Programming in Python for Linguists
A Gentle Introduction

Dirk Hovy
USC's Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

dirkh@isi.edu

May 28, 2012

Contents

1 Introduction

Few, if any, linguistics programs do require an introduction to programming. That is a shame. Programming is a useful tool for linguists, just as word processing or statistics. It can help you do things faster and at larger scale. Most people think of professional hackers when they hear programming, but the level necessary to be useful for linguistic work does not require a special degree. Programming is not a black art. It is a skill that can be learned. Programming is actually very similar to learning a language—one can always get better, learn more, be more proficient. Sure, it does take some time. You can't learn it in a day or a week. But you can already do interesting things after the first few lessons, and – most importantly – it's fun! Running your program and see it produce what you wanted is an amazing experience. You'll see!

This tutorial is designed to do two things: to take you the fear of programming, and to show you how to write code in Python. Instead of presenting each concept as an isolated problem, I will introduce new concepts as part of actual programs. Don't worry if you do not understand the full program at first. It will help you understand the concept better if you see it in action.

Since this is a tutorial for linguists, I will not cover many of the more intricate aspect of programming—they might be indispensable for a professional programmer or computer scientist, but you will probably do fine without them. I will, however, use the official terms. They will be marked in bold and repeated in the margin. That will make it easier for you to search for a problem if you program yourself: there is a lot of information on the web about how to solve pretty much any problem you might encounter. You just have to know what to search for.

It is important that you start programming yourself right away! You cannot learn programming from reading, so feel free to copy the code and play around with it. The more you play, the better! If you would like the code for the scripts in this tutorial, please send me an email.

There will also be little quizzes throughout this tutorial. The answers are at the end. Do not peek ahead, but try to solve the quizzes before you move on: it will help you to assess how much you have really understood. If you feel unsure, go back and reread it, implement it, and make sure you are comfortable with the concept. Some things will come easy, others take time. Don't worry: that's normal. Even many computer scientists take a long time to pick up certain concepts.

If you get stuck, take a break. This is one of the most important principles in programming. But now: let's start!

2 Programming

Programming, irrespective of the language you use, has four main elements:

1. **Assignment**: linking a name to a value. These names are also called **variables**. We will talk about this in section 4.1 on page 9. Assignment variables

2. **Loops**: sometimes we want to do the same thing repeatedly, either a fixed number of times, or until something happens. This is what loops are for. We will look into loops in section 4.2 on page 11. Loops

3. **I/O, Input/Output**: this refers to everything that has to do with getting information into and out of our programs, e.g. files (opening, closing, reading from or writing to them) or output on the screen. We will look at I/O in section 4.3 on page 13 I/O, Input/Output

4. **Control structures**: sometimes, we need to make decisions. I.e., if a variable has a certain Control structures

value, do X, otherwise, do Y. Control structures are simple if...then...else constructs that evaluate the alternatives and make this decision. We will look at this in section 5.1 on page 18.

All problems that can be solved by programming can be decomposed into these four elements!

If you can divide a problem into smaller parts that can be solved with these four techniques, you can write a program that does it for you. It takes some practice and can sometimes seem a little counterintuitive, but you will soon get used to it. After all, as a linguist (or any other kind of researcher), you are used to dissecting problems into smaller parts.

Do not be confused or intimidated by vocabulary. You can express useful things with very few words, and you can program with very few commands. In both cases, you will pick up more expression to express things more elegantly or efficiently as you go along, but it is not that crucial. Programming is first and foremost about how you think about a problem. How you write it down comes much later.

You can –and should!– start at a very high level! Identify what you are provided with in the beginning (your **input**), and what you want to have in the end (the **output**). This is a good starting point, and it is good to keep that in mind when you feel you get lost in the programming details. input output

One of the most trivial, yet best advices for programming comes from the 18th century bishop Joseph Butler:

> Every thing is what it is, and not another thing.

This might sound trite, but I actually use it quite often when I am trying to decide what I need. Write down the different "things" you are dealing with (words, sentences, counts, affixes, morphemes, what-have-you...). Which of these are important? Are they really all distinct, or are any two the same? We will have to represent these important things in our program.

