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.

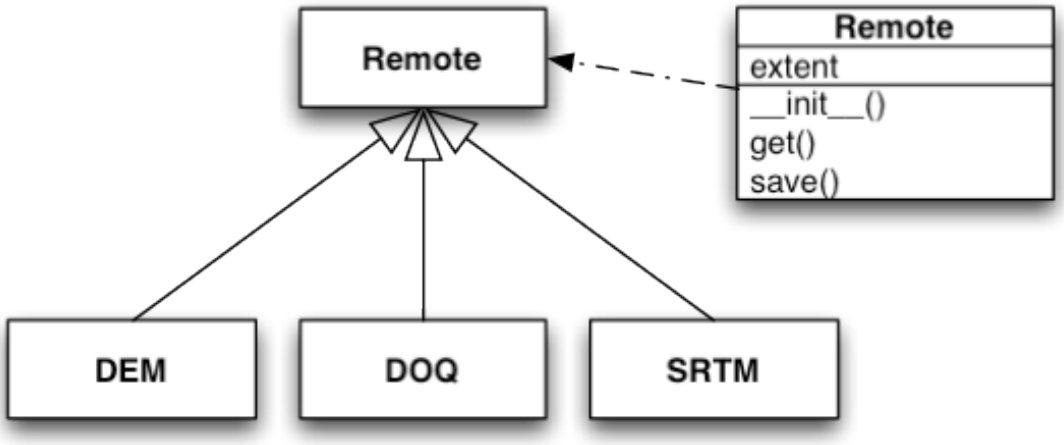

Figure 1. The *Remote* class implements the three methods – *__init__*, *get*, and *save* – that each of the three data types need. We will subclass *Remote* for all three and provide our own implementation of get() for each.

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.

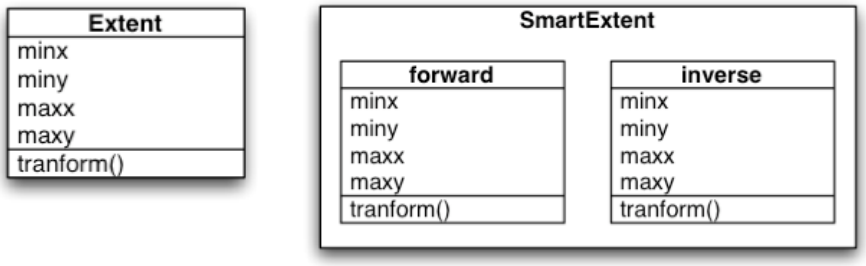

Figure 2. *Extent* and *SmartExtent* objects. The *SmartExtent* object just acts as a container and takes care of calling the *transform()* method for us.

```
1   class Extent(object):
2       def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
3           self.epsgcode = epsgcode
4           self.minx = minx
5           self.maxx = maxx
6           self.miny = miny
7           self.maxy = maxy
8       def transform(self, target_epsg_code):
9           mins = ogr.Geometry(type=ogr.wkbPoint)
10          maxs = mins.Clone()
11
12          mins.AddPoint(self.minx, self.miny)
13          maxs.AddPoint(self.maxx, self.maxy)
14          ref = osr.SpatialReference()
15          ref.ImportFromEPSG(self.epsgcode)
16          maxs.AssignSpatialReference(ref)
17          mins.AssignSpatialReference(ref)
18          out_ref = osr.SpatialReference()
19          out_ref.ImportFromEPSG(target_epsg_code)
20          t_mins = mins.Clone()
21          t_mins.TransformTo(out_ref)
22          t_maxs = maxs.Clone()
23          t_maxs.TransformTo(out_ref)
24          ext = Extent(t_mins.GetX(), t_mins.GetY(),
25                       t_maxs.GetX(), t_maxs.GetY(),
26                       epsgcode = target_epsg_code)
27          return ext
```

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

```
28  class SmartExtent(object):
29      def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
30          self.epsgcode = epsgcode
31          self.forward = Extent(minx, miny, maxx, maxy, epsgcode)
32          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).

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

Some test code

