GIS 4653/5653: Spatial Programming and GIS

Introduction to Python Programming

Why Python?

Which version of Python?

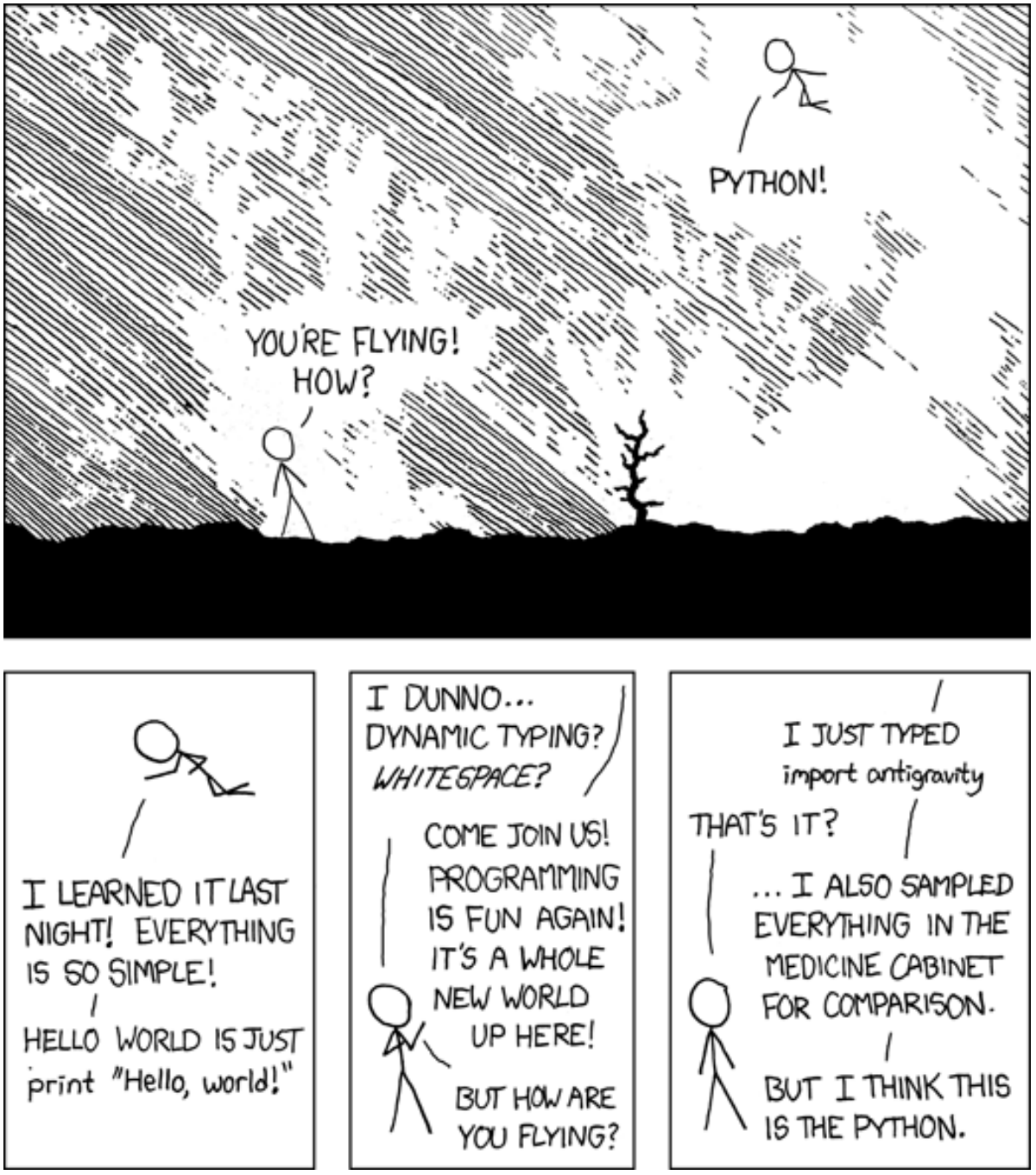
- Will use Python 2.7.x
  - Python 3.x is the latest version of Python, but it is not supported by ArcGIS
  - Syntax differences between Python 2.7 and Python 3
    - Python 2.7 scripts will not run under Python 3 (or vice versa)
    - Programs available to convert Python 2.7 to Python 3 (and vice versa)

Why Python?

- Programs are clean, easy to understand and maintainable
- Programs are portable across Windows, Mac, Linux, etc.
  - ArcGIS Desktop only on Windows, but as we will see, lots of GIS tasks can be achieved directly using Python modules
- Many open-source libraries and modules available
  - Including with spatial programming support
- Plays nicely with programs written in other languages
  - ArcGIS is written in C++, but many Toolbox functions are written in Python

Why not language X?