Write down what needs to be done to the things, and in which order, to get from the input to the output. At this level, it does not matter which language you use, or whether you already know all the commands to do it. You should just devise a general plan on how to attack the problem. Write it out, make a flow chart, or describe it to a friend and see whether it makes sense. The process from input to output should be logical even to somebody who does not know how to program.

Keep in mind that there are always several ways in which you can solve a problem, and often, there is no one best way! Don't stress out about details. There might be a faster or a more straightforward way to solve a given problem (they are often not the same), but as long as it gets you from the input to the output and you know what you are doing, your way is the right way! You can always go back and make things faster, better, or prettier.

In this tutorial, we will train this way of thinking by looking at problems from a very high level. We will identify input and output, all the things we need, and all the processes that need to be done, before we look at the Python program that implements it. The programs are written to be as readable as possible—not as efficient as possible. Once you program more, you will automatically learn how to write more efficient code. It might actually be an interesting experience to go back and rewrite the programs we study here.

3 Python

Python was invented by the Dutchman Guido van Rossum and was named after the British comedy group Monty Python. Python is just one language among many, and each has their advantages and disadvantages. Python is very easy to learn and extremely versatile, but it is a lot slower than C or Java (which you will probably not care too much about). The main idea behind Python was that code is read more often than it is written, so it should be easy to understand. The syntax of Python is thus very similar to English.

Python does not put a lot of restrictions on you: one of the core principles is “We are all adults”. It is assumed that you know what you do, and Python will not throw you a wrench. This makes it easy to start, but brings the danger of doing something inefficiently. However, if you follow common sense and some simple ideas (like the bishop’s), you will not have many problems.

3.1 Installing Python

Before you can implement any problems in Python, you need to get it onto your machine. There are different versions of Python. Many Mac and Linux systems already come with Python installed. I will cover Python 2.7 here, but if you already have 2.4 or higher installed, you should be able to use it just fine. Windows usually does not come with Python, so you will have to install it. I will give a brief overview, but since there are many different operating systems and installers, what you have to do might be slightly different. Usually, the packages make it easy to install them, though. When in doubt, follow their instructions and read the help.

3.1.1 Windows

This is the exemplary install process for ActivePython on Windows. Depending on when you get it and what version of Windows you are using, some names might differ.

1. Go to `http://www.activestate.com/activepython/downloads`
2. Search for a version that starts with 2.7 and download the one that is right for your machine
3. Once downloaded, open the installer file by either double-clicking it or selecting ’Open’ in the Download window
4. Follow the instructions and install Python. Remember where it is installed! It will most likely be `C:\Python27`

You should now have a working Python installation. We will also create a shortcut to the editor, so you can access it from the Desktop.

1. Once finished installing, open the Explorer, and go to the directory where Python was installed.
2. Search for the folder `Libs` and go there
3. Search for the folder `site-packages` and go there
4. Search for the folder `pythonwin` and go there. The full path is now something like `C:\Python27\Libs\site-packages\pythonwin`

5. Search for the file `Pythonwin.exe`. If you have extensions turned off, it will just be called `Pythonwin`. It has a little snake as symbol. Right-click that file and select Send To > Desktop (create shortcut)

The little snake symbol should now also appear on your Desktop.

3.1.2 Mac/Linux

You can either use the pre-installed version, or download the Active State version. Follow the same steps as above, up until the shortcut. Unfortunately, ActivePython does not come with an editor for Mac/Linux.

3.2 Using Python

In order to use Python, you will need a text editor (to write your programs) and a command line environment (to execute the program). The files should have the ending `.py`, so that your computer knows they are Python files.

3.2.1 Windows

ActivePython in Windows gives you both an editor and an environment to run it in. In the previous section, we installed a shortcut to PythonWin. If you double-click to open it, you should see a new window with a smaller window inside that says "Interactive Window". You will see the results of your programs here, and you can test out Python commands here, but we will mainly use saved scripts in this tutorial. To open a new file

1. Select "File > New", or press the blank page window in the upper left corner, or press "CTRL+n"
2. Select "Python Script" from the box that pops up

You can write your program in the new window. Save the program in a place where you can find it. You can use something like `C:\python_tutorial`. PythonWin will automatically add the file ending `.py`.

