Python for R users in the Social Sciences: Lesson 1

Aaron Erlich

March 17, 2013

1 Abstract

Most Political Scientists (and social scientists more generally) come to Python, after having learned R. They realize R is slow and not good at interacting with the web or operating on strings. They then turn to Python, which is much better at these things. Learning R and then learning Python is not a typical computer language learning trajectory for computer scientists (or for anyone else). Therefore, the materials designed for those who want to learn Python are not optimized for the social science community.

This tutorial helps those with some familiarity with R better understand how Python works and get them up to get them up to speed on the basics, so they can later perform tasks they are likely going to need to know: web-scraping, string parsing, API querying, and natural language processing. The lessons assume a low level of computer science and programming theory.

While there has been a move to copy and paste code from trainings, I keep the code in the form it would like on the command prompt in these beginning lessons, to help you familiarize yourself with what the command prompt looks like when code is being iteratively tested. All the code is available from `https://github.com/aserlich/Content_Analysis`, if you would like to test it out. I would not recommend copying and pasting from these lectures.

2 Lists

Both R and Python are object oriented programming languages. However, in R the most common object type you come across is the dataframe, which is a 2 dimensional array of data. The dataframe object does not exist in Python standard modules—though it does in additional add-on libraries. Moreover, vectors and matrices also do not exist. Learning Python as an R user, therefore, requires us to **not** think in terms of dataframes and other one, two or n-dimensional matrices or arrays. How then does Python store data? Python has several main storage types, but the main one that R and Python share is the `list` data type. Unlike any other treatment of Python, we will start with the common ground between R and Python: the list. Lists are containers in which to put objects. The beauty of the list

is that you can add elements to it (they are mutable), and they can store different types of objects. This is true for both Python and R. But they operate in slightly different ways.

```
> Rlist1 <- list('hello world', 'hello Seattle')
> print(Rlist1)

[[1]]
[1] "hello world"

[[2]]
[1] "hello Seattle"
```

Now let's look at the Python code.

```
>>> PythonList1 = ['hello world', 'hello Seattle']
>>> print PythonList1
['hello world', 'hello Seattle']
```

We immediately begin to see the differences between R and Python. First, we note the assignment operator in Python is the equals sign and not the arrow `<-`, as it is in R. Next, we see in Python that when we use the brackets `[ ]`, Python immediately recognizes that that the object is of type list. In R, we have to use the function `list()`, otherwise, R as no idea what we are storing. In Python, the brackets always represent list elements.

2.1 Assigning, Slicing, Sub-setting

Once the data has been assigned to the object, we also notice some key differences. In order to slice the data and access a list element in R, we need to employ double brackets. in Python, the first element of the list is accessible with one set of brackets. **Also note that the list starts at 0**. Let's look at these differences in more detail.

```
> Rlist1 <- list("hello world", "hello Seattle")
> Rlist2 <- list(c("hello World", "hello Seattle"), "hello Tokyo") #we can have a vector of
> print(Rlist1[[1]])  #the first element in the R object

[1] "hello world"

> print(Rlist1[[1]][1])  #but we can't get to the letters

[1] "hello world"

> print(Rlist2[[1]][2])  #and if we have a list of string vectors

[1] "hello Seattle"
```

With R, there is is no easy way to access the strings via slicing. This, however, is very easy in Python. This is because once inside the list, Python encounters an object of type "string". Python knows that strings are made up of characters, so we can use our brackets to slice up that string. This cannot be done in R. Ease of string manipulation is one of the main reasons to use Python and we will return to it in Lesson 2.

```
>>> PythonList1 = ['hello world', 'hello Seattle']
>>> #we cannot have a vector of strings in Python.
>>> #It can be a list of lists
>>> PythonList2 = [['hello world', 'hello Seattle'], 'hello Tokyo']
>>> print PythonList1[0]
hello world
>>> #the 1st 4 chars of elem 0
>>> print PythonList1[0][0:4]
hell
>>> #the 1st list and the 1st item in that list
>>> print PythonList2[0][0]
hello world
>>> #1st char
>>> print PythonList2[0][0][1]
e
>>> #start at end
>>> print PythonList2[1][-1]
o
>>> #5th to end -> to end
>>> print PythonList2[1][-5:]
Tokyo
```

We mentioned that the list starts at 0, but we can also access the data from the other end. In Python, -1 refers to the last item in the list or the last character in a string.

2.2 Punctuation and For Loops

In Python, as you are noticing, there is much less punctuation. There are several reasons for this

1. In Python indentation matters! If you are inside a loop we have to be properly indented–either 4 spaces or a tab

2. Python abstract away a lot of things like iteration in for loops

Let's look at for loops with lists. In R and Python you often iterate over elements—we will stick with lists for now. Both R and Python know whether objected are *iterable*—that is, whether they can be iterated upon. Another way of putting the idea of iterability is that the