- Python competes with two classes of languages:
  - Scripting languages such as Perl
    - Python tends to be more readable, maintainable and of higher quality
    - Python is minimalist whereas Perl prides itself on many ways to do things
    - Perl can be terse and hard to understand
  - High-level languages such as Java
    - Python easier to get started with
    - Do not need to do OO programming unless you need to
    - Python code tends to run half the size of equivalent Java code
- Fast becoming the glue language of choice
  - Why Python is the preferred language to write Arc Toolkits in



http://xkcd.com/353/

Drawback of Python

- Python is not the fastest kid on the block
  - Especially for heavy-duty number crunching
  - May want to write such modules in C/C++ and invoke them from Python
    - Another option to write such modules in Python, compile it and then link the compiled extensions into your code
- Still, Python is sufficient for many programming tasks
  - And hardware is getting more and more capable

Uses of Python

- Python can be used to write:
  - Graphical User Interfaces (GUIs): IDLE is written in Python
  - Dealing with operating systems portably for control purposes
  - Web services and web applications, because of HTML and XML modules that are available for Python
  - Glue to invoke or be called from C++ or Java
  - Working with relational (and spatial) databases

Why is Python popular?

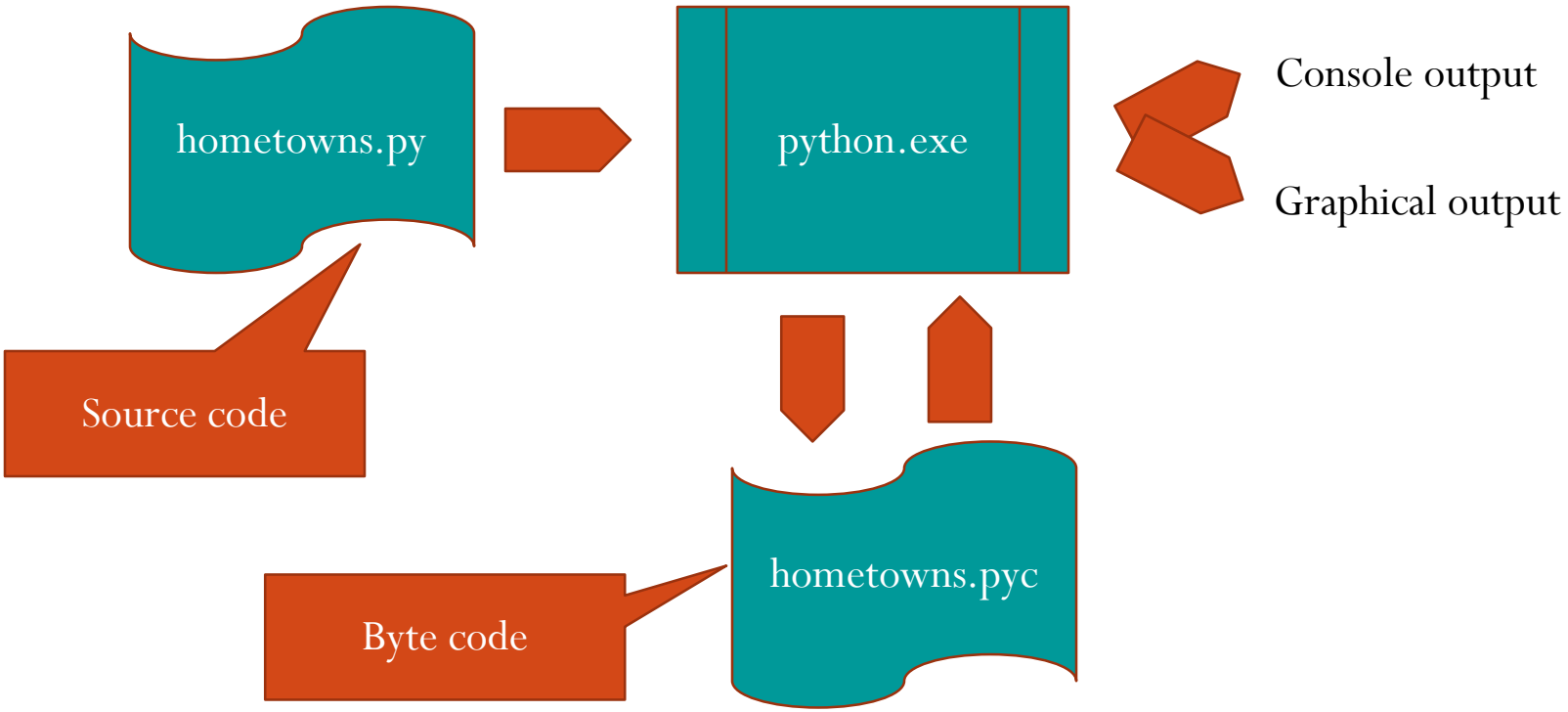
- Object oriented (although you can write procedural code)
- Portable to most major platforms
- Open-source, free
- Dynamic typing (no need to declare types of variables)
- Garbage collection for memory
- Nice built-in and 3rd party libraries
- Can mix it with C++ or Java

Source:Wikipedia

The Python Interpreter and IDE

How Python programs are run

- Python programs are not executables
  - They are executed by a Python interpreter (python.exe on Windows)



