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.

Geometry Operations: OGR and GEOS

The GEOS library

http://geos.refractions.net

provides the spatial predicates originally used in PostGIS, now OGR, and soon MapServer. In this exercise we'll explore unions, intersections, differences, buffers, and work our way up to the task of creating a buffered union of many features from a shapefile.

Matplotlib

Along the way we are going to use the matplotlib package for visualization of our results. This is matlab-like software that is attracting a lot of attention from Python users. If we have time at the end of the workshop, some of you may be interested in digging deeper into matplotlib.

```
>>> from matplotlib import pylab
>>> pylab.plot()
[]
>>> pylab.show()
```

This creates an output window into which we'll render geometries.

Geometries

Let's create two simple, overlapping polygons using the same string interpolation and WKT factory method as in the previous exercise:

```
>>> r1 = {'minx': -5.0, 'miny': 0.0, 'maxx': 5.0, 'maxy': 10.0}
>>> r2 = {'minx': 0.0, 'miny': -5.0, 'maxx': 10.0, 'maxy': 5.0}
>>> template = 'POLYGON ((%(minx)f %(miny)f, %(minx)f %(maxy)f, %(maxx)f %(maxy)f, %(maxx)f %(miny)f, %(minx)f %(miny)f))'
>>> w1 = template % r1
>>> w2 = template % r2
```

You could print these to verify. Next we import the ogr module and use its WKT factory to create instances of ogr.Geometry:

```
>>> from gdal import ogr
```

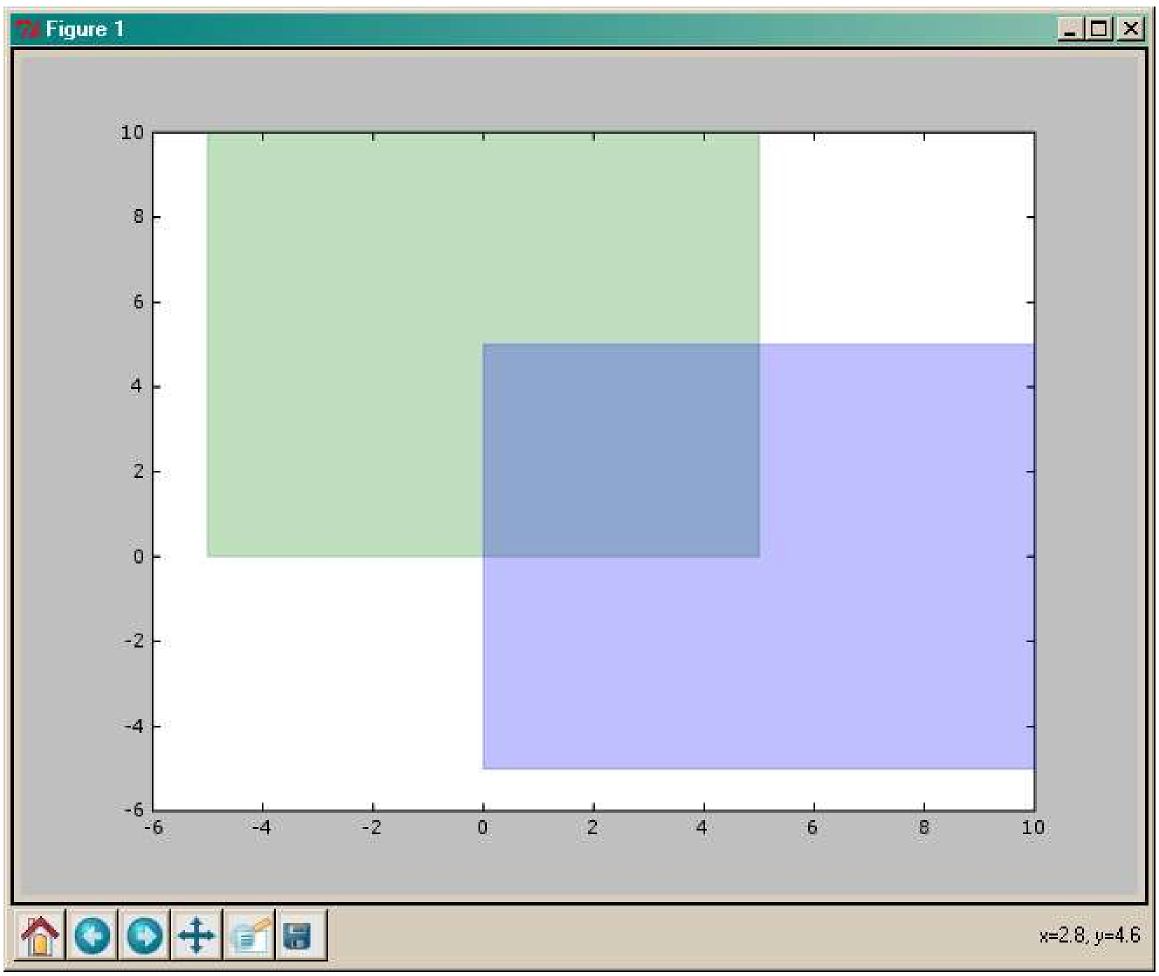
```
>>> g1 = ogr.CreateGeometryFromWkt(w1)
>>> g2 = ogr.CreateGeometryFromWkt(w2)
```

Plotting

Initially we downloaded a helper file named plot.py. It contains two functions for plotting geometries in the matplotlib window.

```
>>> from plot import plot_poly, plot_line
>>> plot_poly(g1, color='green', alpha=0.25)
>>> plot_poly(g2, color='blue', alpha=0.25)
```

The result should be something like

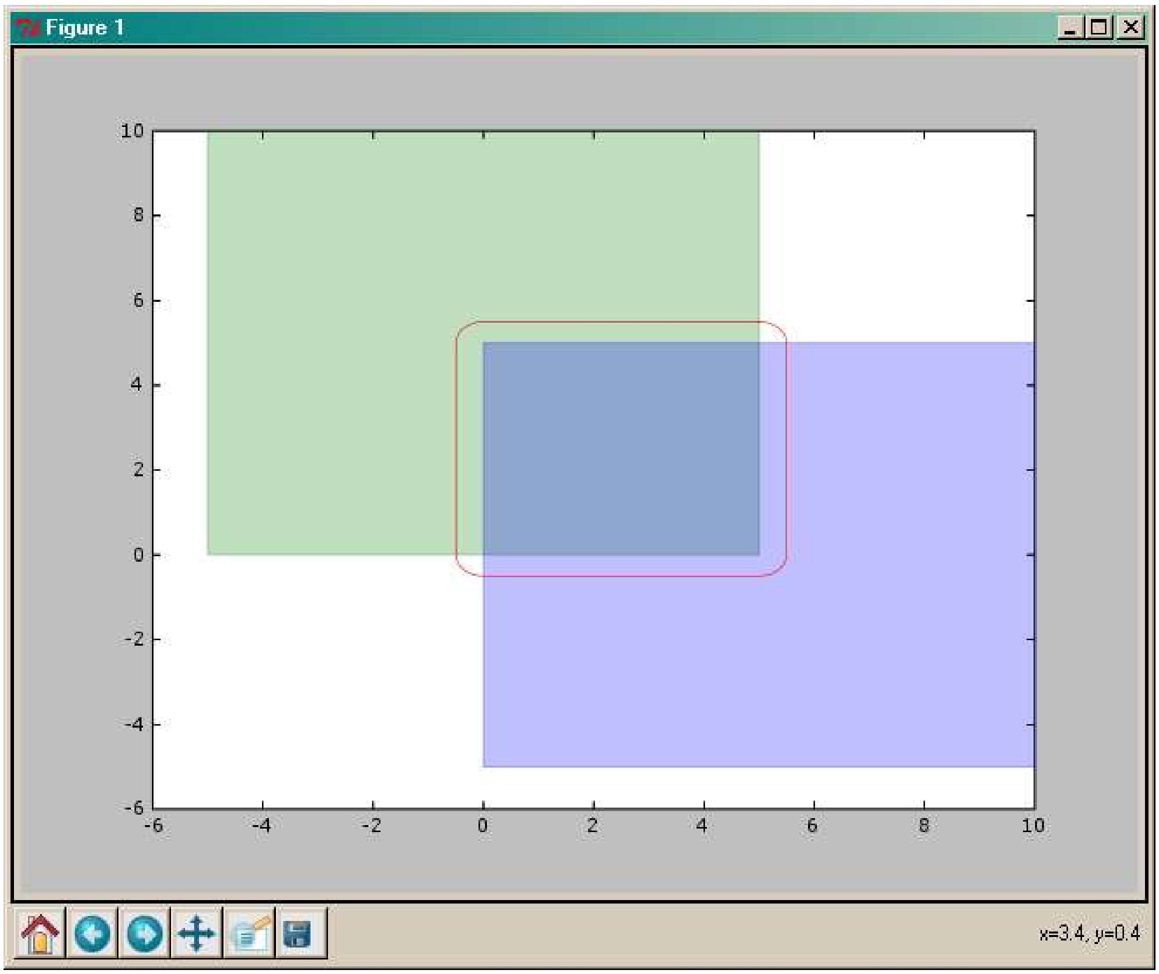


Intersection

Let's try the `Intersection()` and `Buffer()` methods of `ogr.Geometry` first.

```
>>> inter = g1.Intersection(g2)
>>> buffered_inter = inter.Buffer(0.5)
>>> plot_line(buffered_inter, color='red')
```

The result:

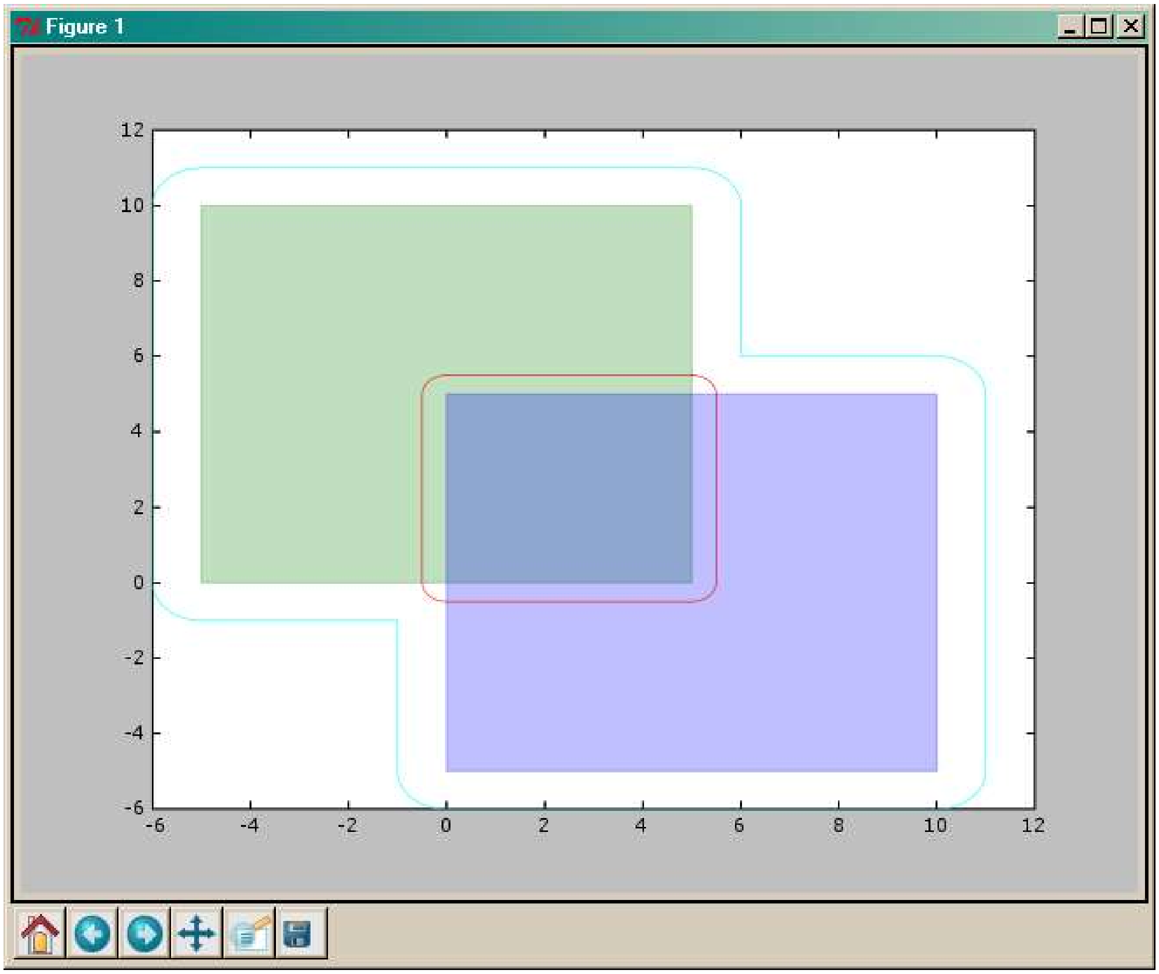


Union

Now the `Union()` method.

```
>>> union = g1.Union(g2)
>>> buffered_union = union.Buffer(1.0)
>>> plot_line(buffered_union, color='cyan')
```

and the results



Lifelike Geometries

Let's close up that output window and move on to less artificial geometries. At c:\ms4w\python\data\world_borders.shp is a world borders shapefile derived from VMAP0 by Schuyler Erle, Rich Gibson, and Jo Walsh. We'll use the OGRFeatureIterator class from the fiter.py helper module to select several of the features from this shapefile:

```
>>> from fiter import OGRFeatureIterator
>>> filename = r'c:\ms4w\python\data\world_borders.shp'
```

Now, define a spatial bounding box and an OGR attribute filter to constrain features. The GEOS `Union()` operation is very slow, and we don't want to wait for too many polygons.

```
>>> bounds = (-10.0, 30.0, 20.0, 60.0)
>>> attrfilter = "fips_cntry = 'UK'"
```

Next, we create a list to hold selected features, and declare the name `u`, for our union geometry, to begin with the value `None`.

```
>>> geoms = []
>>> u = None
```

