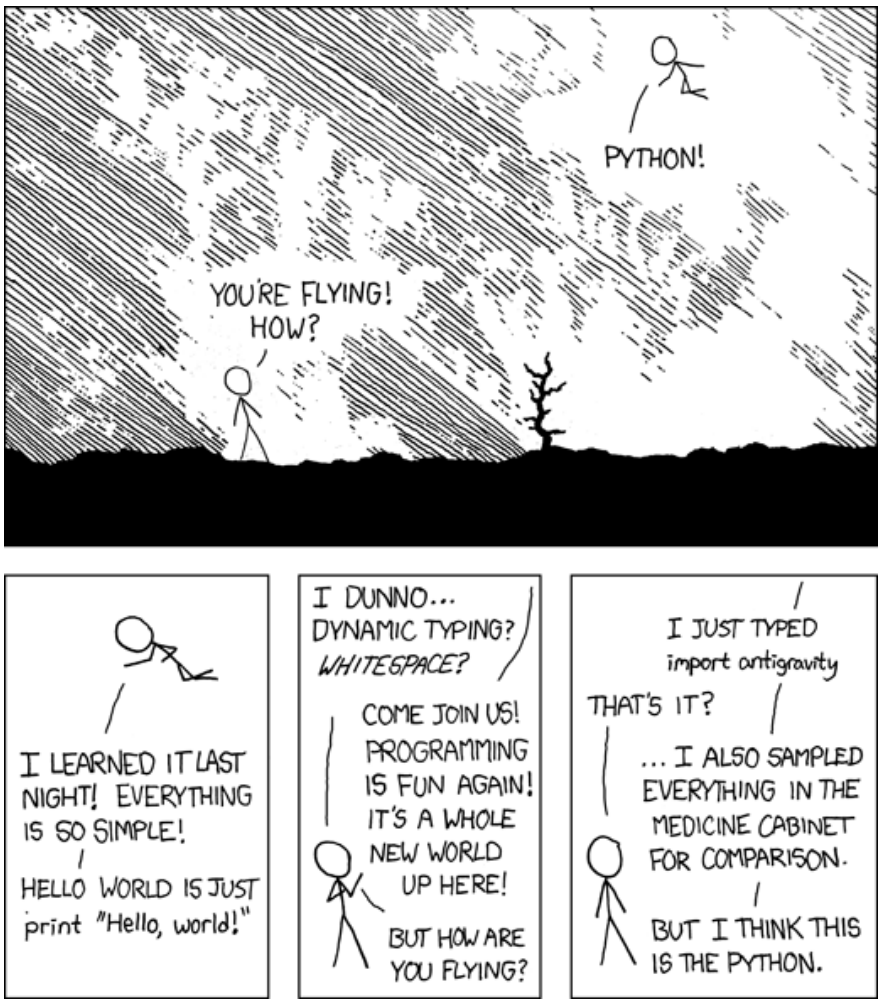
Python for Economists

Alex Bell

alexanderbell@fas.harvard.edu



http://xkcd.com/353

This version: October 2016.

If you have not already done so, download the files for the exercises here.

Contents

1 Introduction to Python

I've been a student of three college classes that taught Python from scratch, but I've never seen a way of teaching Python that I thought was appropriate for economists already familiar with scripting languages such as Stata. I also believe economists are seeking something different from programming languages like Python from what computer scientists look to do. It it not my intention to delve into scary computational estimation methods, rather, I believe the programming flexibility that Python affords opens doors to research projects that can't be reached with Stata or SAS alone. Whenever possible, I present material throughout the introduction in ways I believe are most useful when using Python to aid economic research. The two applications of Python I have found most useful to this end are for text processing and web scraping, as discussed in the second part of this tutorial. I hope you enjoy using Python as much as I do.

1.1 Getting Set-Up

Python is quite easy to download from its website, python.org. It runs on all operating systems, and comes with IDLE by default. You probably want to download the latest version of Python 2; Python 3 works a bit differently.

This tutorial was written for Python 2. Even if you're interested Python 3 it's sensible to do the tutorial in Python 2 then have a look at the differences. By far the most salient difference that beginner should know is that in Python 2, `print` is a statement whereas it is a function in Python 3. That means `print ''Hello World''` in Python 2 becomes `print(''Hello World'')` in Python 3.

1.2 Syntax and Basic Data Structures

Pythonese is surprisingly similar to English. In some ways, it's even simpler than Stata – it may feel good to ditch Stata's "&" and "|" for "and" and "or." You still need to use "==" to test for equality, so that Python knows you're not trying to make an assignment to a variable.

Unlike in Stata, indentation matters in Python. You need to indent code blocks, as you will see in

examples. Capitalization also matters. Anything on a line following a "#" is treated as a comment (the equivalent of "//" in Stata).

You can use any text editor to write a Python script. My favorite is IDLE, Python's Integrated DeveLopment Environment. IDLE will usually help you with syntax problems such as forgetting to indent. Unlike other text editors, IDLE also has the advantage of allowing you to run a script interactively with just a keystroke as you're writing it. The example code shown throughout the notes shows interactive uses of Python with IDLE.

Just as you can run Stata interactively or as do-files, you can run Python interactively or as scripts. Just as you can run Stata graphically or in the command line, you can run Python graphically (through IDLE) or in the command line (the executable is "python").

1.2.1 Variables: What Stata Calls Macros

In most programming languages, including Python, the term "variable" refers to what Stata calls a "macro." Just like Stata has local and global macros, Python has global and local variables. In practice, global variables are rarely used, so we will not discuss them here.

As with Stata macros, you can assign both numbers and strings to Python variables.

```
>>> myNumber = 10
>>> print myNumber
10
>>> myString = "Hello, World!"
>>> print myString
'Hello, World!'
>>> myString = 10 ## Python changes the type of the variable for you on the fly
>>> print myString
10
```

You can use either double or single quotation marks for strings, but the same string must be enclosed by one or the other.

Task 1: Assign two variables to be numbers, and use the plus symbol to produce the sum of those numbers. Now try subtraction and multiplication. What about division? What is 5/4? What about 5./4.? How about float(5)/float(4), or int(5.0)/int(4.0)? If you enter data without a decimal point, Python

generally treats that as an integer, and truncates when dividing.

Task 2: Assign "Hello" to one variable and "World!" to another. Concatenate (combine) the two string variables with the plus sign, just as you would add numbers. Doesn't look right to you? Add in some white space: *var1* + " " + *var2*.

Task 3: What about multiplying a string? What is '-'*50?

1.2.2 Lists

Lists are another common data type in Python. To define a list, simply separate its entries by commas and enclose the entry list in square brackets. In the example below, we see a few ways to add items to a list.

```
>>> myList = [1, 2, 3] # defines new list with items 1, 2, and 3
>>> myList.append(4)
>>> myList = myList + [5]
>>> myList += [6] # this is a shortcut
>>> myList # here is the new list; items appear in the order they were added
[1, 2, 3, 4, 5, 6]
```

In the example above, we saw the syntax myList.append(..). In Python, we use *objects*, such as lists, strings, or numbers. These objects have predefined *methods* that operate on them. The *list* object's append(..) method takes one parameter, the item to append.

Task 4: Define a list in which the items are the digits of your birthday.