Alternatively, you can write your programs with the text editor and save them with the extension `.py`.

In order to run your file,

1. Select "File > Run", or press the little running man button in the top bar, or press "CTRL+r"
2. Usually, the currently open file is already selected, so you can just press "OK" in the next box. If there are arguments (we'll see that later), you can specify them here in the text box.

You will see the output of your file in the Interactive Window. Any error messages will appear here as well.

You can also run the program from the command line. Go to "Start > Run" and type `command` in the next window, then click "OK". This opens a **DOS prompt**. You can navigate around by using the **`cd`** (change directory) command, and see what files and directories are in your current directory via **`dir`**. Once you have reached the folder with the program you would like to run, type

DOS prompt
`cd`
`dir`

```
python <name>
```

where `<name>` is the name of your program. So, for the file `text_statistics.py`, we would type

```
python text_statistics.py
```

The program would run and print its output to the DOS prompt. If there is an error, it would also occur in the command line.

3.2.2 Mac/Linux

Under Mac or Linux, you can use you favorite editor (I use Smultron or Emacs) to write your code. If it supports Python syntax highlighting, even better. To run your programs, you can use the command line, or shell.

After you have written your program, save it in a folder where you can find it again. I created a special folder called `python_tutorial`, and saved all my files in there. The full path to it is

```
/Users/dirkhovy/python_tutorial
```

To run it, open the command line (in Mac OS X, got to Applications, then Utilities, and look for a program called **Terminal**). If you open the shell, you will most likely be in your home directory. In my case, that would be `/Users/dirkhovy`. Navigate to the folder where you saved your program by using the **`cd`** (change directory) command:

Terminal

`cd`

```
cd python_tutorial
```

In order to run your file, you would use the command **`python`** and the name of your program. So if I wanted to run the program `text_statistics.py`, I would type

`python`

```
python text_statistics.py
```

The program would run and print its output to the command line. If there is an error, it would also occur in the command line.

4 Program 1: Word Statistics — Assignment, For-Loops, and IO

Here is our first problem: we have a text of several sentences and want to know for each sentence how many words and characters there are, and the average word length. Let's first think about the input and output: we have a text, and we want three number for each of the sentences (words, characters, average word length).

So we deal with a number of things here: the text, the sentences, the words, the characters, and their respective counts. Somehow, we need to represent those in the computer, manipulate them, and then report the result.

How could the computer get from a sentence to the numbers? It has to take each sentence, count its words, count its characters, remember those numbers, divide them, and then let us know what it found.

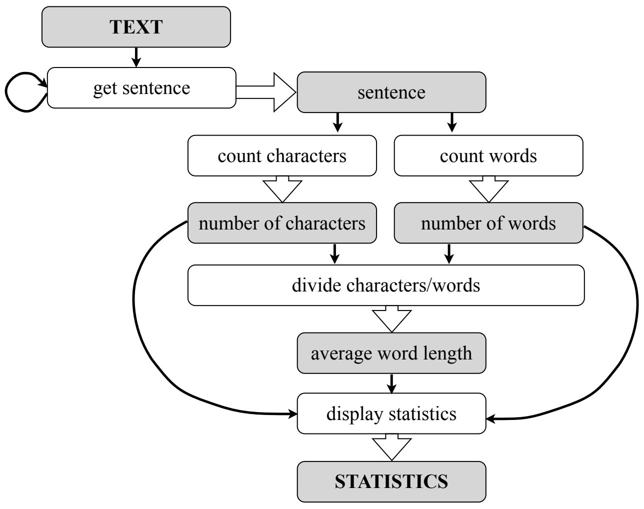

Figure 1: Flow chart for our word statistics problem

At a very high level, what we want to do can be represented as in Figure 1. The input and output are at the top and bottom of the diagram. Shaded boxes denote things we deal with, and white boxes are actions we have to take. The big arrows denote results of an action. The looping arrow indicates that we need to do this step several times. Note that it does not matter whether we first count the characters and then the words or vice versa, but we need both before we can compute the average word length.