```
37  minx = 437142.35
38  miny = 4658582.96
39  maxx = 436521.25
40  maxy = 4659253.80
```

```
41  extent = SmartExtent(minx, miny, maxx, maxy, epsgcode=26915)
42  print extent
43  >> 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.

```
44  class Remote(object):
45      def __init__(self, extent):
46          self.extent = extent
47
48      def get(self):
49          pass
50
51      def save(self, filename):
52
53          infile = self.get()
54
55          o = gdal.OpenShared(infile)
56          dst_driver = gdal.GetDriverByName('HFA')
57          outref = osr.SpatialReference()
58          outref.ImportFromEPSG(self.extent.epsgcode)
59          dst_wkt = outref.ExportToWkt()
60          inref = osr.SpatialReference()
61
62          can_import = inref.ImportFromWkt(o.GetProjection())
63          if can_import != 0:
64              inref.ImportFromEPSG(4326)
65          src_wkt = inref.ExportToWkt()
66
67
```

```
68          gdal.CreateAndReprojectImage(o,
69                                       filename,
70                                       src_wkt = src_wkt,
71                                       dst_driver=dst_driver,
72                                       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).

```
73  class SmartExtent(object):
74  …
75  def get_zone(self):
76          zones = {10:[-126,-120],
77                   11:[-120,-114],
78                   12:[-114,-108],
79                   13:[-108,-102],
80                   14:[-102,-96],
81                   15:[-96,-90],
82                   16:[-90,-84],
83                   17:[-84,-78],
84                   18:[-78,-72],
85                   19:[-72,-66],
86                   20:[-66,-60]
87                  }
88
89          minx = self.inverse.minx
90          maxx = self.inverse.maxx
91          for i in zones:
92              #build the epsg code
93              min,max = map(float,zones[i])
94              if minx > min and minx < max:
95                  min_utmzone = 26900+i
96              if maxx > min and maxx < max:
97                  max_utmzone = 26900+i
```

```
98          if min_utmzone == max_utmzone:
99              return min_utmzone
100         else:
101             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.

```
102 class DOQ(Remote):
103
104     def get(self):
105         thescale = 'Scale1m' # scale of the DOQ from TS
106         thetype = 'Photo'# Photo or Topo
107
108         # a TerraImage must know its zone
109         thezone = self.extent.get_zone() - 26900
110         upperLeft = TerraImage.point(self.extent.inverse.maxy,
111                                      self.extent.inverse.minx)
112         lowerRight = TerraImage.point(self.extent.inverse.miny,
113                                       self.extent.inverse.maxx)
114
115         ti = TerraImage.TerraImage(upperLeft,
116                                    lowerRight,
117                                    thescale,
118                                    thetype,
119                                    thezone)
120         self.ti = ti
121         temp_filename = os.path.join(temp_dir, get_timestamp()) + '.jpg'
122         self.ti.write(temp_filename)
123         self.ti.write_worldfile(temp_filename+"w")
124
125         ds = gdal.Open(temp_filename)
126         drv = gdal.GetDriverByName('GTiff')
127         tiff_filename = temp_filename.replace('.jpg','.tiff')
128         tiff_ds = drv.CreateCopy(tiff_filename, ds)
129         ref = osr.SpatialReference()
130         ref.ImportFromEPSG(self.extent.epsgcode)
131         tiff_ds.SetProjection(ref.ExportToWkt())
132
133         return tiff_filename
```

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

```
134 doq = DOQ(extent)
135 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.

```
136 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.