Indexing into a list is simple if you remember that Python starts counting at 0.

```
>>> myList
[1, 2, 3, 4, 5, 6]
>>> myList[0] # first item in myList
1
>>> len(myList) # length of myList
6
>>> myList[6] ## this will create an error, shown below, with comments added
'Traceback (most recent call last):' # Python tells me about what was happening
  'File ''<pyshell 29>'', line 1, in <module>' # The probelmatic line (in this case, line 29
```

```
                                        # in the Python interpreter I had open)
    'myList[6]' # The problematic command
'IndexError: list index out of range' # a description of the problem
>>> myList[5] # oh -- that was what I meant!
6
```

Task 5: From the list you defined in the previous task, retrieve the first item. Use the len(..) function to find out how long the list is. Now, retrieve the last item.

Task 6: Lists can store any data structure as their items. Make a list in which the first item is the name of the month of your birthday (a string, so enclosed in quotation marks), the second item is the day of the month of your birthday (a number), and the last item is the year of your birthday (also a number).

Task 7: Lists can even contain lists! Ask your neighbor what his or her birthday is. Make a list in which the first item is the list you declared in the previous task, and the second item is the list for your neighbor's birthday.

1.2.3 Functions

Functions are the equivalent of programs in Stata. A function definition starts with *def*, then the function name followed by parentheses. Any parameters the function takes in should be named in the parentheses. A colon follows the parentheses, and the rest of the function declaration is indented an extra level.

```
>>> def printWord(word): # define a function called printWord that takes in parameter 'word'
        print "The word you gave me was "+word
>>> printWord("amazing") # what will this do?
'The word you gave me was amazing'
```

Task 8: Define and test a function "helloWorld()" that takes in no parameters, and just prints the string "Hello, World!" Note that IDLE will auto-indent the first line after the colon for you when you hit the enter key after typing the colon.

The word **return** has special meaning within a function.

```
>>> def addNums(num1, num2):
        return num1+num2
```

```
>>> result = addNums(1,10) # now, what is the value of the variable result?
>>> print result
11
```

Task 9: Define a function multNums that *returns* the product of two numbers. Test it by assigning result=multNums(2,3), then print result. What is multNums(2, result)?

Throughout the rest of the exercises, you can choose whether you'd like to define functions for specific tasks. Sometimes functions are nice if you think you'd like to do something repetitively.

1.2.4 Statements

Python and Stata both support if/else statements, for loops, and while loops. Table 1 presents a comparison.

Table 1: Syntax for Common Loops / Statements

Common Name	Stata	Python

for (each)	foreach item in 'myList' {
	  di 'item'
	}
	//or
	foreach var of varlist * {
	  sum 'var'
	}
		for item in myList:
		  print item

for (values)	forvalues num=1/100 {
	  di 'num'
	}
		for num in range(0,101):
		  print num

while	local i = 1
	while 'i' <=5 {
	  count if v1 == 'i'
	  local i = 'i' + 1
	}
		while len(myList)<10:
		  myList+=[myOldList.pop()]
		  i+=1

if / else / else-if	if 'n'==1 {
	  local word "one"
	}
	else if 'n'==2 {
	  local word "two"
	}
	else if 'n'==3 {
	  local word"three"
	}
	else {
	  local word "big"
	}
		if n==1:
		  w="one"
		elif n==2:
		  w="two"
		elif n==3:
		  w="three"
		else:
		  w="big"

try/catch	cap drop price
	if _rc != 0 {
	  di "Return code was: " _rc
	  di "Variable may not exist"
	}
		myListofVars = [[1,2,3], [2,4,6], [1,3,5]]
		try:
		  myList = myList[:1]
		except IndexError:
		  print "Got an Index Error!"

As we are getting into some more advanced programming, IDLE has a few tricks that may be of use. So far, we have been using IDLE interactively. In the interactive Python interpreter, to recall a block of code you already submitted, simply click once on it then press return. The code will appear at your command prompt. You can also highlight just a portion of code you've entered then hit return.

When writing loops and statements, indentation is critical. Because the interactive Python interpreter puts >>> at the beginning of each command prompt, keeping track of your indentation can be tricky. As you might write a do-file in Stata, you can write a similar script in Python by clicking IDLE's File menu, then New Window. If you save your script file as a .py, IDLE will even highlight the syntax as you type in it.

Task 10: Use a for loop to print each item of the list ["apples", "bananas", "oranges"].

Task 11: Use a for loop to print each number from 50 to 100, inclusive on both ends.

Task 12: Define a function evaluate(*name*) that takes in a string, and returns "cool" if name=="Python" or name=="Stata". Confirm that evaluate("Python") and evaluate("Stata") return "cool". But what is evaluate("Java")? Modify your function to return "lame" in any other condition, using an else statement.

Task 13: Assign myList = [-2,-1,0,1,2]. For each *item* of myList, print *item*. If *item* is less than zero, print "negative". Or else, if it is greater than zero, print "positive". Or else, print "zero". So within a for loop, there should be an if statement, followed by an $elif$, followed by an $else$.

If you are in search of a more nuanced discussion of compound statements in Python, consult Python's compound statements documentation.

1.2.5 Truth Value Testing

In if statements and *while* or for loops, we need to evaluate whether a condition is true. The intricacies of Python's truth value testing are discussed in brief below and in documentation.

Python uses familiar comparison operators, shown in Table 2. The "is" and "is not" operators may be new to you; these will be discussed shortly in a task.

And you can construct more complex boolean statements easily: statement x **or** statement y, statement x **and** statement y, statement x and **not** statement y.

That handles comparisons. So $3 > 1$ is **True**, while $3 < 1$ is **False**. What is the truth value of 1 or 2? Those are always **True** – a loop that starts with "while 1:" can run forever! (Try it if you want – control-c

Table 2: Comparison Operators

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
! =	not equal
is	object identity
is not	negated object identity

will kill it.) What about of a list? In general, the truth value of objects is **True**. The following objects will evaluate to **False** in an if statement or a loop:

- **None** – a special object in Python, similar in some respects to Stata's missing observations (.) or, more closely, other languages' "null".
- **False**
- 0
- An empty sequence of any sort: e.g.“”, [ ]

Task 14: Type into the Python interpreter "print 3==1". What does the expression evaluate to? What about 3>1? 3==1+2? How about "three"=="three"?

Task 15: The word "in" is a special word that tests to see if an item is in a list (or other more advanced data structures we'll soon discuss). What is the truth value of "0 in [1,2,3]"? "1 in [1,2,3]"?

Task 16: Confirm that [1]==[1] is True; that is to say, a list composed of the number one is equal to another list composed of the number one. What is the truth value of [1] is [1]? In fact, though these two lists are equal, they do not point to the same location in memory; they are not the same objects. Now, assign myList = [1]. What is myList==myList? What is the value of the expression myList is myList?

1.3 Advanced Data Structures

So far, the main data structure we have been working with is a list. Lists are *mutable*, meaning that you can add and delete items from them. You can even change an item:

```
>>> myList = ['a','b','c']
>>> myList[0] = 'z' # change the first item
>>> myList
['z', 'b', 'c']
```

For a more in-depth discussion of built-in methods to mutate lists, consult Python's documentation of mutable sequence types.

What about strings? Strings are mutable also, in similar ways. We will give more attention to strings soon, but first let us examine two *immutable* data structures, tuples and sets, followed by a powerful mutable data structure called a dictionary.

