Basic Review of Python

Overview

1. What is python?
2. Basic data types
3. Flow control
4. Functions
5. Classes
6. Importing Functions and Classes from Libraries

What is python?

What is python?

A high-level dynamic programming language with extensive GIS and data analysis applications

- Interpreted
- Dynamically typed
- Uses whitespace
- Cross-platform (caveat: windows seems to have less support)
- Current version is 3.4.x, but most scientific and geospatial packages require 2.7.x

What is python?

As python is interpreted:

- Can be written interactively using the python interpreter at the command line
    - Open command line, type `python`
- Can be run from .py files for longer or more complicated applications
    - Write .py files in `IDLE` or `Spyder`
- Can use a combination of these two
    - `ipython notebook`

DEMO

What is python?

Do you have my favorite fruit, a simple example:

```
favorite = "peach"
inventory = ["apple", "banana", "pear", "orange", "guava"]

# iterate through list of fruit inventory
for fruit in inventory:

    # see if fruit is same as favorite
    if fruit == favorite:  # test inline comment

        # .format is string method, replaces {} with fruit
        print "You have my favorite fruit, {}.".format(fruit)

        # in this case, break out of loop to stop testing
        break

# notice indentation level: else goes with for loop, not if
else:
    print "You don't have my favorite fruit."
```

Basic Data Types

Basic Data Types

- Integers

Integers store positive and negative numbers without decimals:

```
>>> type(5)
<type 'int'>  # int means integer

# ints support typical numerical operations:
>>> (2 + 5) ** 2 - 10 * 8
-31
```

Basic Data Types

- Integers
- Floats

Like integers, but used for decimal values:

```
>>> type(13.45)
<type 'float'>
```

WARNING: Watch out for integer division:

```
>>> type(17 / 5)
<type 'int'>  # what? why?

>>> 17 / 5
3             # integer division drops decimals!

>>> 17.0 / 5  # add a .0 to either side for
3.4           # floating-point division
```

Basic Data Types

- Integers
- Floats
- Strings

Strings hold text and other characters:

```
>>> type("5 is a number.")
<type 'str'>

>>> 'this is also a string'

>>> """as is this"""  # used for spanning multiple lines

>>> "strings have special characters like \t, \n, \r..."

>>> print "a \\t looks like \t."  # escape \ with another \
'a \t looks like        .'        # \t is a tab character
```

Basic Data Types

- Integers
- Floats
- Strings

String can also be concatenated and sliced:

```
>>> s1 = "foo"

>>> s2 = "bar"

>>> concat = s1 + s2
'foobar'

>>> concat[4]  # get character at index 4 (the fifth letter)
'a'

>>> concat[0:2]  # slice from 1st character (0) TO 3rd (2)
'fo'

>>> concat[:2]  # does the same thing
'fo'

>>> concat[2:]  # ommitting 2nd index goes to end
'obar'

>>> concat[-2:]  # can use negative indicies for slice
'ar'
```

Basic Data Types

- Integers
- Floats
- Strings
- Lists

Lists hold collections of items:

```
>>> fruit = ["apple", "banana", "pear", "orange", "guava"]

>>> things = ["car", 3, 5.32, 't']  # holds multiple types
```

Lists can also be concatenated and sliced:

```
>>> l1 = [1, 2, 3, 4]

>>> l2 = [5, 6, 7, 8]

>>> l1[0] + l2[2:]
[1, 7, 8]
```

Individual elements can be replaced as lists are *mutable*:

```
>>> l1 = [1, 2, 3, 4]

>>> l1[3] = 67

>>> print l1
[1, 2, 3, 67]
```

More Basic Data Types

- Tuples

Tuples are like lists, but *immutable*:

```
>>> t1 = (1, 2, 3, 4)

>>> t2 = (5, 6, 7, 8)

>>> t1 + t2
(1, 2, 3, 4, 5, 6, 7, 8)

>>> t1[:3]
(1, 2, 3)

>>> t1[2]
3

>>> t1[2] = 356
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

More Basic Data Types

- Tuples

- Sets

Sets are like tuples and lists, functioning as a collection of objects. However, sets can only contain one of any object:

