# Introduction to Python (part 2)

## 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.

# Processing a file

At the moment, our program runs on only two sentences, and we had to write these into the code by hand. That isn't very useful. In order to solve our problem, we'll need to do the following things:

- Allow the program to process any number of sentences.
- Read the sentences in from a file.
- Keep track of the number of words by gender, not just the overall total.

## The for loop

The solution to the first problem is called a **loop** and we'll start by covering that.

The most common (and useful) Python loop is the for loop. It has the following syntax:

```
for [variable] in [list]:
    [statements]
```

For every item in *list*, the *for* loop sets *variable* equal to that item and then runs all the *statements*. Here's an example:

```
for number in [1, 2, 3]:
    number *= 2
    print(number)
```

What does this print? You can create equivalent code using the copy-and-paste method.

```
#--------processing 1----------
number = 1
number *= 2
print(number)

#--------processing 2----------
number = 2
number *= 2
print(number)

#--------processing 3----------
number = 3
number *= 2
print(number)
```

Notice that the *statements* under *for* have to be indented. Python uses this indentation to keep track of your program's structure, so it's important to get it right. If you had instead typed:

```
for number in [1, 2, 3]:
    number *= 2
print(number)
```

The program would be equivalent to:

```
#--------processing 1----------
number = 1
number *= 2

#--------processing 2----------
number = 2
number *= 2

#--------processing 3----------
number = 3
number *= 2

#--------after the loop--------
print(number)
```

It's possible (but slightly annoying) to try out for loops in the interpreter. When you type the beginning of the loop, you get a ... prompt instead of the normal >>> prompt.

```
>>> for number in [1, 2, 3]:
...
```

You then need to produce the indentation (you can use the space bar, but the tab key is probably neater).

```
>>> for number in [1, 2, 3]:
...     print(number)
...
```

When you're ready to end the for loop, hit *return* and the whole loop will execute, after which the normal >>> prompt will reappear.

## Exercise

With this in mind, use a for loop to simplify our *sentence.py* script from above. The idea should be to replace the two copies of the *processing* block with a single copy inside a loop. To do this, you'll need to create a list containing the two sentences, write a for loop operating over this list, and put the *processing* block inside that loop. I won't show the code, but don't go on until you can do this.

## File handling

At the moment, we've got our program working, but only on sentences we write into the code by hand. To read files from the corpus directory, we'll need to understand a bit about Python's tools for dealing with files.

A Python **file object** is a datatype, like *string* or *int*. It's possible to do all sorts of things with a file object, but we'll only cover the most basic file operation for now--- reading each line in order.

The first thing to do with your file is **open** it. *Open* is a Python function which takes the *name* of the file (also called its *path*) and returns a *file object* connected to that file. You can think of the file object as if it were an old-fashioned cassette tape player into which you load the actual file in computer memory like a tape. Once the tape is loaded into the player, you can play the content, going forward until you get to the end--- and then you're done. You'd need to rewind the tape to get back to the beginning. Of course, the contents of the tape itself (the actual file on the disk) are still all there.

We can test out valid file names to use in our program by checking them with *ls* at the Unix command line. Python understands both absolute paths (starting with a /, as in */home/YOURNAME/ling/example.txt*) and relative paths from the current directory (no /, or *./*, as in *example.txt*). The only thing *ls* understands that *Python* doesn't is the ~ abbreviation for your home directory--- don't use this in computer code.

For now, our example filename will be *Fisher/065/fe_03_06500.txt*. I've got this in my current directory, which I can check...

```
$ ls Fisher/065/fe_03_06500.txt
Fisher/065/fe_03_06500.txt
```

But of course you might have it somewhere else. Make sure you have a proper path to this file before going on.

Now let's *open* our *file object*:

```
>>> fisherFile = open("Fisher/065/fe_03_06500.txt")
>>> fisherFile
<_io.TextIOWrapper name='Fisher/065/fe_03_06500.txt' mode='r' encoding='UTF-8'>
```

The type of the filename is a string (in double quotes); the type of the *file object* is *_io.TextIOWrapper* (but *file object* is easier to say).