1.3.1 Tuples

Like a list, a tuple is an ordered set of values. Unlike a list, tuples are *immutable*, meaning a tuple cannot be changed once you define it, in the way that you would append to a list, for instance. If you were reading in a dataset, you might read in each row as a list, or as a tuple. It is also important to know about tuples because some methods return tuples, not lists. While lists are declared with brackets, tuples are declared with parentheses.

```
>>> row1 = ("name", "animal")
>>> row2 = ("Miss Piggy", "pig")
>>> row3 = ("Kermit", "frog")
>>> row2[0]
'Miss Piggy'
>>> row2.append("oink") # trying to append to a tuple will not make Python happy!
'Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    row2.append("oink")
AttributeError: tuple object has no attribute append '
```

1.3.2 Sets

A set is an *unordered* collection of *distinct* items. In older versions of Python declaring a set is a bit cumbersome: i.e., set([1,2,3]) would declare a set with elements 1, 2, and 3. In newer versions of Python, you can also declare that set with curly braces: {1,2,3}.

Task 17: define A = set([1,2,3,4]) and B=set([2,4,6,8]). What is A.union(B)? What is A.intersection(B)? What does A==A evaluate to? A==B? What about A==set([1,1,1,2,3,4])?

For more on sets, visit sets documentation.

1.3.3 Dictionaries (also known as hash maps)

In the real world, when you want to know the meaning of a word, you look the word up in a dictionary. A dictionary maps words to meanings.

In Python, dictionaries map keys to values. Given a key, you can quickly know its value: like a real dictionary, Python will keep your keys in order so it can quickly retrieve a key's value.[1] The example below shows how to define a dictionary. Like sets in the newer versions of Python, dictionaries are enclosed in curly braces. A colon should separate each key and value, and key-value pairs are separated by commas. Values can be retrieved from a dictionary similarly to how one would index into a list.

```
>>> myDict = {"Miss Piggy":"pig", "Kermit":"frog"}
>>> myDict["Kermit"]
'frog'
```

Sometimes you may find it useful to have all of a dictionary's keys in one list. Then, you can iterate over that list with a for loop. Take your time looking over the following example and map it out in your head. Dictionaries can be difficult to grasp at first.

```
>>> myDict.keys() # the keys
['Miss Piggy', 'Kermit']
```

[1] Unlike a real dictionary, Python rarely keeps its dictionaries in alphabetical order. It applies a *hash function* to each key you give it. For example, a simple hash function would be to match each letter to its position in the alphabet: A maps to memory location 1, B maps to location 2, and so forth. If Python needed to look up the *value* of "C", it would find that at location 3, just like you would find the meaning of "cat" under the dictionary entry for "cat". However, a more complex function is needed to hash numbers and more obscure characters. Regardless, when you go to look up that key, Python re-applies the same hash function it used to store the key's value, and knows exactly where in memory to find that value again. For this reason, some people refer to Python dictionaries as *hash maps*. When searching through large datasets, they will give you significant performance gains because they can quickly find values from keys.

```
>>> myDict.values() # the values
['pig', 'frog']
>>> myDict.items() # a list of keys AND values, in tuples of the form (key, value)
[('Miss Piggy', 'pig'), ('Kermit', 'frog')]
>>> for key in myDict.keys():
    print "our records show "+key+" is a "+myDict[key]
    ## myDict[key] will look up key's value in myDict
'our records show Miss Piggy is a pig
our records show Kermit is a frog'
```

Task 18: Define mydict = {1:"A", 2:"B", 3:"C"}. What is mydict[1]? Use a for loop to print each key separately. Now print each value separately. Can you put an if statement within a for loop that prints each key if its value is "C"?

For more on dictionaries, visit dictionaries documentation.

1.3.4 Casting and a Recap of Data Types

Before moving on to regular expressions, Table 3 recaps the data types we have covered so far.

It is also appropriate to note at this point that we sometimes need to convert an object of one data type to that of another data type. For example, if we wanted to make a tuple into a list, it's possible to ask Python to *reinterpret* a tuple as a list. In programming languages, we often refer to this as "casting."

Task 19: Define myNumber as your favorite number. For instance, you might enter myNumber = 7. Ask Python to print the following: "My favorite number is: "+myNumber. This should throw a TypeError, and Python will inform you that it cannot join together a string and an integer. Try casting myNumber as a string by having Python print: "My favorite number is: "+*str(myNumber)*.

All of the data types we have discussed so far have casting functions that take in objects of another type, and these functions are also listed in Table 3. It takes some playing around to decipher what objects each function can take: for example, Python can handle changing any integer to a string, but it can't always handle changing any string to an integer (the string "1" can be casted as an integer, but not "one", and certainly not a word like "apple").

Table 3: Data Types

Data Type	Example	Mutable?	Preserves Ordering?	Casting Function
Integer	1	Yes	N/A	int(..)
String	"word" or 'word'	Yes	Yes	str(..)
List	[1, 2, 3]	Yes	Yes	list(..)
Tuple	(1, 2, 3)	No	Yes	tuple(..)
Set	set([1,2,3]) or {1,2,3}	No	No	set(..)
Dictionary	{'A':'apple','C':'cat', 'B':'book'}	Yes	No	dict(..)

1.4 String Operators and Regular Expressions

One of the hardest parts of working with strings in Python is to remember that Python starts indexing at 0. "Slicing" into a string is similar to indexing into a list. The slicing functionality shown in the next example holds for both strings and lists.

```
>>> alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> alphabet[0] # first (ie, position 0)
'A'
>>> alphabet[1:] # from position 1 on
'BCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> alphabet[1:25] # from position 1 to before position 25
'BCDEFGHIJKLMNOPQRSTUVWXY'
>>> alphabet[:25] # everything before position 25
'ABCDEFGHIJKLMNOPQRSTUVWXY'
>>> alphabet[:-1] # negative indices start counting from the right
'ABCDEFGHIJKLMNOPQRSTUVWXY'
```

Task 20: Python's len(..) function takes in a string or a list, and returns its length. Using the len(..) function, for each letter in the string "Mississippi", print "Letter i of Mississippi is: " and the letter, where i is that letter's index in the string. When concatenating a string and an integer, don't forget to cast the integer as a string, as shown in Table 3.

Task 21: Now, print the index i and the first i letters of Mississippi.

When reading in or writing out data, which we'll get to soon, you'll often need to use line breaks and tabs. These are the two most frequently used special characters, also called escape sequences, and you use them similarly to how you would use any other character you might type from your keyboard. Signal a line

break with "\n" and a tab with "\t". It's also occasionally useful to enter text verbatim with three double-quotation marks, shown below.

```
>>> lines = """
a	1
b	2
c	3
"""
>>> lines
'\na\t1\nb\t2\nc\t3\n' # do you see the line breaks and tabs?
>>>  #below, verbatim allows us to have " within the string
>>> quoted = """Hello world," he said."""
>>> quoted
'"Hello world," he said.'
>>> #or
>>> quoted = '"Hello world," he said.'
>>> quoted
'"Hello world," he said.'
```

Task 22: Write two words separated by a line break. Write two words separated by a tab.

For more built-in methods you can use on strings, visit the string documentation.