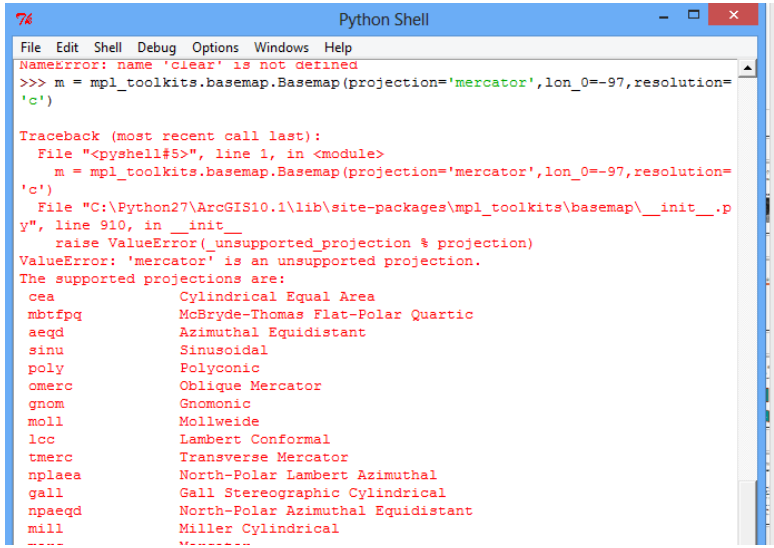
Other Python environments

- What we talked about is called CPython
  - The default version of Python
  - The one we will be using
- Other versions of Python
  - Jython compiles Python source code to Java byte code
    - Executed on JVM, usually to take advantage of Java libraries and the heavy research that has gone into JVM optimization
  - IronPython is like Jython except that it compiles to Microsoft's CLR, to run on the .NET runtime
  - Cython makes it possible for Python to call C and vice-versa
- Can distribute "frozen binaries" i.e. the pyc files
  - Pyc files are not produced by default for single-file programs

IDLE

- Several integrated development environments (IDEs) for Python exist
  - Syntax coloring, method completion, etc.
  - Can use any IDE you want
  - Do not use Notepad/Wordpad
- IDLE is packaged with the default Python distro
  - May not be the best, but it suffices for our purposes
  - On Linux, you will need to install RPM for pythontools

Developing Python programs with IDLE



Use interactive window to try out commands

Save sets of commands as a script so that you can repeat steps by simply executing the script

Some short-cuts in IDLE

- Ctrl+P
  - Get previous line back
- Ctrl + Space
  - Context-specific help
- F5
  - Run the script

Python Basic Syntax

Variables, Operators, Comments

- Variables can take different values at different times
  - Operators allow you to do arithmetic on variables and constants

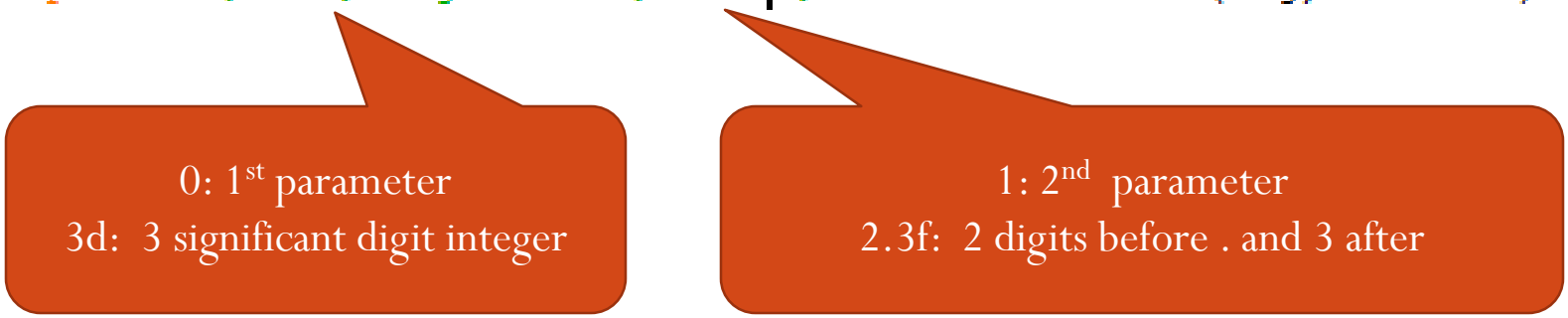
```
import math;  # needed to get pi

deg = 10;
radians = deg * math.pi / 180;
print(radians);
```

  - Unlike other programming languages you may be familiar with, you do not declare the "type" of a variable (or even call it a variable)
  - Python internally keeps track of what type it is
- Comments start with a # and go to the end of the line

Printing

- Can format output as follows:

```
import math;  # needed to get pi

deg = 10;
radians = deg * math.pi / 180;
print '{0:3d} degrees = {1:2.3f} radians'.format(deg, radians)
```



- Done this way to enable internationalization
  - Read format string out of translated files

Defining a function

- Writing equations on variables is usually a prelude to defining functions

```
import math;  # needed to get pi

def degrees_to_radians(deg):
    radians = deg * math.pi / 180;
    return (radians)

print '{0:3d} degrees = {1:2.3f} radians'.format(30, degrees_to_radians(30))
```

