Introduction to Python

You need a quickie refresher on Python to get started.

Python's reputation precedes it. You've probably heard that is Python an interpreted language, that it has significant whitespace (which some find repulsive), and that it powers some of the most well known websites and computing systems in the world.

You may have heard that Python runs slow (true in certain circumstances). It doesn't support this or that programming construct (it might eventually if it is worthy enough). Every language has its warts, but Python is one of the few languages that both *trusts* and puts the developer *first*. By trust, I mean that Python doesn't cut off your nose to spite your face. You generally won't find yourself jumping through hoops to make the language do what you want. It doesn't put you in a padded room to protect you from yourself (although it doesn't dangle you from a cliff like C can either).

By putting the developer first, I mean that Python puts your productivity first. The key insight that the Python developers had (Guido in particular) is that a developer spends most of his/her time *reading* code, not *writing* it. Getting up to speed with someone else's (and your own if you've been away from it for awhile) code is easy because all of the stylistic choices have been made for you (no arguing about brace styles, indenting, and function layout). This frees the developer to focus on good code that does what it is supposed to, not extraneous details that don't matter much in the end.

Enough proselytizing. Let's do some Python. Start by opening up ActiveState Python by choosing Start – Programs – ActiveState ActivePython 2.4 – Pythonwin IDE. The Python interpreter will open up in a document window.

```
PythonWin 2.4.1 (#65, Mar 30 2005, 09:33:37) [MSC v.1310 32 bit
(Intel)] on win32.Portions Copyright 1994-2004 Mark Hammond
(mhammond@skippinet.com.au) - see 'Help/About PythonWin' for further
copyright information.
>>>
```

The interpreter is the thing that *runs* your program. It combines the process of *compiling* and *running* your code at the same time. You can run a Python program in two ways – by opening up an interpreter and running it interactively, or by calling the interpreter to run a program in a non-interactive mode (in the background).

We'll start with the ubiquitous "Hello World."

```
>>> print "Hello World"
Hello World
```

Data Types

Here is some example code that demonstrates the three ultra-basic data types that you'll need when working with Python.

```
>>> an_integer = 3
>>> an_integer
3
>>> a_float = 3.0
>>> a_float
3.0
>>> a_string = '3.0'
>>> a_string
'3.0'
>>> an_integer + a_float
6.0
>>> an_integer + an_integer
6
>>> a_string + a_float
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'float' objects
```

Notice that attempting to add the string and float throws an exception called *TypeError*. This error was thrown because Python can't automatically coerce the objects of type string and float. We can cast the string object into a float by calling the *float()* method on it.

```
>>> float(a_string) + a_float
6.0
```

Of course, if the string is really text and not numeric, the *float()* method method will throw an exception complaining about it.

```
>>> float('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for float(): a
```

Data Structures

Next, we'll cover the three basic data structures that you'll find when working with Python programs.

The first is a list. A list is your basic, integer indexed-based data structure.

```
>>> a_list = ['a','b','c']
>>> another_list = ['3', 3, 3.0]
```

Notice that *another_list* has objects of type *string, integer,* and *float.* Lists (actually all of the data structures) can contain objects of heterogeneous type.

The second is a dictionary, or hash table. A dictionary is used when you want to be able to access something by *key*, rather than by index alone. Use a dictionary when you want to search through a large group of things, rather than interating through a list and testing each member. Also thing to note is that a dictionary's keys are always strings (or hashable objects) and that duplicates are not allowed (you can't have two items in a dictionary with the key *'a'* for example).

```
>>> a_dictionary = {'a':1, 'b':2, 'c':3}
>>> a_dictionary['a']
1
```

The third major data type is the tuple. A tuple is just like a list, except that it cannot have items added or removed from it once it is instantiated. One way to think of a tuple is as a "read-only" type of list.

```
>>> a_tuple = ('a','b','c')
```

Conditionals

Decisions, decisions, decisions… a program isn't really a program unless you can alter an operation based on some input. You universally do this with a conditional statement. In Python, as with many languages, this is done using an *if…else* construct.

```
a_string = 'a'
if a_string == 'a':
    print 'it was a'
else:
    print 'it was not a'

it was a
if a_string == 'a':
    print 'it was a'
elif a_string == 'b':
    print 'it was b'
else:
    print 'it was neither'
```

```

it was a
```

Notice that the print statements are indented underneath the conditional statements. Python denotes code blocks with indentation, rather than using curly braces or some other punctuation. As long as the code blocks are all evenly indented, it will work. The convention is to use 4 spaces for indenting each code block, and usually great care is taken to not mix in tabs and spaces to make it easy to send code around the internet – compensating for the various system and tab stops that might be out there.