1.4.1 Regular Expression Syntax

Regular expressions are an entirely separate language. They fill a certain niche. Consider, for example, asking a computer to find all email addresses in a document. How would you go about this problem? Perhaps you would break an email address into its elements: some characters that aren't spaces, followed by @, followed by some other characters that aren't spaces. You might also check to make sure there is a period sometime after the @. Still, how would you tell a computer to look for even something so simple as"one or more characters that aren't spaces?" It is for these types of problems that the regular expression language began to be developed in the 1950s, primarily for Unix text editors.[2]

To work with regular expressions, you'll need to import the Python module re. In Python, "import" is a special word: it means load all the functions and variables of another file, and let me use those. Think about it sort of like an add-on, but it's included when you install Python. To refer to a *function* of a

[2]The command for searching one of the early text editors for a regular expression *re* was g/*re*/p. For this reason, Unix/Linux users are very familiar with the Unix *grep* shell command.

module, just type *module.function*. A method of the re module, re.search(..) tells you whether your regular expression has been found in a string.

```
>>> import re
>>> if re.search("fun", "Python is fun!"):
  print "we will keep going"
'we will keep going' # glad it printed the right string here
```

In the example above "fun" is a very simple regular expression. What if we had a more complex problem? The regular expression language has several special metacharacters. For example, the * metacharacter matches 0 or more occurrences of the preceding character.

```
>>> re.findall("fun", "fuuun")
[]
>>> re.findall("fu*n", "fuuun")
['fuuun']
>>> re.findall("fu*n", "fn")
['fn']
>>> re.findall("fu*n", "fn fun fuuun fuuunnn")
['fn', 'fun', 'fuuun', 'fuuun']
```

Table 4 presents some of the more useful regular expression special characters. A complete list of special characters can be found in the documentation for Python's **re** module.

Regular expressions, along with all special characters, should be enclosed in double or single quotation marks as if they were ordinary strings.

Task 23: For the tasks below, import the re module and use the re.findall(*reg*, *string*) function to find all occurrences of regular expression *reg* in *string*.

A In the word Mississippi, find:

i Groups of one or more 's'. This should return ['ss', 'ss'].

ii Groups of 'i' followed by 0 or more 's'. This should return ['iss', 'iss', 'i', 'i'].

iii Groups of 'i' followed by 0 or one 's'. This should return ['is', 'is', 'i', 'i'].

iv An s followed by one or more non-linebreak characters followed by a p. This should return ['ssissipp'].

v Groups of one or more characters in the set [is]. This should return ['ississi', 'i'].

Table 4: Special Characters in Regular Expressions

.	Matches any character except a new line. (e.g. "f.n" would match an f, followed by any character except a new line, followed by an n.)
*	Matches 0 or more repetitions of the preceding character, as many as possible. (e.g "f.*n" would match f, followed by 0 or more non-linebreak characters, then an n.)
+	Matches 1 or more repetitions of the preceding character, as many as possible
?	Matches 0 or 1 of the preceding character
{m}	Matches m occurrences of the preceding character
{m,n}	Matches m to n occurrences of the preceding character, as many as possible
{m,n}?	Matches m to n occurrences of the preceding character, as *few* as possible
\	Escape character (eg. \. would match all periods in a string)
$A\|B$	Matches expression A or B. Can separate as many terms as you'd like with |, but the leftmost ones will be tried first, and matching stops when a match is made.
[ ]	Used to indicate a set. In a set: • You can list characters individually: [amk] will match a, m, or k • You can specify ranges of characters specified by a dash: [a-z] will match lowercase letters, [A-Z] uppercase, [0-9] the digits. • Special characters lose their special meanings in sets. • *Set negation* can be quite useful. The caret (^) takes on special meaning when it is the first character in a set: for example, "[^\n]" would match any character other than a new-line character; "[^\n]*" matches 0 or more (as many as possible) characters before the next line – this tends to be a useful expression.

B In the string 03Jan1991, find:

i Exactly three uppercase or lowercase letters. You should use sets for this: [A-Za-z] will be read by Python as uppercase or lowercase letters, and use curly braces as shown in Table 4 to match exactly three.

ii Two digits followed by a letter.

iii A letter followed by four digits.

iv All occurrences of the string "Dec" or "Jan" or "Feb" (use the syntax "$A|B|C$").

1.4.2 Regular Expression Methods

In the above examples, you saw regular expression methods re.search(..) and re.findall(..). Below are some of the most common methods of python's **re** module; once again, a complete list can be found in the

documentation.

Table 5: Regular Expression Methods

Function	Return Type	Description
re.findall(*pattern*, *string*)	List of Strings	Return all non-overlapping matches of *pattern* in *string*, in the order in which they were found.
re.split(*pattern*, *string*)	List of Strings	Return *string*, but with each element of *pattern* breaking apart pieces of *string*. Splitting on \n, for example, would return a list where each item is a line of the original *string*.
string.join(*list*)	String	This method isn't actually in the **re** module, but is very useful. Concatenate items of a list (or any iterable – such as a tuple or set), with *string* between each item. In other words, the opposite of re.join(..)
re.sub(*pattern*, *repl*, *string*)	String	Return *string*, with all instances of *pattern* replaced by *repl*. Similar to Stata's subinstr(..) and regexr(..).
re.search(*pattern*, *string*)	MatchObject instance (or None if no match)	Scan through *string* looking for a location where the regular expression *pattern* produces a match. Similar to Stata's regexm(..).
re.match(*pattern*, *string*)	MatchObject instance (or None if no match)	Same as re.search(..), but only test for a match starting at the beginning of *string* (ie, position 0).

The last two methods return MatchObject instances if they found a match. Don't worry too much about this now – we will see very soon how to make these very useful with grouping.

Task 24: Split "This is a sentence" into a list of words. Now, join that list into one string, with spaces separating the words.

Task 25: You now have the ability to load comma-separated values into a dictionary. In the next task, we will create a dictionary mapping each column of comma-separated data to a list. Since this is a longer task, feel free to use Python's **print** statement liberally to help you check that what you're doing is correct, and it may be a good idea to jot down some notes on what your loops will look like before you touch the keyboard.

Define myData = " " " v1, v2, v3
1, 2, 3
2, 4, 6
4, 8, 12 " " "

Use re.split(..) to define myRows as the rows of myData. Recall that a linebreak appears as \n. Define myDict as a dictionary in which the numbers 0, 1, and 2 each map to an empty list, written as [].

Now, for each row in myRows, define r as a list, whose items are the three values of the row separated by commas (use re.split(..) again). You must append each item $r[i]$ of r to myDict[i]; you will iterate over values from 0 to len(r).

Remember, you've defined myDict[i] to be a list, so you can use all the list functions you learned in section 1.1.2. You may also want to consult Table 1 to recall how to iterate over a range of values.

1.4.3 Grouping RE's

Sometimes we want to search for a regular expression, but are only interested in a piece of it. In the example below, we want to search for a client's name. Having examined our data, we know that the client's name appears as all letters between the string "\nCLIENT:" and the end of the line, "\n". We could search for the expression "\nCLIENT:[^\n]+", but it would be nice not to have that identifying text at the beginning of every client record we find.