Define function

Call function

- Alternate way to call function
  - Can be more readable especially with lots of parameters; you can flip order of parameters around

```
), degrees_to_radians(deg=30)
```

The “for” loop

- To give a bunch of values to a variable and do something with each variable:

```
for d in [45, 90, 135]:
    print '{0:3d} degrees = {1:2.3f} radians'.format(d, degrees_to_radians(d))
```

A list of values is specified by listing the values within square brackets and separating them using commas

- Note the syntax of a for loop
  - For variable-name in list-of-values
  - Put “body” of statements within an indent (whitespace is key!)

```
for d in [45, 90, 135]:
    rad = degrees_to_radians(d)
    print '{0:3d} degrees = {1:2.3f} radians'.format(d, rad)
```

Using range()

- Listing all possible values can get tiresome
  - Usually, there is a simple rule to generate the values
  - Rules like these are what functions are for

```
import math;  # needed to get pi

def degrees_to_radians(deg):
    radians = deg * math.pi / 180;
    return (radians)

for d in range(0,361,20):
    print '{0:3d} degrees = {1:2.3f} radians'.format(d, degrees_to_radians(d))
```

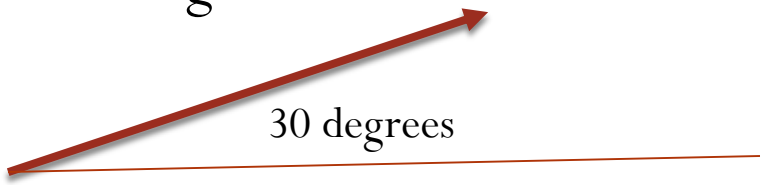
range() is a built-in function where you can specify start, stop and step

  - Why have I specified 361 as the end?
  - Change it to 360. What happens?
    - Look at the documentation for the range() function
- Why is the output "lined" up?

```
  0 degrees = 0.000 radians
 20 degrees = 0.349 radians
 40 degrees = 0.698 radians
 60 degrees = 1.047 radians
 80 degrees = 1.396 radians
100 degrees = 1.745 radians
120 degrees = 2.094 radians
```

In-class assignment: (plane) #1

- Imagine an aircraft rising at a constant speed at an angle of 30 degrees to the ground
  - Print out the height the aircraft reaches as its distance increases



  - Approximately given by distance*sin(30)
    - In Python, one would call Math.sin(), but pass in the value in radians

Distance from starting point	Altitude above ground
0 km	0 km
10 km	
20 km	

Solution: plane

```
import math

def height_for_distance(dist, elev_angle):
    elev_angle_radians = elev_angle * math.pi / 180
    height = dist * math.sin( elev_angle_radians )
    return height

for d in range(0, 50, 10):
    h = height_for_distance(d, 30)
    print '{0:3.2f} km distance --> {1:3.2f} km height'.format(d,h)
```

In-class assignment #2: car

- A car is traveling at 30km/hr
  - Print out the distances traveled after 0, 10, 20, 30 …, 60 minutes

Solution: “car”

```
def distance_traveled(time_min, speed_kmhr):
    time_hr = time_min/ 60.0
    dist = speed_kmhr * time_hr
    return dist

for time in range(0,60,10):
    print '{0:3.1f} min --> {1:3.1f} km'.format(time,
distance_traveled(time,30) )
```

Python Modules

What's a Python Module?

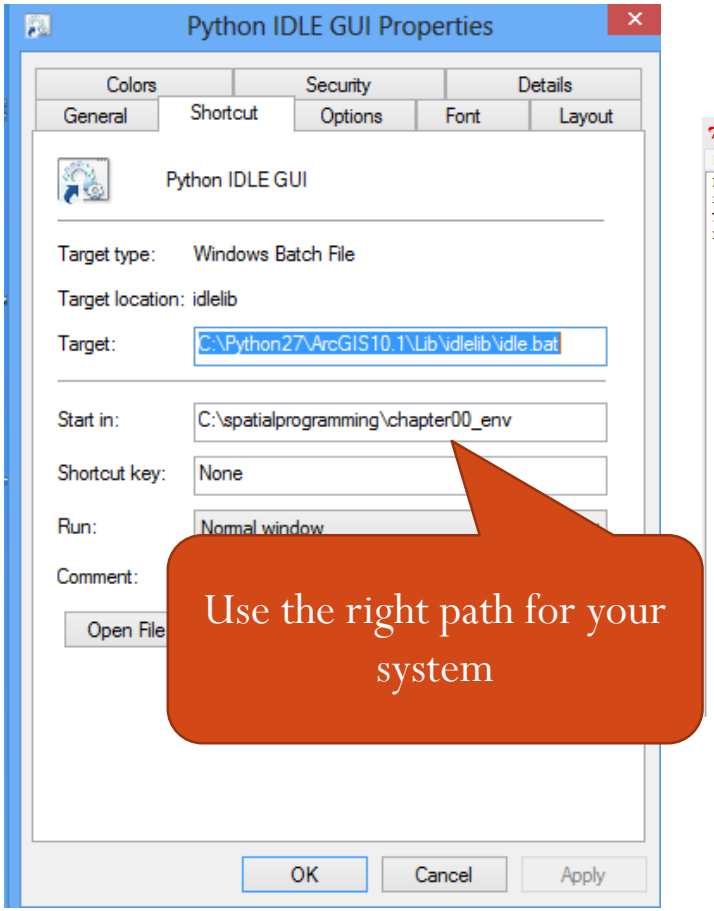
- A Python module is simply a file with Python code
  - The entire file is run when the module is imported by some other Python code
- Try this:
  - Close any IDLE sessions you have running
  - Open up IDLE and at the prompt, type:

