Chapter 20
Working with HTML and KML

Abstract GIS tasks often involve working with HTML and KML files. You may need to generate an HTML page showing the results of some GIS analysis or you may need to parse the data attributes in the description that pops up when you click on a KML element in Google Earth. HTML and KML files are simply text files that use tags to delineate elements within the data. Chapter 19 discussed how to read and write text files with Python; This chapter explains some additional techniques for working with HTML and KML formats.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Identify basic HTML tags, such as links, images, and text formatting.
- Set HTML tag attributes.
- Create an HTML file with text formatting, links, and embedded images.
- Identify KML tags that define geographic features.
- Write HTML and KML with Python.
- Download and save Web site contents with Python.
- Extract files from Zip and KMZ archives.
- Parse HTML and KML with the Python.
- Convert KML files to ESRI shapefiles with Python.

20.1 Working with HTML

HTML, which stands for Hyper Text Markup Language is a popular format for Web pages. This section provides a brief introduction to the format of these files. Readers who are interested in learning more should search for interactive HTML tutorials online which allow you to enter HTML code and view the results.

HTML is a language for encoding web content. Files containing HTML code have '.html' or '.htm' file extensions. HTML is not a programming language like Python, but rather a language that uses tags to create web pages. A *tag* is a set of characters surrounded by angle brackets, such as `<html>` and `<body>`, as shown in Example 20.1. This simple HTML file is named 'elephant1.html'.

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_20

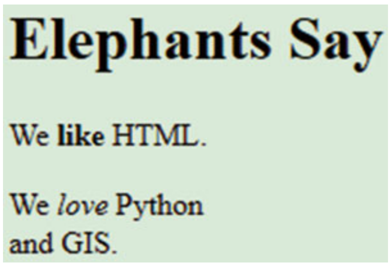
Example 20.1: HTML code from the file 'C:\gispy\data\ch20\htmlExamplePages\elephant1.html'.

```
<!DOCTYPE html>
<html>
    <body bgcolor='Aquamarine'>
          <!-- blablabla -->
          <h1>Elephants Say</h1>
          <p> We <b>like</b> HTML.</p>
          We <i>love</i> Python <br /> and GIS.
    </body>
</html>
```

If you open 'elephant1.html' (found in 'C:\gispy\data\ch20\htmlExamplePages\') without specifying a program, by default it opens in a Web browser, such as Chrome or Firefox. To view the HTML tags, open the file using a text editor. The freely downloadable text editor named 'Notepad++' is good for viewing HTML because it provides context highlighting for HTML code elements; That is, it colors the HTML file contents to differentiate amongst various code components as IDE's do for Python scripts.

The tags are visible when the file is viewed in a text editor, but not when the same file is viewed in a Web browser. The 'elephant.html' file looks as pictured in Figure 20.1 when viewed in a Web browser. The browser interprets the tags instead of displaying them. For example, the tag makes the word 'like' display with bold font. The tags in Example 20.1, serve the following purposes:

- The DOCTYPE tag defines the document type as HTML5. HTML5 defines a set of standards for HTML.
- The text between `<html>` and `</html>` are wrapped around the entire contents of the page so that it will be interpreted as HTML.
- The text between `<body>` and `</body>` are wrapped around all of the page content that is meant to be displayed. Other content, such code that controls the page style, will be between the `<html>` tags, but not between the `<body>` tags. The background color of the page is specified using `bgcolor="Aquamarine"`.



Elephants Say

We **like** HTML.

We *love* Python
and GIS.

Figure 20.1 The HTML page created with the code in Example 20.1.

- The `<!--blablabla-->` tag is a code comment. Like Python comments, HTML comments are only there for the human reader. In Python, anything placed on a line following a # sign is a comment; However in HTML, comments are wrapped in a set of characters. Start comments with `<!--` and end them with `-->`. The text between these characters is a comment.
- The text between `<h1>` and `</h1>` is displayed as a heading.
- The text between `<p>` and `</p>` is displayed as a paragraph, hence the vertical space surrounding the 'we like HTML' paragraph.
- The text between `<b>` and `</b>` is displayed as bold.
- The `<br/>` tag inserts a line break, so that the phrase 'We love Python' and the phrase 'and GIS' appear on separate lines.
- Can you guess what the `<i>` and `</i>` tags do?

Many tags pair a *start tag* with an *end tag*. For example, `<b>` is a start tag, which starts the bold text and `</b>` is an end tag which ends the bold text. The end tag always starts with a forward slash. Only the word 'like' is bold because the `</b>` tag ends the bold font setting. We say pairs of tags are *wrapped* around the *tag content* because tag instructions are applied to whatever appears between them in the HTML text. The word 'like' is the tag content wrapped in tags that apply bold font. Some tags such as the line break tag and comment tags are not paired. Some tags have *tag attributes* that provide more information about the tag. Most tag attributes are optional and they are placed inside the tag after the tag name before the closing angle bracket. An attribute is specified with the attribute name equal to the attribute value in quotes (similar to a Python string assignment statement). For example, the `body` tag has an attribute for specifying the background color (`bgcolor`). Example 20.1 sets the background color to `"Aquamarine"`. Replace `"Aquamarine"` with `"Pink"` or another named color (there are over 100 allowable color names) to change the background. HTML5 files usually control the background with other code in cascading style sheets (css files). The background color is only used here as a simple example of a tag attribute. Tags and attributes for links, lists, and tables are discussed next.

20.1.1 Specifying Links

The text between `<a>` and `</a>` can be linked to a Web address. The `ref` attribute specifies the hypertext reference, the linked Web page. If the file resides in the same directory, specify the reference as the name of the file. For example, since 'elephant1.html' is in the same directory as 'elephant2.html', the following code is placed in 'elephant2.html' to link to the 'elephant1.html' file:

```
<a href="elephant1.html">A link</a>
```

If the file is within the same Web site, use a relative path for the hypertext reference (review relative paths in Section 16.1). If the file is not on the same Web site, the full Web address needs to be specified. The second link in ‘elephant2.html’ uses the following code to point to Google’s home page:

```
<a href="https://www.google.com/">Another link</a>
```

20.1.2 Embedding Images

The `<img>` tag is used to embed images in a web page. The `src` attribute specifies the image source, a path to an image. The image source can be specified with a relative or complete path. Other attributes such as `width` and `height` can be used to format the image. When more than one attribute is used, they are separated by spaces, as in the following example:

```
<img src="../pics/lakshmi.jpg" width="32" height="32">
```

The file ‘elephant2.html’ specifies two links and an image, as shown in Figure 20.2. The code uses the width and height attributes to reduce the image size. Add the following line of code to ‘elephant2.html’ to embed a full-sized version of the image (and refresh your browser to view the change):

```
<img src="../pics/lakshmi.jpg">
```

Notice, the image source is specified by a relative path to the image. If you move the html file, you would need to place the image in the same relative position. If the relative path to an image is incorrect, it will not appear in the page. When you add the following code for a third image link to ‘elephant2.html’, you will still only see two pictures, since there is no ‘lakshmi.jpg’ image in the ‘htmlExamplePages’ directory:

```
<img src="lakshmi.jpg">
```

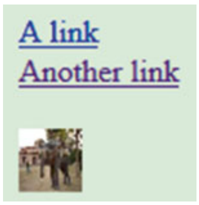


Figure 20.2 The HTML page created with the code in ‘C:\gispy\data\ch20\htmlExamplePages\elephant2.html’.