What can we do? One solution is to slice the string that matches our regular expression: we know it starts with "\nCLIENT:", so we can remove the first 8 characters (\n counts as one character). An easier solution is to denote groups in the regular expression with parentheses. Two more special regular expression characters are (and) – they enclose groups, and in the example below, we show how to extract matches from each group.

```
>>> document = """
CLIENT: MIKE SHORES
HAIR COLOR: BROWN
"""
>>> document
'\nCLIENT: MIKE SHORES\nHAIR COLOR: BROWN\n'
>>> import re
>>> result = re.search("(\nCLIENT: )([^\n]+)", document)
    # group 1: "\nCLIENT: "
    # group 2: one or more non-linebreak characters following that
>>> result ## it's an instance of some scary looking Match object...
<_sre.SRE_Match object at 0x0147B608>
```

```
>>> result.groups() ## returns a tuple of strings
('\nCLIENT: ', 'MIKE SHORES')
>>> result.group(1) ## the garbage we can forget about
'\nCLIENT: '
>>> result.group(2) ## what we were trying to extract!
'MIKE SHORES'
```

Task 26: In the string 03Jan1991, find:

1. Two digits followed by a letter. Use groups to return only the two digits.
2. A letter followed by four digits. Use groups to return only the four digits.

Instead of referring to groups by their numbers, it's sometimes convenient to name groups. You can name a group *name* by putting ?P< *name* > at the beginning of the group, as shown below.

```
>>> result = re.search("(?P<garbage>\nCLIENT: )(?P<name>[^\n]+)", document)
>>> result.group('garbage')
'\nCLIENT: '
>>> result.group('name')
'MIKE SHORES'
```

1.4.4 Assertions: Non-Capturing Groups

In the previous section, we talked about dividing a regular expression into groups when we need the whole regular expression to identify a string, but we only want to retrieve part of it. Another option is to use non-capturing groups.

For example, perhaps I want to find a match for a regular expression only if that regular expression is not followed by another regular expression. We can use what is called a *negative lookbehind assertion* to accomplish this. Negative lookbehind assertions are written similarly to groups: (? <!*str*)*re* will match anytime *re* matches, and the string *str* does not come before it. Similarly, (? <= *str*)*re* will match anytime *re* matches, and the string *str* does come before it.

```
>>> re.findall("[0-9]", "I like 1, 2, not 3, 4, 5, not 6, maybe 7 a little.")
['1', '2', '3', '4', '5', '6', '7']
>>> re.findall("(?<!not )[0-9]", "I like 1, 2, not 3, 4, 5, not 6, maybe 7 a little.")
['1', '2', '4', '5', '7']
```

```
>>> re.findall("(?<=not )[0-9]", "I like 1, 2, not 3, 4, 5, not 6, maybe 7 a little.")
['3', '6']
```

There are four types of assertions:

Table 6: Non-Capturing Groups (aka Assertions)

$re(?=str)$	A *positive lookahead assertion* matches if *str* matches next.
$re(?!str)$	A *negative lookahead assertion* matches if *str* does not match next.
$(?<=str)re$	A *positive lookbehind assertion* matches if *str* matches before.
$(?<!str)re$	A *negative lookbehind assertion* matches if *str* does not match before.

Below is an example of positive and negative *lookahead assertions*.

```
>>> re.findall("[0-9]+(?=g)", "The fat content of fried chicken has decreased 11 percent
    from 9g to 8g per bite.")
['9', '8']
>>> re.findall("[0-9]+(?!g)", "The fat content of fried chicken has decreased 11 percent
    from 9g to 8g per bite.")
['11']
```

1.4.5 Portability of REs (REs in Stata)

Python's regular expression syntax is based largely on the regular expression syntax of an older scripting language, called Perl. Programs like SAS and Stata support some regular expression functionality as well. Both Stata and Perl claim to be based off of the same library, called *regex*, written by Canadian computer scientist Henry Spencer in the late 1980s.

Most of what we have learned so far about regular expressions will carry over to most other programs: . virtually always matches any single character, * means 0 or more, [ ] denotes sets, etc. However, groups beginning with (? ...) were added in Perl 5 in the 1990s, and though it has been implemented in Python, that syntax may not carry everywhere.[3] Syntax of the form (?P...) is specific to Python.

Table 7 and the example below it demonstrate how to apply regular expression syntax in Stata.

In the example below, we show Stata code to find a match for a regular expression, then retrieve the whole match (group 0), followed by each of the matched groups.

[3] According to documentation, the (? ...) syntax was chosen for the new extensions because coming at the beginning of a group, ? had nothing to repeat and was therefore a syntax error before the extensions were implemented.

Table 7: RE Methods in Stata

regexr(*s1*, *re*, *s2*)	Replace the first substring of *s1* that matches *re* with *s2*. Returns the altered string, or else the original *s1* if *re* did not match.
regexm(*s*, *re*)	Returns 1 if *re* matches *s*, and 0 otherwise. More importantly, the stored match can be retrieved with regexs(..)
regexs(*n*)	Returns group *n* of the most recent match. Like in Python, group 0 is the entire match. See example below.

```
. di regexm("Client: Mike Shores", "(Client: )(.*)")
1
. di regexs(0) // the whole match
Client: Mike Shores
. di regexs(1) // the first group
Client:
. di regexs(2) // the second group
Mike Shores
```

The regexm(..) and regexs(..) functions are commonly used in tandem to generate new variables, as shown in this example from the Stata help files that changes some formatting of a string variable containing a phone number. Note that ^at the beginning of a regular expression means to match the beginning of the string (in Python, this is more commonly accomplished by using re.match(..) instead of re.search(^..)).

```
. list number
     +----------------+
     |         number |
     |----------------|
  1. | (123) 456-7890 |
  2. |  (800) STATAPC |
     +----------------+
. gen str newnum = regexs(1) + "-" + regexs(2) if regexm(number, "^\(([0-9]+)\) (.*)")
. list number newnum
     +------------------------------+
     |         number         newnum |
     |------------------------------|
```

```
1. | (123) 456-7890   123-456-7890 |
2. |  (800) STATAPC    800-STATAPC |
   +--------------------------------+
```

1.5 Working with the Operating System

The two most useful modules for working with the operating system are os and sys.

os is useful for manipulating your working directory. After importing the **os** module, os.getcwd() returns the current directory, and os.chdir(*dir*) changes your directory to *dir*, where *dir* is a string. os.listdir(*dir*) returns a list in which each item is a file or sub-directory in the directory *dir*.

When moving around Windows directories, keep in mind that backslash is a special escape character in strings, so you may need to *escape* that special property of backslash. How? By using the escape character itself! As shown in the example below, you can insert a backslash before each backslash that you want to be treated as a normal backslash character. If you're on Unix or Mac, you will not have this backslash problem because those operating systems separate directories with forward slashes, which do not have special meaning in Python.

```
>>> import os
>>> os.getcwd()
'H:\\'
>>> home = "C:\\Documents and Settings\\abell1\\My Documents"
>>> os.chdir(home)
>>> os.getcwd()
'C:\\Documents and Settings\\abell1\\My Documents'
>>> if len(os.listdir(home)) > 10: print "time to clean out my home directory :("
'time to clean out my home directory :('
```

Task 27: In Python, import the os module. What is your current directory? What is in it? Change directory to your "My Documents" folder. You may want to open Windows Explorer to see the full path to this folder.