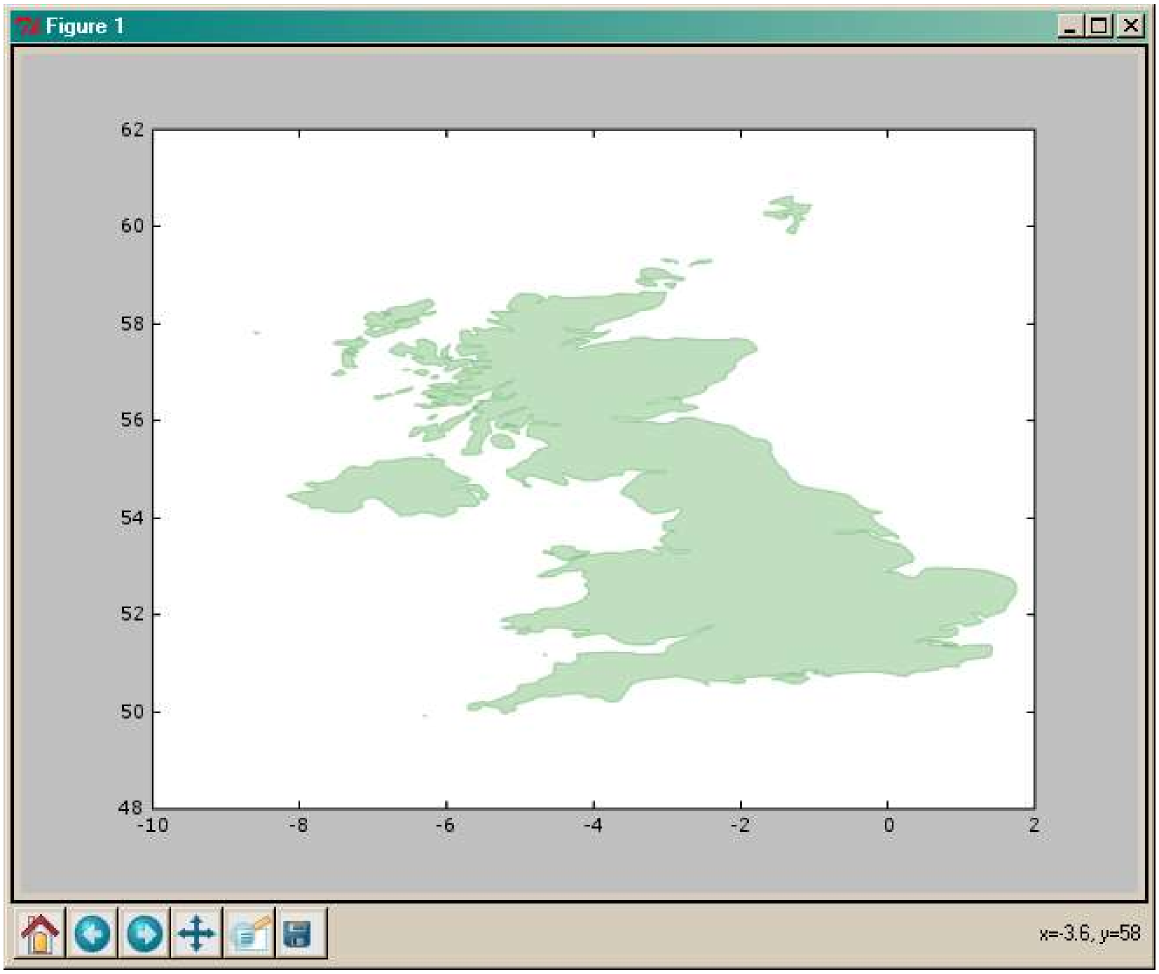
The following iteration appends each selected geometry g and builds up the union of all selected geometries. Iterators are a very common construct, and a big component of Python flavor. The `if/else` blocks below ensure that we begin our union geometry as the clone of a selected geometry, and clone only once.

```
>>> for g in OGRFeatureIterator(filename, bounds,
attrfilter):
...     geoms.append(g)
...     if u:
...         u = u.Union(g)
...     else:
...         u = g.Clone()
...
>>>
```

Now, let's plot the selected geometries using the previously imported `plot_poly()` function.

```
>>> for g in geoms:
...     plot_poly(g, color='green', alpha=0.25)
...
>>>
```

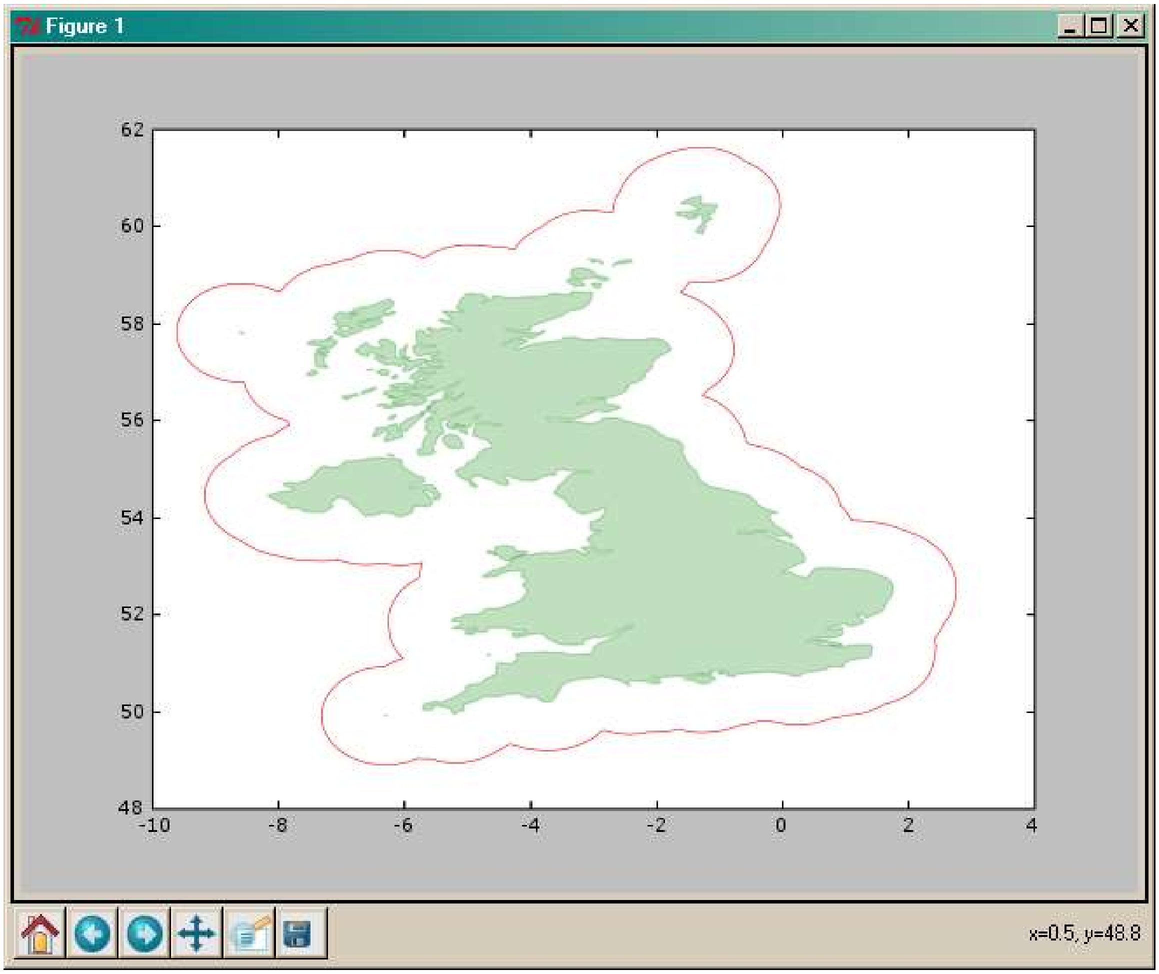
the result



Now, buffer the union and plot it. This is a fairly lengthy operation ...

```
>>> buffer = u.Buffer(1.0)
>>> plot_line(buffer, color='red')
>>>
```

The results



Continuation

In the workshop's extra time, some of you may want to try saving these geometries to a file using ogr.py as we did in the previous tileindex exercise, and display them in OpenEV. Some may be interested in grabbing some features via WFS and plotting them in the same window with the buffered UK features.

Geocode an address and plot it on an orthophoto in the coterminous US using MapScript

> You want to use Python MapScript to geocode addresses and return a DOQ map with the address plotted on it as a point.

Geocoding is very popular these days. Users frequently ask questions on the MapServer list how to incorporate geocoding into existing applications. Websites like MapQuest, Yahoo! Maps, and Google Maps have popularized the concept of using an address as an initial map navigation tool.

The process of geocoding, or turning a street address into a longitude/latitude pair, is a difficult one. It requires two key components – complete and specialized data, and algorithms to turn that data into coordinates. Luckily, Schuyler Erle of *Mapping Hacks* has an Open Source geocoding solution available. Using the US Census Tiger street data, http://geocoder.us provides Perl software and a SOAP and XMLRPC remote query mechanism for geocoding. This means you can download your own copy of the Tiger database and provide geocoding for your own commercial applications. (In fact, if you plan to use the software commercially, you **must** build your own database rather than utilizing the remote query interface provided by geocoder.us.)

Getting a lat/lon for an address

We will be using the remote query functionality of geocoder.us today. Start by importing the *xmlrpc* library and setting up a proxy to geocoder.us's XMLRPC service.

```
import xmlrpclib
geocode_url = 'http://rpc.geocoder.us/service/xmlrpc'
p = xmlrpclib.ServerProxy(geocode_url)
address = "615 WASHINGTON AVE SOUTHEAST, MINNEAPOLIS, MN"
```

Next, execute a call to the XMLRPC service

```
result = p.geocode(address)
print result
C:\>geocode.py
[{'city': 'Minneapolis', 'prefix': '', 'suffix': 'SE', 'zip': 55455, 'number': 6
15, 'long': -93.229894000000002, 'state': 'MN', 'street': 'Washington', 'lat': 4
4.973664999999997, 'type': 'Ave'}]
```

Just like that, we have our coordinates. The XMLRPC service returns a list of dictionaries that we can use to get our coordinate information. Next, we'll import mapscript and setup a mapObj that will draw our map.

Generating a basic DOQ map with TerraServer and MapScript

```
try:
    import mapscript.mapscript as mapscript
except ImportError:
    import mapscript

amap = mapscript.mapObj()
```

Our mapObj is named amap because *map* is a function name in python. Normally, you would instantiate a mapObj with an existing mapfile. In our case, however, we want the entire thing to be self-contained in the script. We will build up our mapObj, layerObj's, and styling all in mapscript…

The other kind of funny thing we do here compensates for the way the mapscript package is installed on the workshop machines. We try to import our workshop mapscript, and if it isn't there in the package format, we just try to import it the regular way.

```
amap.height = 800
amap.width = 1100

debug = 1
ms_debug = 0

if ms_debug:
    amap.debug = mapscript.MS_ON
amap.setProjection('init=epsg:2163')
```

We next define our map width and height. In addition, we define some debugging variables. This will allow us to see and set some diagnostics as we develop that we can easily turn off once we have a finished script. A more sophisticated approach would utilize Python's *logging* module, but this is a short hack, right?

```
lat, lon = result[0]['lat'], result[0]['long']
pt = mapscript.pointObj()
pt.x = lon
pt.y = lat

if debug:
    print '-------  DD Coordinates  -------'
    print 'x: %s y: %s' % (pt.x, pt.y)
    print '--------------------------------'
```

We need to take the coordinates that the geocoder gave us and turn them into a pointObj. We'll project that point into our mapObj's coordinate system, use it to define the extent of the map, and then use it to plot a point showing the location of the address.

```
ddproj = mapscript.projectionObj('proj=latlong,ellps=WGS84')
origproj = mapscript.projectionObj(amap.getProjection())
pt.project(ddproj,origproj)

if debug:
    print '-----  Albers Coordinates  -----'
    print 'x: %s y: %s' % (pt.x, pt.y)
    print '--------------------------------'
```

Now that we've projected a point into our mapObj's coordinate system, we'll make an extent by adding some buffer (in map units, not decimal degrees).

```
buffer = 600
extent = mapscript.rectObj()
extent.minx = pt.x - buffer
extent.miny = pt.y - buffer
extent.maxx = pt.x + buffer
extent.maxy = pt.y + buffer
amap.setExtent(extent.minx, extent.miny,
               extent.maxx, extent.maxy)
```

The last few mapObj properties we need to set have to do with the output format and giving the webObj a place to store temporarily downloaded WMS requests.

```
outputformat = mapscript.outputFormatObj('GD/JPEG')
amap.setOutputFormat( outputformat)

amap.web.imagepath = os.environ['TEMP']
```

Now that we have a mapObj, we create a layerObj to describe the TerraServer WMS connection. The gotchas here to remember are to make sure you set metadata for the *wms_srs* and *wms_title* and make sure to set the projection of the layer to EPSG 4326.

```
layer = mapscript.layerObj(amap)
layer.connectiontype = mapscript.MS_WMS
layer.type = mapscript.MS_LAYER_RASTER
layer.metadata.set('wms_srs','EPSG:4326')
layer.metadata.set("wms_title", "USGS Digital Ortho-Quadrangles")
ts_url =
"http://terraservice.net/ogcmap.ashx?VERSION=1.1.1&SERVICE=wms&LAYERS=DOQ&F
ORMAT=jpeg&styles="
layer.connection = ts_url
layer.setProjection('init=epsg:4326')
layer.status = mapscript.MS_ON
```

Currently, we have a map with only a black and white orthophoto from TerraServer, centered on the latitiude/longitude point that geocoder.us returned to us. Pretty boring, I admit.

```
img = amap.draw()
f = open(r'c:\foo.jpg','wb')
f.write(img.getBytes())
f.close()
```

Symbology

MapScript doesn't currently support the ability to add a FontSet using only MapScript…it needs a mapObj that was defined from a mapfile to do that. This ability will be added in the near future. Because we're attempting to build a map with no mapfile, we'll instead use a pixelmap as our symbol. We won't use one on our local machine, however – we'll use *urllib2* and *StringIO* to download and save our pixelmap symbol and dynamically incorporate it in our map.

I googled for a house symbol on Google's image search. I found one at http://www.worldcommunitygrid.org/images/agent/house.jpg, but you could substitute any URL to a small jpg that you wanted. Next, we'll download the image and stuff it into a cStringIO instance. This will allow MapScript's image reading machinery to treat it as it would any normal file.

```
url = 'http://www.worldcommunitygrid.org/images/agent/house.jpg'
f = urllib2.urlopen(url).read()
f = cStringIO.StringIO(f)
```

The next bits were cribbed from http://ms.gis.umn.edu/docs/howto/mapscript_imagery. We create a new symbol, set it to type MS_SYMBOL_PIXMAP, give the symbol the imagery, and append it to the mapObj's symbolset.

```
symbol = mapscript.symbolObj('from_img')
symbol.type = mapscript.MS_SYMBOL_PIXMAP
img = mapscript.imageObj(f)
symbol.setImage(img)
symbol_index = amap.symbolset.appendSymbol(symbol)
```

With the symbol in hand, we can now go through the process of creating a point and a MS_INLINE layer to place our house pixelmap on the map. We first have to build up a shapeObj. shapeObjs are composed of lineObjs, which themselves are composed of points.

```
line = mapscript.lineObj()
line.add(pt)
shape=mapscript.shapeObj(mapscript.MS_SHAPE_POINT)
shape.add(line)
shape.setBounds()
```

Now that we have our shapeObj, we can build up an inline layer, add our point to it, and set some properties on the layer.

```
inline_layer = mapscript.layerObj(amap)
inline_layer.addFeature(shape)
inline_layer.setProjection(amap.getProjection())
inline_layer.name = "housept"
inline_layer.type = mapscript.MS_LAYER_POINT
inline_layer.connectiontype=mapscript.MS_INLINE
inline_layer.status = mapscript.MS_ON
inline_layer.transparency = mapscript.MS_GD_ALPHA
```

With our layer built, we can add our symbology to it. Notice we create a classObj on the inline layer, give it a name, and then create a styleObj that points to our symbol. This is the preferred way of doing styling in MapScript (styleObjs inside of classObjs) instead of merely putting the symbology on the classObj.

```
cls = mapscript.classObj(inline_layer)
cls.name='classname'
style = mapscript.styleObj(cls)
style.symbol = amap.symbolset.index('from_img')
```

Finally, draw our map again and save it to a temporary file.

```
img = amap.draw()

f = open(r'c:\temp\foo.jpg','wb')
f.write(img.getBytes())
f.close()
```

Figure 1. Our final output.