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

```
138 class RemoteSRTM(RemoteDEM):
139     def get(self):
140         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'
141
142         extent_string = '%s,%s,%s,%s' % (self.extent.t_mins.GetX(),
143                                          self.extent.t_mins.GetY(),
144                                          self.extent.t_maxs.GetX(),
145                                          self.extent.t_maxs.GetY()
146                                          )
147         url = url % extent_string
148         response = urllib2.urlopen(url)
149
150         astring = response.read()
151
152         temp_filename = os.path.join(temp_dir, get_timestamp()) + '.tiff'
```

```
153         fo = open(temp_filename,'wb')
154         fo.write(astring)
155         fo.close()
156         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.

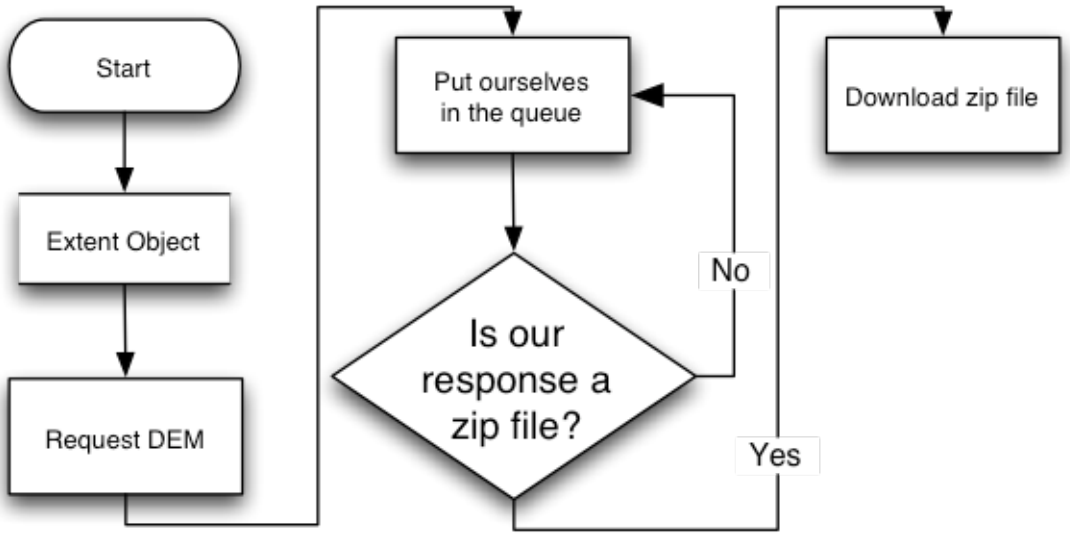

Figure 3. A process diagram of requesting a DEM from the USGS site.

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.

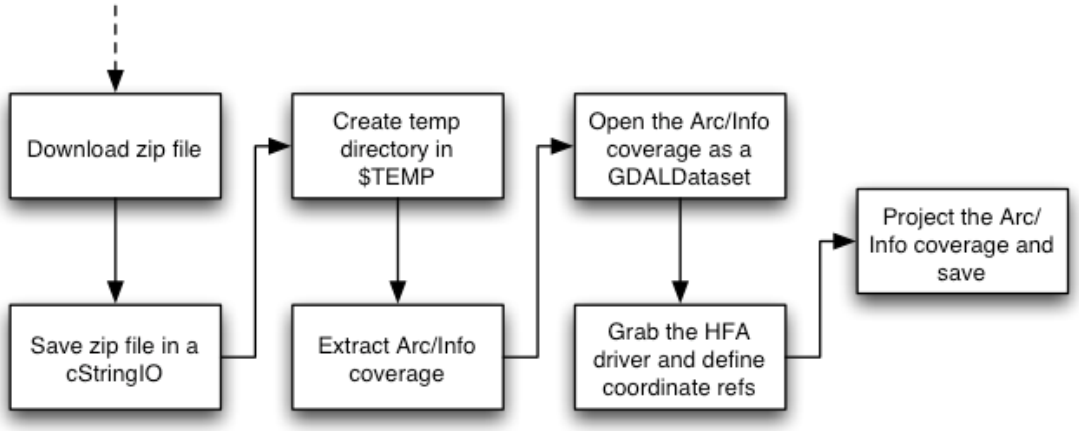

Figure 4. get() method of *DEM* (continued).