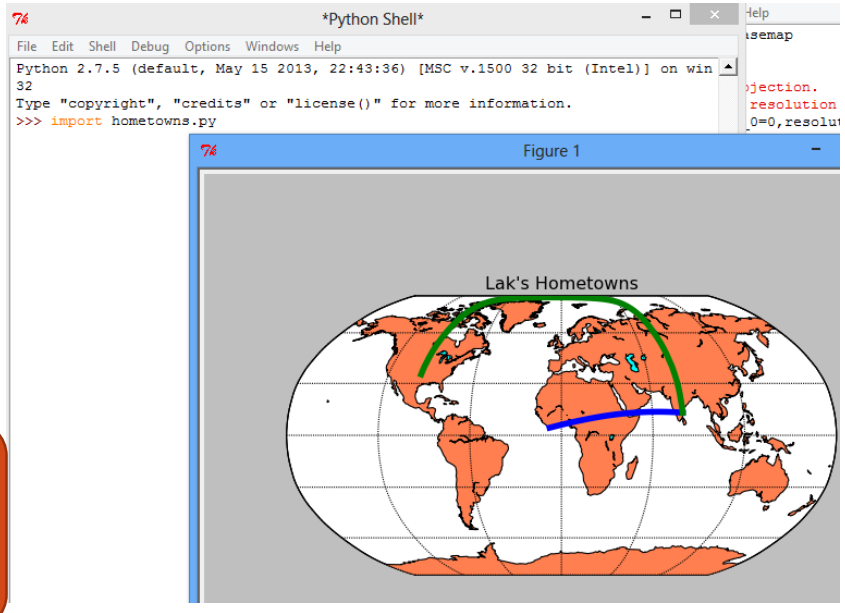
```
>>> import hometowns.py

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    import hometowns.py
ImportError: No module named hometowns.py
>>>
```

  - Why doesn't this work?

Change start path of IDLE

- Change the directory in which IDLE runs:





No need for extension

- Close the window and note strange error:

```
>>> import hometowns.py

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import hometowns.py
ImportError: No module named py
>>>
```

Hmm . . .

- To import a module, drop the ".py" extension
  - Restart IDLE and type in:

```
Type "copyright", "cred
>>> import hometowns
>>>
```

  - Plot gains control of the program, so close the window to get the prompt back

Why did you have to restart IDLE?

- Why did I ask you to restart IDLE?
  - What happens if you type "import" twice?

```
Type "copyright", "credi
>>> import hometowns
>>> import hometowns
>>>
```