Task 28: Explore your computer a bit until you find a directory with several files with different extensions. Use os.listdir(..) to retrieve the contents of that directory, and then print out all file names that have a particular extension (e.g., .doc, .exe, .do, etc.)

os.system(*command*) can also be useful: it simply executes *command* in the shell, where *command* is a string. Use it in combination with os.listdir(..) to move or rename several files of a certain type, for instance. Soon, we will see how to remove a file on Windows. In Unix, you can rename or move files with the "mv" command and remove them with the "rm" command. Any commands you can execute in the terminal, you can also execute with os.sytem(..). Below we show how we can use some simple Python to move all .dta files to a new directory, on Unix, called "data".

```
import os, re
os.chdir("~/messyfolder") # change directory to a folder in my Unix home directory (aka ~)
for filename in os.lisdir(os.getcwd()): # list the contents of my current directory
  if re.search("\.dta", filename):
    os.system("mv "+filename+" data/"+filename)
```

So far, you have been running Python interactively. Sometimes, it's useful to save a script and call it from the shell. You can even call a script from Stata by using Stata's built-in **shell** command: something like "shell python file.py arg1 arg2," where arg1 and arg2 are arguments you'd like to pass to your program from a Stata do-file. Just import sys in your Python file, and when the file is called from the commandline sys.argv will be a list of parameters passed to Python. In this example, sys.argv would return ['test.py', 'arg1', 'arg2'].

1.6 Working with Files

Open a file object with the method open(*filename*, *mode*). *filename*, a string, should either be the full path to the file, or else the relative path to the file within the current working directory (which you now know how to manipulate). The three most common *mode*s are:

- 'r': read-only mode; the default if no mode is specified
- 'w': write-only mode (an existing file with the same name will be overwritten)
- 'a': append mode; any data written to the file is automatically added to the end (if the file does not yet exist, Python will create it)

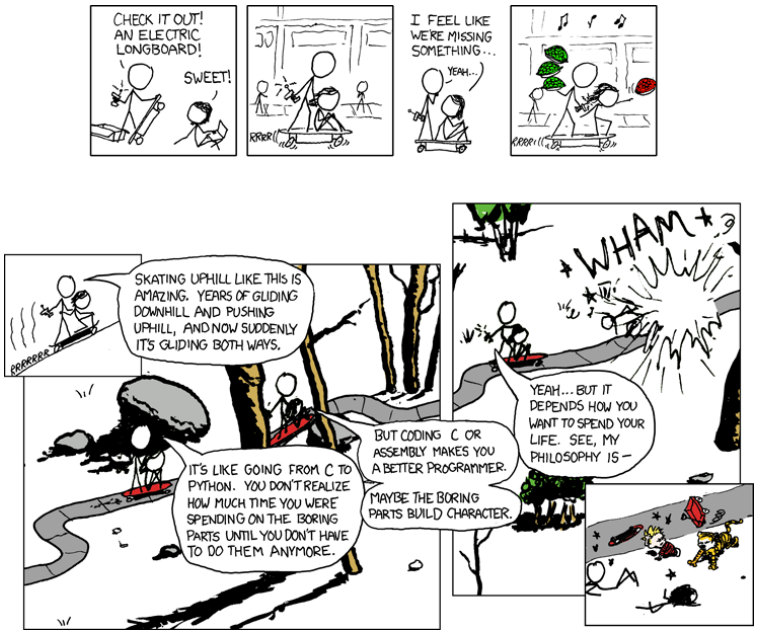
open(..) returns a file object, which is essentially a pointer to a file on disk. Interacting with the file object is intuitive. *file*.read() returns the data stored in the file, and *file*.write(*data*) writes string *data* to the file. When you're done with *file*, call *file*.close(). This both frees up system resources and ensures the data you've been telling Python to write to the file is actually flushed from the memory cache to the disk.

There may come a day when you want to process a file that is too large to fit into memory, and you want Python to read it from the disk line by line rather than all at once. For this reason, Python can interpret a file as a list of lines, and, if *file* is an open file object you created with the open(..) constructor, you can use the syntax "for line in *file*:" to access each line of the file as a string; one line will be read from the disk each time you cycle through the for loop.

Task 29: Use the os module to navigate to your My Documents directory. Open a file called "file.txt", in write mode, and write to it the words "Hello, World!". Now, write "\n 1 \n 2" then close it. Can you see the file in Windows Explorer? Can you open it with a text editor like Notepad or Wordpad? Try opening the file in append mode, and append the line "\n appended", then close it again. Open the file in read mode, and call the read() method. Close it once more, and use the os.system() command to delete file.txt using the Windows shell command "del file.txt". Confirm that the file is no longer in your My Documents folder.

For more on files, see Python's documentation for working with files.

2 Applications



http://xkcd.com/409

2.1 Text Processing

In this section, we discuss ways to pick up where programs such as Stata leave off with regard to handling strings. It's important that you are confident moving data between Stata and Python.

Task 30: In the "materials" folder, inventors.dta is a sampling of names and locations of first named inventors on US patent applications. The field appl_id is the application's ID number. There is a field called "inventor", but you'd like to split it into name, city, state, and country. This may be possible to do in Stata, but there are some formatting challenges that make using Stata's split command difficult. You may also encounter situations when the length of string variables exceeds Stata's 244 character limit, making text processing in Stata impossible.

Open inventors.dta with Stata (on the training computers, \\apps\local\STATA12MP\StataMP.exe). Outsheet it; the default tab-delimited is good since you know your data doesn't have tabs, but it does have commas. Specify noquote and nonames options: do not put string variables in quotation marks, and do not put names of variables on the first line. Name it locations_raw.txt.

In the "materials" folder is a Python script location_cleaner.py. It will open your outsheeted data and do a bit of textual processing on it. The sort of processing it does is nothing you couldn't do with a few hours to play with the raw data.

Run the script and open the result, locations_output.txt, with a text editor to see how it looks. Insheet it with Stata.

Now, use Stata's shell command to call Python from the commandline. Notice the two try/except blocks at the top of the Python file. It will look for two optional arguments, the name of the input file and the name of the output file. You can pass that information right from a do file. You already know a good deal of Python, and hopefully this exercise has shown you how you can use that knowledge as a very natural extension of the work you already do in programs such as Stata.

2.1.1 Extraction from Word Documents

You can open just about any file with Python's open(..) and read() functionality, but you won't always get something pretty out.

Task 31: A company under investigation has given you records of every transaction they've made in the last year. Unfortunately, each transaction was logged in its own Microsoft Word file! Can you extract clients' names, dates, and zip codes? Open each .doc file in read mode for binary files, "rb". Confirm that each series is uniquely identified in each .doc file by the second group of each of the following regular expressions:

1. "(CLIENT NAME:)([^\r]+)"
2. "(DATE:)([^\r]+)"
3. "(ZIP:)([^\r]+)"

Save the extracted data to a file you open in write mode at the start of the script. Each line of the output file should represent a Word document, and you should separate your series with tabs – result.write("\t".join([name,date,zip])+"\n") for each Word document. Don't forget to close the output file at the end of your script.

2.1.2 Word Frequency Dictionaries

Word frequency dictionaries have many uses. For instance, you might be matching names between two lists of companies. Say one contains the company "Pfizer", and the other contains a "Pfizer, Inc.". It may help you to know something about the relative frequency of the term "Pfizer" in your dataset, as compared to the term "Inc." in your dataset.

Similarly, if you're looking for likely matches between datasets by each word, it would take a very long time to examine all potential matches of "Pfizer, Inc." that just contain the word "Inc.", so you might want to omit some of the most frequent words from your matching strategy.

Task 32: The "materials" folder contains a folder called "constitutions". In that folder are the full text of the US Constitution, the Articles of Confederation, and the Magna Carta. The file stopwords.txt is for something we will discuss shortly.