computer language knows which objects can be systematically gone through sequentially—element by element—starting from the first and ending in the last. In both languages, lists are objects that are *iterable*. However, we will see from the code blocks that programmers in the two languages often take different approaches.

```
> countryCodes <- c("RUS", "AFG", "GER")
> # we often iterate by indexing
> for (i in 1:length(countryCodes)) {
+     print(paste(countryCodes[i], "is country number", i))
+ }

[1] "RUS is country number 1"
[1] "AFG is country number 2"
[1] "GER is country number 3"

> # but we could do this -- which is more like Python
> for (i in countryCodes) {
+     print(i)
+ }

[1] "RUS"
[1] "AFG"
[1] "GER"
```

```
>>> countryCodes = ['RUS', 'AFG', 'GER']
>>> #semicolon starts the for loops
>>> for i in countryCodes:
...   #indentation matters and you need a blank non-indented line after
...   print i
...
RUS
AFG
GER
>>> #we can also make Python more like R
>>> for i,j in enumerate(countryCodes):
...   print j + " is country number " + str(i)
...
RUS is country number 0
AFG is country number 1
GER is country number 2
>>> #this will yield an error message
>>> for i,j in enumerate(countryCodes):
```

```
...    print j + " is country number " + i
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

As we can see Python requires many fewer keystrokes to undertake the same task. If we need to count the numbers we are iterating through in Python, we need to use the built in function `enumerate()`. Enumerate takes two inputs separated by a comma. The first input– in our case the letter `i` is the number of the element being iterated through and the then second input is the contents of the element itself.

Here we can also note the difference in operands will operate across different types of variables and Python knows when it sees a bunch of strings plus signs in between it means something different then when there are numbers. Hence, in Python a + sign in between two string elements, simply means we need to concatenate them. Since `i` from the enumerate command is a number I convert it to string with the `str()` command before I concatenate it. If I didn't convert it to a number, I could get an error message.

2.3 Adding and Removing

Adding and removing elements from a list in R and Python also occur in different ways. In R, we usually take a subset of our data or concatenate things on to a list.

```
> countryCodes <- c("RUS", "AFG", "GER")
> countryCodes <- c(countryCodes, "USA")  #bind an element to itself-safe
> print(countryCodes)

[1] "RUS" "AFG" "GER" "USA"

> countryCodes[[5]] <- "BRA"  #add an element in the fifth spot
> countryCodes[[7]] <- "GEO"  #what happened to six? it's NA
> print(countryCodes)

[1] "RUS" "AFG" "GER" "USA" "BRA" NA    "GEO"
```

```
>>> countryCodes = ['RUS', 'AFG', 'GER']
>>> #puts on the end
>>> countryCodes.append('USA')
>>> #put 'BRA' at the beginning
>>> countryCodes.insert(0,'BRA')
>>> #put 'GEO' in after the 4th slot and shift everything down
>>> countryCodes.insert(4, 'GEO')
```

```
>>> print countryCodes
['BRA', 'RUS', 'AFG', 'GER', 'GEO', 'USA']
>>> #what will Python do here? Will it create NA's like R?
>>> countryCodes.insert(15, 'YUG')
>>> print countryCodes
['BRA', 'RUS', 'AFG', 'GER', 'GEO', 'USA', 'YUG']
```

Here we see another way the Python language differs from R. The ". in Python following an object tells Python that a function will be called on that object. These functions are called methods. All objects have different methods that can be called upon them. You can see all the methods that can be called on lists here. `http://docs.python.org/2/tutorial/datastructures..html`. So here we are using the `append` and `insert` methods. These methods changes the object without the user assigning country codes. Beware, while this is very efficient, you can also make trouble. Consider whaty ou could do with the method `replace`.

In R and Python, we often want to subset or slice our data. In R, we generally do this with either the `subset()` function or through slicing. In Python, we can use our methods to subset or slicing as well. However, remember if we use our methods then the underlying object will change.

```
> countryCodes <- as.list(countryCodes)  #make my vector a list
> subset(countryCodes, countryCodes != "AFG") #remove AFG but doesn't change underlying obj

[[1]]
[1] "RUS"

[[2]]
[1] "GER"

[[3]]
[1] "USA"

[[4]]
[1] "BRA"

[[5]]
[1] "GEO"

> countryCodes[1:4]

[[1]]
[1] "RUS"
```

```
[[2]]
[1] "AFG"

[[3]]
[1] "GER"

[[4]]
[1] "USA"
```

```
>>> countryCodes = ['RUS', 'AFG', 'GER', 'USA', 'BRA', 'GER']
>>> countryCodes.sort()
>>> print countryCodes
['AFG', 'BRA', 'GER', 'GER', 'RUS', 'USA']
>>> countryCodes.remove('AFG')
>>> #returns last and removes from list
>>> countryCodes.pop()
'USA'
>>> #returns item in the second position and removes from list
>>> countryCodes.pop(1)
'GER'
>>> print countryCodes
['BRA', 'GER', 'RUS']
```

2.4 List Comprehensions

Besides all the other features of lists that Python has, it has something that has no equivalent in R: the list comprehension. At it's most basic level, a list comprehension creates a new list by using a syntax to iterate like a for loop, but only uses one line of code. Basically, the list comprehension turns the for loop inside out and starts with the operation you are going to do to each object. It then names the iterator and then and only then tells Python what the object is that is going to be iterated over. List comprehensions get much fancier, but this will start us out.

```
>>> countryCodes = ['RUS', 'AFG', 'GER']
>>> #the traditional for loop
>>> for country in countryCodes:
...   country.lower()
...
'rus'
'afg'
'ger'
```

```
>>> #list comprehension approaches
>>> [country.lower() for country in countryCodes]
['rus', 'afg', 'ger']
>>> [country.lower() for country in countryCodes if country.startswith('G')]
['ger']
```

Notice how efficient this code is. We can get a lot done in just a few lines. That's why folks like Python.