Python for R Users in the Social Sciences: Lesson 2

Aaron Erlich

March 17, 2013

1 Strings

Strings in R and Python look deceptively similar. They are very different, however. Let's consider the following R code.

```
> oneCountry <- "USA"
> oneCountry_Vector <- vector(mode = "character", 1)
> oneCountry_Vector[1] <- "USA"
> oneCountry == oneCountry_Vector

[1] TRUE

> length(oneCountry)

[1] 1

> oneCountry[1]

[1] "USA"

> oneCountry[2]

[1] NA
```

In the above code, we create what looks like a string. Then, we create a vector of length one of mode `character`. Why are they the same? Because when we assign "USA" to the object `oneCountry`, R encodes it as a vector of length one of mode `character`. In R, vectors have a mode character. Other modes are `numeric` and `logical`.

Now let's look at the Python code.

```
>>> oneCountry = "USA"
>>> type(oneCountry)
<type 'str'>
>>> oneCountry[0]
'U'
>>> len(oneCountry)
3
```

In the Python code, Python encodes the object `oneCountry` as type `str`. As the code above shows we can look at the type of any object in Python. This will come in handy later when were are dealing with objects which are defined by external libraries like NLTK. Remember the Python standard libraries do not have vectors or matrices. However, it does have strings. And strings can be operated on. The fact we that Python has a string data type, makes it much better than R for operating on strings, which is one of the main goals of automated content analysis.

We met Python methods with lists. There are also specific methods for strings. We can see the methods for string with the directory function `dir()`. I show a sample of these methods. Note, that in order to show them to you so that they fit on the line I use the libarary `pprint`, which stands for "pretty print." I encourage you to look at all of methods for strings in more detail, as you will inevitable use most of them.

```
>>> import pprint
>>> oneCountry = "USA"
>>> pprint.pprint(dir(oneCountry)[45:60])
['isdigit',
 'islower',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition']
```

In discussing lists we discovered they are *mutable*. That means that the methods we call on them will change the underlying list. Strings are *immutable*. This means that the methods that we call on them will not change the underlying string data. It will return the data to be stored in a different object, however. The fact that strings are *immutable* is important because we generally don't want to change the underlying raw data we read into Python; and Python prevents us from doing so.

```
>>> oneCountry = "USA"
>>> oneCountry.lower()
'usa'
>>> print(oneCountry)
USA
>>> oneCountry.split('S')
['U', 'A']
```

```
>>> print(oneCountry)
USA
```

R, one the other hand has string functions, which can be called on vectors of mode ”character”. When these R functions are called they generally return a list. They return a list because they will return a vector for each element of the vector of mode string which was passed to the function. Python also returns a list for many of the string methods for the same reasons, but the input is substantively different. Note: Hadley Wickham’s `stringr` package is a better way to deal with strings in R. However, Python is even better, so we won’t be discussing `stringr`

```
> strsplit("I don't wanna go to school, but I have to, but I don't want to",
+     "but")

[[1]]
[1] "I don't wanna go to school, " " I have to, "
[3] " I don't want to"

> twoSchoolStatements <- c("I don't wanna go to school, but I have to, but I don't want to",
+     "I really wanna go to school, but I am sick, but I really don't have to")
> strsplit(twoSchoolStatements, "but")

[[1]]
[1] "I don't wanna go to school, " " I have to, "
[3] " I don't want to"

[[2]]
[1] "I really wanna go to school, " " I am sick, "
[3] " I really don't have to"
```

In Python, we can simply apply the `split` method to the the string. And it will return the response.

```
>>> "I don't wanna go to school, but I have to, but I don't want to".split("but")
["I don't wanna go to school, ", ' I have to, ', " I don't want to"]
```

So, how would I do this on a list of strings in Python? There are two easy ways to do this. Let us force R to put strings into a list of vectors of length one to illustrate the comparison. If we put the string vectors into a list in R, we could then use the `lapply` function to apply a function to each object in the list. This is known as functional programming. Functional programming is much faster than iteration. Hence,if you have massive data—as is often the case in automated content analysis—it can be useful.

```
> twoSchoolsStatementsList <- list("I don't wanna go to school, but I have to, but I don't want to",
+     "I really wanna go to school, but I am sick, but I really don't have to")
> lapply(twoSchoolsStatementsList, strsplit, "but")
```

```
[[1]]
[[1]][[1]]
[1] "I don't wanna go to school, " " I have to, "
[3] " I don't want to"


[[2]]
[[2]][[1]]
[1] "I really wanna go to school, " " I am sick, "
[3] " I really don't have to"

> lapply(twoSchoolsStatementsList, function(x) strsplit(x, "but"))

[[1]]
[[1]][[1]]
[1] "I don't wanna go to school, " " I have to, "
[3] " I don't want to"


[[2]]
[[2]][[1]]
[1] "I really wanna go to school, " " I am sick, "
[3] " I really don't have to"
```

