Introduction to Python

Basics

This tutorial is intended as a companion to the official Python tutorial.

We'll be focusing on our case study problem of whether men or women talk more. Let's start with a single sentence from the Fisher corpus:

```
I'm in graduate school
```

A reasonable sentence, right?

Strings

In Python, we can represent this sentence as a string, which is Python's type for text data. Strings are separated from the rest of your program by quotes. Fire up Python and give this a try:

```
$ python3
Python 3.2.3 (default, Feb 20 2013, 14:44:27)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "I'm in graduate school"
"I'm in graduate school"
```

If you type this string at the Python prompt, Python echoes it right back at you. That isn't very useful, but it does confirm that you've entered it correctly, and in a way that doesn't cause any errors.

Normally, you can also use single quotes to indicate a string.

```
$ python3
 Python 3.2.3 (default, Feb 20 2013, 14:44:27)
 [GCC 4.7.2] on linux2
 Type "help", "copyright", "credits" or "license" for more information.
 >>> 'hello'
'hello'
```

But what is happening when we use single quotes for our Fisher sentence?

(I'm going to omit Python's startup messages in these examples from now on. You can tell I'm using the Python interactive interpreter if the prompt is >>>, and I'm using the Linux command line if the prompt is *$*.)

```
>>> 'I'm in graduate school'
  File "<stdin>", line 1
     'I'm in graduate school'
        ^
  SyntaxError: invalid syntax
>>>
```

When you enter this, you get an error message... Why?

Since this string includes a single quote *inside* it (the one in "I'm"), it will cause an error... which you can fix by switching back to double quotes, or by putting a backslash before the troublesome character.

```
>>> 'I\'m in graduate school'
"I'm in graduate school"
```

The length function

Let's start out by measuring the length of our string in characters. To do this, we'll use the **len** function:

```
>>> len("I'm in graduate school")
22
```

This works just like typing a string. We've typed an *expression* at the Python prompt, and Python printed out its *value* for us. The value of a string is just itself; the value of the *len* function, *applied to* the string, is the length of the string in characters.

22 is a new data type, an *int* (short for *integer*). By the way, you can find the type of something with the **type** function:

```
>>> type("I'm in graduate school")
<class 'str'>
>>> type(len("I'm in graduate school"))
<class 'int'>
```

'str' stands for string. To find out more about the *len* function, we can use *help*:

```
>>> help(len)
Help on built-in function len in module builtins:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
(END)
```

(Once you're done with the help text, hit the 'q' key to get rid of it.)

help tells us several things. *len(object) -> integer* tells us the **type signature** of the function--- *len* expects a single argument, which can be almost anything, and its value is an integer. It also tells us what *len* actually does (return the number of items in a sequence).

Numbers

We can use the integer type to do math with Python:

```
>>> 2 + 2
4
```

Since the value of *len("I'm in graduate school")* is a number, we can do math with it just like anything else:

```
>>> len("I'm in graduate school") + len("what school are you in")
44
>>> 3 * len("I'm in graduate school")
66
```

The *number* 22 is different from the *string* "22" (written with quote marks around it)! If you use the string version in a mathematical expression by mistake, you might get the wrong operation, or you might get an error.

```
>>> 2 * 22 #right
44
>>> 2 * "22" #not what you meant
'2222'
>>> 2 + "22" #error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Again, the error message is pretty sensible--- this is a TypeError, because Python can't add an integer to a string.

The tutorial has many more useful math facts. At some point, you should spent a while playing with mathematical expressions. For instance, how do you write *two to the power of the sum of three and five divided by four*?

Strings and words

Of course, when we ask whether men or women talk more, we don't necessarily want to measure the lengths of their utterances in characters! There are a variety of questions we might ask (total speech time, total syllables, etc). For now, let's try measuring the length of an utterance in words.

I'm in graduate school is four words long. We can ask Python to split the string into words like this:

```
>>> "I'm in graduate school".split()
["I'm", 'in', 'graduate', 'school']
```

There are two new things going on here. One is the expression itself. We used the *.split()* function to split the string into words; this function places a word boundary wherever it sees whitespace. *.split()* has a slightly different syntax from *len*. It's called a **class function** or **method**, and it has the syntax *argument1.function(argument2, argument3)*. Python uses the first argument to this kind of function to look up the code it's going to have to execute.

The second thing is the list data type, which is the type of the value we just got. A list is delimited with square brackets, and its items are separated by commas. You can put anything you want in a list (including items of different types, and even other lists). And like any other kind of object, you can enter a list at the prompt:

```
>>> ["in", "school"]
['in', 'school']
>>> [1, 2, 3]
[1, 2, 3]
```

The *split* function takes a string and produces a list of strings. Before looking at what we can do with this list, we'll figure out how to get the *help* page for *split*. Just typing *help(split)* isn't good enough; since *split* is a method, Python needs to know the type of its first argument. Simple enough... just provide an argument of the correct type, or the name of that type.