20.1.3 HTML Lists

HTML lists and tables are important for reporting GIS results. These structures both have tags nested inside of tags. There are two types of HTML lists, ordered (ol) and unordered (ul) lists. A list is either wrapped in `<ol>` and `</ol>` tags or `<ul>` and `</ul>` tags. The items in the list are nested inside these tags and each item is wrapped in list item tag pairs (`<li>` and `</li>`). The following HTML code creates an ordered list:

```
<ol>
    <li>Savannah</li>
    <li>Bush</li>
    <li>African</li>
</ol>
```

The ordered list automatically generates a number or letter for each list item. By default, numbers are used. The HTML code above creates the following ordered list:

1. Savannah
2. Bush
3. African

The unordered list tag generates a bulleted list. The following code specifies an unordered list with two items:

```
<ul>
    <li>Savannah</li>
    <li>Bush</li>
</ul>
```

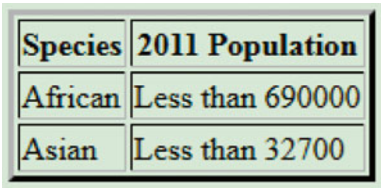
The HTML code above creates the following bulleted list with one bullet for each ‘li’ tag:

- Savannah
- Bush

Indentation is not required in HTML, but indented list items in the code are easier to read.

20.1.4 HTML Tables

HTML tables are wrapped in `<table>` and `</table>` tags and the contents are inserted in row major order. In other words, the first row is specified left to right, followed by the second, and so forth. Each row is wrapped in `<tr>` and `</tr>` tags.



Species	2011 Population
African	Less than 690000
Asian	Less than 32700

Figure 20.3 The table created by the HTML table code example in 'C:\gispy\data\ch20\htmlExamplePages\elephant3.html'.

The cells in each row are again individually wrapped in `<th>` and `</th>` or `<td>` and `</td>` tags ('th' stands for table heading and 'td' stands for table data). The following code sets table border thickness to 3 and the three `tr` tag pairs create three rows, while the pairs of nested `th` or `td` tags create two columns within each row (see the resulting table in Figure 20.3):

```
<table border="3">
    <tr>
        <th>Species</th>
        <th>2011 Population</th>
    </tr>
    <tr>
        <td>African</td>
        <td>Less than 690000</td>
    </tr>
    <tr>
        <td>Asian</td>
        <td>Less than 32700</td>
    </tr>
</table>
```

20.1.5 *Writing HTML with Python*

HTML can be used to present GIS analysis results. Since HTML is simply a text file containing HTML tags, you can use Python file handling to write HTML. There are specialized packages for writing HTML which are not covered in this book; Instead, the upcoming examples show how to do basic HTML file writing with standard Python file handling, familiar from Chapter 19. The Python script in Example 20.2 hard-codes a string variable `mystr` with HTML code, opens a file for writing, and writes the string to file. The triple quotes are wrapped around the string literal value to preserve the line breaks in the HTML code.

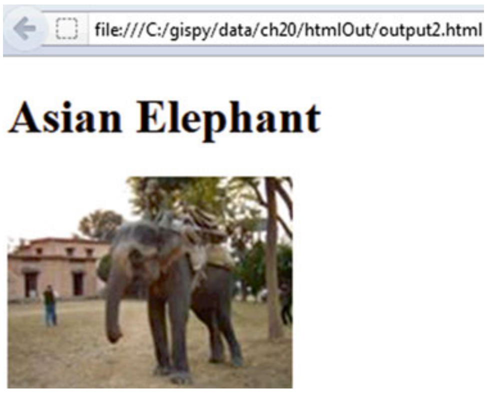


Figure 20.4 HTML page generated by the Python code in Examples 20.2 and 20.3.

Example 20.2

```
# writeSimpleHTML.py
mystr = '''<!DOCTYPE html>
<html>
    <body>
        <h1>Asian Elephant</h1>
        <img src="../data/ch20/pics/lakshmi.jpg" alt="elephant">
    </body>
</html>
'''

htmlFile = 'C:/gispy/scratch/output.html'
outf = open(htmlFile, 'w')
outf.write(mystr)
outf.close()
print '{0} created.'.format(htmlfile)
```

As the HTML file becomes more complex, breaking the HTML code into three parts (beginning, middle, and end) can make it easier to manage the string contents. Example 20.3 breaks the HTML code from Example 20.2 into three parts and writes each part to file. Example 20.3 also uses string formatting to dynamically generate the HTML content based on user input (the resulting Web page is shown in Figure 20.4).

Example 20.3

```
# writeSimpleHTML2.py
# Purpose: Write HTML page in 3 parts.
# Usage: workspace title image_path
# Example input:
# C:/gispy/scratch "Asian Elephant" ../data/ch20/pics/lakshmi.jpg
```

```
import os, sys
workspace = sys.argv[1]
title = sys.argv[2]
image = sys.argv[3]

beginning = '''<!DOCTYPE html>
<html>
    <body>'''

middle = '''
             <h1>{0}</h1>
             <img src='{1}' >\n'''.format(title, image)

end = '''     </body>
</html>
'''

htmlfile = workspace + '/output2.html'
with open(htmlfile,'w') as outf:
    outf.write(beginning)
    outf.write(middle)
    outf.write(end)
print '{0} created.'.format(htmlfile)
```

Compound structures can be built separately too. The `python2htmlList` function in Example 20.4 converts a Python list to an HTML list. First each item in the Python list is wrapped in list item tags. Then the items are joined into a single string. Finally, the list tag (`ol` or `ul`) is wrapped around the string.

Example 20.4

```
# excerpt from printHTMLList.py
def python2htmlList(myList, listType):
    '''Convert a Python list to HTML list.
    For example, convert [rast1,rast2] to:
    <ul>
       <li>rast1</li>
       <li>rast2</li>
    </ul>
    '''
    # Wrap items in item tags.
    htmlItems = ['<li>' + str(item) + '</li>' for item in myList]

    # Join the item list into a string with a line break
    # after each item.
    itemsString = '''\n    '''.join(htmlItems)
    # Wrap the string of items in the list tag.
    htmlList = '''
    <{0}>
        {1}
    </{0}>
    '''.format(listType, itemsString)
    return htmlList
```

The following code creates a Python list, calls the function, and prints the resulting unordered HTML list:

```
>>> rasts = [u'elev', u'landcov', u'soilsid', u'getty_cover']
>>> htmlList = python2htmlList(rasts, 'ul')
>>> print htmlList
<ul>
    <li>elev</li>
    <li>landcov</li>
    <li>soilsid</li>
    <li>getty_cover</li>
</ul>
```

By calling this function on Python lists, you can generate HTML lists to write in HTML files. In an upcoming chapter, you'll also learn how to generate screen captures of maps with Python, enabling you to automatically generate HTML output reports to show GIS analysis results, with graphical elements (using the 'img' tag).