Another important item to note here is that = is different than ==. One equals sign is for *assignment* and two equals signs are for *comparison*. For example, this code snippet isn't going to do what you'd hoped for.

```
>>> if a_string = 'b':
  File "<stdin>", line 1
    if a_string = 'b':
                ^
SyntaxError: invalid syntax
```

Loops

Computers are computers because they can do things a lot of times in a row and they don't complain about it. There are two ways to do a lot of things in a row in Python. The first is a *for* loop and the second is a *while* loop.

```
for item in a_list:
    print 'lowercase: ', item, 'uppercase: ', item.upper()
lowercase:  a uppercase:  A
lowercase:  b uppercase:  B
lowercase:  c uppercase:  C
```

Another way of printing the results is to use string interpolation. The string substitution syntax is very similar to the *printf* substitution in C. If you find yourself adding a lot of strings together into one larger one, use string interpolation instead of the + operator. It will make things easier to read and easier to change.

```
for item in a_list:
    print 'lowercase: %s uppercase:%s'% (item, item.upper())
```

Functions

Functions allow you to consolidate operations, eliminate code redundancy, and clean up your code. Unlike other languages, functions in Python rely on something that is casually called "duck typing." Duck typing means "if it acts like a duck and quacks like a duck, we'll treat it like a duck." As long as the object passed into the function has the proper attributes and/or methods, the function will happily call and work with it.

```
def print_it(astring):
    print astring
>>> print_it('Howard')
Howard
```

A function is started with a *def* for define. Then comes the name and the list of parameters inside of parenthesis. Our *print_it* function takes a single parameter, *astring*, and prints it.

You can also define default arguments in function. This is commonly done to reduce line noise in the code and allow flexibility.

```
def print_it_two(astring, salutation="Mr."):
    print salutation, astring
>>> print_it_two('Howard Butler')
Mr. Howard Butler
>>> print_it_two('Cunningham', salutation="Mrs.")
Mrs. Cunningham
```

Objects

In Python, everything is an object. This includes things like functions, class definitions, and code itself. All of this object stuff doesn't mean that you *have* to program in an object-oriented way (unlike some languages like Ruby, for example). You can still write a straight-ahead, linear program that manipulates some text, or a module that is just a bunch of functions that are called in a specific order.

Even though you aren't required to program in an object-oriented way, it is helpful to understand how to use objects in Python. All of the code that you'll import and use, including stuff from the standard library, is arranged in objects.

I find it helpful when working with object-oriented code to think of verbs. Objects *have* things, objects *are* things, and objects *do* things.

Have

When we say that objects *have* things, we mean that we use objects to carry data. You will hear the words *property* and *attribute* to describe this. There are slight differences between a property and an attribute of an object, but in Python, for the most part, you shouldn't have to care. Just remember when someone says that an object *has* something, they are referring to the data that it carries.

Are

When we say that objects *are* things, we mean that an object is of some type. A type might sometimes be coerced into another type, or it might inherit attributes and methods from a parent type (called a subclass or subtype).

Do

When we say that objects *do* things, we mean that we use objects to perform an action on data. You will hear the words *method* or *function* to describe this. It might perform this action on or using one of its own attributes or data that you give it to act on.

You define an object by using the *class* keyword.

```
class Bear:
    def __init__(self, name='Yogi'):
        self.name = name
    def growl(self):
        print 'grrrr'
    def eat(self, food):
        print self.name, 'eats', food
    def __str__(self):
        return 'My name is %s' % (self.name)
```

The first thing we do is define an __init__ method. __init__ is a special or "magic" method in Python in which we define the data the class will carry along with it (or *have*). Note the use of a default method, with the Bear's name defaulting to Yogi. The __str__ method defines what is returned when we try to get a string representation of the Bear. In our case, we just return a string that reports the Bear's name…

growl and *eat* are methods that define something that the Bear class *does*.

```
>>> yogi = Bear()
>>> yogi.eat('tomatoes')
```

```
Yogi  eats  tomatoes
>>> yogi = Bear()
>>> yogi.eat('tomatoes')
Yogi eats tomatoes
>>> print yogi
My name is Yogi
>>> yogi.growl()
grrrr
```