```
>>> help(split) #nope!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'split' is not defined
```

```
>>> help("I'm in graduate school".split) #this works
>>> help(str.split) #so does this

Help on method_descriptor:

split(...)
   S.split([sep[, maxsplit]]) -> list of strings

   Return a list of the words in S, using sep as the
   delimiter string.  If maxsplit is given, at most maxsplit
   splits are done. If sep is not specified or is None, any
   whitespace string is a separator and empty strings are
   removed from the result.
```

Lists

We now know how to split the sentence into a list of words... What can we do with this list?

First, we can compute its length. Guess how?

```
>>> len("I'm in graduate school".split())
4
```

Second, we can look at individual items within the list. For instance, suppose the sentence was prefixed with the identity and gender of its speaker:

```
B-f: I'm in graduate school
```

There are now five 'words'... the first one is the identity tag, and the other four are the actual words.

```
>>> "B-f: I'm in graduate school".split()
['B-f:', "I'm", 'in', 'graduate', 'school']
```

In Python, list elements are numbered starting with 0 and counting up to *len(list) - 1*. We can refer to these elements by number using square brackets, in the format *list[number]*.

```
>>> ['B-f:', "I'm", 'in', 'graduate', 'school'][0]
'B-f:'
>>> ['B-f:', "I'm", 'in', 'graduate', 'school'][1]
"i'm"
>>> ['B-f:', "I'm", 'in', 'graduate', 'school'][2]
'in'
>>> ['B-f:', "I'm", 'in', 'graduate', 'school'][3]
'graduate'
>>> ['B-f:', "I'm", 'in', 'graduate', 'school'][4]
'school'
```

Of course, as always, we can substitute anything whose *value* is a list, and anything whose *value* is a number, and get the same answer.

```
>>> "B-f: I'm in graduate school".split()[2 - 1]
"I'm"
```

We can also use **slices** to pull out a sequence of elements in a row. A list slice has the notation *list[begin:end]* (where *begin* and *end* are numbers). Its value is another list, containing all the items starting at *begin* and ending just before *end*.

```
>>> ['B-f:', "I'm", 'in', 'graduate', 'school'][1:3]
["I'm", 'in']
```

Conveniently, Python lets you leave out the *end* number if you want to grab the whole rest of the list.

```
>>> ['B-f:', "I'm", 'in', 'graduate', 'school'][1:]
["I'm", 'in', 'graduate', 'school']
```

Processing a sentence

Writing a Python script

We're now in a position to do some actual work on our problem... at least on a single sentence! Rather than do this all at the Python prompt, where it's kind of ephemeral, we'll write down our work in a script file so that we can run it over and over.

To start with, create a file named *sentence.py* in your editor. We'll start by putting a simple Python command into this file so we can tell if it's running properly. By convention, this line will be:

```
print("hello world")
```

(You can try this out at the interactive prompt first, if you feel like it.) Save the file. Now open up a terminal and try the following:

```
$ python3 sentence.py
hello world
```

Once you've gotten this working, we can start counting the words in our sentence. We can start with this code:

```
print("Here are all the words", "B-f: I'm in graduate school".split())
print("the sentence has", len("B-f: I'm in graduate school".split()), "words")
```

When we run this, we get the following:

```
$ python3 sentence.py
Here are all the words ['B-f:', "I'm", 'in', 'graduate', 'school']
the sentence has 5 words
```

Variables

Typing the sentence over and over makes our code difficult to adapt to a new sentence. We can make our code more flexible and easier to read using **variables** (covered by the official tutorial back in the section on numbers). A variable is a name which you link to some piece of data in memory. Its *value* is then that piece of data. The equal sign (=) *assigns* a value to a variable.

```
>>> sentence = "B-f: I'm in graduate school" #assign string to sentence variable
>>> sentence #check value of sentence variable; will get string
"B-f: I'm in graduate school"
```

Let's go back to our script. We'll create some variables, *sentence* and *words*, to make things shorter and simpler.

```
sentence = "B-f: I'm in graduate school" #the sentence we're analyzing
words = sentence.split()               #a list of the words

print("Here are all the words", words)

speaker = words[0]
actualWords = words[1:]                #everything after the speaker

print("the sentence is spoken by", speaker)
print("their actual utterance was", actualWords)
print("the sentence has", len(actualWords), "words")
```

This script does a lot of the analysis we need for the male/female study... for a single utterance, anyway. To deal with multiple utterances, we'll need to learn some more sophisticated programming tools. But before we go on, we'll add one more utterance to the program to get an idea of how the full-scale case should look.

To do so, we'll copy and paste the code we already used. When we execute the first line, *sentence* is assigned a new value, "B-f: yeah I can hear you", which overwrites the old value. Now *words* is assigned a new value (the list of words from the *new* sentence). The rest of the code works just as before.

