Episode 9
Files

Reading and writing files can become rather complicated in Python if the files have no inherent structure. The most general methods of reading files will read all the contents in as strings, leaving it up to the programmer to extract the information from the strings or strings. To accomplish this requires more knowledge about string handling than our short introduction to Python will cover. However, much scientific data is in the form of columns of data, often separated by commas, leading to most such files being called CSV (comma-separated values) even if the separator is something else such as a space or semicolon.

Our sample file contains data about bird populations. The data were compiled by the US Geological Survey's Patuxent Wildlife Research Center in Maryland, in collaboration with the USGS, the Environment Canada/Canadian Wildlife Service, and Mexico's CONABIO agency. The Patuxent Wildlife Research Center's website https://www.pwrc.usgs.gov/ has much more information about their research programs.

Download the file `BBSVa.csv` from the Data Files resource area. In Spyder, create a new file and call it `BBS.py`. Start with

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
input_file="BBSVa.csv"
bbs=pd.read_csv(input_file)
```

Run this code to check whether the file is available. If it isn't, it may not be in the correct folder. It will need to reside in your "working directory" in order for the script to be able to find it. You can see the *path* of your working directory in Spyder in a textbox at the top. Look in the File Explorer of Spyder to see the list of files visible in that folder. If the data file you just downloaded is not already there, move it to that folder.

Now let's use Pandas to explore the data a little.

```
bbs.columns
```

We choose a bird called the yellow-billed cuckoo to study. From the colums we find one called `Species` so let's look at that.

```
bbs['Species']
```

This gives us the exact string `(Yellow-billed Cuckoo)` as it is specified in the file. We want to find the row number corresponding to the data. We find it from the Pandas output; it is 241 (recall that Python counts from zero). We can now use a Pandas function iloc to look at the data.

```
bbs.loc[241]
```

gives us information from that row. The first item is a string, however, and we need just the numbers for

plotting. Pandas lets us do that all in one line.

```
bbs.ix[241,1:].plot()
```

We really didn't want only to look at the yellow-billed cuckoo; it was just a choice we made to familiarize ourselves with the data. What we want to do is read all the data from any file of this type, determine which birds are declining, and write their information to another file for later examination. To do this we'll make use of some SciPy to fit a line to each set of observations; we will consider a negative slope to indicate a decline.

Let's start by fitting a line to the yellow-billed cuckoo data to make sure our basic logic is correct. You should always write small pieces of code and test them as you go; do not plunge into writing a lot of code and then attempting to test it afterward. This will make your debugging much easier and will increase your confidence that your code is correct.

Change your code in the Spyder Editor pane to

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.stats as stats
bbs=pd.read_csv("BBSVa.csv")
years_list=[]
for year in bbs.columns[1:]:
     years_list.append(float(year))
years=np.array(years_list)
```

This extracts the years as numbers so that we can use them as the independent variable. The entire process first uses Pandas to read the data into a DataFrame. The column names after the first one are the years, so we need to pull them out and convert them to numbers. We create an empty list to hold the year strings. We write a loop to process through the columns, ignoring the zeroth element (which is the word "Species"). We append each year to our list while simultaneously converting it to a floating-point number using the `float` *type-conversion* function. Finally we convert the list to a NumPy array. There are other ways to accomplish this task, of course—there is rarely only one way to write a program—but this accomplishes what we need.

Now we will write the trend-detecting code. We'll use `linregress` from SciPy's `stats` package (`scipy.stats`) to fit a least-squares regression line to the data. The documentation for linregress shows that we must pass it a one-dimensional array of an independent variable `x` and a corresponding array of `y` and it returns the slope, the intercept, the r-value, and the standard error

```
slope,intercept,rvalue,pvalue,stderr=stats.linregress(x,y)
```

We add this to our code for the single bird selected earlier. Add the lines:

```
data=bbs.ix[241,1:]
slope,intercept,rvalue,pvalue,stderr=stats.linregress(years,data)
line=slope*years+intercept
plt.figure()
plt.plot(line)
bbs.ix[241,1:].plot()
```