Code Listing

```
# ----------------------------------------
# Geocoding and MapScript
# (c) 2005 Howard Butler
# hobu@iastate.edu
# ----------------------------------------

import os
import xmlrpclib

# geocode our address
geocode_url = 'http://rpc.geocoder.us/service/xmlrpc'
p = xmlrpclib.ServerProxy(geocode_url)
address = "615 WASHINGTON AVE SOUTHEAST, MINNEAPOLIS, MN"

result = p.geocode(address)
print result

# import mapscript as package first ... then the regular way
try:
    import mapscript.mapscript as mapscript
except ImportError:
    import mapscript

amap = mapscript.mapObj()

amap.height = 800
amap.width = 1100

debug = 1
ms_debug = 0

if debug:
    print
    print '------- Address  --------'
    print '%s' % address
    print '-------------------------'

if ms_debug:
    amap.debug = mapscript.MS_ON

# set projection to US laea
amap.setProjection('init=epsg:2163')

# grab the first address geocoder.us gives back to us
# and turn it into a pointObj
lat, lon = result[0]['lat'], result[0]['long']
pt = mapscript.pointObj()
pt.x = lon
pt.y = lat
```

```

if debug:
    print '-------  DD Coordinates  -------'
    print 'x: %s y: %s' % (pt.x, pt.y)
    print '--------------------------------'

# project our point into the mapObj's projection
ddproj = mapscript.projectionObj('proj=latlong,ellps=WGS84')
origproj = mapscript.projectionObj(amap.getProjection())
pt.project(ddproj,origproj)

if debug:
    print '-----  Albers Coordinates  -----'
    print 'x: %s y: %s' % (pt.x, pt.y)
    print '--------------------------------'

# create an extent for our mapObj by buffering our projected
# point by the buffer distance.  Then set the mapObj's extent.
buffer = 600
extent = mapscript.rectObj()
extent.minx = pt.x - buffer
extent.miny = pt.y - buffer
extent.maxx = pt.x + buffer
extent.maxy = pt.y + buffer
amap.setExtent(extent.minx, extent.miny,
               extent.maxx, extent.maxy)

# set the output format to jpeg
outputformat = mapscript.outputFormatObj('GD/JPEG')
amap.setOutputFormat(outputformat)

# give the WMS client a place to put temp files
amap.web.imagepath = os.environ['TEMP']

# define the TerraServer WMS layer
layer = mapscript.layerObj(amap)
layer.connectiontype = mapscript.MS_WMS
layer.type = mapscript.MS_LAYER_RASTER
layer.metadata.set('wms_srs','EPSG:4326')
layer.metadata.set("wms_title", "USGS Digital Ortho-Quadrangles")
ts_url =
"http://terraservice.net/ogcmap.ashx?VERSION=1.1.1&SERVICE=wms&LAYERS=DOQ&F
ORMAT=jpeg&styles="
layer.connection = ts_url
layer.setProjection('init=epsg:4326')
layer.status = mapscript.MS_ON
if ms_debug:
    layer.debug = mapscript.MS_ON

# import the libraries we'll need to make our pixelmap symbol
import urllib2
```

```
import cStringIO

# get a jpeg image from somewhere on the web and read it into
# a StringIO.
url = 'http://www.worldcommunitygrid.org/images/agent/house.jpg'
f = urllib2.urlopen(url).read()
f = cStringIO.StringIO(f)

# create the symbol using the image
symbol = mapscript.symbolObj('from_img')
symbol.type = mapscript.MS_SYMBOL_PIXMAP
img = mapscript.imageObj(f)
symbol.setImage(img)
symbol_index = amap.symbolset.appendSymbol(symbol)

# create a shapeObj out of our address point so we can
# add it to the map.
line = mapscript.lineObj()
line.add(pt)
shape=mapscript.shapeObj(mapscript.MS_SHAPE_POINT)
shape.add(line)
shape.setBounds()

# create our inline layer that holds our address point
inline_layer = mapscript.layerObj(amap)
inline_layer.addFeature(shape)
inline_layer.setProjection(amap.getProjection())
inline_layer.name = "housept"
inline_layer.type = mapscript.MS_LAYER_POINT
inline_layer.connectiontype=mapscript.MS_INLINE
inline_layer.status = mapscript.MS_ON
inline_layer.transparency = mapscript.MS_GD_ALPHA

# add the image symbol we defined above to the inline
# layer.
cls = mapscript.classObj(inline_layer)
cls.name='classname'
style = mapscript.styleObj(cls)
style.symbol = amap.symbolset.index('from_img')

# draw the map and save it somewhere.
img = amap.draw()
f = open(r'c:\temp\foo.jpg','wb')
f.write(img.getBytes())
f.close()
```

Creating Aggregate Rasters for MapServer or GDAL

MapServer tileindex and GDAL VRT

Tileindexes

Although the gdaltindex utility meets the needs of most users, creating a tileindex shapefile is a good introduction to gdal.py. It can also be useful to have a tileindex file with more attributes for reuse in your map.

os.path

The os.path module implements functions on pathnames. Create a new text file in your working directory named `hobu.txt`. No contents are needed. We'll use this file to explore `os.path`.

The `abspath` function returns the absolute path given a relative path.

```
>>> import os.path
>>> os.path.abspath('./hobu.txt')
'P:\\OSG05\\aggregation\\hobu.txt'
>>>
```

The `basename` function returns the filename with all directories stripped from the path.

```
>>> os.path.basename('P:\OSG05\aggregation\hobu.txt')
'hobu.txt'
>>>
```

the `getctime` function returns the file creation time in seconds past the epoch

```
>>> os.path.getctime('hobu.txt')
1118386365
>>>
```

glob

Just like a shell glob. `glob.glob` returns a possibly empty list of paths that match the input pattern:

```
>>> import glob
>>> glob.glob('*.txt')
['hobu.txt']
```

```
>>>
```

Putting it together

Now we'll combine these to print information about a batch of files:

```
>>> for path in glob.glob('*.txt'):
...     print os.path.basename(path), \
...           os.path.abspath(path), \
...           os.path.getctime(path)
...
hobu.txt P:\OSG05\aggregation\hobu.txt 1118386365
>>>
```

And now we'll try this on the workshop raster data. Replace the pattern below with the path to the workshop data:

```
>>> paths = glob.glob('P:\OSG05\python-tests\data\*.tif')
>>> for path in paths:
...     print os.path.basename(path), \
...           os.path.abspath(path), \
...           os.path.getctime(path)
...
escalante30_zip.tif P:\OSG05\python-
tests\data\escalante30_zip.tif 1044213876
mtnwest_zip.tif P:\OSG05\python-tests\data\mtnwest_zip.tif
1044212332
waterpocket30_zip.tif P:\OSG05\python-
tests\data\waterpocket30_zip.tif 104421366
6
zion30_zip.tif P:\OSG05\python-tests\data\zion30_zip.tif
1044211340
cameron30_zip.tif P:\OSG05\python-
tests\data\cameron30_zip.tif 1044129070
wasatch30_zip.tif P:\OSG05\python-
tests\data\wasatch30_zip.tif 1044129100
>>>
```

gdal

OK, so we can obtain all kinds of OS info about the raster data. Now we'll get to the the important geo properties using GDAL's `gdal` Python module.

In the following steps, don't bother with typing the paths. Type the leading quotation mark, drag the file from the file explorer to the interpreter, and the close the quotes.

Let's open one of the workshop raster files in the default read-only mode:

```
>>> from gdal import gdal
>>> dataset = gdal.Open('P:\OSG05\python-
tests\data\cameron30_zip.tif')
>>> dataset
<gdal.gdal.Dataset instance at 0x008E48C8>
>>>
```

The gdal module is extensive. In this exercise we're going to limit ourselves to the following attributes of a Dataset:

```
>>> dataset.RasterCount
3
>>> dataset.RasterXSize
999
>>> dataset.RasterYSize
1586
>>> dataset.GetGeoTransform()
(-106.05969999999999, 0.00027777777777799998, 0.0,
40.842500000000001, 0.0, -0.0
00277692307691999998)
>>>
```

These are the number of bands, the number of pixels and lines, and the geo transform parameters. The elements at indexes 0 and 1 of this tuple are the upper left x value and the x pixel size. The elements at indexes 3 and 5 are the upper left y value and -1 times the y pixel size.

Let's use these properties and methods to compute the bounding boxes for our raster data files:

```
>>> paths = glob.glob('P:\OSG05\python-tests\data\*.tif'):
>>> for path in paths:
...     ds = gdal.Open(path)
...     geo = ds.GetGeoTransform()
...     pixels = ds.RasterXSize
...     lines = ds.RasterYSize
...     minx = geo[0]
...     maxx = minx + pixels * geo[1]
...     maxy = geo[3]
...     miny = maxy + lines * geo[5]
...     print os.path.basename(path), (minx, miny, maxx, 
maxy)
...
escalante30_zip.tif (-111.705, 37.686388888056207, 
-111.22944443999971, 38.06583
3333055551)
mtnwest_zip.tif (-115.5, 36.500000000000022, 
-103.50000000000048, 42.0)
waterpocket30_zip.tif (-111.28472222194445, 
37.298055554999578, -110.72666665999
982, 38.340000000000003)
zion30_zip.tif (-113.21111111, 37.106111111111382, 
-112.74444443999984, 37.63166
6666111109)
cameron30_zip.tif (-106.05969999999999, 
40.402080000000488, -105.78219999999978,
 40.842500000000001)
wasatch30_zip.tif (-111.85889999999999, 40.389999999999951, 
-111.40639999999964,
40.77028)
>>>
```

There's no `close` method for a GDAL dataset. The dataset is closed at the end of the interior block above when Python's garbage collection sweeps out the local `ds` object. You might want to be explicit about it, appending

```
...     del ds
```

to the end of the block.

ogr

That's all we need from `gdal.py` in order to create our raster tileindex. Now we'll need to learn to create an output vector dataset and push features into it. Here, in a nutshell, is creation and saving of a polygon type shapefile using GDAL's `ogr.py` module:

```
>>> from gdal import ogr
>>> driver = ogr.GetDriverByName('ESRI Shapefile')
>>> tileindex_shp = driver.CreateDataSource
('tileindex.shp')
>>> tileindex = tileindex_shp.CreateLayer('tileindex',
geom_type=ogr.wkbPolygon)
>>> tileindex_shp.Destroy()
>>>
```

The Destroy method is more bark than bite. It doesn't delete the file on disk, just closes the output stream and releases allocated memory. Look in your working directory and you will find a shapefile – a rather pointless shapefile with no records, no fields.

Shapefile fields

Let's address that now. Delete the three shapefile components, and repeat the following lines. Try using your interpreter's command history.

```
>>> tileindex_shp = driver.CreateDataSource
('tileindex.shp')
>>> tileindex = tileindex_shp.CreateLayer('tileindex',
geom_type=ogr.wkbPolygon)
```

Next we'll define a string type field named 'location' and set its width to 200 characters:

```
>>> field = ogr.FieldDefn('location', ogr.OFTString)
>>> field.SetWidth(200)
```

and add this field to the layer

```
>>> tileindex.CreateField(field)
0
```

we'll leave the data source open.

Adding Features

A record in our shapefile layer is represented by ogr's `Feature` class. The constructor requires a `FieldDefn` argument, and we obtain one from the layer itself. The value of our single 'location' field is set using the feature's SetField method. Note the return of the `abspath` function and our `hobu.txt` file.

```
>>> feature = ogr.Feature(tileindex.GetLayerDefn())
>>> feature.SetField(0, os.path.abspath('hobu.txt'))
```

A complete feature needs a geometry. We won't dive too deep into ogr.Geometry yet, but will use Python's string interpolation to hack a WKT (well-known text) string and exploit ogr's WKT geometry factory. This time we are using a Python mapping as the object of the interpolation operator instead of a tuple as we did earlier:

```
>>> wkt = 'POLYGON ((%(minx)f %(miny)f, %(minx)f %(maxy)f,
%(maxx)f %(maxy)f, %(maxx)f %(miny)f, %(minx)f %(miny)f))'
>>> wkt = wkt % {'minx': -10, 'miny': -10, 'maxx': 10,
'maxy': 10}
>>> wkt
'POLYGON ((-10.000000 -10.000000, -10.000000 10.000000,
10.000000 10.000000, 10.000000 -10.000000, -10.000000
-10.000000))'
```

Next we create an ogr.Geometry from this string and set the feature's geometry from it:

```
>>> geom = ogr.CreateGeometryFromWkt(wkt)
>>> feature.SetGeometryDirectly(geom)
0
```

create a new feature in our layer based upon this one, and close the data source.

```
>>> tileindex.CreateFeature(feature)
0
>>> tileindex_shp.Destroy()
```

Open the shapefile in OpenEV to see the results.

Aside for mapscript users

The `mapscript.pointObj` and `mapscript.rectObj` classes each have magic methods to support Python's built in `str()` function. Give these a quick try:

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

```
>>> p = mapscript.pointObj(1, 2)
>>> str(p)
"{ 'x': 1 , 'y': 2, 'z': 0 }"
>>> r = mapscript.rectObj(-10,-10,10,10)
>>> str(r)
"{ 'minx': -10 , 'miny': -10 , 'maxx': 10 , 'maxy': 10 }"
>>>
```

Hey, what do you know? Looks a lot like a Python dict, and with the help of the built in `eval()` function, we can turn it into a dict and interpolate the values into a WKT string:

```
>>> wkt = 'POLYGON ((%(minx)f %(miny)f, %(minx)f %(maxy)f,
%(maxx)f %(maxy)f, %(maxx)f %(miny)f, %(minx)f %(miny)f))'
>>> wkt = wkt % eval(str(r))
>>> wkt
'POLYGON ((-10.000000 -10.000000, -10.000000 10.000000,
10.000000 10.000000, 10.000000 -10.000000, -10.000000
-10.000000))'
>>>
```

Complete tileindex script

A complete tileindexing script is included in the workshop at c:/ms4w/apps/python/aggregation/aggtindex.py and can be run using the accompanying aggregation.bat file. Aim it at the workshop raster files in c:/ms4w/apps/python/python/data and check the results again in OpenEV.

Virtual Datasets

GDAL's virtual dataset, or VRT, driver is a means of (among other things) aggregating raster data. The document at http://www.gdal.org/gdal_vrttut.html describes how to express a virtual dataset using XML. We're going to create a VRT that aggregates the workshop raster files, allowing them to be visualized or processed as if they were a single dataset.

XML and Elementtree

Python has a standard XML library, and a great range of other available libraries for parsing and writing XML. The `elementtree` package

http://effbot.org/zone/element-index.htm is a good match for VRT's lightweight XML.

Here's a very simple example that's easy to type in the interpreter:

```
>>> from elementtree.ElementTree import Element,
SubElement
>>> html = Element('html')
>>> body = SubElement(html, 'body')
>>> heading = SubElement(body, 'h1')
>>> heading.text = 'Introducing ElementTree'
>>> para = SubElement(body, 'p')
>>> para.text = 'Package for manipulating hierarchical
data'
```

Now let's import the tostring function so that we can see how this is encoded:

```
>>> from elementtree.ElementTree import tostring
>>> tostring(html)
'<html><body><h1>Introducing ElementTree</h1><p>Package
for manipulating hierarchical data</p></body></html>'
```

On second thought, let's add some CSS to demonstrate element attributes:

```
>>> head = SubElement(html, 'head')
>>> style = SubElement(head, 'style')
>>> style.attrib['type'] = 'text/css'
>>> style.text = 'H1{color:red} P{color:blue}'
>>> from elementtree.ElementTree import tostring
>>> tostring(html)
'<html><body><h1>Introducing ElementTree</h1><p>Package
for manipulating hierarchical data</p></body><head><style
type="text/css">H1{color:red} P{color:blue}
</style></head></html>'
```

and then use the `ElementTree` class to write this to a file

```
>>> from elementtree.ElementTree import ElementTree
>>> tree = ElementTree(html)
>>> tree.write('example.html')
```

Open example.html in a web browser. Minus the standard preamble, it's XHTML, and easy to generate using `elementtree`.