20.1.6 Parsing HTML with BeautifulSoup

By reading and parsing markup language content with Python, you can access linked or embedded GIS data. The previous section showed that HTML pages can be generated using Python `file` objects in write `'w'` (write) mode. HTML can also be read with Python `file` objects in `'r'` (read) mode; However, the challenge comes in deciphering the HTML, i.e., parsing the content. To simplify this process, it's worth learning a few things about a module that supports markup language parsing. Python does have a built-in module for HTML parsing (named `HTMLParser`), but the non-built-in module named `BeautifulSoup`, in reference to a Lewis Carroll verse, is easier to use for high-level tasks such as finding all the links in a page.

The `BeautifulSoup` module has methods and objects for reading tags and their attributes and content. Online documentation explains how to download and install the latest version of `BeautifulSoup`, but for consistency, a stand-alone version which consists of just one Python file named 'BeautifulSoup.py' is included with the sample scripts for this chapter. Use this module while you're learning the basics, so that you don't get caught up in the installation procedures. Code may need minor changes to use the latest installable versions.

The given module doesn't require any special installation to be imported, but since it is not a built-in Python module, you have to make sure Python looks in the right directory. Handle this just as you handle importing user-defined modules. To import `BeautifulSoup`, first import the `sys` module and append the path for `BeautifulSoup` to the path list, then use a separate import statement to import `BeautifulSoup`:

```
import sys
sys.path.append('C:/gispy/sample_scripts/ch20')
import BeautifulSoup
```

Once you have imported it, you can make use of its methods and objects. To use the module, the first step is to create a `soup` object. We'll start by using a hard-coded HTML string to demonstrate how it works. The basic approach to using the module is to put the HTML content in a Python string variable (either by hard-coding or by reading it from a file), create a `soup` object, then call the soup methods on the `soup` object. The following statement hard-codes `mystr` with some HTML code:

```
>>> mystr = '<!DOCTYPE html><html><body><h1>Asian Elephant</h1><img
src="lakshmi.jpg" alt="elephant"></body></html>'
```

To create a `soup` object, call the `BeautifulSoup` class in the `BeautifulSoup` module with dot notation (the module and the class do have the same name) and pass in HTML content as an argument. The following code shows this use of dot notation with the HTML string variable passed in as content:

```
>>> soup = BeautifulSoup.BeautifulSoup(mystr)
```

The `soup` object has a function named `prettify`. Though HTML doesn't require each tag to be on a separate line or nested tags to be indented, this kind of spacing certainly makes the HTML easier to read. The `prettify` function, as the name implies, makes the HTML code pretty. Use dot notation on the `soup` object to call this object method and it returns a prettified HTML string:

```
>>> h = soup.prettify()
>>> print h
<!DOCTYPE html>
<html>
      <body>
        <h1>
            Asian Elephant
        </h1>
        <img src="lakshmi.jpg" alt="elephant" />
      </body>
</html>
```

The input HTML string was difficult to decipher when it was written on a single line. Now that it has been prettified, you can clearly see that this HTML sets a title and embeds an image, like the HTML page ('output.html') generated in Example 20.2.

The `BeautifulSoup` module makes it easy to find HTML elements. The `find` method finds the first occurrence of a tag:

```
>>> t = soup.find('h1')
>>> t
<h1>Asian Elephant</h1>
```

The `find` method returns a `BeautifulSoup Tag` object which has properties such as `name`, `attrs`, and `contents`. The `name` property contains the name of the tag, the `attrs` property contains a list of tag attributes, and the `contents` property contains a list of tag contents:

```
>>> type(t)
<class 'BeautifulSoup.Tag'>
>>> t.name
u'h1'
>>> t.attrs
[]
>>> t.contents
[u'Asian Elephant']
```

This `h1` tag has no attributes, so the `attrs` property is an empty list. The tag contents property is a list of items, even if there is only one content item, as in this example. List indexing can be used to access individual content items.

To access the first item in the list, use index zero, as in the following example:

```
>>> t.contents[0]
u'Asian Elephant'
```

The `img` tag has two attributes. The `Tag` object stores the attributes as a list of (attribute name, attribute value) tuples:

```
>>> t2 = soup.find('img')
>>> t2
<img src="lakshmi.jpg" alt="elephant" />
>>> t2.attrs
[(u'src', u'lakshmi.jpg'), (u'alt', u'elephant')]
```

The src attribute specifies the image file and the alt attribute provides a text alternative to be displayed in case there is trouble loading the image. You can access the value of an attribute by using the name, as you use a key in a dictionary:

```
>>> t2['src']
u'lakshmi.jpg'
>>> t2['alt']
u'elephant'
```

You can also loop through the attribute list in a way that's reminiscent to looping through a dictionary getting both key and value for each entry. For the attribute list, you can use two comma separated variables between the for and in keywords to get both the names and the values of the attributes:

```
>>> for name, value in t2.attrs:
...     print 'Name: ' + name + ' Value: ' + value
...
Name: src Value: lakshmi.jpg
Name: alt Value: elephant
```

Another soup object method named findAll returns a list of Tag objects for tags with the specified name. The following code finds all the list item (li) tags in htmlList:

```
>>> htmlList = '\n <ul>\n <li>elev</li>\n <li>landcov</li>\n
<li>soilsid</li>\n <li>getty_cover</li>\n </ul>\n'
>>> soup2 = BeautifulSoup.BeautifulSoup(htmlList)
>>> tags = soup2.findAll('li')
>>> tags
[<li>elev</li>, <li>landcov</li>, <li>soilsid</li>, <li>getty_cover</li>]
```

Loop through this list to retrieve the individual items. Access the first item in the tag contents lists by using a zero index:

```
>>> for t in tags:
...     print t.contents[0]
...
elev
landcov
soilsid
getty_cover
```

So far, the examples have used hard-coded HTML strings to create the soup object. In practice, useful HTML content will usually come from HTML files (instead of hard-coded strings). A soup object can also be created from a text file object open for reading. Example 20.5 opens an HTML file in read mode and creates a `soup` object from `infile`, the file object. Then it parses the HTML content using several of the soup navigation methods. The `FOR`-loop prints the two hypertext links found in the ‘elephant2.html’:

```
>>> Link: elephant1.html
Link: https://www.google.com/
```

Example 20.5

```
# getLinks.py
# Purpose: Read and print the links in an html file.
import sys
basedir = 'C:/gispy/'
mPath = 'data/ch20/htmlExamplePages/elephant2.html'
sys.path.append(basedir + 'sample_scripts/ch20')
import BeautifulSoup
# Read the HTML file contents.
with open(basedir + mPath, 'r') as infile:
    # Create a soup object and find all the hyperlinks.
    soup = BeautifulSoup.BeautifulSoup(infile)
    linkTags = soup.findAll('a')
    # Print each hyperlink reference.
    for linkTag in linkTags:
        print 'Link: {0}'.format(linkTag['href'])
```

20.2 Fetching and Uncompressing Data

Python can also enable you to *fetch* web content (retrieve it from a Web site), so that you can automatically harvest online GIS content that is made available through Web site links.