We can find out more about what yogi *is* by asking its type with the *type()* function.

```
>>> type(yogi)
<type 'instance'>
```

And we can check what type it is by comparing it to its class.

```
>>> isinstance(yogi, Bear)
True
```

Modules and Packages

Python Module

A module is a file containing Python statements with a .py extension. Modules are used to reduce the amount of typing you do at the interpreter prompt, and, of course, to reuse code in different applications.

For example, with an editor create a new file called `wkt.py` in your current directory and type into it the following:

```
def wktpoint(x, y):
    return 'POINT (%f %f)' % (x, y)
```

This defines a function which takes a coordinate in the form of two floats, interpolates the coordinate values into a well-known text representation of a point, and returns this string.

The module is loaded using a Python `import` statement

```
>>> import wkt
```

dropping the .py extension, and afterwards the function is callable using :

```
>>> wkt.wktpoint(1, 2)
'POINT (1.000000 2.000000)'
>>>
```

Notice that after you import the wkt module, your current directory now contains a `wkt.pyc` file. This is the module as compiled bytecode, and speeds up the next import of the module. The Python interpreter compares the timestamps on the compiled and source module so that it is recompiled whenever the source has been changed.

Module Search Path

Note that we didn't specify any path to the wkt module. How is it found? By default Python will search for files in the following directory order:

1. current directory (interpreter prompt) or directory of the input script
2. directories specified in the `PYTHONPATH` environment variable
3. installation-dependent system paths, such as `c:\python24\lib` for the library of standard modules and `c:\python24\lib\site-packages` for installed non-standard modules.

The `PYTHONPATH` variable is useful with uninstalled bundles such FWTools.

The dir() function

The built-in dir() function returns a sorted list of the names defined in a module. This is all names: variables, functions, classes. Using our wkt.py as an example:

```
>>> import wk
>>> dir(wkt)
['__builtins__', '__doc__', '__file__', '__name__',
'wktpoint']
>>>
```

The first four names are common to all modules and then there is our `wktpoint` function.

Finding Module Constants

A module is a great place to keep constants, and all of our GIS modules define a few. If you want to see all the mapscript integer constants and their values:

```
>>> from mapscript import mapscript
>>> [(n, eval('mapscript.%s' % (n))) \
... for n in dir(mapscript) \
... if type(eval('mapscript.%s' % (n))) == type(1)]
[('FTDouble', 2), ('FTInteger', 1), ('FTInvalid', 3),
('FTString', 0), ('MAX_PARAMS', 10000),
('MESSAGELENGTH', 2048), ('MS_AUTO', 9), ('MS_BITMAP', 1),
('MS_CC', 8), ('MS_CGIERR', 13), ('MS_CHILDERR', 31),
('MS_CJC_BEVEL', 1), ...]
```

the eval function evaluates a string as a Python expression. For example:

```
>>> eval('1 + 1')
2
>>>
```

we use it above within a Python list comprehension to generate a list of names, filter those that have integer type values and return the name and value as a tuple. List comprehensions are an increasingly popular Python construction. The one above is quite complex. Here are simpler examples that build up to the same level of complexity:

```
>>> [x for x in [1, 2, 3]]
[1, 2, 3]
>>> [(x, 2*x) for x in [1, 2, 3]]
[(1, 2), (2, 4), (3, 6)]
>>> [(x, 2*x) for x in [1, 2, 3] if x > 1]
[(2, 4), (3, 6)]
```

Packages

A package is a directory of modules and allows us to structure the module namespace. It also allows developers to avoid module name conflicts. We can all have our own `geometry` module as long as its contained within a unique package.

Previously we imported the mapscript module from the mapscript package

```
>>> from mapscript import mapscript
```

Another example is the xml package from the standard library. Browse to `C:\Python24\Lib\xml` and note that it contains, among other things, `sax` and `dom` sub-packages. This separation is for efficiency as much as namespace structure, as the SAX and DOM approaches to XML are not usually combined in a single application, and there's no point in loading a module that won't be used.