Easy VRT

For a first example, we're going to quickly create a VRT that simply proxies a single band of one of our workshop rasters much like in the first example on the VRT tutorial page.

```
>>> from gdal import gdal
>>> ds = gdal.Open
(r'c:\ms4w\python\data\wasatch30_zip.tif')
>>> geo = ds.GetGeoTransform()
>>> pixels = ds.RasterXSize
>>> lines = ds.RasterYSize
```

You could print the values of these if you wanted. That's all we need from `gdal`, and now we begin by creating our top level element:

```
>>> vrt_elem = Element('VRTDataset',
                       rasterXSize=str(pixels),
                       rasterYSize=str(lines))
```

Note that all `Element` attributes must be strings. Next we add a GeoTransform `SubElement` and set its text node to a string representation of the raster dataset's geotransform.

```
>>> geo_elem = SubElement(vrt_elem, 'GeoTransform')
>>> geo_elem.text = '%f, %f, %f, %f, %f, %f' % (geo)
```

Next we'll add a band element to the root

```
>>> band_elem = SubElement(vrt_elem, 'VRTRasterBand',
dataType='Byte', band='1')
```

and then take a preview of our VRT under construction

```
>>> tostring(vrt_elem)
'<VRTDataset rasterXSize="1629"
rasterYSize="1369"><GeoTransform>-111.858900, 0.000278,
0.000000, 40.770280, 0.000000,
-0.000278<GeoTransform><VRTRasterBand band="1"
dataType="Byte" /></VRTDataset>'
```

Only thing left to do is to define the source data for the band. This involves several new levels of sub elements. Take care that they are subbed from the proper parent element. If you mistakenly insert an element into another, you can take advantage of the fact that all Elements are list-like and delete the sub element at a certain index.

```
>>> source_elem = SubElement(band_elem, 'SimpleSource')
```

```
>>> filename_elem = SubElement(source_elem,
'SourceFilename', relativeToVRT='0')
>>> filename_elem.text =
r'c:\ms4w\python\data\wasatch30_zip.tif'
>>> sband_elem = SubElement(source_elem, 'SourceBand')
>>> sband_elem.text = '1'
>>> srect_elem = SubElement(source_elem, 'SrcRect',
xOff='0', yOff='0', xSize=str(pixels), ySize=str(lines))
>>> drect_elem = SubElement(source_elem, 'DstRect',
xOff='0', yOff='0', xSize=str(pixels), ySize=str(lines))
```

Now let's wrap this up in an ElementTree and write it to disk.

```
>>> vrttree = ElementTree(vrt_elem)
>>> vrttree.write('first.vrt')
```

This first.vrt file can be opened in OpenEV. You should see a gray scale image of the Wasatch Range centered roughly on the Alta ski area at the head of Little Cottonwood Canyon.

Further VRT Element Hacking

A handy feature is that our elements are entirely mutable. Set the source band to "2" and write to a new file

```
>>> sband_elem.text = '2'
>>> vrttree.write('second.vrt')
```

repeat for the third band

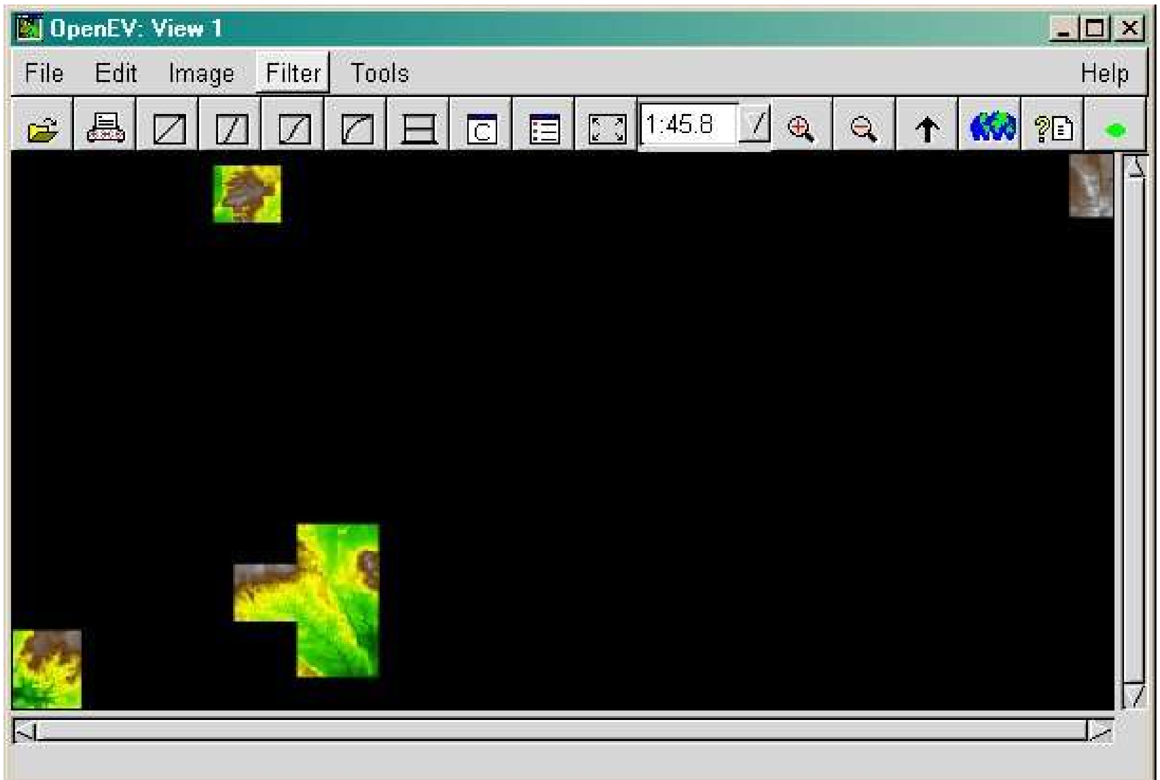
```
>>> sband_elem.text = '3'
>>> vrttree.write('third.vrt')
```

Raster hackers might find this a good way to tweak pixel scaling, color tables, or even filter kernels. See http://www.gdal.org/gdal_vrttut.html for more VRT options.

Complete VRT Script

Finally, we return to the objective: a VRT that aggregates source rasters of a single class (same band count, same projection, and pixel resolution). It's not much more involved than our previous example. The VRT raster size and extents are expanded as each input raster is read, and the individual raster data is mapped into the aggregate output by calculating the appropriate destination rectangle.

The completed script is at c:/ms4w/apps/python/aggregation/aggvrt.py and can be run using the accompanying aggregation.bat file. Aim it at the 5 workshop raster files matching the pattern c:/ms4w/apps/python/python/data/*30*.tif , redirect the output to a .vrt file and check the results again in OpenEV. You should see results like this



Get DEM, DOQ, and SRTM data for any area of interest in the coterminous US

> You need digital elevation data and ortho imagery for any area of interest in the coterminous US.

One approach might be the "Google" approach, ie download **all** of the US NED and USGS DOQ data for the entire US, process it, and then store it. The approach has some disadvantages, however. First, USGS updates both the NED and DOQ data at different intervals for different parts of the country. If you were to pre-process everything, you would only have a single "snapshot" that was only valid for a single point it time – you want the latest and greatest data. Second, the storage requirements for this approach are humungous. The cost of maintaining all three datasets would be very high, and you would still have the problem of refreshing the data.

Another, more timely approach, would be to automate the process of getting each on demand, depend on the infrastructure that already manages them, and use the data in whichever application needs it.

This hack will utilize three methods to request, acquire, and transfer the data. The DEM will be "scraped" off of the USGS site, the DOQ will be requested through TerraServer's SOAP API, and the SRTM data will come from a MapServer-based WCS (Web Coverage Service) source.

What you need:

- Python (obviously)
- GDAL (for projecting the DEM, and creating the output data)
- pyTerra (for requesting the DOQ from TerraServer)
- OpenEV (to view your output)
-

Some strategerizing

All three of our data types – DEM, DOQ, and SRTM – need to be saved out to Imagine (HFA) format in the same coordinate system as the extent that we'll specify. Even though we're hacking, a little object-orient design could save us some time. One thing to notice is that each of the data types is given the same starting point – an extent, and each has the same end point – saved to an Imagine file.

The part that is different for each of the three data types is how the data are actually gotten. If we create a class that can be subclassed for each of the three types, we only have to implement the part that gets the data in each.

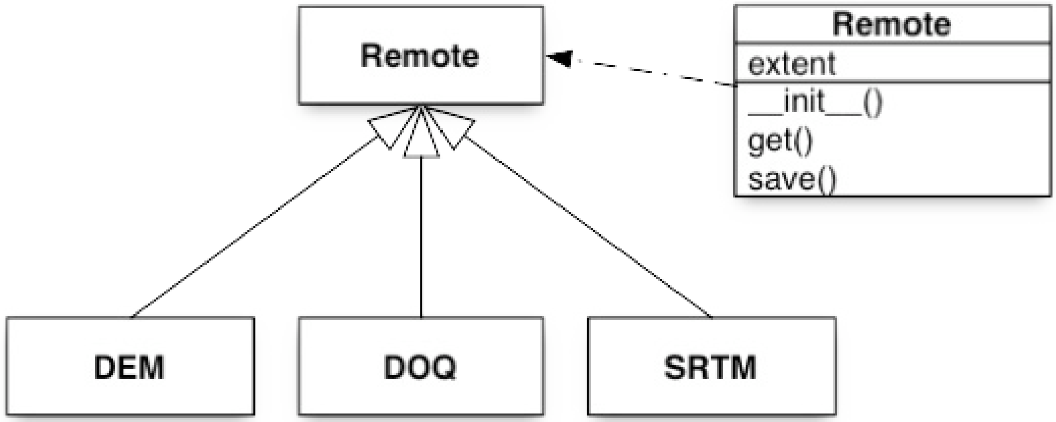


Figure 1. The *Remote* class implements the three methods – *__init__*, *get*, and *save* – that each of the three data types need. We will subclass *Remote* for all three and provide our own implementation of get() for each.

A smart extent object

To start, we need something that is a "smart" extent that knows how to project itself. We will use a class to do this, and the class will take in minx, miny, maxx, and maxy parameters on instantiation as well as an optional EPSG code telling us which coordinate system the extent is in (defaulting to 4326). The *Extent* object will provide a *transform*() method that can transform the extent into any other coordinate system.

To make things a bit easier, we will make a class called *SmartExtent* that will store both the forward and inverse extents to make it easy to get both the projected and unprojected coordinates.

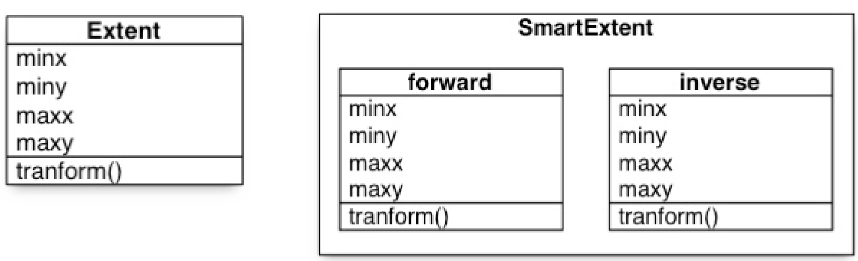


Figure 2. *Extent* and *SmartExtent* objects. The *SmartExtent* object just acts as a container and takes care of calling the *transform()* method for us.

```
class Extent(object):
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        self.epsgcode = epsgcode
        self.minx = minx
        self.maxx = maxx
        self.miny = miny
        self.maxy = maxy
    def transform(self, target_epsg_code):
        mins = ogr.Geometry(type=ogr.wkbPoint)
        maxs = mins.Clone()

        mins.AddPoint(self.minx, self.miny)
        maxs.AddPoint(self.maxx, self.maxy)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.epsgcode)
        maxs.AssignSpatialReference(ref)
        mins.AssignSpatialReference(ref)
        out_ref = osr.SpatialReference()
        out_ref.ImportFromEPSG(target_epsg_code)
        t_mins = mins.Clone()
        t_mins.TransformTo(out_ref)
        t_maxs = maxs.Clone()
        t_maxs.TransformTo(out_ref)
        ext = Extent(t_mins.GetX(), t_mins.GetY(),
                     t_maxs.GetX(), t_maxs.GetY(),
                     epsgcode = target_epsg_code)
        return ext
```

We'll use the *SmartExtent* object to act as a container for our transformed extents.

```
class SmartExtent(object):
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        self.epsgcode = epsgcode
        self.forward = Extent(minx, miny, maxx, maxy, epsgcode)
        self.inverse = self.forward.transform(4326)
```

Next, the instance needs to know how return the transformed (in 4326) coordinates whenever we try to get the string representation of it (this way we can easily substitute it into the URL for the area-of-interest query).

```
def __str__(self):
        outstring = "%s,%s,%s,%s"
        return outstring % (self.inverse.maxy, self.inverse.miny,
                            self.inverse.maxx, self.inverse.minx)
```

Some test code

```
minx = 437142.35
miny = 4658582.96
maxx = 436521.25
maxy = 4659253.80
```

```
extent = SmartExtent(minx, miny, maxx, maxy, epsgcode=26915)
print extent
>> 42.0827943476,42.0768029037,-93.7674817555,-93.759900979
```

Now that we have a smart extent, we can input a bounding box in whatever projection system we need. The advantages of doing it this way instead of just using a simple lat/lon box are twofold. First, if we need to, we can reuse this extent and add more smarts to it when we need to (and will for downloading the TerraServer imagery). Second, providing the convenience of an auto-projecting extent protects our little application from changes in requirements up the line. That way, when your boss asks, "Can I feed this a Lambert Conformal Conic extent instead?", you'll be ready for it.

The super class's *save()* method

While each subclass implements its own *get()* method, the *Remote* class will be the one implementing the *save()* method so that each of the three data types will behave similarly. It also defines the *__init__()* method that takes in one of our extents.

One thing to note here is that the *save*() method takes care to get the projection information from the native-format files that the *get()* method returns. It also makes sure that the raster is projected into the coordinate system that was given in the extent.

```
class Remote(object):
    def __init__(self, extent):
        self.extent = extent

    def get(self):
        pass

    def save(self, filename):

        infile = self.get()

        o = gdal.OpenShared(infile)
        dst_driver = gdal.GetDriverByName('HFA')
        outref = osr.SpatialReference()
        outref.ImportFromEPSG(self.extent.epsgcode)
        dst_wkt = outref.ExportToWkt()
        inref = osr.SpatialReference()

        can_import = inref.ImportFromWkt(o.GetProjection())
        if can_import != 0:
            inref.ImportFromEPSG(4326)
        src_wkt = inref.ExportToWkt()
```

```
        gdal.CreateAndReprojectImage(o,
                                    filename,
                                    src_wkt = src_wkt,
                                    dst_driver=dst_driver,
                                    dst_wkt=dst_wkt)
```

Getting the DOQ

The work of getting the DOQ from TerraServer has already been done for us. The pyTerra (http://hobu.biz/software/pyTerra/) library has a class called TerraImage that does all of the work that we implemented in the get() method of RemoteDEM. All we need to do is to create a *get()* method that does the work of downloading the TerraServer image, setting the coordinate system to the coordinate system (UTM zone) that TerraServer gave us, and return the filename back to the instance so that the *save*() method can pick it up and reproject it into the our coordinate system of choice.