The results look reasonable so now we'll remove this code and replace it with code to go through all the data. Examining the data file, we notice that the last line is always the total over all species. The first item in each row of data is the species name, which we will store in a different array.

```
data=bbs.values[:-1,1:]
birds=bbs.values[:-1,0]
```

We are going to loop through the data, apply linregress to each row, and check whether the slope is negative. To test that our logic is working, we'll just print the name of the bird species to the console if the condition is `True`.

```
nrows,ncols=data.shape
for i in range(nrows):
    slope,intercept,rvalue,pvalue,stderr=stats.linregress(years,data[i,:])
    if slope<0:
         print birds[i]
```

That result looks reasonable, but we don't want to print to the console; we want to print to a file.

Like reading, writing arbitrary data to a file can be complicated, so we'll keep our example simple. It's important to understand that a file must be "attached" before your program can do anything with it. The file must be *opened* before we can do anything to it. When we use the Pandas function `read_csv` to ingest our data, the function takes care of opening the file for us. When we write, we will generally have to open the file ourselves. We must also declare our intention; read only, write only, or read and write. These are specified by

adding a second, optional argument to `open()` to specify the *access mode*; `'r'` for read only, `'w'` for write only, or `'r+'` for read or write. If we do specify, the default is `'r'` (read only). In this example we will only write to the file.

```
output_file="DecliningVaBirds.csv"
output=open(output_file,'w')
```

The `open` function returns a variable called a *file object* which contains information about the file. Henceforth we refer to the file with the variable and not by its name. Add the two lines to open the file before you read the input data or immediately afterward, so it will be ready when we enter the loop.

Since we open the file to attach it, you might guess we must close it to detach it. Add the line

```
output.close()
```

as the last line of your program. Now save the script file and run it. Check your folder for the presence of a new file, DecliningVaBirds.csv. Since we have written nothing to it, it should be empty.

If you open an existing file with `'w'` it will always destroy any existing contents. If you wish to append without overwriting, open the file with `'a'` instead. Both `'w'` and `'a'` will create a new file if it did not previously exist. Another option you may wish to use for reading is `'U'` for universal format. Windows, OSX, and Linux have different conventions for formatting lines and `'U'` can generally read any of them. For example, to open a file for reading you can use

```
infile=open(input_file,'rU')
```

Sometimes you should use `'rb'`, `'wb'`, or `'ab'`, rather than plain `'r'`,`'w'`, or `'a'`, especially with Windows computers. This is necessary if your file is in binary (not text) format but is sometimes needed for text files on systems with different newline conventions.

There are several ways to write to a file, including built-in functions in NumPy, Pandas, and other modules, but the basic method uses the `write` function. Attach `.write(something)` to the file object variable. The argument within the parentheses is the string you wish to write. Note that `write` handles only strings, so if you want to write numbers you will have to convert them with the `str()` function. Moreover, it will not write a newline unless you explicitly include it.

Add a line

```
output.write("Hello World!")
```

just before the `output.close()` statement. Now check the contents of your DecliningVaBirds.csv file.

We left out the newline; this really should have been

```
output.write("Hello World!\n")
```

The special character `\n` is used to mark the end of the line and go to a new line.

What we really want to write is the species name of each declining bird, the years observed, and the trend (slope) of the line. Remove the "Hello world" write line. Change your loop to

```
nrows,ncols=data.shape
```

```
for i in range(nrows):
    slope,intercept,rvalue,pvalue,stderr=stats.linregress(years,data[i,:])
    if slope<0:
         line=birds[i]+","+bbs.columns[1]+"-"+bbs.columns[-1]+","+str(slope)+"\n"
         output.write(line)
```

Now run your program and check the output file for the results.

There are many other ways to write files. A good introduction is at https://www.tutorialspoint.com/python/python_files_io.htm

Pay attention to some differences between Python 2.7 and Python 3.+ in input/output. The major differences are in input and output to the console; `print` behaves slightly differently between the two, and for user input from the console Python 2.7 uses `raw_input()` and Python 3.+ uses `input()`.