People often say the Constitution discusses the federal government more, whereas the Articles of

Confederation discusses mostly states' rights. To test this claim, read in the text files. Use the built-in function *str*.lower(), which can be called on any string, to make all letters lowercase. To get a list of words, split each document on the regular expression '[.!?]+' – one or more spaces, periods, exclamation points, or question marks. Define an empty dictionary myDict={}.

For each *word* in the list of words you've created, **try:** myDict[*word*]+=1. Since accessing a dictionary on a key that doesn't (yet) exist will raise a KeyError, we must follow with a clause starting with **except KeyError:**, and in that case, set myDict[*word*]=1, because this is the first time we've seen that word. The syntax of try/except is shown in Table 1.

What are the most frequent words in the documents? Assign "freqList = sorted(myDict.items(), key=lambda(k,v):(v,k))," and freqList[-10:] will be your ten most frequent words. You've probably picked up some words that appear commonly, but don't mean much – words like "the" or "and." Modify your code to only use words not in stopwords.txt, then check again.

2.1.3 Soundex: Surname Matching by Sounds

The Soundex algorithm is used to match subjects' last names. In the early 1900s, American inventor Robert C. Russell filed two US patents "to simplify and improve an index wherein names are to be entered and grouped phonetically rather than in accordance with the alphabetical construction of the names." The details of his coding system are described in the patent – he breaks letters' sounds up into classes such as the "labials and labio-dentals", not to be confused with the "labio-nasal represented by the m" or the "dental-mutes represented by t and d." After applying a set of rules, the algorithm spits out the first letter of the surname followed by three digits, which are supposed to be distinct to the sound of the name. Russell's Soundex algorithm has been important in the field of surname matching, notably for census work. The National Archives maintains its own, updated copy of the algorithm for census researchers.

A version of Soundex is implemented in Stata: In Stata, display soundex("Robert") returns R163, as does display soundex("Rupert").

A (slightly different)[4] version of the Soundex algorithm is also available for your convenience and research dabblings in the "materials" folder. To use it, either open it with IDLE or navigate to the "materials"

[4]Mine is based off of the version on Wikipedia

directory and type "import soundex" then "soundex.getSoundex(*name*)."

2.1.4 Levenshtein's "Edit Distance"

In Stata and Python, testing the equality of strings is simple. For example, "Fahrenheit"=="Fahrenheit" is True (both strings are spelled the same, correctly), but "F**ahr**enheit"=="F**ar**enheit" (a common mispelling) is False, so these strings do not match. But for many applications, we would want these strings to match.

The Levenshtein algorithm tells you the "edit distance" between two words: the minimum number of insertions, deletions, or substitutions required to transform one word into the second. Suppose someone entering data meant to type "Jason" but his finger slipped when hitting the "s" and he hit an extra "a" by accident – the data now says"Jaaon".

Table 8: Levenshtein Processing of "Jaaon" vs. "Jason"

		J	a	*a*	o	n
	0	1	2	3	4	5
J	1	**0**	1	2	3	4
a	2	1	**0**	1	2	3
s	3	2	1	***1***	2	3
o	4	3	2	2	**1**	2
n	5	4	3	3	2	**1**

The most intuitive way to follow the process of the Levenshtein algorithm is along the diagonal, in bold. What is the distance between an empty string and an empty string? 0 – the first item of the diagonal. Between 'J' and 'J'? Also 0 – move one to the right and one downward. Same for the distance between 'Ja' and 'Ja' – it is the next 0. But what about 'Jaa' and 'Jas'? That is 1, so there is a 1 in that position (italicized).

Where would you look to find the distance between 'Jason' and an empty string? 'Jason' and 'J'? What about the distance between 'Jason' and 'Jaaon'? In your mind, verify that the numbers you're finding there are correct.

What is the distance between "Farenheit" and "Fahrenheit"? In all cases, the total Levenshtein distance between two strings is the number in the bottommost and rightmost position.

A Python implementation of the Levenshtein algorithm is included in the "materials" folder. It is called

Table 9: Levenshtein Processing of "Farenheit" vs. "Fahrenheit"

		F	a	r	e	n	h	e	i	t
	0	1	2	3	4	5	6	7	8	9
F	1	0	1	2	3	4	5	6	7	8
a	2	1	0	1	2	3	4	5	6	7
h	3	2	1	1	2	3	3	4	5	6
r	4	3	2	1	2	3	4	4	5	6
e	5	4	3	2	1	2	3	4	5	6
n	6	5	4	3	2	1	2	3	4	5
h	7	6	5	4	3	2	1	2	3	4
e	8	7	6	5	4	3	2	1	2	3
i	9	8	7	6	5	4	3	2	1	2
t	10	9	8	7	6	5	4	3	2	**1**

lev.py. To represent the matrices shown above, it uses a list in which each item is a list, as we did for a task in Section 1.1.2.

2.2 Web Scraping

I use the term Web Scraping to refer to the process of taking data from the internet, and putting it into usable form for a research project. In general, web scraping involves breaking a URL down into static and variable pieces.

A common example of web scraping is stock data. If you're looking for historical information on a company, say stock ticker **GOOG** (Google, Inc.), you can get its current price at http://finance.yahoo.com/q/pr?s=**GOOG**. When you access that link with your browser, Yahoo Finance sends a bunch of HTML and other formatting that your browser makes sense of – it translates that into different colors, alignments, etc. But Python can read that source code, too, and from that source code, you can write a regular expression to pick out the current price. If you wanted to get every company's current price, it would only be a matter of getting a list of all ticker symbols, and putting that in a for loop. Many sites make historical data available also; to get historical comma-separated data, you'd just modify your loop to pass through pages of the form http://ichart.finance.yahoo.com/table.csv?s=**GOOG**&d=11&e=3&f=2012&g=d&a=0&b=2&c=2010&ignore=.csv.

Some sites, such as Yahoo Finance, make data readily accessible to web scrapers. Google Insights for Search is another easily scrapable site worth looking into. They allow you to see relative frequencies of search terms, by date and by country (in the US, broken down to states and metropolitan areas). At the

top right, you have an option to download the results as comma-separated data. There may be some interesting patterns out there.

2.2.1 Using urllib2

With Python's urllib2 module, opening websites is about as easy as opening files.

```
>>> import urllib2, re
>>> page = urllib2.urlopen("http://www.nytimes.com")
>>> text = page.read()
>>> re.search('(<title>)(.+)(</title>)', text).group(2) ## What is the title of the page?
'The New York Times - Breaking News, World News &amp;amp; Multimedia'
>>> ## let's look over some stories from the front page
>>> big_leads = re.findall('(<p class="summary">\n*)(.+)(</p>)', text)
>>> big_leads[0][1]
'President Obama refused to apologize for his remarks to Mitt Romney on Saturday,
    barnstorming through Virginia as his aides continued their attacks.        '
>>> big_leads[2][1]
'The gadget in your purse or jeans that you think is a cellphone is actually a tracking
    device that happens to make calls.'
```

You don't have to know HTML to be able to extract useful information from web sites. Oftentimes, it's a matter of looking at the website through your web browser, finding an example of a field you'd like to extract (e.g., "President Obama refused to apologize. . . "), then searching for that in the HTML source code that Python sees.

Task 33: In this task, you will make queries to and extract data from a website.