```
#-----------processing sentence 2--------------------

sentence = "B-f: yeah i can hear you"    #another sentence
words = sentence.split()               #change value to words of sentence 2

print("Here are all the words", words)
...
```

We can collect some statistics grouping both sentences together. To do so, we'll create two variables, *totalWordsSpoken* and *totalUtterances*, which will be numbers. As we analyze each sentence, we'll update our running totals to account for the data from that sentence.

To keep track of a running total, we need two ingredients. First, we set the total to 0. And then, we add each term on to the total. Notice that we can use the same variable to do this each time:

```
>>> total = 0
>>> total
0
>>> total = total + 3 #equiv. 0 + 3
>>> total
3
>>> total = total + 2 #equiv. 3 + 2
>>> total
5
```

The pattern $x = x + y$ occurs so often in programs like this that Python provides an abbreviation for it: +=: x += y means $x = x + y$. So, for instance, when we want to update our total number of words to include the most recent utterance, we can do this.

```
totalWordsSpoken += len(actualWords)
```

(By the way, += is just a labor-saving device. There's nothing wrong with continuing to write out the full sum if you're worried you'll forget what it means. Python has a lot of similar labor-saving abbreviations. It's often good programming style to use them, since it can make your code easier for an experienced programmer to read, but you should never feel required to use something you find confusing.)

Program structure

The full program looks like this:

```
#------------initialization of tracking variables------------------
totalWordsSpoken = 0
totalUtterances = 0

#-----------processing sentence 1---------------------

sentence = "B-f: I'm in graduate school" #the sentence we're analyzing
words = sentence.split()               #a list of the words

print("Here are all the words", words)

speaker = words[0]
actualWords = words[1:]                #everything after the speaker

print("the sentence is spoken by", speaker)
print("their actual utterance was", actualWords)
print("the sentence has", len(actualWords), "words")

totalWordsSpoken += len(actualWords)
totalUtterances += 1

#-----------processing sentence 2---------------------

sentence = "B-f: yeah i can hear you"    #another sentence
words = sentence.split()               #change value to words of sentence 2

print("Here are all the words", words)

speaker = words[0]
actualWords = words[1:]                #everything after the speaker

print("the sentence is spoken by", speaker)
print("their actual utterance was", actualWords)
print("the sentence has", len(actualWords), "words")

totalWordsSpoken += len(actualWords)
totalUtterances += 1

#---------done with all the sentences; post-analysis----------

print("the total number of words spoken was", totalWordsSpoken)
print("the total number of utterances was", totalUtterances)
print("the average number of words per utterance was",
      totalWordsSpoken / totalUtterances)
```

The program has a fairly typical structure for this kind of analysis. There's an initialization block at the beginning which sets up our running total variables. Next, there's a processing block which we run once

for each sentence. This block is just copied over and over, once per data item; the only thing that changes is the sentence itself. The processing block is responsible for computing the quantity of interest (words in the sentence) and adding it to the appropriate total variable. Finally, there's an analysis block at the end which uses our running totals to compute something of interest and print it out.

When we run the program, we see this:

```
$ python3 sentence.py
Here are all the words ['B-f:', "i'm", 'in', 'graduate', 'school']
the sentence is spoken by B-f:
their actual utterance was ["i'm", 'in', 'graduate', 'school']
the sentence has 4 words
Here are all the words ['B-f:', 'yeah', 'i', 'can', 'hear', 'you']
the sentence is spoken by B-f:
their actual utterance was ['yeah', 'i', 'can', 'hear', 'you']
the sentence has 5 words
the total number of words spoken was 9
the total number of utterances was 2
the average number of words per utterance was 4.5
```

We can see which output comes from which block.

```
$ python3 sentence.py
  <----initialization ran here and didn't print anything
  <----processing sentence 1 started here
Here are all the words ['B-f:', "i'm", 'in', 'graduate', 'school']
the sentence is spoken by B-f:
their actual utterance was ["i'm", 'in', 'graduate', 'school']
the sentence has 4 words
  <----processing sentence 1 ended here
  <----processing sentence 2 started here
Here are all the words ['B-f:', 'yeah', 'i', 'can', 'hear', 'you']
the sentence is spoken by B-f:
their actual utterance was ['yeah', 'i', 'can', 'hear', 'you']
the sentence has 5 words
  <----processing sentence 2 ended here
  <----postprocessing started here
the total number of words spoken was 9
the total number of utterances was 2
the average number of words per utterance was 4.5
  <----program end
```

This "block" structure isn't actually part of the Python language; it's not something the computer keeps track of while it's running your script. It's part of our mental representation of the program. But thinking about the program this way will be very important when we extend it to deal with the full dataset.