```
157 class DEM(Remote):
158     def get(self):
159         self.add_to_queue()
160         self.download()
161         temp_file = self.write_temp_file()
162         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.

```
163 def add_to_queue(self):
164
165         url =
    'http://extract.cr.usgs.gov/Website/distreq/RequestSummary.jsp?AL=%s&PR=0&P
    L=NED01HZ'
```

```
166         url = url %(self.extent)
167
168         req = urllib2.Request(url)
169         handle = urllib2.urlopen(req)
170
171         url =
    'http://extract.cr.usgs.gov/diststatus/servlet/gov.usgs.edc.RequestStatus?s
    iz=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_n
    ed&msd=NED.CONUS_NED_METADATA&zun=METERS&prj=0&csx=2.777777777999431E-
    4&csy=2.777777777999431E-4&bnd=&bndnm='
172
173         url = url % (self.extent.inverse.minx,
174                      self.extent.inverse.maxx,
175                      self.extent.inverse.maxy,
176                      self.extent.inverse.miny)
177
178         req2 = urllib2.Request(url)
179         handle = urllib2.urlopen(req2)
180         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.

```
181 def download(self):
182         if debug:
183             print 'requesting DEM download...'
184
185         # ask forever until the website returns
186         # application/x-zip-compressed as the content type
187         # I've had this go for over an hour sometimes.
188         wait, newurl = self.handle.info()['refresh'].split(';')
189         newurl = newurl.replace('URL=','').strip()
190         url2 = 'http://extract.cr.usgs.gov%s'%newurl
191         while 1:
192
193             time.sleep(int(wait))
194             request = urllib2.Request(url2)
195             response = urllib2.urlopen(request)
196
197             if 'text/html' not in response.info()['content-type']:
198                 if debug:
199                     print "it's our turn... downloading DEM..."
200                 output = response.read()
201                 break
```

```
202             else:
203                 if debug:
204                     print 'still in the queue. requesting again... '
205 
206         zip_output = cStringIO.StringIO()
207         zip_output.write(output)
208         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.

```
209 def write_temp_file(self):
210         # reset the cStringIO to read at the beginning
211         self.zip_output.reset()
212 
213         # make a temp directory in $TEMP
214         temp_filename = get_timestamp()
215         outdir = os.path.join(temp_dir,temp_filename)
216         os.mkdir(outdir)
217 
218         # the arcinfo files use the *opposite* path separator than
219         # than the system's.
220         if os.sep == '\\':
221             info_separator = '/'
222         else:
223             info_separator = '\\'
224 
225         # extract the info coverage into the temp directory
226         z = zipfile.ZipFile(self.zip_output)
227         arcinfo_dir = z.filelist[0].filename.split(info_separator)[0]
228         tempdir = os.path.join(outdir, arcinfo_dir)
229         os.mkdir(tempdir)
230         tempdir = os.path.join(outdir, arcinfo_dir, arcinfo_dir)
231         os.mkdir(tempdir)
232         tempdir = os.path.join(outdir, arcinfo_dir, 'info')
233         os.mkdir(tempdir)
234 
235         for name in z.namelist():
236             outfile = open(os.path.join(outdir, name), 'wb')
237             outfile.write(z.read(name))
238             outfile.flush()
239             outfile.close()
240 
241         # USGS's zipfile buries the data in another arcinfo directory
242         # so we have to use two
243         infile = os.path.join(outdir,arcinfo_dir,arcinfo_dir)
244         return infile
```

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

```
245 maxx = 437142.35
246 miny = 4658582.96
247 minx = 436521.25
248 maxy = 4659253.80
249 extent = Extent(minx, miny, maxx, maxy, epsgcode=26915)
250 print extent
251 dem = DEM(extent)
252 dem.save(r'c:\temp\dem.img')
253 42.0828443199,42.076752942,-93.7599730671,-93.7674089552
254 requesting DEM download...
255 still in the queue. requesting again...
256 it's our turn... downloading DEM...
```