There is one complication, however. The TerraImage class of pyTerra requires that the UTM zone also be given with the request. Because we made the "smart" extent, providing this won't be too hard. The smart extent already contains the information we need (the longitude) to calculate a UTM zone in its t_mins and t_maxs attributes. We can use these attributes and a lookup dictionary to find the UTM zone of the extent. If the extent crosses two UTM zones, nothing is returned (TerraServer can't process requests across UTM zones in a single pass anyway).

```
class SmartExtent(object):
…
def get_zone(self):
        zones = {10:[-126,-120],
                 11:[-120,-114],
                 12:[-114,-108],
                 13:[-108,-102],
                 14:[-102,-96],
                 15:[-96,-90],
                 16:[-90,-84],
                 17:[-84,-78],
                 18:[-78,-72],
                 19:[-72,-66],
                 20:[-66,-60]
                }

        minx = self.inverse.minx
        maxx = self.inverse.maxx
        for i in zones:
            #build the epsg code
            min,max = map(float,zones[i])
            if minx > min and minx < max:
                min_utmzone = 26900+i
            if maxx > min and maxx < max:
                max_utmzone = 26900+i
```

```
        if min_utmzone == max_utmzone:
            return min_utmzone
        else:
            return None
```

In our *get*() method, we set all of the information needed for the TerraImage instance and save the JPEG and worldfile into the temporary directory, open it with GDAL, convert it to a GeoTIFF, and add the coordinate reference. The *save()* method will then pick this up when reprojecting the DOQ into the coordinate system that we defined in our extent.

```
class DOQ(Remote):

    def get(self):
        thescale = 'Scale1m' # scale of the DOQ from TS
        thetype = 'Photo'# Photo or Topo

        # a TerraImage must know its zone
        thezone = self.extent.get_zone() - 26900
        upperLeft = TerraImage.point(self.extent.inverse.maxy,
                                     self.extent.inverse.minx)
        lowerRight = TerraImage.point(self.extent.inverse.miny,
                                      self.extent.inverse.maxx)

        ti = TerraImage.TerraImage(upperLeft,
                                   lowerRight,
                                   thescale,
                                   thetype,
                                   thezone)
        self.ti = ti
        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.jpg'
        self.ti.write(temp_filename)
        self.ti.write_worldfile(temp_filename+"w")

        ds = gdal.Open(temp_filename)
        drv = gdal.GetDriverByName('GTiff')
        tiff_filename = temp_filename.replace('.jpg','.tiff')
        tiff_ds = drv.CreateCopy(tiff_filename, ds)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.extent.epsgcode)
        tiff_ds.SetProjection(ref.ExportToWkt())

        return tiff_filename
```

The usage of this class is a simple, two-line call:

```
doq = DOQ(extent)
doq.save(r'C:\temp\doq.img')
```

Getting the SRTM data

As of MapServer 4.4, support for WCS (Web Coverage Service) is available. Whereas WMS provides a rendered map image, WCS allows a requestor to obtain the actual raw raster data through a structured URL request. Frank Warmerdam provides the SRTM (Shuttle Radar Topography Mission) through a WCS service at http://maps.gdal.org . Here is an example of the "Capabilities" request that you can use to find out more information about a WCS server – in our case Frank's.

```
http://maps.gdal.org/cgi-bin/mapserv_dem?&version=1.0.0&service=WCS&request=GetCapabilities
```

Hitting this URL in our web browser, we can see that there is one layer, called *srtmplus_raw*, which appears to have what we want.

Next, we'll use a *DescribeCoverage* method to find out more information about the layer.

```
http://maps.gdal.org/cgi-bin/mapserv_dem?&version=1.0.0&service=WCS&request=DescribeCoverage&layer=srtmplus_raw
```

The XML listing of this request shows us that that the layer is provided in the EPSG:4326 and EPSG:4269 coordinate systems, has an output format of **GEOTIFF_INT16**, and has a resolution of 0.00833333 degrees per pixel.

With this information in hand, we have enough information to build a Python class to do the work of downloading the image for us. We've already done most of the work, however. The *RemoteDEM* class already defines a way to take in an extent, and turn a temporary GDAL DataSet into a projected Imagine file.

All our RemoteSRTM class needs to define is the *_save_tempfile()* method. This method needs to formulate the request, download it, and save it to a temporary file.

```
class RemoteSRTM(RemoteDEM):
    def get(self):
        url = 'http://maps.gdal.org/cgi-bin/mapserv_dem?&crs=EPSG:4326&coverage=srtmplus_raw&version=1.0.0&service=WCS&request=GetCoverage&bbox=%s&width=100&height=100&format=GEOTIFF_INT16'

        extent_string = '%s,%s,%s,%s' % (self.extent.t_mins.GetX(),
                                         self.extent.t_mins.GetY(),
                                         self.extent.t_maxs.GetX(),
                                         self.extent.t_maxs.GetY()
                                         )
        url = url % extent_string
        response = urllib2.urlopen(url)

        astring = response.read()

        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.tiff'
```

```
        fo = open(temp_filename,'wb')
        fo.write(astring)
        fo.close()
        return temp_filename
```

USGS NED DEM

USGS provides one product of digital elevation models called the NED (National Elevation Dataset) on their website at http://seamless.usgs.gov. The user interface for requesting DEMs is pretty clunky, but it amounts to manually defining an area of interest, choosing the "Download" link, waiting in a queue, and then saving the zip file of the DEM on your local machine.

Python provides excellent capabilities for working with URLs in the standard libraries *urllib* and *urllib2*. We can use *urllib2* and *cookielib* (another standard library as of Python 2.4) to simulate the user requesting an area of interest, waiting in the queue, and saving the resulting zip file on the local file system.

We need to make a series of requests to the USGS site to simulate a user doing the same thing. The first request asks the USGS for a session, tells the site which area of interest we want, and returns us the session cookie that we will use for subsequent requests. The second request actually puts us in the queue. The final series of requests asks the website if our data is ready every 10 seconds, and when it is, downloads the data to our local machine.

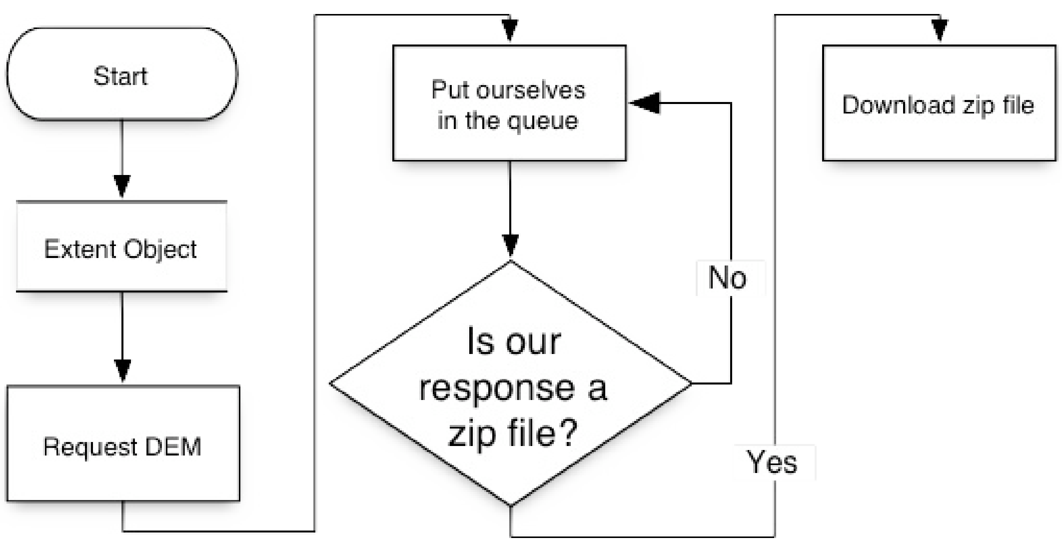


Figure 3. A process diagram of requesting a DEM from the USGS site.

Download and save the DEM

Now we need to make a class that will model the behavior of our remote DEM from the USGS site. Looking at Figure 3, we can see that we will want to feed this class one of our custom extent objects. Once returned, the DEM will be a binary Arc/Info coverage that is stored in a zip file. We will then need some methods to both get the DEM and save it to a temp directory that we can then use to create our shaded DOQ.

The __init__ method of our DEM (subclassed from Remote) class will take in our custom extent. All of the specific implementation exists in the *get()* method, and it does all of the work of actually getting the DEM from the USGS site.

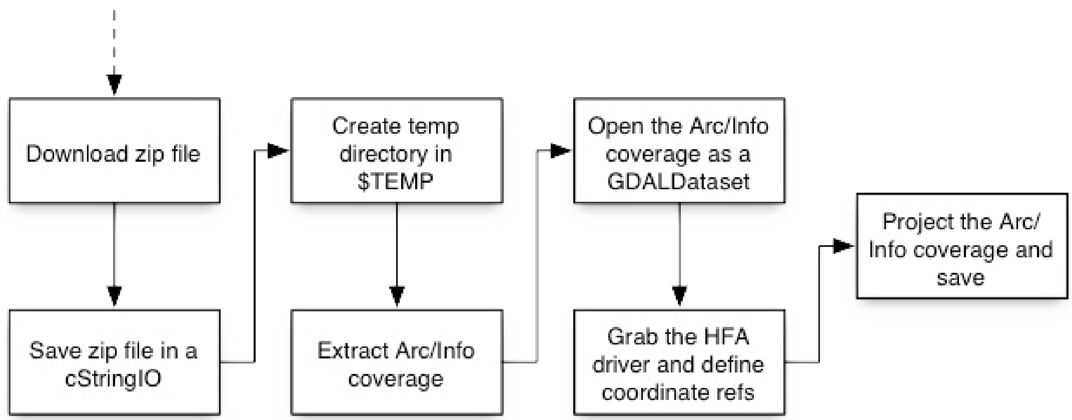


Figure 4. get() method of *DEM* (continued).

```
class DEM(Remote):
    def get(self):
        self.add_to_queue()
        self.download()
        temp_file = self.write_temp_file()
        return temp_file
```

The *get()* method of *DEM* is really just a driver function. The main pieces of requesting, downloading, and saving a DEM from the USGS site are implemented separately to make things a bit easier to read, break up the code, and allow us to make localized changes if the USGS changes their site (which they did at least once while I was writing this exercise).

The *add_to_queue()* method makes the initial request to the USGS site and gives it our extent, gets a cookie (done automatically by *cookielib*), and returns our place in the queue.

```
def add_to_queue(self):

        url = 
'http://extract.cr.usgs.gov/Website/distreq/RequestSummary.jsp?AL=%s&PR=0&P
L=NED01HZ'
```

```
        url = url %(self.extent)

        req = urllib2.Request(url)
        handle = urllib2.urlopen(req)

        url = 'http://extract.cr.usgs.gov/diststatus/servlet/gov.usgs.edc.RequestStatus?siz=1&key=NED&ras=1&rsp=1&pfm=GridFloat&lay=-1&fid=-1&lft=%s&rgt=%s&top=%s&bot=%s&wmd=1&mcd=NED&mdf=TXT&arc=ZIP&sde=NED.conus_ned&msd=NED.CONUS_NED_METADATA&zun=METERS&prj=0&csx=2.777777777999431E-4&csy=2.777777777999431E-4&bnd=&bndnm='

        url = url % (self.extent.inverse.minx,
                     self.extent.inverse.maxx,
                     self.extent.inverse.maxy,
                     self.extent.inverse.miny)

        req2 = urllib2.Request(url)
        handle = urllib2.urlopen(req2)
        self.handle = handle
```

Now that we're in the queue, we need a method to actually wait in the queue, make a request to the website to ask if it is done with our DEM every 10 seconds, and download the zip file with our data. The *download*() method does this. Instead of saving the data to a file, however, we will be storing it in a cStringIO instance. You can think of this as a string buffer. By storing it here, it will be in memory with our instance. We are using this because we want our *write_temp_file*() method to actually extract the stuff out of the zip file and return its location to the caller so that the *save()* method can do what it needs.

```
def download(self):
        if debug:
            print 'requesting DEM download...'

        # ask forever until the website returns
        # application/x-zip-compressed as the content type
        # I've had this go for over an hour sometimes.
        wait, newurl = self.handle.info()['refresh'].split(';')
        newurl = newurl.replace('URL=','').strip()
        url2 = 'http://extract.cr.usgs.gov%s'%newurl
        while 1:

            time.sleep(int(wait))
            request = urllib2.Request(url2)
            response = urllib2.urlopen(request)

            if 'text/html' not in response.info()['content-type']:
                if debug:
                    print "it's our turn... downloading DEM..."
                output = response.read()
                break
```

```
            else:
                if debug:
                    print 'still in the queue. requesting again... '

        zip_output = cStringIO.StringIO()
        zip_output.write(output)
        self.zip_output = zip_output
```

Python comes with the module **zipfile** as a standard library, and we'll now use that help us implement the *write_temp_file*() method. This method does all of the mundane business of extracting the Arc/Info coverage out to a temporary file and returning the location of the file to the caller.

```
def write_temp_file(self):
        # reset the cStringIO to read at the beginning
        self.zip_output.reset()

        # make a temp directory in $TEMP
        temp_filename = get_timestamp()
        outdir = os.path.join(temp_dir,temp_filename)
        os.mkdir(outdir)

        # the arcinfo files use the *opposite* path separator than
        # than the system's.
        if os.sep == '\\':
            info_separator = '/'
        else:
            info_separator = '\\'

        # extract the info coverage into the temp directory
        z = zipfile.ZipFile(self.zip_output)
        arcinfo_dir = z.filelist[0].filename.split(info_separator)[0]
        tempdir = os.path.join(outdir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, 'info')
        os.mkdir(tempdir)

        for name in z.namelist():
            outfile = open(os.path.join(outdir, name), 'wb')
            outfile.write(z.read(name))
            outfile.flush()
            outfile.close()

        # USGS's zipfile buries the data in another arcinfo directory
        # so we have to use two
        infile = os.path.join(outdir,arcinfo_dir,arcinfo_dir)
        return infile
```

Finally, here's how we use the *DEM* class…

```
maxx = 437142.35
miny = 4658582.96
minx = 436521.25
maxy = 4659253.80
extent = Extent(minx, miny, maxx, maxy, epsgcode=26915)
print extent
dem = DEM(extent)
dem.save(r'c:\temp\dem.img')
42.0828443199,42.076752942,-93.7599730671,-93.7674089552
requesting DEM download...
still in the queue. requesting again...
it's our turn... downloading DEM...
```

Code Listing

```
import gdal.ogr as ogr
import gdal.osr as osr
import gdal.gdal as gdal

import cookielib
import urllib2
import cStringIO
import zipfile


from pyTS import TerraImage
from pyTS import pyTerra

import os
import sys
import time

debug = 1
cookiejar = cookielib.LWPCookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))
urllib2.install_opener(opener)

temp_dir= os.environ['TEMP']

def get_timestamp():
    """returns a big, unique, string that we can use for nonsensical filenames"""
    import md5
    q = md5.md5(str(time.time()))
    return q.hexdigest()

class Extent(object):
    """An extent that can transform itself into other coordinate systems"""
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        """MapServer-style... minx, miny, maxx, maxy, with an optional epsgcode
        defaults to EPSG:4326"""
        self.epsgcode = epsgcode
        self.minx = minx
        self.maxx = maxx
        self.miny = miny
        self.maxy = maxy
    def transform(self, target_epsg_code):
        """Transforms the extent into the target EPSG code and returns it."""
        mins = ogr.Geometry(type=ogr.wkbPoint)
        maxs = mins.Clone()

        mins.AddPoint(self.minx, self.miny)
```

```
        maxs.AddPoint(self.maxx, self.maxy)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.epsgcode)
        maxs.AssignSpatialReference(ref)
        mins.AssignSpatialReference(ref)
        out_ref = osr.SpatialReference()
        out_ref.ImportFromEPSG(target_epsg_code)
        t_mins = mins.Clone()
        t_mins.TransformTo(out_ref)
        t_maxs = maxs.Clone()
        t_maxs.TransformTo(out_ref)
        ext = Extent(t_mins.GetX(), t_mins.GetY(),
                     t_maxs.GetX(), t_maxs.GetY(),
                     epsgcode = target_epsg_code)
        return ext



class SmartExtent(object):
    """A class that acts as a container for our extents by storing the
    forward and inverse projections."""
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        """MapServer-style... minx, miny, maxx, maxy, with an optional epsgcode
        defaults to EPSG:4326"""
        self.epsgcode = epsgcode
        self.forward = Extent(minx, miny, maxx, maxy, epsgcode)
        self.inverse = self.forward.transform(4326)

    def __str__(self):
        """Prints out the inverse extent in the form that the USGS site needs."""
        outstring = "%s,%s,%s,%s"
        return outstring % (self.inverse.maxy, self.inverse.miny,
                            self.inverse.maxx, self.inverse.minx)

    def get_zone(self):
        """Returns the UTM zone of the extent."""
        zones = {10:[-126,-120],
                 11:[-120,-114],
                 12:[-114,-108],
                 13:[-108,-102],
                 14:[-102,-96],
                 15:[-96,-90],
                 16:[-90,-84],
                 17:[-84,-78],
                 18:[-78,-72],
                 19:[-72,-66],
                 20:[-66,-60]
                }