20.2.1 Fetching HTML

You can use the built-in module named `urllib2` to *fetch* (i.e., get the contents of) online html pages, by importing the module and calling the `urlopen` function. This function takes a URL (a web address) as input and returns a `file` object open for reading (like calling the built-in `open` function in `'r'` mode). You can use the standard Python text file reading functionality on the response. The following code fetches HTML from Google's Web site, calls `read` (which returns the entire contents of the HTML file as a string), and prints the number of characters in the HTML (the length of the string):

```
#fetchHTML.py
import urllib2
url = 'http://www.google.com'
response = urllib2.urlopen(url)
contents = response.read()
response.close()
print len(contents)
```

Once you fetch the page and read the contents into a string variable, you can parse the content string using the `BeautifulSoup` module as in the previous examples:

```
soup = BeautifulSoup.BeautifulSoup(contents)
>>> pics = soup.findAll('img')
```

When you run this code on the Google page, you may see different results from day to day, since the content is dynamic. The following output lists one image with seven image attributes defined:

```
>>> pics
[<img alt="Google" height="95" src="/intl/en_ALL/images/srpr/
logo1w.png" width="275" id="hplogo" onload="window.lol&&lol()"
style="padding:28px 0 14px" />]
```

20.2.2 Fetching Compressed Data

Online GIS data is often stored in *compressed files*, such as Zip or KMZ files that archive multiple files within a single file. You can also fetch these sorts of files, but they need to be handled differently. These are *binary* (non-text) file formats. The '.shp' extension file from a shapefile dataset is another example of a binary file. Try opening a '.shp' file in Notepad to view the indecipherable contents. Binary files consist mostly of unprintable characters and are not readable in a standard text editor.

The code for fetching and reading compressed files is the same as the code for fetching and reading HTML files. However, for binary compressed files, a binary `file` object is returned from the `urlopen` function, so the output from the `read` command is a binary string. Instead of using the `BeautifulSoup` module to parse the contents, you need to save the contents to your computer by writing them to a file. You can do so by opening a file for output in `wb` (write binary) mode and calling the `write` command. In Example 20.6, the binary `file` object is fetched and the contents of the response are read, as usual. The remainder of the code differs from HTML handling. A new file, `outFile`, is opened for writing in binary mode and the binary contents are written to the file, a new zip file is written in the output directory. The input for the script was a local Zip file URL, one starting with `file:///`, followed by the full path file name. The same function can be used for online URLs such as those starting with `http://` or `ftp://`. Additionally, the same approach works for KMZ files.

Example 20.6

```
# fetchZip.py
# Purpose: Fetch a zip file and place it in an output directory.
import os, urllib2

def fetchZip(url, outputDir):
    '''Fetch binary web content located at 'url'
    and write it in the output directory'''
    response = urllib2.urlopen(url)
    binContents = response.read()
    response.close()

    # Save zip file to output dir (write it in 'wb' mode).
    outFileName = outputDir + os.path.basename(url)
    with open(outFileName,'wb') as outf:
        outf.write(binContents)
outputDir = 'C:/gispy/scratch/'
theURL = 'file:///C:/gispy/data/ch20/getty.zip'
fetchZip( theURL, outputDir )
print'{0}{1}created.'.format(outputDir,os.path.basename(theURL))
```

20.2.3 Expanding Compressed Data

Once the Zip or KMZ file archive has been fetched and saved locally, it can be decompressed using Python. The built-in `zipfile` module can handle extracting the files from a Zip or KMZ archive. You work with a `Zipfile` object which has `namelist` and `extractall` methods. Example 20.7 shows how to do this for a Zip file. The `unzipArchive` function takes two arguments, the name of a Zip or KMZ file and a destination for the output. It works like a standard file extraction program, extracting files to the given location. It does so by first, creating a `Zipfile` object `zipObj`. Then it extracts the contents of each archived file to the destination directory (`dest`). Next it gets a lists of the files in the archive using the `namelist` method and loops through the list of archived files to report that they have been extracted. The script's printed output appears as follows:

```
Unzip C:/gispy/data/ch20/park.zip to C:/gispy/scratch/park/
Extract file: park.prj ...
Extract file: park.sbn ...
Extract file: park.sbx ...
```
(… and so forth)

Example 20.7

```
# extractFiles.py
# Purpose: Extract files from an archive;
#    Place the files into an output directory.
# Usage: No script arguments

import os, zipfile

def unzipArchive(archiveName, dest):
    '''Extract files from an archive
    and save them in the destination directory'''
    print 'Unzip {0} to {1}'.format( archiveName, dest )
    # Get a Zipfile object.
    with zipfile.ZipFile(archiveName, 'r') as zipObj:
        zipObj.extractall(dest)
        # Report the list of files extracted from the archive.
        archiveList = zipObj.namelist()
        for fileName in archiveList:
            print ' Extract file: {0} ...'.format(fileName)
    print 'Extraction complete.'

archive = 'park.zip'
baseDir = 'C:/gispy/'
archiveFullName = baseDir + 'data/ch20/' + archive
destination = baseDir + 'scratch/' +     \
              os.path.splitext(archive)[0] + '/'
if not os.path.exists(destination):
    os.makedirs(destination)
unzipArchive(archiveFullName, destination)
```