The file object is now connected to the actual file on disk... but how do we get at the contents? There are several ways to do this. But the easiest is to write a for loop. Just like a list, a file can be used inside a for loop.

```
>>> for line in fisherFile:
...     print(line)
...
0.33 2.13 B-m: (( hi how's it going ))
<ETC>
```

This loop will spew the entire contents of the file to your screen. Once the contents of the file has been consumed (the cassette tape player is at the end of the tape), it stops. If you try writing the same loop again, you get nothing. In order to get the whole thing to work again, *open* the file again. The new file object is connected to the file at the beginning, and the for loop will work just as before.

What if the file can't be found? Well, then you get an error.

```
>>> fisherFile = open("nonexistent")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'nonexistent'
```

When this happens, use *ls* to try to figure out what's going on. Remember that relative paths care about what directory you're in when you run the Python program, so if *ls* works and your code doesn't, you might be running them from different places.

Use what you've learned to rewrite your program so that it counts all the words in the example file. The lines in the actual Fisher data have two extra fields which are timestamps, before the speaker, so you'll also need to change some indices to make sure these don't get counted as words.

## Decision-making with if

Now that we can process a whole file, we can actually start to figure out something about gender. To start off with, we'll need some new tracking variables, one set per gender.

```
#------------initialization of tracking variables--------------------
totalWordsM = 0            #words spoken by men
totalUtterancesM = 0
totalWordsF = 0            #words spoken by women
totalUtterancesF = 0
```

The missing piece here, of course, is that inside the *processing* block, we'll need to make a decision--- if the sentence is spoken by a man, we need to update *totalWordsM*, but if it is spoken by a woman, we instead need to update *totalWordsF*.

Python allows you to make this decision using the if statement. It has the following syntax:

```
if [condition]:
   [statements]
```

Just like *for*, *if* has a line ending with a colon, and then an indented block of statements which it controls. In this case, the statements execute if *condition* is true.

In Python, truth values have a type of their own, **bool** (short for Boolean, from the logician George Boole). There are two of them: *True* and *False*. Thus, the simplest *if* statements are:

```
>>> if True:
...    print("most things will never happen; this one will")
...
most things will never happen; this one will
```

```
>>> if False:
...     print("this one won't")
...
>>>
```

Of course, these statements are rather trivial--- why write a statement that can never be executed? Luckily, Python has a variety of logical operators whose *value* is a bool.

## Comparisons

The most important set of operators with boolean values are the *comparison* operations. These are < (less), == (equal) and > (greater). (Why ==? We're already using the single equals sign for variable assignment, as in *sentence = "hello world"*, and using it for both would be confusing. Luckily, if you try to use the wrong one, you usually get an error.)

```
>>> 1 == 1
True
>>> 1 < 5
True
>>> 1 > 5
False
```

These operators also work on non-numerical data:

```
>>> "cat" == "cat"
True
>>> [1, 2, 3] == [1, 2]
False
```

For these data types, <-style comparisons usually get you alphabetical (lexicographic) order. For instance:

```
>>> "anna" < "cat"
True
>>> ["beta", "gamma"] > ["beta", "alpha", "zebra"]
True
```

There are also some compound operators for your convenience: <= (less than or equal), >= (greater than or equal) and *!*= (not equal). Once you're used to the basic comparisons, it's easy to figure out what these do.

## Logic

You can also do logic with booleans. Python provides three logical operators, *and*, *or* and *not*. These do what you think they do.

```
>>> True and False
False
```

Of course, you can also use these operators with expressions whose *value* is boolean.

```
>>> not (5 > len("you are a cat".split()) and [1, 2, 3][2] < 4)
```

What does this print? Why?

## Using 'if' to check the gender

Now we know how to get boolean values, we can use *if* to assign the word counts to the correct gender-based variable. Let's think about the speaker values again--- currently, our program prints out lines like:

```
the sentence is spoken by B-f:
```

We want to know what the third letter of this string is... and we can find that out using the brackets notation, just as if the string were a list. What number do we have to write?

Now we need some code of the form:

```
if [that letter] == "f": #check if that letter is the string 'f'
   [add stuff to the female totals]
```

Obviously, we can do exactly the same thing for the men (except changing the string "f" to "m", or "==" to "!="). Again, there's a shortcut... Python's *else* keyword. This has the syntax:

```
if [condition]:
   #this stuff happens only if *condition* is True
   [statements]
else:
   #this stuff happens only if *condition* is False
   [other statements]
```

*else* can make your code easier to read, but you don't have to use it. Either way, make your code compute all the total variables correctly and then run it.