Code Listing

```
1   import gdal.ogr as ogr
2   import gdal.osr as osr
3   import gdal.gdal as gdal
4
5   import cookielib
6   import urllib2
7   import cStringIO
8   import zipfile
9
10
11  from pyTS import TerraImage
12  from pyTS import pyTerra
13
14  import os
15  import sys
16  import time
17
18  debug = 1
19  cookiejar = cookielib.LWPCookieJar()
20  opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))
21  urllib2.install_opener(opener)
22
23  temp_dir= os.environ['TEMP']
24
25  def get_timestamp():
26      """returns a big, unique, string that we can use for nonsensical
    filenames"""
27      import md5
28      q = md5.md5(str(time.time()))
29      return q.hexdigest()
30
31  class Extent(object):
32      """An extent that can transform itself into other coordinate systems"""
33      def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
34          """MapServer-style... minx, miny, maxx, maxy, with an optional
    epsgcode
35          defaults to EPSG:4326"""
36          self.epsgcode = epsgcode
37          self.minx = minx
38          self.maxx = maxx
39          self.miny = miny
40          self.maxy = maxy
41      def transform(self, target_epsg_code):
42          """Transforms the extent into the target EPSG code and returns
    it."""
43          mins = ogr.Geometry(type=ogr.wkbPoint)
44          maxs = mins.Clone()
45
46          mins.AddPoint(self.minx, self.miny)
```

```
47          maxs.AddPoint(self.maxx, self.maxy)
48          ref = osr.SpatialReference()
49          ref.ImportFromEPSG(self.epsgcode)
50          maxs.AssignSpatialReference(ref)
51          mins.AssignSpatialReference(ref)
52          out_ref = osr.SpatialReference()
53          out_ref.ImportFromEPSG(target_epsg_code)
54          t_mins = mins.Clone()
55          t_mins.TransformTo(out_ref)
56          t_maxs = maxs.Clone()
57          t_maxs.TransformTo(out_ref)
58          ext = Extent(t_mins.GetX(), t_mins.GetY(),
59                       t_maxs.GetX(), t_maxs.GetY(),
60                       epsgcode = target_epsg_code)
61          return ext
62
63
64
65  class SmartExtent(object):
66      """A class that acts as a container for our extents by storing the
67      forward and inverse projections."""
68      def __init__(self, minx, miny, maxx, maxy, epsgcode=4326):
69          """MapServer-style... minx, miny, maxx, maxy, with an optional
    epsgcode
70          defaults to EPSG:4326"""
71          self.epsgcode = epsgcode
72          self.forward = Extent(minx, miny, maxx, maxy, epsgcode)
73          self.inverse = self.forward.transform(4326)
74
75      def __str__(self):
76          """Prints out the inverse extent in the form that the USGS site
    needs."""
77          outstring = "%s,%s,%s,%s"
78          return outstring % (self.inverse.maxy, self.inverse.miny,
79                              self.inverse.maxx, self.inverse.minx)
80
81      def get_zone(self):
82          """Returns the UTM zone of the extent."""
83          zones = {10:[-126,-120],
84                   11:[-120,-114],
85                   12:[-114,-108],
86                   13:[-108,-102],
87                   14:[-102,-96],
88                   15:[-96,-90],
89                   16:[-90,-84],
90                   17:[-84,-78],
91                   18:[-78,-72],
92                   19:[-72,-66],
93                   20:[-66,-60]
94                  }
95
```

```
96          minx = self.inverse.minx
97          maxx = self.inverse.maxx
98          for i in zones:
99              #build the epsg code
100             min,max = map(float,zones[i])
101             if minx > min and minx < max:
102                 min_utmzone = 26900+i
103             if maxx > min and maxx < max:
104                 max_utmzone = 26900+i
105         if min_utmzone == max_utmzone:
106             return min_utmzone
107         else:
108             return None
109
110
111 class Remote(object):
112     """A super class that defines the input for the three data types."""
113     def __init__(self, extent):
114         """Take in one of our SmartExtent objects."""