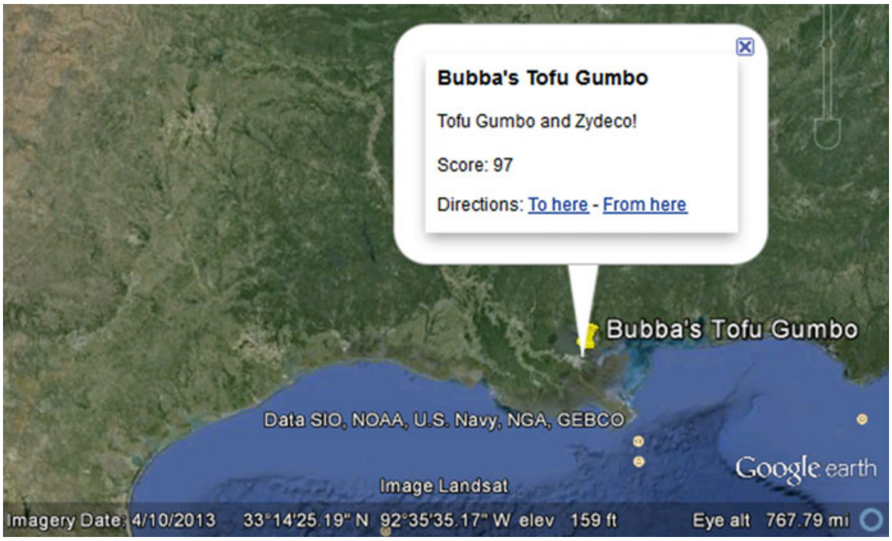
To collect online data, you can combine fetching and parsing HTML with fetching and unzipping compressed files. Many government and agency Web sites link to compressed data that can be downloaded by a script. For example, the government of North Carolina hosts an online data repository for Wake County (search online for 'wakegov data download'). The Data Download page links to zipfiles containing transportation, census, and other local GIS datasets that are updated periodically. The following workflow would be used to harvest these datasets:

```
FETCH the data download page.
CREATE a soup object.
GET a list of the links in the data download page.
FOR each link
    IF the link references a Zip file THEN
        CALL fetchZip to fetch the Zip file, saving it locally.
        CALL unzipArchive to extract the compressed files.
    ENDIF
ENDFOR
```

The sample script ‘wakeStreets.py’ provides a partial-Python implementation; A few lines of code are left as an exercise.

20.3 Working with KML

Now that you know something about HTML and writing/reading it with Python, you’ll also be able to work with other markup languages. Keyhole Markup Language (KML) files are of particular interest for GIS work, since this is a markup file format for geographic information. Like HTML, KML uses tags wrapped around content to encode data in a text file. KML files contain tags relating to places with geographic positions. They are designed to be viewed in Google Earth (Figure 20.5) and other GIS. ArcGIS can import these datasets into shapefiles for geographic analysis, using the ArcGIS KML to Layer (Conversion) tool, but this tool does not necessarily


Bubba's Tofu Gumbo

Tofu Gumbo and Zydeco!

Score: 97

Directions: To here - From here

Bubba's Tofu Gumbo
Data SIO, NOAA, U.S. Navy, NGA, GEBCO
Image Landsat
Google earth
Imagery Date: 4/10/2013 33°14'25.19" N 92°35'35.17" W elev 159 ft Eye alt 767.79 mi

Figure 20.5 A KML placemark balloon from ‘restaurants.kml’ viewed in Google Earth.

import all of the information encoded in the file. Geographic features are imported, but important attributes may not be imported. Since KML is a markup language, Python scripting with `BeautifulSoup` can be used to solve this problem and parse the data. Then ArcGIS cursors can be used to build tables from the parsed data.

20.3.1 The Structure of KML

To use Python for parsing KML, you'll need some understanding of the structure of a KML file. The most common way to specify a geographic feature is to use a `Placemark` tag. Each placemark tag contains a `Geometry` object, such as a point, line, or polygon (specified with `Point`, `LineString`, and `Polygon` tags). The location of an object is further nested within `coordinate` tags. Placemark properties such as a name and a description can also be specified.

Example 20.8 shows a KML file containing two points. The first line indicates the file type. XML, which stands for Extensible Markup Language, contains tags that define the structure of a document (such as `<chapter>` and `<section>` tags). KML is a special type of XML file for encoding geospatial objects. The KML opening and closing tags are wrapped around all of the KML content. The `Document` tag allows you to put more than one placemark in a single file. The file in Example 20.8 only contains two placemarks, one for each of the two points. The tags inside the placemark specify a name and description for each point. The contents of these tags appear in the balloon attached to the placemark when it is selected in Google Earth (see Figure 20.5). The `Point` tags are wrapped around a pair of `coordinates` tags with comma separated longitude, latitude, and altitude values for the placemark.

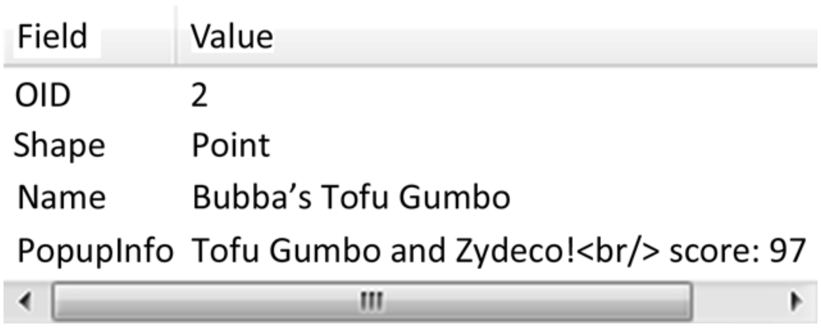
Example 20.8: KML code from the 'C:\gispy\data\ch20\restaurants.kml' file.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
 <Document>
   <Placemark>
     <name>Bubba's Tofu Gumbo</name>
         <description>Tofu Gumbo and Zydeco!<br />
                    Score: 97</description>
     <Point>
       <coordinates>-90,30,0</coordinates>
     </Point>
   </Placemark>
   <Placemark>
     <name>Joe Bob's Good Cookin'</name>
     <description>The best tree top grits n' greens restaurant
                south of the Mason-Dixon line.<br />
                Score: 94</description>
     <Point>
       <coordinates>-78,35,0</coordinates>
```

```
    </Point>
  </Placemark>
 </Document>
</kml>
```

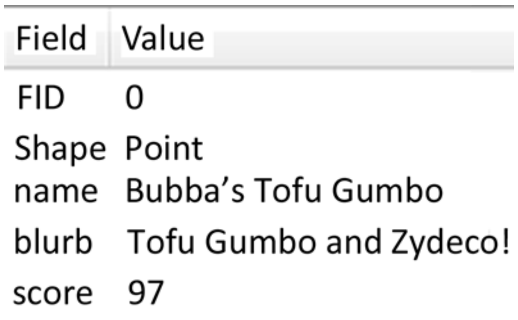
20.3.2 Parsing KML

The ArcGIS KML to Layer (Conversion) tool may not be sufficient for bringing KML data into ArcGIS for analysis. A common problem is that more than one piece of information can be encoded within the description. For example, the descriptions in 'restaurants.html', contain a blurb and an inspection score. The KML to Layer tool does not create a field for each of these pieces of information, so analysis can't be performed on description content. Figure 20.6 shows the field values for one of the restaurants in a file created by converting with this tool. Average scores can't be calculated because the descriptive blurb and the score, which are both within the description tag, are both imported together into the `PopupInfo` field. Whereas, we would prefer to obtain the blurb and score information in separate fields as shown in Figure 20.7. The `BeautifulSoup` module can be used to extract this information.



Field	Value
OID	2
Shape	Point
Name	Bubba's Tofu Gumbo
PopupInfo	Tofu Gumbo and Zydeco!<br/> score: 97

Figure 20.6 ArcGIS conversion results. The `PopupInfo` field contains all of the `description` tag contents.



Field	Value
FID	0
Shape	Point
name	Bubba's Tofu Gumbo
blurb	Tofu Gumbo and Zydeco!
score	97

Figure 20.7 Script conversion results. The `blurb` and `score` fields contains the `description` tag contents.

Example 20.9