Now that we are clear on what we want to do, we can look at how we can do it in Python. It defines a number of sentences, and then counts the number of words and characters in each, as well as the average number of characters per word.

```
###########
# strings #
###########
# strings can be enclosed by double quotes...
sentence1 = "Monty Python is full of memorable quotes."
```

```
# ...or single quotes
sentence2 = 'My hovercraft is full of eels.'
# single quotes can be used within double quotes
sentence3 = "You're all individuals"
# double quotes can be used within single quotes
sentence4 = 'We are the knights that always say "ni!"'
# if we have both, we need to "escape" the outer one with a '\'
sentence_with_escaped_quotes = "Brian: \"You're all individuals\" Voice: \"I'm not!\""

#########
# lists #
#########
mini_text = [sentence1, sentence2, sentence3, sentence4, sentence_with_escaped_quotes]

############
# integers #
############
number_of_sentences = 5

#########
# loops #
#########
# look at each list element in turn
for current_sentence in mini_text:

    number_of_characters = 0
    # go through each character in the string
    for character in current_sentence:
        # increase the character count by 1
        number_of_characters = number_of_characters + 1

    number_of_words = 0
    # split the sentence at spaces into a list and look at each element in turn
    for word in current_sentence.split():
        # increase the word count by one (alternative syntax)
        number_of_words += 1

    # in order to make sure the result is a comma value, we have to "cast"
    average_word_length = float(number_of_characters)/number_of_words

    # print everything out on screen
    print current_sentence
    print "number of words:", number_of_words
    print "number of characters:", number_of_characters
    print "this sentence has about %s characters per word" % (average_word_length)
    print   # this just prints an empty line
```

You can probably figure out what some things do, and are a little confused about others. Don't worry, it will all become clear! The first thing you see is a comment. Whatever you put behind a hash ('#') will be ignored by the program. Use as many comments as possible! They will help you (and others) later on to figure out what your code does. You can put comments on their own line or at the end of a line that contains code (as in line 51).

4.1 Assignment

In line 5 we see the first Python command.

```
sentence1 = "Monty Python is full of memorable quotes."
```

We take the sentence `Monty Python is full of memorable quotes.` and assign it the name `sentence1`. This name is called a **variable**. You can just think of it as a shorthand, so you don't have to type the whole sentence every time, and as a placeholder: you might know there is a sentence1, even if you don't know yet how it goes. Variables are simply the names that refer to a thing. Assignment is the process of attaching the name to the thing. You can change the thing (i.e., the value of the variable) without changing the name of the variable. This is like name reference: the reference to a person does not change, even if the person does. I.e., if Norbert grows a beard, you still refer to him as Norbert. Conversely, you can also give an existing thing a new name, but it won't change the thing itself. So if you want to call Norbert "Nobbs", it does not change him (see Figure 2).[1] Another way to imagine a variable is as a container: it has a label on the outside (the name), and contains some information in it (the value). We can change the contents without changing the label, or slap on a new label without changing the content.

variable

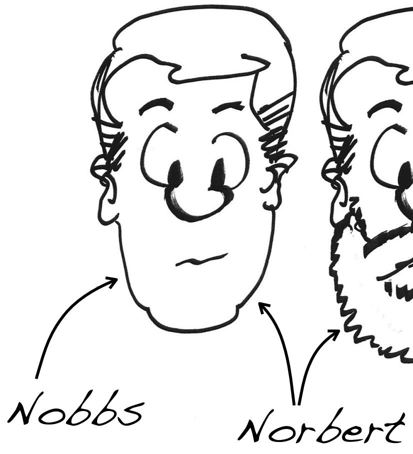

Figure 2: Variables are just names for objects. Changing one does not affect the other!

You can name your variables whatever you want. It makes sense to use a descriptive name, rather than `x` or `y` (you will not remember what those meant 5 weeks down the line). Do not think you'll save time by making your variables short! You will spend more time figuring out what `stuff` does than it took you to write out `number_of_verb_occurrences` every time. Remember: you'll read your code more often than you write it!

There are a few rules for **valid names**:

valid names

[1]Whether he likes that name is a different question...