115         self.extent = extent
116
117     def get(self):
118         """Dummy method.  Should not be called directly."""
119         pass
120
121     def save(self, filename):
122         """Saves the given filename in the coordinate system that
123         was given by the SmartExtent."""
124         infile = self.get()
125
126         o = gdal.OpenShared(infile)
127         dst_driver = gdal.GetDriverByName('HFA')
128         outref = osr.SpatialReference()
129         outref.ImportFromEPSG(self.extent.epsgcode)
130         dst_wkt = outref.ExportToWkt()
131         inref = osr.SpatialReference()
132
133         can_import = inref.ImportFromWkt(o.GetProjection())
134         if can_import != 0:
135             inref.ImportFromEPSG(4326)
136         src_wkt = inref.ExportToWkt()
137
138         gdal.CreateAndReprojectImage(o,
139                                      filename,
140                                      src_wkt = src_wkt,
141                                      dst_driver=dst_driver,
142                                      dst_wkt=dst_wkt)
143
144 class DEM(Remote):
145     """A class to get DEMs from the USGS Seamless site at
146     http://seamless.usgs.gov"""
```

```
147     def get(self):
148         """Returns the downloaded file for the save() method"""
149         self.add_to_queue()
150         self.download()
151         temp_file = self.write_temp_file()
152         return temp_file
153 
154     def add_to_queue(self):
155         """Adds our request for a DEM to the USGS queue."""
156         url =
    'http://extract.cr.usgs.gov/Website/distreq/RequestSummary.jsp?AL=%s&PR=0&P
    L=NED01HZ'
157         url = url %(self.extent)
158 
159         req = urllib2.Request(url)
160         handle = urllib2.urlopen(req)
161 
162         url =
    'http://extract.cr.usgs.gov/diststatus/servlet/gov.usgs.edc.RequestStatus?s
    iz=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_n
    ed&msd=NED.CONUS_NED_METADATA&zun=METERS&prj=0&csx=2.777777777999431E-
    4&csy=2.777777777999431E-4&bnd=&bndnm='
163 
164         url = url % (self.extent.inverse.minx,
165                      self.extent.inverse.maxx,
166                      self.extent.inverse.maxy,
167                      self.extent.inverse.miny)
168 
169         req2 = urllib2.Request(url)
170         handle = urllib2.urlopen(req2)
171         self.handle = handle
172 
173     def download(self):
174         """Waits in the USGS queue and downloads the DEM in
175         Arc/Info format when it is our turn."""
176         if debug:
177             print 'requesting DEM download...'
178 
179         # ask forever until the website returns
180         # application/x-zip-compressed as the content type
181         # I've had this go for over an hour sometimes.
182         wait, newurl = self.handle.info()['refresh'].split(';')
183         newurl = newurl.replace('URL=','').strip()
184         url2 = 'http://extract.cr.usgs.gov%s'%newurl
185         while 1:
186 
187             time.sleep(int(wait))
188             request = urllib2.Request(url2)
189             response = urllib2.urlopen(request)
190 
```

```
191             if 'text/html' not in response.info()['content-type']:
192                 if debug:
193                     print "it's our turn... downloading DEM..."
194                 output = response.read()
195                 break
196             else:
197                 if debug:
198                     print 'still in the queue. requesting again... '
199 
200         zip_output = cStringIO.StringIO()
201         zip_output.write(output)
202         self.zip_output = zip_output
203 
204     def write_temp_file(self):
205         """Writes out the file given by the download() method
206         in a temporary directory and returns the location to the
207         caller."""
208         # reset the cStringIO to read at the beginning
209         self.zip_output.reset()
210 
211         # make a temp directory in $TEMP
212         temp_filename = get_timestamp()