```
# parseKMLrestaurants.py
# Purpose: Print KML placemark names and descriptions.
import sys

baseDir = 'C:/gispy/'
sys.path.append(baseDir +'sample_scripts/ch20')
import BeautifulSoup

fileName = baseDir + 'data/ch20/restaurants.kml'

# Get the KML soup.
with open(fileName, 'r') as kmlCode:
    soup = BeautifulSoup.BeautifulSoup(kmlCode)

# Print the names and descriptions.
names = soup.findAll('name')
descriptions = soup.findAll('description')
for name, description in zip(names, descriptions):
    print name.contents[0]
    print '\t{0}'.format(description.contents)
```

Parsing KML with the `BeautifulSoup` module is just like parsing HTML with this module; Only the tags differ. Example 20.9 parses the ‘restaurants.kml’ file and prints the names and description of each placemark using the `name` and `description` tags. This example imports `BeautifulSoup`, then opens the KML file for reading. KML files are ASCII text, so the `kmlCode` text file object open for reading KML code, can be used to create the `soup` object. The Python built-in `zip` function is then used in this example to loop through both lists simultaneously and print each name and description. The output is as follows:

```
Joe Bob's Good Cookin'
    [u'Tofu Gumbo and Zydeco!', <br />, u'Score: 97']
Bubba's Tofu Gumbo
    [u"The best tree top grits n' greens restaurant south of the
       Mason-Dixon line.", <br />, u'Score: 94']
```

Remember that tag contents are always returned as a list, even when there is only one value. The `name` tag contents list simply contains a name. Example 20.9 prints the name by indexing the first item in the list. The description contains a list of three items, the descriptive blurb, a line break, and the restaurant's sanitation score:

```
>>> description.contents
[u'Tofu Gumbo and Zydeco!', <br />, u'Score: 97']
```

To enable GIS analysis of the sanitation scores, we can extract their numeric value from the description contents list. The score is the third item in the list, so it can be retrieved with index 2, but the numeric portion of this item needs to be separated from the string portion. The following lines of code use string methods and casting to retrieve the score as a float:

```
>>> scoreString = description.contents[2]
>>> scoreString
u'Score: 97'
>>> scoreList = scoreString.split(':')
>>> scoreList
['Score', ' 97']
>>> score = float(scoreList[1])
>>> score
97.0
```

20.3.3 Converting KML to Shapefile

Once you've determined how to parse the desired information from a KML file, you can use an insert cursor to convert the extracted values into a GIS shapefile format. A shapefile can be created from a KML file using the following steps:

```
CALL the Create Feature Class (Data Management) tool
SET the field names and types
FOR each field name
    CALL the Add Field (Data Management) tool
ENDFOR
CREATE a soup object from the KML file contents
GET tag lists from the soup (findAll)
CREATE an insert cursor
FOR each item in the tag lists
    GET the value for each field
    PUT field values in a list
    INSERT the new row into the shapefile
ENDFOR
```

Example 20.10 shows an excerpt from sample script 'restaurantKML2shp.py' that represents an implementation of the `FOR each item in tag lists` loop to handle the 'restaurants.kml' file. Other KML files with point features that have multiple pieces of information within the `description` tag can be created by making modifications to the portions of this script that specify the field and types and the loop contents.

Example 20.10

```
# Excerpt from restaurantKML2shp.py...
coordinates = soup.findAll('coordinates')
names = soup.findAll('name')
descriptions = soup.findAll('description')
# Populate the shapefile.
with arcpy.da.InsertCursor( fc, [ 'SHAPE@XY' ] + fieldNames ) as ic:
    for c,n,d in zip(coordinates, names, descriptions):
        # Get field values.
        [x, y, z] = c.contents[0].split(',')
        myPoint = arcpy.Point(x, y)
        name = n.contents[0]
        blurb = d.contents[0]
        scoreString = d.contents[2]
        scoreList = scoreString.split(':')
        score = float(scoreList[1])
        # Put row values in a list & insert the new row.
        newRow = [myPoint, name, blurb, score]
        ic.insertRow(newRow))
```

20.4 Discussion

This chapter briefly introduced two markup languages, illustrated writing and reading markup files with standard built-in Python `file` objects and functions, and showed how to use Python to fetch and consume Web content. The `BeautifulSoup` module for parsing markup languages was used to parse HTML and KML content. Finally, insert cursors were used to import parsed KML into a shapefile. One difficulty in parsing markup files is that there can be errors in the markup code. Also, since the content in markup files varies, writing code to step through with the debugger to watch soup `Tag` objects and their contents is a pragmatic approach for developing scripts with `BeautifulSoup`. Once you see the tag contents (such as the KML description tag list contents), you can refine your code to consume the information you need. This code may need to be highly customized to handle your data, which explains the difficulty in creating a general tool that handles any HTML or KML content. Though this chapter only used HTML and KML examples, these approaches can be adapted for other markup language files, such as XML files.

20.5 Key Terms

HTML	Wrapped tags
KML	Beginning and ending tags
HTML lists	Fetch
HTML tables	URL
Tags	Compressed file
Start tag and end tag	Binary file
Tag content	The `BeautifulSoup` module
Tag attributes	

20.6 Exercises

1. **HTMLvocab** For the HTML in the code sample below, identify the following components: (a) the start or single tag names, (b) the names of attributes used in each tag, (c) the values of the attribute, and (d) the tag contents (the data between the opening and closing tags).