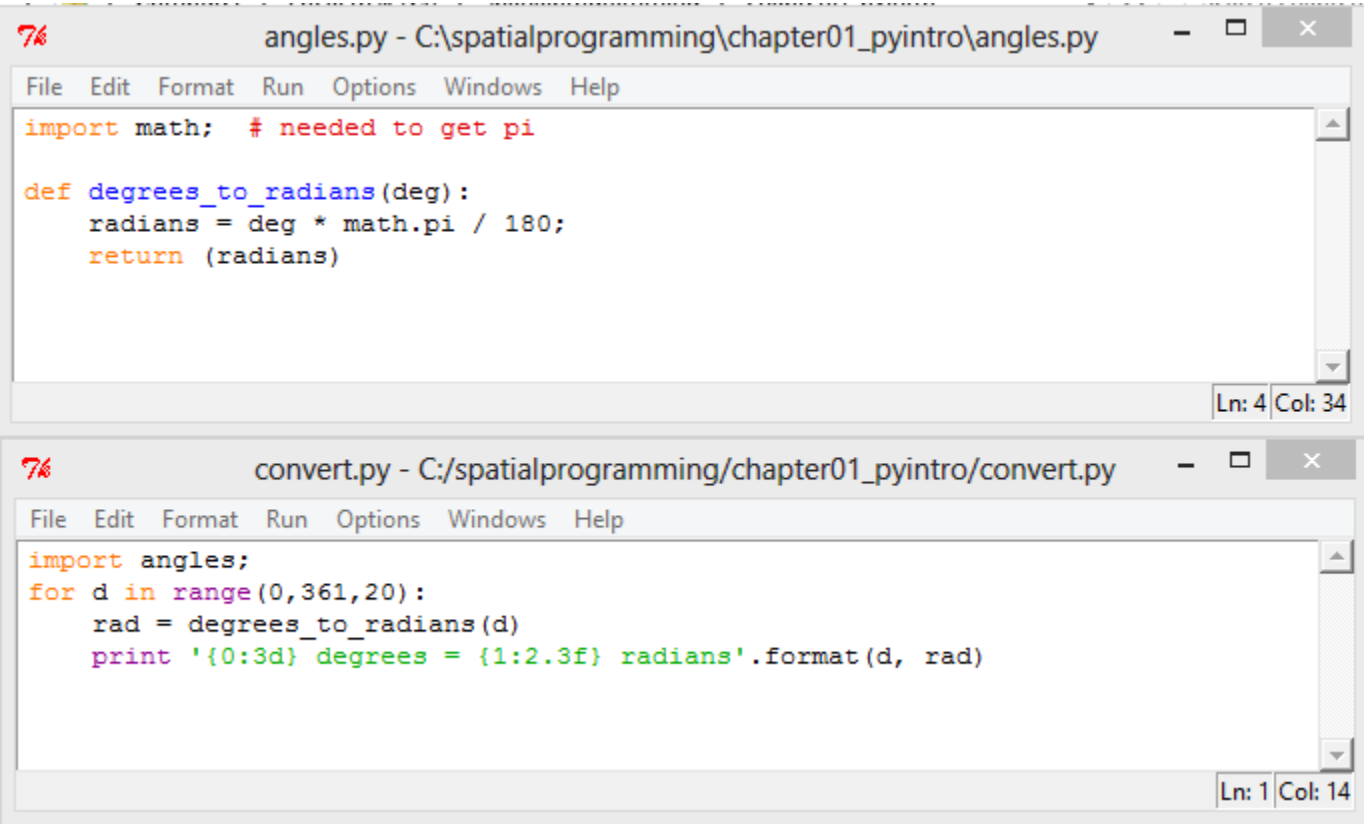
  - Did the plot happen again?
- Modules are imported only once
  - Second, and subsequent, imports are silently ignored
  - Not a good practice to do plotting, etc. from modules
  - Typically, one defines functions, classes, etc. in a module
  - Invoke those functions from a "main" or "driver" script
  - This way, "import" only pulls in the definitions

Search path for Python modules

- Obviously, matplotlib, numpy, etc. are not in the current directory
  - So how did those modules get found?
- Python's search path consists of:
  - The Python built-in libraries
  - Directory containing the calling script
    - Current directory if the calling script is the interpreter
  - The environment variable PYTHONPATH
    - Is it set on your system?
  - Modules placed in special directory in the Python installation
    - The matplotlib installer unpacked the Python modules here

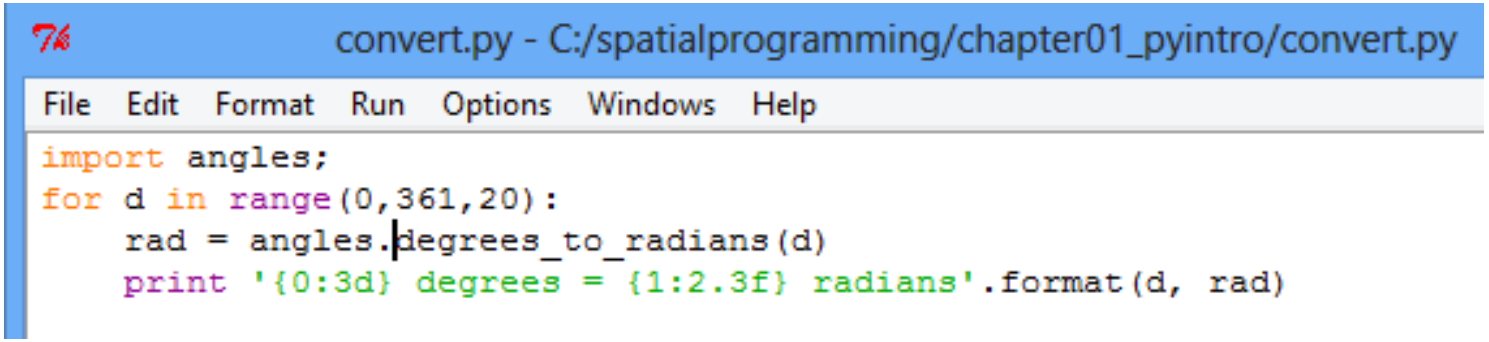
Defining and using modules

- Create two separate files in the same directory
  - angles.py, convert.py and then run the convert script
  - What is your result? Why? (Hint: see how pi was used)

