## 3
 a[0]       ## 'hi'
 a[2] = 'ho'  ## Can change an existing element
```

Python Lists

- The .append(value) method on a list adds an element to its end, and the sorted(list) function takes in a list and returns a new list sorted into increasing order:

```
a = ['hi', 'there']
  a.append('aa')  ## use .append() to add elements to the end
  a.append('bb')
  ## now a is ['hi', 'there', 'aa', 'bb']
  b = sorted(a)  # b is ['aa', 'bb', 'hi', 'there'], a is unchanged
```

Python List Loop

The easiest way to access elements in a list with a **loop**:

```
a = [1, 2, 3]
sum = 0
for num in a:                    ## iterate num over values 1, 2, 3
  sum = sum + num
```

Another way to loop over a list is using the **range(n) function** witch returns the sequence 0, 1, 2, ... n-1, so for i in **range(len(list)):** iterates over the index numbers of a list, like this:

```
a = ['hi', 'there', 'ok']
result = ''
for i in range(len(a)):
  # i will be 0, 1, 2 ... use a[i] to look at each element.
  # Here we just accumulate the a[i] strings
  result = result + a[i]
```

Python List Loop

The easiest way to access elements in a list with a **loop**:

```
a = [1, 2, 3]
sum = 0
for num in a:                    ## iterate num over values 1, 2, 3
  sum = sum + num
```

Another way to loop over a list is using the **range(n) function** witch returns the sequence 0, 1, 2, ... n-1, so for i in **range(len(list)):** iterates over the index numbers of a list, like this:

```
a = ['hi', 'there', 'ok']
result = ''
for i in range(len(a)):
  # i will be 0, 1, 2 ... use a[i] to look at each element.
  # Here we just accumulate the a[i] strings
  result = result + a[i]
```

Python List Loop

```
# The easiest way to access elements in a list with a loop:
 a = [1, 2, 3]
 sum = 0
 for num in a:  ## iterate num over values 1, 2, 3
  sum = sum + num
# Another way to loop over a list is using the range(n) function witch returns the sequence 0, 1, 2, ... n-1, so for i in range(len(list)): iterates over the index numbers of a list, like this:
 a = ['hi', 'there', 'ok']
 result = ''
 for i in range(len(a)):
  # i will be 0, 1, 2 ... use a[i] to look at each element.
  # Here we just accumulate the a[i] strings
  result = result + a[i]
```

- This form of loop gives flexibility to refer to the element to the left (a[i-1]) or the next element (a[i+1]) within the loop, however be careful not to refer past the end of the list, len(a)-1 is the max allowed index.
- Python lists also support the "slice" syntax to refer to subparts of a list -- slices are discussed in the Python Strings doc, and work analogously for lists.
- **Sorting**: the easiest way to sort a list is with the **sorted(list) function** which takes in any collection and returns a new list, sorted into increasing order.

Exercise 17: Maximum of two numbers function

Define a function max() that takes two numbers as arguments and returns the largest of them.

Use the if-then-else construct available in Python.

(It is true that Python has the max() function built in, but writing it yourself is nevertheless a good exercise.)

```
x =0
y =0
def maxNumber(x,y):
    x = raw_input('Enter  the first number: ')
    y = raw_input('Enter  the second number: ')
    if x>y:
        print ' The maximum is: ', str(x)
    elif x==y:
        print ' The two numbers are equal'
    else:
        print ' The maximum is:  ', str(y)
    return
#x = raw_input('Enter  the first number: ')
#y = raw_input('Enter  the second number: ')
maxNumber(x,y)
```

Exercise 20: Print Even numbers from a defined range

- Write a program, which will find all such numbers between 1000 and 3000 (**both included**) such that each digit of the number is an **even number**.
- The numbers obtained should be printed in a **comma-separated sequence** on a **single line**.

```
# First Method
values = []
for i in range(1000, 3001):
    if i %2 == 0:
        print i
        values.append(i)
print values
```

Exercise 20: Print Even numbers from a defined range

- Write a program, which will find all such numbers between 1000 and 3000 (**both included**) such that each digit of the number is an **even number**.
- The numbers obtained should be printed in a **comma-separated sequence** on a **single line**.

```
values = []
for i in range(1000, 3001):


    s = str(i)
    if (int(s[0])%2==0) and (int(s[1])%2==0) and (int(s[2])%2==0) and (int(s[3])%2==0):
        values.append(s)
print ",".join(values)
```

Exercise 21: Count Digits and Numbers from input

Write a program that accepts a **sentence** and calculate the number of letters and digits.
Suppose the following input is supplied to the program:
hello world! 123
Then, the output should be:
LETTERS 10
DIGITS 3

```
values = []
for i in range(1000, 3001):

    s = str(i)
    if (int(s[0])%2==0) and (int(s[1])%2==0) and (int(s[2])%2==0) and (int(s[3])%2==0):
        values.append(s)
print ",".join(values)
```