```
<img src="tree.gif" alt="tree image" width="80" height="100" />
<a href="http://www.sierraclub.org">Sierra Club</a>
<br />
<ol type='a'>
<li>Sweet Gum</li>
<li>Maple</li>
<li>Oak</li>
</ol>
```

2. **tagMatch** Match the tag HTML or KML tag name with its purpose.

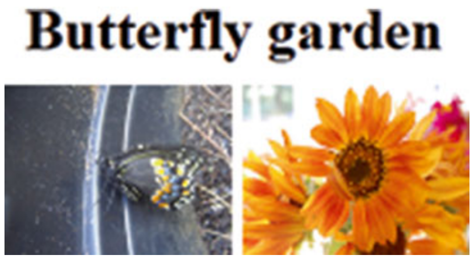
Markup tag name	Purpose
1. placemark	A. bold font setting
2. a	B. line break
3. br	C. hyperlink
4. tr	D. feature
5. ul	E. embedded picture
6. coordinates	F. bulleted list
7. h1	G. table row
8. linestring	H. geometric feature
9. img	I. header
10. b	J. latitude, longitude, and (optionally) altitude

3. Follow the instructions below to practice creating HTML files with Python code. Write one script for each part (a-c). Each script should create an output HTML file in the 'C:/gispy/scratch' directory.

   (a) **writeHTML1.py** Write a Python script to create an HTML page named 'images1.html' by hard-coding the HTML strings and the output directory path (‘C:/gispy/scratch/images1.html’). The HTML page should contain a header (‘Butterfly garden’) and two embedded images (Use a relative path to ‘butterfly.jpg’ and ‘flower.jpg’ in the ‘C:/gispy/data/ch20/pics’ directory). Display the images as 87 × 65 (w × h) pixel thumbnails.

Example input:
(No arguments required)

Example output: C:/gispy/scratch/images1.html created. When opened in a Web browser the output looks like this:



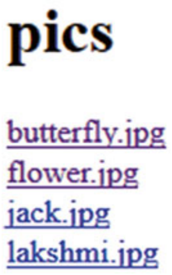
Butterfly garden

(b) **writeHTML2.py** Write a Python script to create an HTML page named ‘images2.html’ with dynamic content. The script should take two arguments, a directory containing images and an output directory where the HTML will be created. The page should have a header that is the name of the image directory and the page body should provide a link (with a relative path) to every JPEG image found in the given directory with each link on a separate line. Use the `os.path.relpath` function to find the relative path from the HTML to the images. (For example,
`relPath = os.path.relpath(imageDir, outDir))`

Example input:
C:/gispy/data/ch20/pics/ C:/gispy/scratch/

Example output:
```
>>> C:\gispy\scratch\images2.html created.
```

When opened in a Web browser the output looks like this:



pics

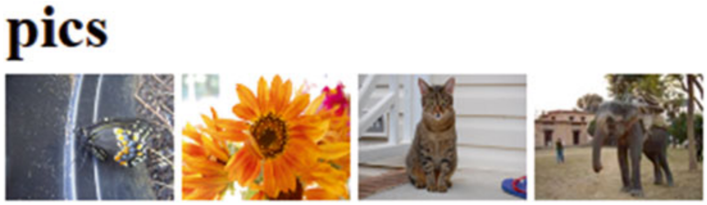
butterfly.jpg
flower.jpg
jack.jpg
lakshmi.jpg

(c) **writeHTML3.py** Write a Python script to create an HTML page named 'images3.html' with dynamic content. The script should take two arguments, a directory containing images and an output directory where the HTML will be created. The page should have a header which is the name of the image directory and the page body should embed every JPEG image found in the given directory as 87×65 (w×h) pixel thumbnails. Use relative paths to the images and use the `os.path.relpath` function to find the relative path from the HTML to the images. (For example,
`relPath = os.path.relpath(imageDir, outDir))`

Example input: C:/gispy/data/ch20/pics

Example output:
```
>>> C:/gispy/scratch/images3.html created.
```

When opened in a Web browser the output looks like this:

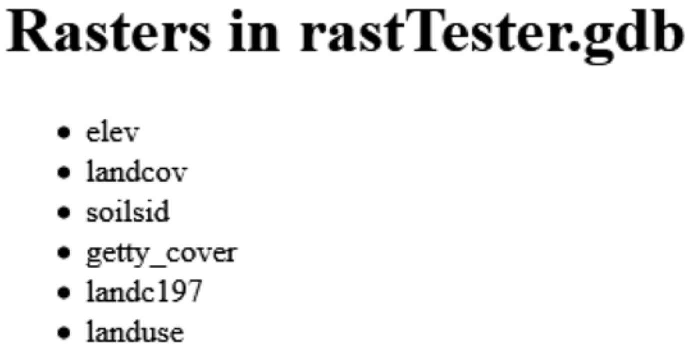


4. **writeRastersHTML.py** Write a Python script to create an HTML page named 'rasters.html' with dynamic content. The script should take two arguments, a workspace containing rasters and an output directory. The page header should say 'Rasters in' plus the name of the workspace and the page body should show a bulleted list of the rasters found in the given workspace. You can use the `python2htmlList` function from the 'printHTMLList.py' sample script to generate the HTML list.

Example input:
C:/gispy/data/ch20/rastTester.gdb C:/gispy/scratch

Example output:
```
>>> C:/gispy/scratch/rasters.html created.
```

When opened in a Web browser the output looks like this (only the first six bullets are shown):



Rasters in rastTester.gdb

- elev
- landcov
- soilsid
- getty_cover
- landc197
- landuse

5. **wakeStreets.py** The sample script 'wakeStreets.py' is missing code which needs to be filled in. Practice handling HTML and Zip files by replacing the five ### comments with one or more lines of code each. This script takes one argument, the URL for a download page for Wake County, NC data and an output directory. An example is given here; if the example input URL is no longer correct, search online for 'Data download Wake GIS'. When the code is complete, the script will fetch and unzip a 'Wake_Streets' Zip file. This script assumes that 'BeautifulSoup.py' resides in the same directory as this script.

   Example input:
   http://www.wakegov.com/gis/services/Pages/data.aspx C:/gispy/scratch/

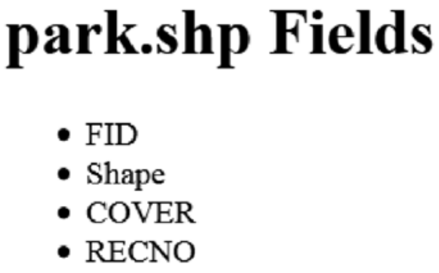
   Example output:
   ```
   >>> Unzip C:/gispy/scratch/Wake_Streets_20xx_xx.zip to
   C:/gispy/scratch/Wake_Streets_20xx_xx/
   Extract file: StreetsMetadata.doc ...
   Extract file: Wake_Streets_20xx_xx.dbf ...
   Extract file: Wake_Streets_20xx_xx.prj ...
   (and so forth)
   Extraction complete.
   ```
   xx will be replaced by the current date information in the form YY_MM.

6. **writeFieldsHTML.py** Write a script that writes an HTML file containing a report of the name of an input data file and a list of the fields in the file. The file name should be formatted as a header and the fields should be formatted as a bulleted list using html tags. The script should take two arguments, the full path file name of an input file and an output directory.

   Example input:
   C:/gispy/data/ch20/park.shp C:/gispy/scratch

   Example output:
   ```
   >>> C:/gispy/scratch/park.html created.
   ```

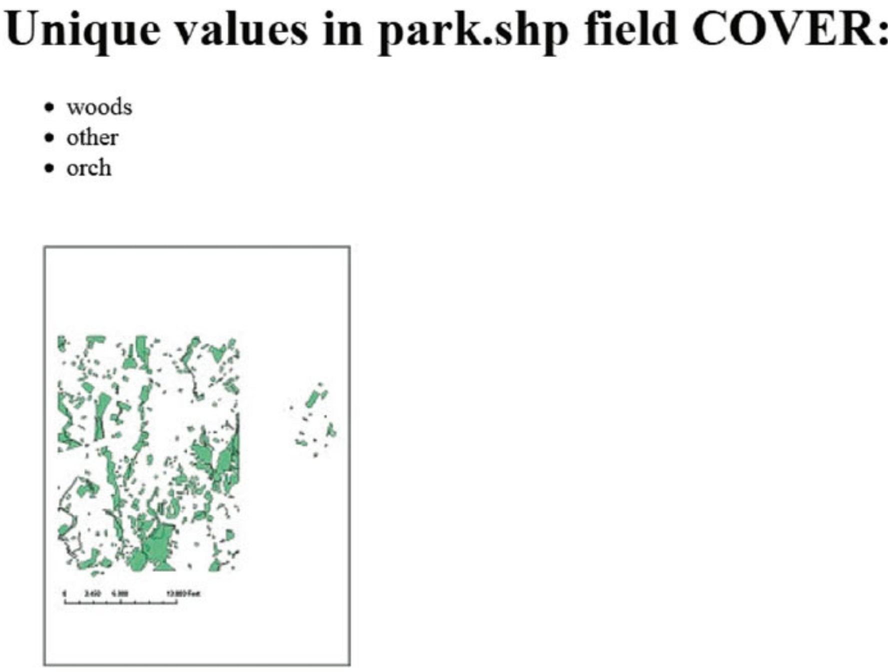
   When opened in a web browser the output file looks like this:



park.shp Fields

- FID
- Shape
- COVER
- RECNO

7. **writeUniqueValuesHTML.py** The sample script named 'writeUniqueValues HTML.py' is missing code which needs to be filled in. Practice handling HTML and lists by replacing the five ### comments with one or more lines of code each. File locations are hard-coded, so no arguments are needed. When the code is complete, the script will create an HTML page like the one shown here:



Unique values in park.shp field COVER:

- woods
- other
- orch

8. **parseHTMLimages.py** Write a script that takes an http URL as input and parses the HTML on the page using the `BeautifulSoup` module in the sample scripts directory. Assume 'BeautifulSoup.py' resides in the same directory as this script. The purpose of the script is to parse the given input for all of the image tags and print the number of images found and the source for each image. Assume that 'BeautifulSoup.py' resides in the same directory as this script.

   Example input: http://www.nytimes.com/

   Example output (only 3 of the 69 image sources are shown here):