```

```
        minx = self.inverse.minx
        maxx = self.inverse.maxx
        for i in zones:
            #build the epsg code
            min,max = map(float,zones[i])
            if minx > min and minx < max:
                min_utmzone = 26900+i
            if maxx > min and maxx < max:
                max_utmzone = 26900+i
        if min_utmzone == max_utmzone:
            return min_utmzone
        else:
            return None


class Remote(object):
    """A super class that defines the input for the three data types."""
    def __init__(self, extent):
        """Take in one of our SmartExtent objects."""
        self.extent = extent

    def get(self):
        """Dummy method.  Should not be called directly."""
        pass

    def save(self, filename):
        """Saves the given filename in the coordinate system that
        was given by the SmartExtent."""
        infile = self.get()

        o = gdal.OpenShared(infile)
        dst_driver = gdal.GetDriverByName('HFA')
        outref = osr.SpatialReference()
        outref.ImportFromEPSG(self.extent.epsgcode)
        dst_wkt = outref.ExportToWkt()
        inref = osr.SpatialReference()

        can_import = inref.ImportFromWkt(o.GetProjection())
        if can_import != 0:
            inref.ImportFromEPSG(4326)
        src_wkt = inref.ExportToWkt()

        gdal.CreateAndReprojectImage(o,
                                     filename,
                                     src_wkt = src_wkt,
                                     dst_driver=dst_driver,
                                     dst_wkt=dst_wkt)

class DEM(Remote):
    """A class to get DEMs from the USGS Seamless site at
    http://seamless.usgs.gov"""
```

```
    def get(self):
        """Returns the downloaded file for the save() method"""
        self.add_to_queue()
        self.download()
        temp_file = self.write_temp_file()
        return temp_file

    def add_to_queue(self):
        """Adds our request for a DEM to the USGS queue."""
        url = 'http://extract.cr.usgs.gov/Website/distreq/RequestSummary.jsp?AL=%s&PR=0&PL=NED01HZ'
        url = url %(self.extent)

        req = urllib2.Request(url)
        handle = urllib2.urlopen(req)

        url = 'http://extract.cr.usgs.gov/diststatus/servlet/gov.usgs.edc.RequestStatus?siz=1&key=NED&ras=1&rsp=1&pfm=GridFloat&lay=-1&fid=-1&lft=%s&rgt=%s&top=%s&bot=%s&wmd=1&mcd=NED&mdf=TXT&arc=ZIP&sde=NED.conus_ned&msd=NED.CONUS_NED_METADATA&zun=METERS&prj=0&csx=2.777777777999431E-4&csy=2.777777777999431E-4&bnd=&bndnm='

        url = url % (self.extent.inverse.minx,
                     self.extent.inverse.maxx,
                     self.extent.inverse.maxy,
                     self.extent.inverse.miny)

        req2 = urllib2.Request(url)
        handle = urllib2.urlopen(req2)
        self.handle = handle

    def download(self):
        """Waits in the USGS queue and downloads the DEM in
        Arc/Info format when it is our turn."""
        if debug:
            print 'requesting DEM download...'

        # ask forever until the website returns
        # application/x-zip-compressed as the content type
        # I've had this go for over an hour sometimes.
        wait, newurl = self.handle.info()['refresh'].split(';')
        newurl = newurl.replace('URL=','').strip()
        url2 = 'http://extract.cr.usgs.gov%s'%newurl
        while 1:

            time.sleep(int(wait))
            request = urllib2.Request(url2)
            response = urllib2.urlopen(request)

```

```
            if 'text/html' not in response.info()['content-type']:
                if debug:
                    print "it's our turn... downloading DEM..."
                output = response.read()
                break
            else:
                if debug:
                    print 'still in the queue. requesting again... '

        zip_output = cStringIO.StringIO()
        zip_output.write(output)
        self.zip_output = zip_output

    def write_temp_file(self):
        """Writes out the file given by the download() method
        in a temporary directory and returns the location to the
        caller."""
        # reset the cStringIO to read at the beginning
        self.zip_output.reset()

        # make a temp directory in $TEMP
        temp_filename = get_timestamp()
        outdir = os.path.join(temp_dir,temp_filename)
        os.mkdir(outdir)

        # the arcinfo files use the *opposite* path separator than
        # than the system's.
        if os.sep == '\\':
            info_separator = '/'
        else:
            info_separator = '\\'

        # extract the info coverage into the temp directory
        z = zipfile.ZipFile(self.zip_output)
        arcinfo_dir = z.filelist[0].filename.split(info_separator)[0]
        tempdir = os.path.join(outdir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, 'info')
        os.mkdir(tempdir)

        for name in z.namelist():
            outfile = open(os.path.join(outdir, name), 'wb')
            outfile.write(z.read(name))
            outfile.flush()
            outfile.close()

        # USGS's zipfile buries the data in another arcinfo directory
        # so we have to use two
        infile = os.path.join(outdir,arcinfo_dir,arcinfo_dir)
```

```
        return infile




class DOQ(Remote):
    """A class that implements getting a DOQ from the Microsoft
    TerraServer."""
    def get(self):
        """Returns the downloaded file for the save() method"""
        thescale = 'Scale1m' # scale of the DOQ from TS
        thetype = 'Photo'# Photo or Topo

        # a TerraImage must know its zone
        thezone = self.extent.get_zone() - 26900
        upperLeft = TerraImage.point(self.extent.inverse.maxy,
                                     self.extent.inverse.minx)
        lowerRight = TerraImage.point(self.extent.inverse.miny,
                                      self.extent.inverse.maxx)

        ti = TerraImage.TerraImage(upperLeft,
                                   lowerRight,
                                   thescale,
                                   thetype,
                                   thezone)
        self.ti = ti
        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.jpg'
        self.ti.write(temp_filename)
        self.ti.write_worldfile(temp_filename+"w")

        ds = gdal.Open(temp_filename)
        drv = gdal.GetDriverByName('GTiff')
        tiff_filename = temp_filename.replace('.jpg','.tiff')
        tiff_ds = drv.CreateCopy(tiff_filename, ds)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.extent.epsgcode)
        tiff_ds.SetProjection(ref.ExportToWkt())

        return tiff_filename

class SRTM(Remote):
    """A class that implements getting the SRTM data from the
    WCS server at http://maps.gdal.org"""
    def get(self):
        """Returns the downloaded file for the save() method"""
        url = 'http://maps.gdal.org/cgi-
bin/mapserv_dem?&crs=EPSG:4326&coverage=srtmplus_raw&version=1.0.0&service=
WCS&request=GetCoverage&bbox=%s&width=100&height=100&format=GEOTIFF_INT16'

        extent_string = '%s,%s,%s,%s' % (self.extent.inverse.minx,
                                          self.extent.inverse.miny,
```

```
                                         self.extent.inverse.maxx,
                                         self.extent.inverse.maxy
                                         )
        url = url % extent_string
        response = urllib2.urlopen(url)

        astring = response.read()

        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.tiff'
        fo = open(temp_filename,'wb')
        fo.write(astring)
        fo.close()
        return temp_filename

maxx = 437142.35
miny = 4658582.96
minx = 436521.25
maxy = 4659253.80
extent = SmartExtent(minx, miny, maxx, maxy, epsgcode=26915)
print extent
dem = DEM(extent)
dem.save(r'c:\temp\usgs.img')
doq = DOQ(extent)
doq.save(r'C:\temp\doq.img')
srtm = SRTM(extent)
srtm.save(r'c:\temp\srtm.img')
```

Exploring Web Feature Services

No fancy client needed, open standards and XML make it easy to explore WFS using Python.

Capabilities

We'll test against my two-bit WFS instance at

http://zcologia.com:9001/mapserver/members/

which has members of the next generation MapServer site as its sole feature type, and if port 9001 is forbidden in the UMN computer lab, we'll try

http://www.refractions.net:8080/geoserver/wfs/GetCapabilities

or another service from the catalog at

http://www.refractions.net/white_papers/ogcsurvey/index.php

Connecting

Start up the Python interpreter and define a GetCapabilities request URL:

```
>>> base =
'http://zcologia.com:9001/mapserver/members/capabilities.r
py'
>>> request = base +
'?service=WFS&request=GetCapabilities'
```

We'll use `urllib` to get a file on this URL and parse the file with an `ElementTree`

```
>>> import urllib
>>> u = urllib.urlopen(request)
>>> from elementtree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> root = tree.parse(u)
```

This method returns the root Element of `tree`. Next print `root`

```
>>> print root
<Element {http://www.opengis.net/wfs}WFS_Capabilities at
99ed78>
>>>
```

the string representation of an Element includes the qualified name of the element in the form *{uri}local*. In an XML file, we usually define a prefix for each URI, and write the element out like '<prefix:local />'.

Service Elements

Elements are list-like, so we have a simple Python-ic way to inspect the children of any element

```
>>> list(root)
[<Element {http://www.opengis.net/wfs}Service at 97a580>, <Element {http://www.opengis.net/wfs}Capability at 99e878>, <Element {http://www.opengis.net/wfs}FeatureTypeList at 9a5148>, <Element {http://www.opengis.net/wfs}Filter_Capabilities at 9a53a0>]
```

Let's pick out the *Service*, or more accurately, the *{http://www.opengis.net/wfs} Service* element and print its children

```
>>> service = root[0]
>>> for e in service:
...     print '%s => %s' % (e.tag, e.text)
...
{http://www.opengis.net/wfs}Title => MapServer Site Member Locations
{http://www.opengis.net/wfs}Name => members
{http://www.opengis.net/wfs}OnlineResource => http://zcologia.com:9001/mapserver/members
{http://www.opengis.net/wfs}Abstract => Demonstrating a lightweight and low budget WFS server using ElementTree and Twisted. Every 5 minutes we use RPC to mine the next generation MapServer website for member locations. These locations are rendered into GML for a WFS response.
{http://www.opengis.net/wfs}AccessConstraints => NONE
{http://www.opengis.net/wfs}Fees => NONE
{http://www.opengis.net/wfs}Keywords => WFS, ELEMENTTREE, TWISTED, PYTHON
>>>
```

FeatureType Elements

The question we ask now is whether this service can give us any point type features. The first step towards the answer is to poke around our root's

FeatureTypeList element. This is at index 2.

```
>>> ftypes = root[2].getiterator
('{http://www.opengis.net/wfs}FeatureType')
>>> ftypes
[<Element {http://www.opengis.net/wfs}FeatureType at
9a5a80>]
>>> list(ftypes[0])
[<Element {http://www.opengis.net/wfs}Name at 9a59e0>,
<Element {http://www.opengis.net/wfs}SRS at 9a5ad0>,
<Element {http://www.opengis.net/wfs}LatLongBoundingBox at
9a5af8>]
>>> for e in ftypes[0]:
...     print '%s => %s' % (e.tag, e.text)
...
{http://www.opengis.net/wfs}Name => member
{http://www.opengis.net/wfs}SRS => EPSG:4326
{http://www.opengis.net/wfs}LatLongBoundingBox => None
>>>
```

Capability Elements

So, we have a feature type named 'member' ... does it have a point property? To answer this, we'll need to make a GetFeatureType request. The base URL for such a request is found by inspecting the root's Capability element:

```
>>> capability = root[1]
>>> list(capability)
[<Element {http://www.opengis.net/wfs}Request at 99ed00>]
>>> request = capability[0]
>>> list(request)
[<Element {http://www.opengis.net/wfs}GetCapabilities at
99eaf8>, <Element {http://www.opengis.net/wfs}
DescribeFeatureType at 9a52d8>, <Element
{http://www.opengis.net/wfs}GetFeature at 9a5580>]
>>> iter = request.getiterator
('{http://www.opengis.net/wfs}Get')
>>> for e in iter:
...     print '%s => %s' % (e.tag, e.items())
...
```

```
{http://www.opengis.net/wfs}Get => [('onlineResource',
'http://zcologia.com:9001/mapserver/members/capabilities.r
py')]
{http://www.opengis.net/wfs}Get => [('onlineResource',
'http://zcologia.com:9001/mapserver/members/description.rp
y')]
{http://www.opengis.net/wfs}Get => [('onlineResource',
'http://zcologia.com:9001/mapserver/members/features.rpy')
]
```

The second of these elements is the one we are after.

DescribeFeatureType

Now we make a DescribeFeatureType request and parse the response.

```
>>> description_base = iter[1].get('onlineResource')
>>> url = description_base +
'?service=WFS&request=DescribeFeatureType&typename=member'
>>> u = urllib.urlopen(url)
>>> dtree = ElementTree()
>>> droot = dtree.parse(u)
>>> list(droot)
[<Element {http://www.w3.org/2001/XMLSchema}import at
a3df30>, <Element {http://www.w3.org/2001/XMLSchema}
element at a3df58>, <Element
{http://www.w3.org/2001/XMLSchema}complexType at a3de40>]
```

Making sense of the schema is a bit beyond the scope of this humble hacking workshop. We'll just print the attributes of the schema elements and look for *location*, *position*, or *pointProperty* types and refs:

```
>>> elems = droot.getiterator
('{http://www.w3.org/2001/XMLSchema}element')
>>> for e in elems:
...     print e.items()
...
[('substitutionGroup', 'gml:_Feature'), ('type',
'member_Type'), ('name', 'member')]
[('type', 'string'), ('name', 'fid')]
[('type', 'string'), ('name', 'mid')]
[('type', 'string'), ('name', 'fullname')]
```

```
[('ref', 'gml:location')]
>>>
```

gml:location ... we have a point property.

Get DEM, DOQ, and SRTM data for any area of interest in the coterminous US

> You need digital elevation data and ortho imagery for any area of interest in the coterminous US.

One approach might be the “Google” approach, ie download **all** of the US NED and USGS DOQ data for the entire US, process it, and then store it. The approach has some disadvantages, however. First, USGS updates both the NED and DOQ data at different intervals for different parts of the country. If you were to pre-process everything, you would only have a single “snapshot” that was only valid for a single point it time – you want the latest and greatest data. Second, the storage requirements for this approach are humungous. The cost of maintaining all three datasets would be very high, and you would still have the problem of refreshing the data.

Another, more timely approach, would be to automate the process of getting each on demand, depend on the infrastructure that already manages them, and use the data in whichever application needs it.

This hack will utilize three methods to request, acquire, and transfer the data. The DEM will be “scraped” off of the USGS site, the DOQ will be requested through TerraServer’s SOAP API, and the SRTM data will come from a MapServer-based WCS (Web Coverage Service) source.

What you need:

- Python (obviously)
- GDAL (for projecting the DEM, and creating the output data)
- pyTerra (for requesting the DOQ from TerraServer)
- OpenEV (to view your output)
-

Some strategerizing

All three of our data types – DEM, DOQ, and SRTM – need to be saved out to Imagine (HFA) format in the same coordinate system as the extent that we’ll specify. Even though we’re hacking, a little object-orient design could save us some time. One thing to notice is that each of the data types is given the same starting point – an extent, and each has the same end point – saved to an Imagine file.

The part that is different for each of the three data types is how the data are actually gotten. If we create a class that can be subclassed for each of the three types, we only have to implement the part that gets the data in each.