Suppose you have a list of phone numbers, and you'd like to know where they are located. In a web browser, navigate to usreversephonedirectory.com. Give it your phone number. Hopefully, it found the number and told you the correct location. Next, examine the URL it took you to. If your phone number were 012-345-6789, this particular website should take you to the address:
*http://usreversephonedirectory.com/results.php?areacode=**012**&phone1=**345**&phone2=**6789**&type=phone&Search=Search&redir_page=results%2Fphone%2F&imageField.x=0&imageField.y=0*

Armed with this intuition, the file phone_numbers.txt in the "materials" folder contains 50 *real* phone numbers generated by a random number generator. Open the file in Python, split the phone numbers on

"-", and make the appropriate URL request. In order to find out how to extract the location information, open a page in Python and in a web browser. From your browser, see what the location is, then from Python, identify what text comes before and after that specific location. Search the page with a regular expression with two groups: the first should be the text that comes before, and the second should be one or more characters that are *not* the character you've identified that signals the end of the field.

For more uses of Python's urllib2 module, visit Python's urllib2 howto.

2.2.2 Logging-in with Cookies

A common problem with web scraping is that many of the sites you want to download information from require a log-in process. Sometimes, it's possible to carefully examine the HTML of the site (e.g., using your browser's "View Source" feature) and determine which fields to post your login credentials using an HTTP request form Python. However, it has also become common for sites to use Javascript functions and special features that make automated log-in more difficult. In those cases, a huge time-saver can be to log in to the site via a web browser, save your cookies, and include those cookies in all of your automated Python requests. The package `pycookiecheat` allows you to load your cookies from Chrome into Python. Note that the module `pycookiecheat` is designed to work with the module `requests` rather than `urllib2`. Here is some example code to log in to OKCupid using your computer's Chrome cookies:[5]

```
import requests, pycookiecheat
url = 'https://okcupid.com/'
s = requests.Session()
cookies = pycookiecheat.chrome_cookies(url)
response = s.get('https://www.okcupid.com/match', cookies = cookies)
```

Task 34: Make an OKCupid profile and log-in from Chrome. Install `pycookiecheat` and use that package to log in to your OKCupid account and download the page that lists links to profiles near you.

2.2.3 Making your Scripts Robust

Sometimes, the internet just hiccups. This could be a fault on your computer's end or the fault of the downloading server.

[5] I have found some of these packages to be more stable in Python 3.

For some purposes, you won't care if your script crashes, because you'll be monitoring it anyway. Other times, if you'd like a job to run overnight or generally without your attention, it will help you keep your sanity to put some safeguards in place to stop the job from failing.

The goal is to have your script continue to run, at some level ignoring the error, but also to log the error so that you know what has been happening, in case there's something you can do about it. Because you may have a lot of output on the screen, for a large job you may wish to have a file that just logs time and description of errors.

Here is a function that essentially *wraps around* urllib2's urlopen(..) method, making it a bit more useful for a big job.

```
def grab(url, num):
    try: # try to download url and return the page
        page = urllib2.urlopen(url)
        return page # only get to this line if no error on urlopen(..)
    except urllib2.HTTPError as e: # there was an error
        err = open("err.txt","a") # open error log, and write to it the error description,
            time, and how many times we've tried to grab this url
        err.write(str(e)+" on "+url+" at "+time.strftime("%d/%m %H:%M%S")+"; attempt "+str(
            num+1)+"\n\n")
        err.close()
        if num > 100: # if we've already tried to grab this url 100 times, let's give up
            print "maybe it's time to give up on this address"
            return None
        else: # if not, let's put Python to sleep for a minute, and then try again.
            hopefully, the problem will resolve itself soon.
            time.sleep(60)
            return grab(url, num+1)
```

2.2.4 Saving Binary Files on Windows

We said there are three main modes for dealing with files: "w", "r", and "a". On Windows, when you aren't dealing with text, you may need to use "wb" and "rb" for dealing with files that aren't text. This includes JPEG and ZIP files you may be downloading. The "b" stands for binary mode. Only Windows makes the distinction between binary and text files, but you can still keep the "b" on other platforms.

2.2.5 Chunking Large Downloads

Most desktops at work have just a few gigabytes of RAM, or Random Access Memory. When you download a file from Internet Explorer, it usually downloads files straight to the hard disk, which is large in comparison to RAM. When you download a file in Python, you download it only into RAM until you open(..) a new file and write the contents of the page to the file. If you're downloading a large file, you may not be able to fit the whole file into RAM at once. Not only can this grind your computer to a halt, but it can also cause Python to throw an error, terminating your script's execution.

To get around this problem, the read() method allows you to pass in an optional parameter of how much data to read from the web page, in bytes. Each time the read(*bytes*) method is called on a page, it starts from where it left off. So all we need to do is read a few bytes of data, save it to the disk, and repeat.

```
page = urllib2.urlopen("http://www.bigpage.com")
fileOut = open("bigpage.html", "w")
CHUNK = 1024 * 100 # 100 kilobyte chunks -- set it what you'd like
while True:
        chunk = page.read(CHUNK)
        if len(chunk)==0: break # this will happen once we've read the whole page
        fileOut.write(chunk)
fileOut.close()
```

2.2.6 Unzipping

If you're downloading lots of zipfiles, it's easy to unzip them. Suppose you've just downloaded a zip file *file*. We will unzip it to *newDirectory*, which will be automatically created.

```
import zipfile
z = zipfile.ZipFile(file)
z.extractall(newDirectory)
```

2.2.7 Email Notifications

If you have a long process running, it could be useful to notify yourself by email whenever it completes, or if it is running into problems and needs your attention.

The Python module smtplib contains useful documentation for sending emails through Python. There is also a Python module called email. A sample script for sending email from a Gmail account is below:

```
import smtplib, platform

fromaddr = '???@gmail.com'  # fill in
toaddr  = '!!!@gmail.com'  # fill in
me = platform.node() # the name the computer calls itself
msg = 'New notification from '+me+'.'   # you could make this more specific to your needs

# Credentials (if needed)
username = 'fakeUserName'  # fill in
password = 'fakePassword'  # fill in

# The actual mail send
server = smtplib.SMTP('smtp.gmail.com:587')
server.starttls()
server.login(username,password)
server.sendmail(fromaddr, toaddr, msg)
server.quit()
```

2.2.8 Crawling

So far, we have discussed web scraping primarily in terms of breaking a URL down into static and variable pieces. With the textual analysis skills we've covered, it's also possible to arrive at one website and pick out the links we want to follow, and continue on until we've found what we're looking for, in some sense "crawling" the web.

Task 35: The full text of all patents granted by the US Patent and Trademark Office from 1976 to today is hosted in weekly zip files at http://www.google.com/googlebooks/uspto-patents-grants-text.html. From that page, can you extract a list of all zip files? You probably shouldn't actually download all the data – it's almost 100GB, and is sure to get you some calls from IT!

2.2.9 A Note on Privacy

Be conscious of what information you're giving to sites when you web scrape, and what computers you use. For example, if you query a site for several thousand phone numbers, and the site sees that traffic for those

specific phone numbers coming from your IP address, you may be giving the site more information than you intended.

3 Extensions

3.1 Scripting ArcGIS

Users of the ArcGIS suite may be interested to know that those programs are deeply integrated with Python. Next time you go to generate a new variable in ArcMap, consider using a built-in Python function, or defining your own in the codeblock. For more advanced automation, there is a Python module called arcpy that allows you to interface with Arc's geoprocessor, and script your workflow. Some documentation of the acrpy module is given on ESRI's website, though I have yet to find an easy-to-follow tutorial (maybe somebody reading this will figure out arcpy and write one).