In Python, we can also use functional programming, but the grammar is slightly different. Consider the following example:

```
>>> import pprint
>>> twoSchoolStatements = [
... "I don't wanna go to school, but I have to, but I don't want to",
... "I really wanna go to school, but I am sick, but I really don't have to"
... ]
>>> pprint.pprint((lambda x: x.split("but"), twoSchoolStatements))
(<function <lambda> at 0x100568a28>,
 ["I don't wanna go to school, but I have to, but I don't want to",
  "I really wanna go to school, but I am sick, but I really don't have to"])
>>> def butSplit(x): return x.split("but")
...
>>> print(butSplit)
<function butSplit at 0x100568a28>

>>> mapWithDef = map(butSplit, twoSchoolStatements)
>>> mapWithLambda = map(lambda x: x.split("but"), twoSchoolStatements)
>>> pprint.pprint(mapWithDef)
[["I don't wanna go to school, ", ' I have to, ', " I don't want to"],
 ['I really wanna go to school, ', ' I am sick, ', " I really don't have to"]]
```

```
>>> pprint.pprint(mapWithLambda)
[["I don't wanna go to school, ", ' I have to, ', " I don't want to"],
 ['I really wanna go to school, ', ' I am sick, ', " I really don't have to"]]
```

So, what is going on here? Let's start at the end and work our way back. The `map` function is like the R `apply` functional family. It applies a function to each element of an iterable. But what function to apply to our list of strings? We have two options. We can either define a function using `def` or we can emulate the R code by using a `lambda` expression, which like the `function(x)` in the R code, is an anonymous function"; that is, it doesn't have a name. However, in Python we call each of these anonymous functions `lambda`. Lambdas have a couple of differences when we compare them with regular functions we define. Refer to Mark Lutz's book for a longer treatment.[1] Either way, we are applying a function to our list. with the `def` we have defined our function, whereas with `lambda` we have created an expression that will later define a function. All this is a bit technical, but you should know it is out there.

Of course, you could also still iterate. And remember in Python we have the list comprehension, which makes iterating syntax short and straightforward.

```
>>> import pprint
>>> twoSchoolStatements = [
... "I don't wanna go to school, but I have to, but I don't want to",
... "I really wanna go to school, but I am sick, but I really don't have to"
... ]
>>> splitListComp = [x.split("but") for x in twoSchoolStatements]
>>> pprint.pprint(splitListComp)
[["I don't wanna go to school, ", ' I have to, ', " I don't want to"],
 ['I really wanna go to school, ', ' I am sick, ', " I really don't have to"]]
```

2 Numbers

So if Python has a type `str` it must have some numeric types, right? Off course it does. In fact it has lots of number types ranging from the simple to the complex. That's a bad joke, but Python does support imaginary numbers. However. in the social science once things become numeric, we generally move over to R. If you are interested in doing statistics and numeric operations in algebra, the course recommends three packages: `NumPy`, `SciPy` and `Pandas`. However looking at numbers allows us to look at a couple of operators that are different in R and Python. Remember also that Python operators are universal, so they don't only work on numbers—if they make sense at all, they will also be applied to strings

The are a few differences because generally Python supports a few more word representations of operations such as ”and”, but also supports many of the same character representations of mathematical operations that R does. I demonstration one small difference between ”in” and ”not in”.

[1]Or see `http://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/`

```
> numbers <- c(1, 45.3, 3.5)
> numbers %in% c(1, 3)

[1]  TRUE FALSE FALSE

> !(numbers %in% c(1, 3))

[1] FALSE  TRUE  TRUE

> numbers[!(numbers %in% c(1, 3))]

[1] 45.3  3.5
```

Remember Python doesn't have vectors, so we have to test each individual element explicitly in the code. In the following code, we are testing to see if the number is either a 1 or a 3 or is not a 1 or a 3. I show both how to carry out the test with a regular for loop and a list comprehension. What does that last list comprehension do?

```
>>> numbers = [1,45.3, 3.5]
>>> for number in numbers:
...   print(number in [1,3])
...
True
False
False
>>> [number in [1,3] for number in numbers]
[True, False, False]
>>> [number not in [1,3] for number in numbers]
[False, True, True]
>>> [number for number in numbers if number in [1,3]]
[1]
```

Numbers are also mutable. We can also introduce the cool += operator here, which is Python shorthand for take the thing and add it to whatever is already there. Would minus work, do you think? And multiplication?

```
>>> oneNumber = 1
>>> oneNumber += 1
>>> oneNumber
2
>>> oneNumber += 15.5
>>> oneNumber
17.5

>>> SocialScience = "S"
>>> SocialScience += "ocial"
>>> SocialScience += " Science"
>>> SocialScience
'Social Science'
```

In R we would have to do the following, which is not as elegant.

```
> oneNumber <- 1
> oneNumber <- oneNumber + 1
> oneNumber <- oneNumber + 15.5
> oneNumber

[1] 17.5
```

Python also supports both the ampersand (&) and the word ”and” as well as the pipe (|) and the word ”or

```
>>> numbers = [1,45.3, 3.5]
>>> for number in numbers:
...   if number > 3 and number % .5 == 0:
...     print(str(number) + " is greater than 3 and has a modulo of 0 when divided by .5")
...   elif number > 45 or number/100 > .3:
...     print("Big number " + str(number))
...   else:
...     print("Puny number " + str(number))
...
Puny number 1
Big number 45.3
3.5 is greater than 3 and has a modulo of 0 when divided by .5
```

Notice, I snuck in an if statement in this last example as well. As always, the indentations matters. Like the for loop it's followed by a semicolon. The `elif` and `else` are options. But they are also both followed by a colon and follow the same indentation pattern as the `if`