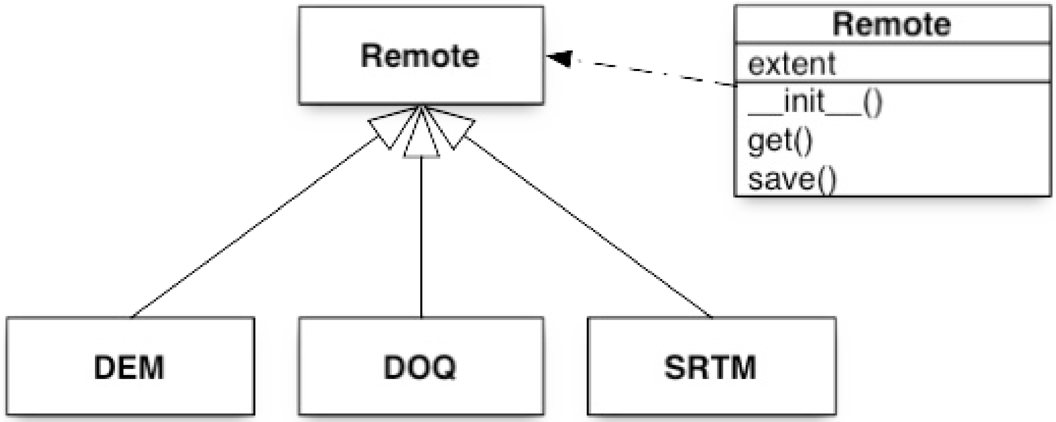


Figure 1. The *Remote* class implements the three methods – *__init__*, *get*, and *save* – that each of the three data types need. We will subclass *Remote* for all three and provide our own implementation of get() for each.

A smart extent object

To start, we need something that is a "smart" extent that knows how to project itself. We will use a class to do this, and the class will take in minx, miny, maxx, and maxy parameters on instantiation as well as an optional EPSG code telling us which coordinate system the extent is in (defaulting to 4326). The *Extent* object will provide a *transform*() method that can transform the extent into any other coordinate system.

To make things a bit easier, we will make a class called *SmartExtent* that will store both the forward and inverse extents to make it easy to get both the projected and unprojected coordinates.

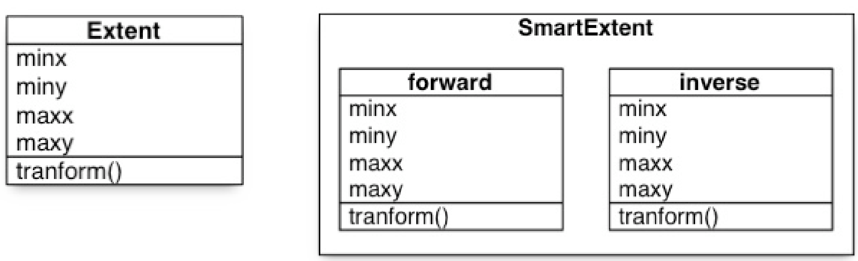


Figure 2. *Extent* and *SmartExtent* objects. The *SmartExtent* object just acts as a container and takes care of calling the *transform()* method for us.

```
class Extent(object):
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        self.epsgcode = epsgcode
        self.minx = minx
        self.maxx = maxx
        self.miny = miny
        self.maxy = maxy
    def transform(self, target_epsg_code):
        mins = ogr.Geometry(type=ogr.wkbPoint)
        maxs = mins.Clone()

        mins.AddPoint(self.minx, self.miny)
        maxs.AddPoint(self.maxx, self.maxy)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.epsgcode)
        maxs.AssignSpatialReference(ref)
        mins.AssignSpatialReference(ref)
        out_ref = osr.SpatialReference()
        out_ref.ImportFromEPSG(target_epsg_code)
        t_mins = mins.Clone()
        t_mins.TransformTo(out_ref)
        t_maxs = maxs.Clone()
        t_maxs.TransformTo(out_ref)
        ext = Extent(t_mins.GetX(), t_mins.GetY(),
                     t_maxs.GetX(), t_maxs.GetY(),
                     epsgcode = target_epsg_code)
        return ext
```

We'll use the *SmartExtent* object to act as a container for our transformed extents.

```
class SmartExtent(object):
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        self.epsgcode = epsgcode
        self.forward = Extent(minx, miny, maxx, maxy, epsgcode)
        self.inverse = self.forward.transform(4326)
```

Next, the instance needs to know how return the transformed (in 4326) coordinates whenever we try to get the string representation of it (this way we can easily substitute it into the URL for the area-of-interest query).

```
def __str__(self):
        outstring = "%s,%s,%s,%s"
        return outstring % (self.inverse.maxy, self.inverse.miny,
                            self.inverse.maxx, self.inverse.minx)
```

Some test code

```
minx = 437142.35
miny = 4658582.96
maxx = 436521.25
maxy = 4659253.80
```

```
extent = SmartExtent(minx, miny, maxx, maxy, epsgcode=26915)
print extent
>> 42.0827943476,42.0768029037,-93.7674817555,-93.759900979
```

Now that we have a smart extent, we can input a bounding box in whatever projection system we need. The advantages of doing it this way instead of just using a simple lat/lon box are twofold. First, if we need to, we can reuse this extent and add more smarts to it when we need to (and will for downloading the TerraServer imagery). Second, providing the convenience of an auto-projecting extent protects our little application from changes in requirements up the line. That way, when your boss asks, “Can I feed this a Lambert Conformal Conic extent instead?”, you’ll be ready for it.

The super class’s *save()* method

While each subclass implements its own *get()* method, the *Remote* class will be the one implementing the *save()* method so that each of the three data types will behave similarly. It also defines the *__init__()* method that takes in one of our extents.

One thing to note here is that the *save*() method takes care to get the projection information from the native-format files that the *get()* method returns. It also makes sure that the raster is projected into the coordinate system that was given in the extent.

```
class Remote(object):
    def __init__(self, extent):
        self.extent = extent

    def get(self):
        pass

    def save(self, filename):

        infile = self.get()

        o = gdal.OpenShared(infile)
        dst_driver = gdal.GetDriverByName('HFA')
        outref = osr.SpatialReference()
        outref.ImportFromEPSG(self.extent.epsgcode)
        dst_wkt = outref.ExportToWkt()
        inref = osr.SpatialReference()

        can_import = inref.ImportFromWkt(o.GetProjection())
        if can_import != 0:
            inref.ImportFromEPSG(4326)
        src_wkt = inref.ExportToWkt()
```

```
        gdal.CreateAndReprojectImage(o,
                                     filename,
                                     src_wkt = src_wkt,
                                     dst_driver=dst_driver,
                                     dst_wkt=dst_wkt)
```

Getting the DOQ

The work of getting the DOQ from TerraServer has already been done for us. The pyTerra (http://hobu.biz/software/pyTerra/) library has a class called TerraImage that does all of the work that we implemented in the get() method of RemoteDEM. All we need to do is to create a *get()* method that does the work of downloading the TerraServer image, setting the coordinate system to the coordinate system (UTM zone) that TerraServer gave us, and return the filename back to the instance so that the *save*() method can pick it up and reproject it into the our coordinate system of choice.

There is one complication, however. The TerraImage class of pyTerra requires that the UTM zone also be given with the request. Because we made the "smart" extent, providing this won't be too hard. The smart extent already contains the information we need (the longitude) to calculate a UTM zone in its t_mins and t_maxs attributes. We can use these attributes and a lookup dictionary to find the UTM zone of the extent. If the extent crosses two UTM zones, nothing is returned (TerraServer can't process requests across UTM zones in a single pass anyway).

```
class SmartExtent(object):
…
def get_zone(self):
        zones = {10:[-126,-120],
                 11:[-120,-114],
                 12:[-114,-108],
                 13:[-108,-102],
                 14:[-102,-96],
                 15:[-96,-90],
                 16:[-90,-84],
                 17:[-84,-78],
                 18:[-78,-72],
                 19:[-72,-66],
                 20:[-66,-60]
                }

        minx = self.inverse.minx
        maxx = self.inverse.maxx
        for i in zones:
            #build the epsg code
            min,max = map(float,zones[i])
            if minx > min and minx < max:
                min_utmzone = 26900+i
            if maxx > min and maxx < max:
                max_utmzone = 26900+i
```

```
        if min_utmzone == max_utmzone:
            return min_utmzone
        else:
            return None
```

In our *get*() method, we set all of the information needed for the TerraImage instance and save the JPEG and worldfile into the temporary directory, open it with GDAL, convert it to a GeoTIFF, and add the coordinate reference. The *save()* method will then pick this up when reprojecting the DOQ into the coordinate system that we defined in our extent.

```
class DOQ(Remote):

    def get(self):
        thescale = 'Scale1m' # scale of the DOQ from TS
        thetype = 'Photo'# Photo or Topo

        # a TerraImage must know its zone
        thezone = self.extent.get_zone() - 26900
        upperLeft = TerraImage.point(self.extent.inverse.maxy,
                                     self.extent.inverse.minx)
        lowerRight = TerraImage.point(self.extent.inverse.miny,
                                      self.extent.inverse.maxx)

        ti = TerraImage.TerraImage(upperLeft,
                                   lowerRight,
                                   thescale,
                                   thetype,
                                   thezone)
        self.ti = ti
        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.jpg'
        self.ti.write(temp_filename)
        self.ti.write_worldfile(temp_filename+"w")

        ds = gdal.Open(temp_filename)
        drv = gdal.GetDriverByName('GTiff')
        tiff_filename = temp_filename.replace('.jpg','.tiff')
        tiff_ds = drv.CreateCopy(tiff_filename, ds)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.extent.epsgcode)
        tiff_ds.SetProjection(ref.ExportToWkt())

        return tiff_filename
```

The usage of this class is a simple, two-line call:

```
doq = DOQ(extent)
doq.save(r'C:\temp\doq.img')
```

Getting the SRTM data

As of MapServer 4.4, support for WCS (Web Coverage Service) is available. Whereas WMS provides a rendered map image, WCS allows a requestor to obtain the actual raw raster data through a structured URL request. Frank Warmerdam provides the SRTM (Shuttle Radar Topography Mission) through a WCS service at http://maps.gdal.org . Here is an example of the "Capabilities" request that you can use to find out more information about a WCS server – in our case Frank's.

```
http://maps.gdal.org/cgi-bin/mapserv_dem?&version=1.0.0&service=WCS&request=GetCapabilities
```

Hitting this URL in our web browser, we can see that there is one layer, called *srtmplus_raw*, which appears to have what we want.

Next, we'll use a *DescribeCoverage* method to find out more information about the layer.

```
http://maps.gdal.org/cgi-bin/mapserv_dem?&version=1.0.0&service=WCS&request=DescribeCoverage&layer=srtmplus_raw
```

The XML listing of this request shows us that that the layer is provided in the EPSG:4326 and EPSG:4269 coordinate systems, has an output format of **GEOTIFF_INT16**, and has a resolution of 0.00833333 degrees per pixel.

With this information in hand, we have enough information to build a Python class to do the work of downloading the image for us. We've already done most of the work, however. The *RemoteDEM* class already defines a way to take in an extent, and turn a temporary GDAL DataSet into a projected Imagine file.

All our RemoteSRTM class needs to define is the *_save_tempfile()* method. This method needs to formulate the request, download it, and save it to a temporary file.

```
class RemoteSRTM(RemoteDEM):
    def get(self):
        url = 'http://maps.gdal.org/cgi-bin/mapserv_dem?&crs=EPSG:4326&coverage=srtmplus_raw&version=1.0.0&service=WCS&request=GetCoverage&bbox=%s&width=100&height=100&format=GEOTIFF_INT16'

        extent_string = '%s,%s,%s,%s' % (self.extent.t_mins.GetX(),
                                         self.extent.t_mins.GetY(),
                                         self.extent.t_maxs.GetX(),
                                         self.extent.t_maxs.GetY()
                                         )
        url = url % extent_string
        response = urllib2.urlopen(url)

        astring = response.read()

        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.tiff'
```

```
        fo = open(temp_filename,'wb')
        fo.write(astring)
        fo.close()
        return temp_filename
```

USGS NED DEM

USGS provides one product of digital elevation models called the NED (National Elevation Dataset) on their website at http://seamless.usgs.gov. The user interface for requesting DEMs is pretty clunky, but it amounts to manually defining an area of interest, choosing the "Download" link, waiting in a queue, and then saving the zip file of the DEM on your local machine.

Python provides excellent capabilities for working with URLs in the standard libraries *urllib* and *urllib2*. We can use *urllib2* and *cookielib* (another standard library as of Python 2.4) to simulate the user requesting an area of interest, waiting in the queue, and saving the resulting zip file on the local file system.

We need to make a series of requests to the USGS site to simulate a user doing the same thing. The first request asks the USGS for a session, tells the site which area of interest we want, and returns us the session cookie that we will use for subsequent requests. The second request actually puts us in the queue. The final series of requests asks the website if our data is ready every 10 seconds, and when it is, downloads the data to our local machine.

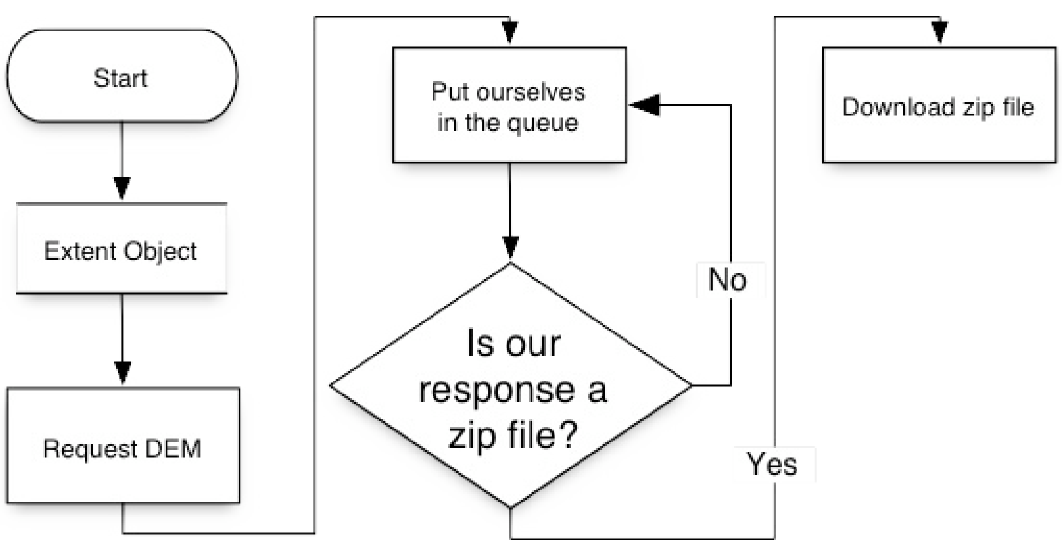


Figure 3. A process diagram of requesting a DEM from the USGS site.

Download and save the DEM

Now we need to make a class that will model the behavior of our remote DEM from the USGS site. Looking at Figure 3, we can see that we will want to feed this class one of our custom extent objects. Once returned, the DEM will be a binary Arc/Info coverage that is stored in a zip file. We will then need some methods to both get the DEM and save it to a temp directory that we can then use to create our shaded DOQ.

The __init__ method of our DEM (subclassed from Remote) class will take in our custom extent. All of the specific implementation exists in the *get()* method, and it does all of the work of actually getting the DEM from the USGS site.