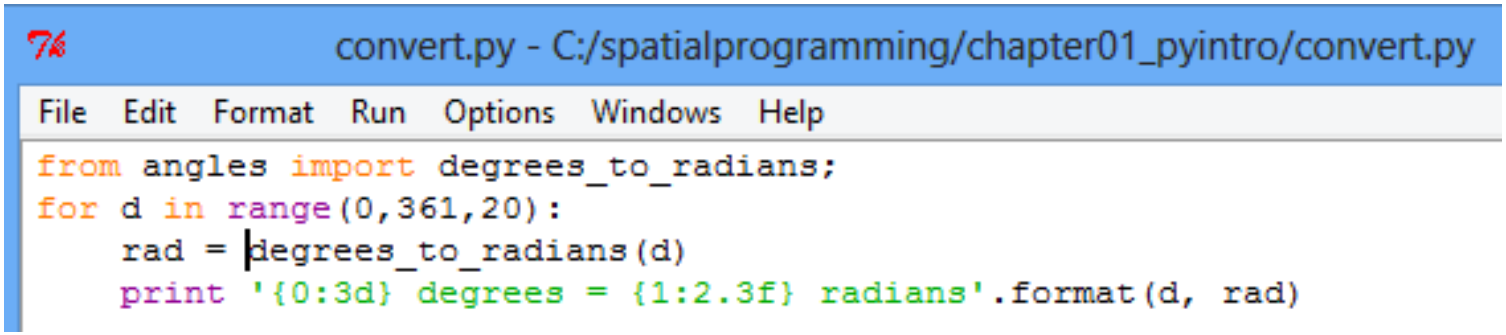


What happens when you import a module

- When you import a module containing function definitions, all the function definitions are read
  - But the functions remain in the "namespace" of that module



  - Alternately, you can bring the function into "this" namespace:



Types

Working with Strings

- What is this code doing? (copy-paste from PDF and try it out)

```
from angles import degrees_to_radians;

fmtA = '{0:3d} degrees';
fmtB = '{1:2.4f} radians';
nEquals = 40 - len(fmtA.format(10)) - len(fmtB.format(0,3.5));

fmt = fmtA + "="*nEquals + fmtB;

for d in range(0,361,45):
    rad = degrees_to_radians(d)
    print fmt.format(d, rad)
```

  - Why am I calling format() on the 4th line?
    - Why am I calling it with one parameter first, then with two parameters?
  - What is the 5th line doing?
    - What does + do? What does * do?

Two types of function calls

- len() is a generic function
  - Happens to work both on lists and on strings
- format() is a function that is defined by str
  - You call it on a string
  - Other methods you can call on a string let you strip white spaces, capitalize, lowercase, etc.
  - Can treat a string as a list of characters with fmt[34] etc.

```
>>> fmt = 'hello world';
>>> fmt.replace('o','E')
'hellE wErld'
>>> len(fmt)
11
>>> fmt.capitalize()
'Hello world'
>>> fmt[3]
'l'
>>> fmt[2]
'l'
>>> fmt[1]
'e'
```

Finding list of functions

- Can use Ctrl-Space to find list of available functions
- Can also call dir() on an object or a module

```
>>> dir(fmt)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__
format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__get
slice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mo
d__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook
__', '_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center',
 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index
', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', '
rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', '
strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'er
fc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gam
ma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'mod
f', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Getting help on a function

- Use the help() function:

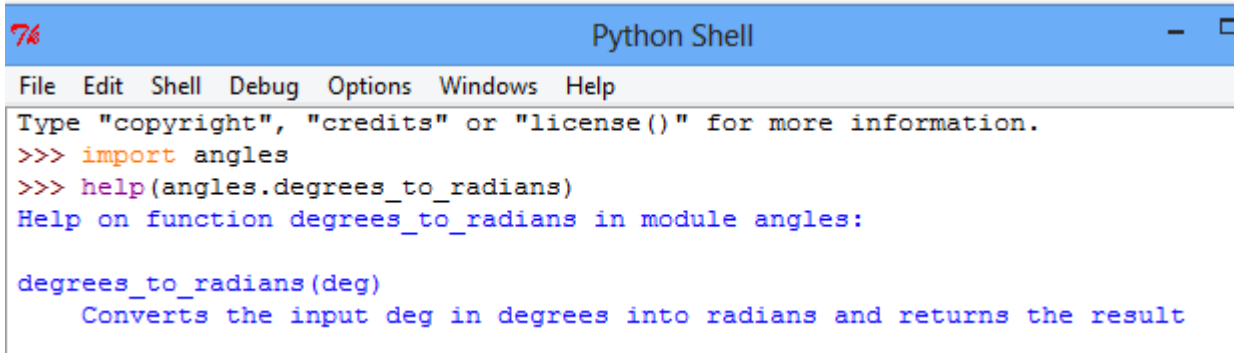
```
>>> help(math.pow)
Help on built-in function pow in module math:

pow(...)
    pow(x, y)

    Return x**y (x to the power of y).
```

How the help string is pulled out

- Documentation embedded in the source code
  - Document any non-obvious functions that you write

```
import math;  # needed to get pi
from math import cos, sin;

def degrees_to_radians(deg):
    """
    Converts the input deg in degrees into radians and returns the result
    """
    radians = deg * math.pi / 180;
    return (radians)
```

Using three quotes makes Python treat the following text verbatim (i.e., raw). However, any string will work.



Python's core data types

Type	Example
Number (includes integer, float)	34.57
String	'monty' "Python"
List	[34, 42.3, 'monty']

Dictionary	{
	'john' : 45345,
	'jane' : 35432
	}

Boolean	True False
Set	{ "harold", "jane", "john" }

- Lot more built-in types associated with specific capabilities (File, etc.); third party libraries define their own types

Working with List