```
>>> l1 = [1, 2, 3, 4]

>>> l1 * 3
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]

>>> set(l1 * 3)
set([4, 1, 2, 3])  # order is not maintained
```

More Basic Data Types

- Tuples
- Sets
- Dictionaries

Dictionaries are key-value stores. That is, look up a key to find its value:

```
>>> d = {"John": 26, "Henry": 44, "Maria": 34, "Olaf": 13}

>>> d["John"]
26
```

Dictionaries can hold other dictionaries:

```
>>> feature41 = {"name": "Library", "size": 5789}

>>> feature93 = {"name": "School", "size": 15765}

>>> features = {41: feature41, 93: feature93}

>>> print features
{41: {'name': 'Library', 'size': 5789},
93: {'name': 'School', 'size': 15765}}

>>> features[93]["name"]
'School'
```

More Basic Data Types

- Tuples
- Sets
- Dictionaries
- Booleans

- Booleans are used for logic, to hold `True` or `False`.
- Cannot have any other values
- `True` and `False` are reserved words in python

Flow Control

Flow Control

- Conditional Statements

Allows different operations depending on given conditions:

```
>>> x, y = 37, 99

# use ==, <, >, <=, >=, != for comparisons
>>> if x >= 25:
...     if y == 99:
...         z = 5
...     elif y > 1:
...         z = 12
... else:
...     z = 17
# what is z?

# False, 0 of any type, an empty sequence like "", [], (),
# an empty mapping like {}, and None all evaluate as false
>>> if 0:
...     print "true"
... else:
...     print "false"
'false'
```

Flow Control

- Conditional Statements

```
# can use not, and, or to combine statements
>>> if not ((5 or 0) and not (True or False)):
...     # what will we get?

# null is None in python, and is an identity
# test idenity using is, not ==
>>> if something is not None:
...     print "There is one of these"
```

Flow Control

- Conditional Statements

- While Loops

While loops are used to repeat operations until a condition is met:

```
# don't do this: infinite loop
>>> while True:
...     pass

# this is okay -- effectively a do-while loop
>>> i = 0
>>> while True:
...     i += 1
...     if i > 5:
...         break

# this is better
>>> i = 0
>>> while i <= 5:
...     i += 1
# what will i be?
```

Flow Control

- Conditional Statements
- While Loops
- For Loops

For loops are used to *iterate* over an *iterable*:

```
# lists are iterable
>>> fruits = ["apple", "banana", "pear", "orange", "guava"]
>>> for fruit in fruits:
...     print fruit
'apple'
'banana'
'pear'
'orange'
'guava'

# range() and xrange() can generate a list of #s
>>> sum = 0
>>> for number in xrange(4, len("nondeterministically"), 2):
...     sum += number
>>> print sum
88

# list comprehension uses for; strings are iterable
>>> str1 = "abcdefghijklmnopqrstuvwxyz"
>>> list1 = [l for l in str1 if l in "zyghseivnsaby"]
>>> print list
['a', 'b', 'e', 'g', 'h', 'i', 'n', 's', 'v', 'y', 'z']
```

- tuples are iterable
- dictionaries are iterable: loop over keys
- file objects are iterable: loop over lines

Flow Control

- Conditional Statements
- While Loops
- For Loops
- Try/Excepts

The try/except pattern can be used to handle *exceptions*:

```
# an exception
>>> x1 = 5

>>> x2 = "hello"

>>> print x1 + x2

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

# execution stops with an unhandled exception


>>> try:
...     print x1 + x2
... except TypeError:
...     print "One or more arguments were of the wrong type."
... # finally executes with or without a handled exception
... finally:
...     print "Thanks for using our adding machine."

One or more arguments were of the wrong type.
Thanks for using our adding machine.