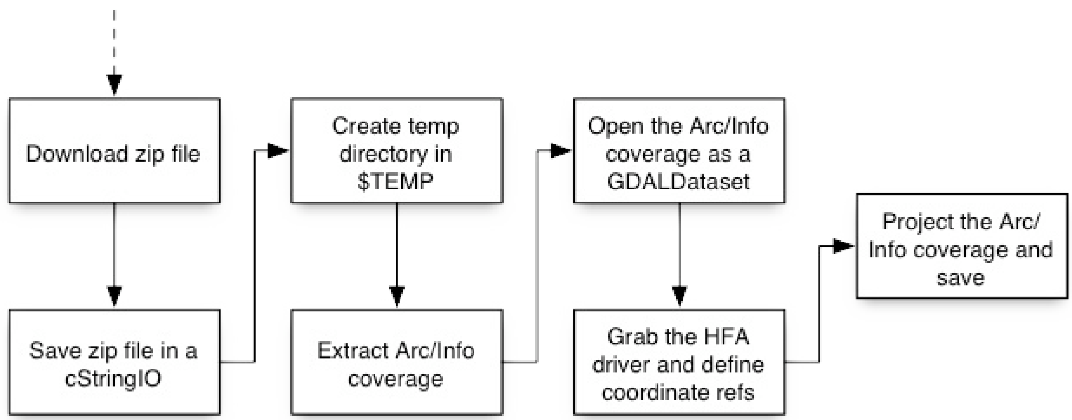


Figure 4. get() method of *DEM* (continued).

```
class DEM(Remote):
    def get(self):
        self.add_to_queue()
        self.download()
        temp_file = self.write_temp_file()
        return temp_file
```

The *get()* method of *DEM* is really just a driver function. The main pieces of requesting, downloading, and saving a DEM from the USGS site are implemented separately to make things a bit easier to read, break up the code, and allow us to make localized changes if the USGS changes their site (which they did at least once while I was writing this exercise).

The *add_to_queue()* method makes the initial request to the USGS site and gives it our extent, gets a cookie (done automatically by *cookielib*), and returns our place in the queue.

```
def add_to_queue(self):

        url = 
'http://extract.cr.usgs.gov/Website/distreq/RequestSummary.jsp?AL=%s&PR=0&P
L=NED01HZ'
```

```
        url = url %(self.extent)

        req = urllib2.Request(url)
        handle = urllib2.urlopen(req)

        url = 'http://extract.cr.usgs.gov/diststatus/servlet/gov.usgs.edc.RequestStatus?siz=1&key=NED&ras=1&rsp=1&pfm=GridFloat&lay=-1&fid=-1&lft=%s&rgt=%s&top=%s&bot=%s&wmd=1&mcd=NED&mdf=TXT&arc=ZIP&sde=NED.conus_ned&msd=NED.CONUS_NED_METADATA&zun=METERS&prj=0&csx=2.777777777999431E-4&csy=2.777777777999431E-4&bnd=&bndnm='

        url = url % (self.extent.inverse.minx,
                     self.extent.inverse.maxx,
                     self.extent.inverse.maxy,
                     self.extent.inverse.miny)

        req2 = urllib2.Request(url)
        handle = urllib2.urlopen(req2)
        self.handle = handle
```

Now that we're in the queue, we need a method to actually wait in the queue, make a request to the website to ask if it is done with our DEM every 10 seconds, and download the zip file with our data. The *download*() method does this. Instead of saving the data to a file, however, we will be storing it in a cStringIO instance. You can think of this as a string buffer. By storing it here, it will be in memory with our instance. We are using this because we want our *write_temp_file*() method to actually extract the stuff out of the zip file and return its location to the caller so that the *save()* method can do what it needs.

```
def download(self):
        if debug:
            print 'requesting DEM download...'

        # ask forever until the website returns
        # application/x-zip-compressed as the content type
        # I've had this go for over an hour sometimes.
        wait, newurl = self.handle.info()['refresh'].split(';')
        newurl = newurl.replace('URL=','').strip()
        url2 = 'http://extract.cr.usgs.gov%s'%newurl
        while 1:

            time.sleep(int(wait))
            request = urllib2.Request(url2)
            response = urllib2.urlopen(request)

            if 'text/html' not in response.info()['content-type']:
                if debug:
                    print "it's our turn... downloading DEM..."
                output = response.read()
                break
```

```
            else:
                if debug:
                    print 'still in the queue. requesting again... '

        zip_output = cStringIO.StringIO()
        zip_output.write(output)
        self.zip_output = zip_output
```

Python comes with the module **zipfile** as a standard library, and we'll now use that help us implement the *write_temp_file*() method. This method does all of the mundane business of extracting the Arc/Info coverage out to a temporary file and returning the location of the file to the caller.

```
def write_temp_file(self):
        # reset the cStringIO to read at the beginning
        self.zip_output.reset()

        # make a temp directory in $TEMP
        temp_filename = get_timestamp()
        outdir = os.path.join(temp_dir,temp_filename)
        os.mkdir(outdir)

        # the arcinfo files use the *opposite* path separator than
        # than the system's.
        if os.sep == '\\':
            info_separator = '/'
        else:
            info_separator = '\\'

        # extract the info coverage into the temp directory
        z = zipfile.ZipFile(self.zip_output)
        arcinfo_dir = z.filelist[0].filename.split(info_separator)[0]
        tempdir = os.path.join(outdir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, 'info')
        os.mkdir(tempdir)

        for name in z.namelist():
            outfile = open(os.path.join(outdir, name), 'wb')
            outfile.write(z.read(name))
            outfile.flush()
            outfile.close()

        # USGS's zipfile buries the data in another arcinfo directory
        # so we have to use two
        infile = os.path.join(outdir,arcinfo_dir,arcinfo_dir)
        return infile
```

Finally, here's how we use the *DEM* class…

```
maxx = 437142.35
miny = 4658582.96
minx = 436521.25
maxy = 4659253.80
extent = Extent(minx, miny, maxx, maxy, epsgcode=26915)
print extent
dem = DEM(extent)
dem.save(r'c:\temp\dem.img')
42.0828443199,42.076752942,-93.7599730671,-93.7674089552
requesting DEM download...
still in the queue. requesting again...
it's our turn... downloading DEM...
```

Code Listing

```
import gdal.ogr as ogr
import gdal.osr as osr
import gdal.gdal as gdal

import cookielib
import urllib2
import cStringIO
import zipfile


from pyTS import TerraImage
from pyTS import pyTerra

import os
import sys
import time

debug = 1
cookiejar = cookielib.LWPCookieJar()
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))
urllib2.install_opener(opener)

temp_dir= os.environ['TEMP']

def get_timestamp():
    """returns a big, unique, string that we can use for nonsensical filenames"""
    import md5
    q = md5.md5(str(time.time()))
    return q.hexdigest()

class Extent(object):
    """An extent that can transform itself into other coordinate systems"""
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        """MapServer-style... minx, miny, maxx, maxy, with an optional epsgcode
        defaults to EPSG:4326"""
        self.epsgcode = epsgcode
        self.minx = minx
        self.maxx = maxx
        self.miny = miny
        self.maxy = maxy
    def transform(self, target_epsg_code):
        """Transforms the extent into the target EPSG code and returns it."""
        mins = ogr.Geometry(type=ogr.wkbPoint)
        maxs = mins.Clone()

        mins.AddPoint(self.minx, self.miny)
```

```
        maxs.AddPoint(self.maxx, self.maxy)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.epsgcode)
        maxs.AssignSpatialReference(ref)
        mins.AssignSpatialReference(ref)
        out_ref = osr.SpatialReference()
        out_ref.ImportFromEPSG(target_epsg_code)
        t_mins = mins.Clone()
        t_mins.TransformTo(out_ref)
        t_maxs = maxs.Clone()
        t_maxs.TransformTo(out_ref)
        ext = Extent(t_mins.GetX(), t_mins.GetY(),
                     t_maxs.GetX(), t_maxs.GetY(),
                     epsgcode = target_epsg_code)
        return ext



class SmartExtent(object):
    """A class that acts as a container for our extents by storing the
    forward and inverse projections."""
    def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
        """MapServer-style... minx, miny, maxx, maxy, with an optional epsgcode
        defaults to EPSG:4326"""
        self.epsgcode = epsgcode
        self.forward = Extent(minx, miny, maxx, maxy, epsgcode)
        self.inverse = self.forward.transform(4326)

    def __str__(self):
        """Prints out the inverse extent in the form that the USGS site needs."""
        outstring = "%s,%s,%s,%s"
        return outstring % (self.inverse.maxy, self.inverse.miny,
                            self.inverse.maxx, self.inverse.minx)

    def get_zone(self):
        """Returns the UTM zone of the extent."""
        zones = {10:[-126,-120],
                 11:[-120,-114],
                 12:[-114,-108],
                 13:[-108,-102],
                 14:[-102,-96],
                 15:[-96,-90],
                 16:[-90,-84],
                 17:[-84,-78],
                 18:[-78,-72],
                 19:[-72,-66],
                 20:[-66,-60]
                }

```

```
        minx = self.inverse.minx
        maxx = self.inverse.maxx
        for i in zones:
            #build the epsg code
            min,max = map(float,zones[i])
            if minx > min and minx < max:
                min_utmzone = 26900+i
            if maxx > min and maxx < max:
                max_utmzone = 26900+i
        if min_utmzone == max_utmzone:
            return min_utmzone
        else:
            return None


class Remote(object):
    """A super class that defines the input for the three data types."""
    def __init__(self, extent):
        """Take in one of our SmartExtent objects."""
        self.extent = extent

    def get(self):
        """Dummy method.  Should not be called directly."""
        pass

    def save(self, filename):
        """Saves the given filename in the coordinate system that
        was given by the SmartExtent."""
        infile = self.get()

        o = gdal.OpenShared(infile)
        dst_driver = gdal.GetDriverByName('HFA')
        outref = osr.SpatialReference()
        outref.ImportFromEPSG(self.extent.epsgcode)
        dst_wkt = outref.ExportToWkt()
        inref = osr.SpatialReference()

        can_import = inref.ImportFromWkt(o.GetProjection())
        if can_import != 0:
            inref.ImportFromEPSG(4326)
        src_wkt = inref.ExportToWkt()

        gdal.CreateAndReprojectImage(o,
                                     filename,
                                     src_wkt = src_wkt,
                                     dst_driver=dst_driver,
                                     dst_wkt=dst_wkt)

class DEM(Remote):
    """A class to get DEMs from the USGS Seamless site at
    http://seamless.usgs.gov"""
```

```
    def get(self):
        """Returns the downloaded file for the save() method"""
        self.add_to_queue()
        self.download()
        temp_file = self.write_temp_file()
        return temp_file

    def add_to_queue(self):
        """Adds our request for a DEM to the USGS queue."""
        url = 'http://extract.cr.usgs.gov/Website/distreq/RequestSummary.jsp?AL=%s&PR=0&PL=NED01HZ'
        url = url %(self.extent)

        req = urllib2.Request(url)
        handle = urllib2.urlopen(req)

        url = 'http://extract.cr.usgs.gov/diststatus/servlet/gov.usgs.edc.RequestStatus?siz=1&key=NED&ras=1&rsp=1&pfm=GridFloat&lay=-1&fid=-1&lft=%s&rgt=%s&top=%s&bot=%s&wmd=1&mcd=NED&mdf=TXT&arc=ZIP&sde=NED.conus_ned&msd=NED.CONUS_NED_METADATA&zun=METERS&prj=0&csx=2.777777777999431E-4&csy=2.777777777999431E-4&bnd=&bndnm='

        url = url % (self.extent.inverse.minx,
                     self.extent.inverse.maxx,
                     self.extent.inverse.maxy,
                     self.extent.inverse.miny)

        req2 = urllib2.Request(url)
        handle = urllib2.urlopen(req2)
        self.handle = handle

    def download(self):
        """Waits in the USGS queue and downloads the DEM in
        Arc/Info format when it is our turn."""
        if debug:
            print 'requesting DEM download...'

        # ask forever until the website returns
        # application/x-zip-compressed as the content type
        # I've had this go for over an hour sometimes.
        wait, newurl = self.handle.info()['refresh'].split(';')
        newurl = newurl.replace('URL=','').strip()
        url2 = 'http://extract.cr.usgs.gov%s'%newurl
        while 1:

            time.sleep(int(wait))
            request = urllib2.Request(url2)
            response = urllib2.urlopen(request)

```

```
            if 'text/html' not in response.info()['content-type']:
                if debug:
                    print "it's our turn... downloading DEM..."
                output = response.read()
                break
            else:
                if debug:
                    print 'still in the queue. requesting again... '

        zip_output = cStringIO.StringIO()
        zip_output.write(output)
        self.zip_output = zip_output

    def write_temp_file(self):
        """Writes out the file given by the download() method
        in a temporary directory and returns the location to the
        caller."""
        # reset the cStringIO to read at the beginning
        self.zip_output.reset()

        # make a temp directory in $TEMP
        temp_filename = get_timestamp()
        outdir = os.path.join(temp_dir,temp_filename)
        os.mkdir(outdir)

        # the arcinfo files use the *opposite* path separator than
        # than the system's.
        if os.sep == '\\':
            info_separator = '/'
        else:
            info_separator = '\\'

        # extract the info coverage into the temp directory
        z = zipfile.ZipFile(self.zip_output)
        arcinfo_dir = z.filelist[0].filename.split(info_separator)[0]
        tempdir = os.path.join(outdir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, arcinfo_dir)
        os.mkdir(tempdir)
        tempdir = os.path.join(outdir, arcinfo_dir, 'info')
        os.mkdir(tempdir)

        for name in z.namelist():
            outfile = open(os.path.join(outdir, name), 'wb')
            outfile.write(z.read(name))
            outfile.flush()
            outfile.close()

        # USGS's zipfile buries the data in another arcinfo directory
        # so we have to use two
        infile = os.path.join(outdir,arcinfo_dir,arcinfo_dir)
```

```
        return infile




class DOQ(Remote):
    """A class that implements getting a DOQ from the Microsoft
    TerraServer."""
    def get(self):
        """Returns the downloaded file for the save() method"""
        thescale = 'Scale1m' # scale of the DOQ from TS
        thetype = 'Photo'# Photo or Topo

        # a TerraImage must know its zone
        thezone = self.extent.get_zone() - 26900
        upperLeft = TerraImage.point(self.extent.inverse.maxy,
                                     self.extent.inverse.minx)
        lowerRight = TerraImage.point(self.extent.inverse.miny,
                                      self.extent.inverse.maxx)

        ti = TerraImage.TerraImage(upperLeft,
                                   lowerRight,
                                   thescale,
                                   thetype,
                                   thezone)
        self.ti = ti
        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.jpg'
        self.ti.write(temp_filename)
        self.ti.write_worldfile(temp_filename+"w")

        ds = gdal.Open(temp_filename)
        drv = gdal.GetDriverByName('GTiff')
        tiff_filename = temp_filename.replace('.jpg','.tiff')
        tiff_ds = drv.CreateCopy(tiff_filename, ds)
        ref = osr.SpatialReference()
        ref.ImportFromEPSG(self.extent.epsgcode)
        tiff_ds.SetProjection(ref.ExportToWkt())

        return tiff_filename

class SRTM(Remote):
    """A class that implements getting the SRTM data from the
    WCS server at http://maps.gdal.org"""
    def get(self):
        """Returns the downloaded file for the save() method"""
        url = 'http://maps.gdal.org/cgi-
bin/mapserv_dem?&crs=EPSG:4326&coverage=srtmplus_raw&version=1.0.0&service=
WCS&request=GetCoverage&bbox=%s&width=100&height=100&format=GEOTIFF_INT16'

        extent_string = '%s,%s,%s,%s' % (self.extent.inverse.minx,
                                         self.extent.inverse.miny,
```

```
                                              self.extent.inverse.maxx,
                                              self.extent.inverse.maxy
                                              )
        url = url % extent_string
        response = urllib2.urlopen(url)

        astring = response.read()

        temp_filename = os.path.join(temp_dir, get_timestamp()) + '.tiff'
        fo = open(temp_filename,'wb')
        fo.write(astring)
        fo.close()
        return temp_filename

maxx = 437142.35
miny = 4658582.96
minx = 436521.25
maxy = 4659253.80
extent = SmartExtent(minx, miny, maxx, maxy, epsgcode=26915)
print extent
dem = DEM(extent)
dem.save(r'c:\temp\usgs.img')
doq = DOQ(extent)
doq.save(r'C:\temp\doq.img')
srtm = SRTM(extent)
srtm.save(r'c:\temp\srtm.img')
```