```
>>> mylist = [ 42, 34.57, "monty", 'python' ]
>>> len(mylist)
4
>>> mylist[2]
'monty'
>>> mylist[-1]
'python'
>>> otherlist = mylist + ["hello", "world"]
>>> len(otherlist)
6
>>> print(otherlist)
[42, 34.57, 'monty', 'python', 'hello', 'world']
>>> otherlist.pop(3)
'python'
>>> print(otherlist)
[42, 34.57, 'monty', 'hello', 'world']
>>> otherlist.append( otherlist.pop(2) )
>>> print(otherlist)
[42, 34.57, 'hello', 'world', 'monty']
>>> print( otherlist.sort() )
None
>>> otherlist.pop(0)
34.57
>>> print(otherlist)
[42, 'hello', 'monty', 'world']
>>> otherlist.reverse()
>>> print(otherlist)
['world', 'monty', 'hello', 42]
```

Can mix types within a List. Normally we do not do this

Pop() returns the thing that was removed

Sort() does not return anything; it just sorts

Index out of range

- Python does bounds checking

```
>>> otherlist[87]

Traceback (most recent call last):
  File "<pyshell#77>", line 1, in <module>
    otherlist[87]
IndexError: list index out of range
>>>
```

2D arrays in Python

- Simply a list of lists

```
>>> arr = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9],
        [10, 11, 12]
           ]
>>> len(arr)
4
>>> len(arr[0])
3
>>> arr[1][2]
6
>>> arr[3]
[10, 11, 12]
```

- "List comprehension" is rather unique to Python:

```
>>> [row[1] for row in arr]
[2, 5, 8, 11]
```

Tuple, Set vs. List

- Tuples are like List
  - But they are immutable
  - Can not reassign a value to an item in a tuple
  - Not very often used
- A set contains unique values

```
>>> vowels = set(['a', 'e', 'i', 'o', 'u'])
>>> first  = set(['a', 'b', 'c', 'd', 'e'])
>>> vowels | first
set(['a', 'c', 'b', 'e', 'd', 'i', 'o', 'u'])
>>> vowels & first
set(['a', 'e'])
>>> vowels - first
set(['i', 'u', 'o'])
>>> 'e' in vowels
True
>>> 'y' in vowels
False
```

Dictionaries are associative arrays

- Allow you to index a type by something other than number

```
>>> dict = {
        "jane" : 34567,
        "john" : 42345,
        "ming" : 67893
           }
>>> len(dict)
3
>>> dict["jane"]
34567
>>> dict["jane"] = 98765
>>> dict["jane"]
98765
>>> dict["raj"] = 67534
>>> len(dict)
4
```

Nest dictionaries to represent objects

- Can use a dictionary to represent objects

```
>>> ktlx = {'site':"Twin Lakes, Oklahoma", 'lat':35.43, 'lon':-97.34}
>>> ktlx
{'lat': 35.43, 'lon': -97.34, 'site': 'Twin Lakes, Oklahoma'}
>>> ktlx['site']
'Twin Lakes, Oklahoma'
```

- Nesting dictionary allows you to represent a complex object

```
>>> ktlx = {'site': {"name":"Twin Lakes, Oklahoma", 'lat':35.43, 'lon':-97.34},
        'id' : "KTLX",
        'nearby' : ["KINX", "KVNX"]
          }
>>> ktlx["nearby"]
['KINX', 'KVNX']
>>> ktlx["id"]
'KTLX'
>>> ktlx["site"]
{'lat': 35.43, 'lon': -97.34, 'name': 'Twin Lakes, Oklahoma'}
```

Keys of a dictionary

- Looping through a dictionary

```
>>> sorted(ktlx)
['id', 'nearby', 'site']
>>> ktlx.keys()
['nearby', 'site', 'id']
>>> for key in ktlx.keys():
        print (key , "=>", ktlx[key])

('nearby', '=>', ['KINX', 'KVNX'])
('site', '=>', {'lat': 35.43, 'lon': -97.34, 'name': 'Twin Lakes, Oklahoma'})
('id', '=>', 'KTLX')
>>>
```

- Check if a key is present in a dictionary

```
>>> "site" in ktlx
True
>>> "location" in ktlx
False
```

Homework

- Print a table of distances between major Oklahoma cities
  - Use the Haversine formula for distance between two cities

$$2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

    - Do not use an external module for the great circle distance computation (I want you to implement the formula above yourself)
  - Look up the location of OKC, Tulsa, Norman, Lawton using any map
- Submit PDF document with code & snapshot of output
  - Example output (note: the numbers are incorrect):

	OKC	Tulsa	Norman	Lawton
OKC	0.00	180.04	69.81	479.22
Tulsa	180.04	0.00	238.46	657.57
Norman	69.81	238.46	0.00	419.96
Lawton	479.22	657.57	419.96	0.00