# execution can continue because the exception was handled
```

Functions

Functions

- Used to break complex operations into small, easily understood pieces
- Create modular, reuseable code

```
>>> def funct(x):
...     return x

>>> funct(56)
56

>>> def square(x):
...     return x * x  # x is not same x in funct()

>>> square(4)
16

# some functions built-in, as we have seen
>>> len('This is a sentence composed of many characters')
46

>>> len([1, 2, 3, 4])
4
```

Functions

We can put a bunch of this together:

```
>>> FILE = "./file.txt."  # line are "first", "second", "third", etc.

>>> def get_sixth_char(string):
...     """Returns 6th char in a string, unless IndexError or newline"""
...     try:
...         char = string[5]
...     except IndexError:
...         char = None
...     else:
...         if char == "\n":
...             char = None
...     return char

>>> linecontent = {}

# with block ensures file closes safely
# open() is a function with two arguments, the first being the file to open,
# the second is the mode, in this case 'r' for read
>>> with open(FILE, 'r') as infile:
...     for linenumber, line in enumerate(infile):
...         linecontent[linenumber] = get_sixth_char(line)

>>> print linecontent
{0: None, 1: 'd', 2: None, 3: 'h', 4: None, 5: None, 6: 't', 7: 'h',
8: None, 9: None}
```

Classes

Classes

- Classes are objects
- Objects can have attributes, and you can do stuff to an object
  - Doors open, switches turn on and off
- Objects can be helpful and can clean up code, but are complex to implement
- GIS libs use objects extensively
- Everything in python is an object
  - strings, dicts, ints: all objects, with properties and methods

*a method is a function attached to an object

Classes

```
class Person(object):

    # __init is a required method to initialize class
    def __init__(self, name, age, heightInches, weightPounds):
        # these are all class properties
        self.name = name
        self.age = age
        self.heightInches = heightInches
        self.weightPounds = weightPounds

    def introduce(self, othername=None):
        """Class method to introduce Person"""
        if othername:
            # notice format is a method of the string class
            print "Hello {}. My name is {}.".format(othername, self.name)
        else:
            print "Hi, I'm {} and I am {}.".format(self.name, self.age)

    def getWeightInKilos(self):
        """Method returns Person weight in kilograms"""
        return self.weightPounds / 2.2

    def getHeightInMeters(self):
        """Method returns Person height in meters"""
        return self.heightInches * 2.54 / 100
```

Classes

We can use our Person class like this:

```
# construct an instance of the Person class
>>> fred = Person("Fred", 52, 73, 189)

# we see fred is a Person object
>>> fred
<__main__.Person object at 0x1088d2ed0>

# get properties of fred instance
>>> print fred.name, fred.age, fred.heightInches, fred.weightPounds
Fred 52 73 189

# use methods of Person class with fred instance
>>> fred.introduce(othername="Juan")
Hello Juan. My name is Fred.

>>> fred.introduce()
Hello, my name is Fred and I am 52 years old.

>>> fred.getWeightInKilos()
85.9090909090909
```

Importing Functions and Classes

Import Statements

- Functions and classes make code modular and reusable
- Many programmers much smarter than you or me have already created frameworks and utilites solving many problems
- Why try to reinvent the wheel?

Import Statements

- Functions and classes make code modular and reusable
- Many programmers much smarter than you or me have already created frameworks and utilites solving many problems
- Why try to reinvent the wheel?

Use what's already been done!

Import Statements

- In python speak:
  - a module contains functions and/or classes
  - a package contains multiple modules
- The standard library contains many modules and packages.
- We can access them like this:

```
>>> import os

>>> os
<module 'os' from '/usr/local/Cellar/python/2.7.8/Frameworks/Python.framework/V...
```

Try it yourself. Then run `dir(os)` to see everything defined in the os package.

Import Statements

- We can import more than just the packages and modules in the standard lib:

```
# any package that has been installed to the python we are using is available
>>> import arcpy

# we want to get OGR and GDAL, but they are modules in a package...
>>> from osgeo import ogr
>>> from osgeo import gdal

# want a specific function or class in a package?*
>>> from arcpy import Buffer_analysis

# think a function is long-winded in the name?*
>>> from arcpy import CreateFileGeodatabase_management as CreateFGDB
```