```
>>> 69 images found.
image src: http://ad.doubleclick.net/ad/N5928.276948.NYTIMES/
B3868511.6;sz=184x90;
image src: http://ad.doubleclick.net/ad/N5928.276948.NYTIMES/
B3868511.7;sz=184x90;
pc=nyt227382A353135;
image src: http://i1.nyt.com/images/misc/nytlogo379x64.gif
...
```

9. **extractMany.py** Write a script that takes two arguments, an input directory and an output directory. The script should unzip every Zip and KMZ file in the input directory and write the results to the output directory. Use the `unzipArchive` function from sample script 'extractFiles.py' to perform the extraction for each file.

   Example input:
   C:/gispy/data/Ch20 C:/gispy/scratch

   Example output:
   ```
   >>> Unzip C:/gispy/data/Ch20/park.kmz to C:/gispy/scratch
   Extract file: doc.kml ...
   Unzip C:/gispy/data/Ch20/getty.zip to C:/gispy/scratch
   Extract file: COVER63p.prj ...
   Extract file: COVER63p.sbn ...
   Extract file: COVER63p.sbx ...
   ```
   (and so forth)

10. The KML file, 'Sample_7_Day_GPS_Data.kml', contains records of stops (points) and routes (linestrings) of parents taking kids to sports activities. Learn more about KML code by writing a script for each part (a and b).

    (a) **printStyles.py** When 'Sample_7_Day_GPS_Data.kml' is viewed in 'Google Earth', the stops and routes are displayed with tags and colored lines. Open the file in a text editor such as 'Notepad++' and you'll see the styles defined in the KML header. The 'normalPlacemark' is set to a 'red-stars.png'. The names 'YellowLineGreenPoly1', 'YellowLineGreenPoly2', and so forth are set to various colors. These styles control the symbology. Write a script to print the symbology for each placemark by using the `BeautifulSoup` module to find all the 'styleurl' tags and printing the *first content item* for each tag. Assume that 'BeautifulSoup.py' resides in the same directory as this script. The script should take one argument, the full path to a KML file.

    Example input:
    C:/gispy/data/Ch20/Sample_7_Day_GPS_Data.kml

    Example output:
    ```
    >>> #highlightPlacemark
    #yellowLineGreenPoly1
    #normalPlacemark
    #yellowLineGreenPoly2
    #normalPlacemark
    #yellowLineGreenPoly3
    ```
    … (and so forth)

    (b) **parseKMLDescription.py** In 'Sample_7_Day_GPS_Data.kml', the route descriptions contains driving time, distance traveled, maximum speed, and average speed. The description tag contents for the linestring placemark return these statistics in a list, as shown in the following example:

```
[<br />, u'Driving Time: 00h:01m:49s', <br />, u'Distance
Traveled: 0 mile', <br />, u'Maximum Speed: 1.25 mile/hour',
<br />, u'Average Speed: 0.05 mile/hour']
```

The sample script 'parseKMLDescription.py' finds all the placemarks and their description tag contents (`descriptionList`). Add code to the script that parses the `descriptionList` to retrieve the distance traveled in each description. Create a function named `getDistance`, that takes `descriptionList` as an argument and returns the numeric distance. Print each distance, sum the values, and finally print the total overall distance. Assume that 'BeautifulSoup.py' resides in the same directory as this script. The script should take one argument, the full path to a KML file.

Example input: C:/gispy/data/Ch20/Sample_7_Day_GPS_Data.kml

Example output:

```
>>> Current distance: 0.0
Current distance: 0.01
Current distance: 13.32
Current distance: 5.9
...
Current distance: 0.04
Total distance 183.36 miles.
```

11. **parseGPSLog.py** The sample script 'parseGPSLog.py' is missing code which needs to be filled in. Practice parsing KML code by replacing the nine ### comments with one or more lines of code each. The KML file, 'Sample_7_Day_GPS_Data.kml', contains records of stops and routes of parents taking kids to sports activities. When the code is complete, this script should create a point shapefile containing the KML points with the GPS stop names, dates and times. The script assumes that 'BeautifulSoup.py' resides in the same directory as this script. This script takes two arguments, the directory where 'Sample_7_Day_GPS_Data.kml' resides and an output directory.

Example input:
C:/gispy/data/ch20 C:/gispy/scratch

Example output:

```
>>> 35.782101 -78.677338 0
...
35.781952 -78.67717 0
Skipping this Linestring placemark.
--Adding entries to C:/gispy/scratch/Sample_7_Day_GPS_Data.shp--
--Output created: C:/gispy/scratch/Sample_7_Day_GPS_Data.shp--
```