For this example file, my code prints:

```
the total number of words spoken by women was 1061
the total number of utterances by women was 121
the average number of words per utterance was 8.768595041322314
the total number of words spoken by men was 809
the total number of utterances by men was 126
the average number of words per utterance was 6.420634920634921
```

So it looks like this particular female speaker talks a little bit more than her male counterpart. Of course, this is a single datapoint--- not a full analysis of the problem!

# Processing multiple files

Think about what you'll need to do to make the program run over multiple files. There are two missing pieces:

- Figure out the list of files to run on
- Write another loop to process each one in turn

In just a second, we'll figure out how to do these problems... but what would you do if you were on your own? Use the internet!

Try googling a solution to the file listing problem. Did you find one?

Searching for *python list directory contents* gets this link as the first result: <*http://stackoverflow.com/questions/2759323/how-can-i-list-the-contents-of-a-directory-in-python*> And the top-rated answer is correct

```
import os
os.listdir("path") # returns list
```

But let's spend a few minutes understanding *why* it's correct.

## Modules

Python supplies a variety of libraries called modules which contain extra functionality you might want to use. Some of these are built in; others are developed independently and have to be installed. And you can also write your own!

To use a Python module, you *import* it:

```
>>> import os
```

*import* allows you to use the functions defined in the module. You have to prefix them with the module name to make it clear where they come from. For instance, now that you've imported *os*, you can use the *os.listdir* function. This also allows you to obtain help.

```
>>> help(os.listdir)

Help on built-in function listdir in module posix:

listdir(...)
   listdir([path]) -> list_of_strings

   Return a list containing the names of the entries in the directory.

       path: path of directory to list (default: '.')

   The list is in arbitrary order.  It does not include the special
   entries '.' and '..' even if they are present in the directory.
```

This gives us some understanding of the *listdir* function; we give it a path (like */Users/<YOURNAME>/Ling5050*) and it gives back a list of strings which are the names of files in the directory.

```
>>> os.listdir("/Users/mcdm/Downloads/Fisher")
['058', '065']
```

You could use this function to build an *ls* program if you wanted. But instead, try using it to get a list of all the files in *Fisher/065* so we can process them with our gender counting program.

As a side note, if the module name is long and you want to avoid retyping it, you can use this syntax:

```
>>> from os import listdir
>>> listdir("/Users/mcdm/Downloads/Fisher")
['058', '065']
```

Like all the shortcuts, you never need to use this one--- the longer form is always acceptable.

## Some important modules

It's worth knowing about a few modules that Python programmers use a lot. Here's a quick list--- you can learn more about them with *help*, or read the tutorial page.

**sys:** *sys.argv* gives you access to your program's command line arguments (as in *python myscript.py ARG1 ARG2 ARG3...*).

**re:** Grep-style regular expressions--- hard to debug, but powerful!

**math:** *log*, *exp*, *sqrt*, trigonometry...

**random:** random number generation

**os:** directory and filesystem queries, plus *os.system* for running external programs

## Modifying the program

How do we need to modify our program to make it run over all the files? *os.listdir* gives us a list of files in a directory, and we'll need to process each one in the same way. This means we'll need a new *for* loop. Which parts of our previous program should be inside it? Which parts need to stay outside?

Modify the program to run over all the files in the *065* directory. What are the results? If you're having trouble doing this, read on for a few hints.

One possible issue is that *os.listdir* gives the names of the files in *065*... but you can't open these files directly. (You can tell this by checking with *ls*. Go back and try this now.)

What's wrong? The *open* function needs the full path from the current directory to the file, not just the name. You need to create a filename:

```
>>> fisherDir = "Fisher/065"
>>> fileName = "fe_03_06518.txt"
>>> fisherDir + "/" + fileName
'Fisher/065/fe_03_06518.txt'
```

Another is that the program won't work if you have any junk files in your Fisher directory. If this is the issue, try writing an *if* statement to check if the filename starts with *fe* and ends with *.txt*. What comparisons should you use?

What are your results? Who talks more?

As a final embellishment, try allowing the user to specify the directory name on the command line, like this:

```
$ python3 processDir.py Fisher/065
```

You'll need to use the *sys* module... try searching the web for an example.