213         outdir = os.path.join(temp_dir,temp_filename)
214         os.mkdir(outdir)
215 
216         # the arcinfo files use the *opposite* path separator than
217         # than the system's.
218         if os.sep == '\\':
219             info_separator = '/'
220         else:
221             info_separator = '\\'
222 
223         # extract the info coverage into the temp directory
224         z = zipfile.ZipFile(self.zip_output)
225         arcinfo_dir = z.filelist[0].filename.split(info_separator)[0]
226         tempdir = os.path.join(outdir, arcinfo_dir)
227         os.mkdir(tempdir)
228         tempdir = os.path.join(outdir, arcinfo_dir, arcinfo_dir)
229         os.mkdir(tempdir)
230         tempdir = os.path.join(outdir, arcinfo_dir, 'info')
231         os.mkdir(tempdir)
232 
233         for name in z.namelist():
234             outfile = open(os.path.join(outdir, name), 'wb')
235             outfile.write(z.read(name))
236             outfile.flush()
237             outfile.close()
238 
239         # USGS's zipfile buries the data in another arcinfo directory
240         # so we have to use two
241         infile = os.path.join(outdir,arcinfo_dir,arcinfo_dir)
```

```
242         return infile
243
244
245
246
247 class DOQ(Remote):
248     """A class that implements getting a DOQ from the Microsoft
249     TerraServer."""
250     def get(self):
251         """Returns the downloaded file for the save() method"""
252         thescale = 'Scale1m' # scale of the DOQ from TS
253         thetype = 'Photo'# Photo or Topo
254
255         # a TerraImage must know its zone
256         thezone = self.extent.get_zone() - 26900
257         upperLeft = TerraImage.point(self.extent.inverse.maxy,
258                                      self.extent.inverse.minx)
259         lowerRight = TerraImage.point(self.extent.inverse.miny,
260                                       self.extent.inverse.maxx)
261
262         ti = TerraImage.TerraImage(upperLeft,
263                                    lowerRight,
264                                    thescale,
265                                    thetype,
266                                    thezone)
267         self.ti = ti
268         temp_filename = os.path.join(temp_dir, get_timestamp()) + '.jpg'
269         self.ti.write(temp_filename)
270         self.ti.write_worldfile(temp_filename+"w")
271
272         ds = gdal.Open(temp_filename)
273         drv = gdal.GetDriverByName('GTiff')
274         tiff_filename = temp_filename.replace('.jpg','.tiff')
275         tiff_ds = drv.CreateCopy(tiff_filename, ds)
276         ref = osr.SpatialReference()
277         ref.ImportFromEPSG(self.extent.epsgcode)
278         tiff_ds.SetProjection(ref.ExportToWkt())
279
280         return tiff_filename
281
282 class SRTM(Remote):
283     """A class that implements getting the SRTM data from the
284     WCS server at http://maps.gdal.org"""
285     def get(self):
286         """Returns the downloaded file for the save() method"""
287         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'
288
289         extent_string = '%s,%s,%s,%s' % (self.extent.inverse.minx,
290                                          self.extent.inverse.miny,
```

```
291                                             self.extent.inverse.maxx,
292                                             self.extent.inverse.maxy
293                                             )
294         url = url % extent_string
295         response = urllib2.urlopen(url)
296
297         astring = response.read()
298
299         temp_filename = os.path.join(temp_dir, get_timestamp()) + '.tiff'
300         fo = open(temp_filename,'wb')
301         fo.write(astring)
302         fo.close()
303         return temp_filename
304
305 maxx = 437142.35
306 miny = 4658582.96
307 minx = 436521.25
308 maxy = 4659253.80
309 extent = SmartExtent(minx, miny, maxx, maxy, epsgcode=26915)
310 print extent
311 dem = DEM(extent)
312 dem.save(r'c:\temp\usgs.img')
313 doq = DOQ(extent)
314 doq.save(r'C:\temp\doq.img')
315 srtm = SRTM(extent)
316 srtm.save(r'c:\temp\srtm.img')
```