1. Variables have to start with a letter (you can use all letters from `a-z` and `A-Z`), but can not start with a number.
2. You can use numbers later on in the variable name (just not as first letter).
3. Variable names cannot contain spaces or dashes (those are for subtraction), but you can use underscores (_) to separate words (as I have done with `sentence_with_escaped_quotes` in line 13).
4. Mind your spelling: it makes a difference whether you use a capital letter or not!

In general, people in Python use lowercase for variables, and separate words by underscores.

QUESTION: Which of these are valid names:
1) `quantifier`
2) `parasitic_gap`
3) `2cool4school`
4) `parasitic_Gap`[1]

There are a few words that are part of Python's vocabulary, for example for-loops (section 4.2 on the following page) or control structures (section 5.1 on page 18). These words are printed in **bold** in the code[2]. They are called **reserved words**. You can not use those words as variable names! reserved words

Variables can be all kinds of things: words, numbers, etc. (but only one of those things: remember the bishop!) This is called the variable **type**. It does not make sense, for example, to divide a word by 15, or to count the number of 'a's in $345,956$. They are different **objects**, and thus have different properties. Python tries to figure out what type a variable has based on how you use it. While Python is pretty good at this, it helps to know the different types and use them accordingly. You can imagine variables to be jars with a label, and the contents are different drinks. Python can take a sip and figure out whether it is beer or milk and whether it is ok to use for what you tell it to do with it. E.g., you can empty the milk from the bucket that says "for the cat" and fill it with beer (i.e., change the type of an existing variable), and Python will let you do that as long as you use it for beer-related things from now on, but it will complain when you command it to do something that you can only do with milk, like give it to the cat (i.e., do something that the type does not support, like dividing "coconut" by 42). So you should always know what is in the jar (irrespective of its label), and what you can do with it (or expect Python to do with it).

type

objects

Any sequence of characters is called a **string**. Strings can be single words or sentences. We denote strings by enclosing them with single or double quotes. Which ones you choose is up to you. string

```
sentence1 = "Monty Python is full of memorable quotes."
# ...or single quotes
sentence2 = 'My hovercraft is full of eels.'
```

If you know you have to use a single quote as part of a sentence, you would enclose it with double quotes, and vice versa (see lines 9 and 11):

```
# single quotes can be used within double quotes
sentence3 = "You're all individuals"
# double quotes can be used within single quotes
sentence4 = 'We are the knights that always say "ni!"'
```

[2]If you use an editor that supports Python, those words will appear in a different color

If you need to use both, you have to **escape** the ones inside the string that you also used to mark the string. That is, you put a backslash in front of them to let Python know: this is not where it ends, this is part of the string. escape

```
sentence_with_escaped_quotes = "Brian: \"You're all individuals\" Voice: \"I'm not!\""
```

The backslashes will not appear in the actual string!

If you want to refer to a group of several objects together, you can use a **list** (see line 18). This is like saying "class", when you actually mean "Marc and Gabe and Emily and...". Lists in Python are denoted by comma separated variables that are enclosed by square brackets: list

```
mini_text = [sentence1, sentence2, sentence3, sentence4, sentence_with_escaped_quotes]
```

Lists are a very powerful structure. They can contain all kinds of variable types (strings, numbers, other lists, or mixes of all of the above). In fact, Python actually treats strings as lists of characters. This is very handy for us, since we can go through a sentence one character at a time and use all the fancy functions that exist for lists on strings. We will see more on lists and their properties later on in section 6 on page 20.

Lastly, we can have numbers. There are two different kinds of numbers Python uses: whole numbers, **integers**, and numbers with a decimal point, **floats**. In line 23, we define the value of `number_of_sentences` to be the integer 5: integers floats

```
number_of_sentences = 5
```

You can see more integer variables being assigned in lines 31 and 37. In line 44, we compute the average number of characters per word by dividing one integer by another integer:

```
average_word_length = float(number_of_characters)/number_of_words
```

However, the result will most likely be a real number. We thus have to explicitly tell Python that the result should be a real number by **casting** it to **`float()`**. If we don't do that, Python will round the result to the nearest integer, which is not what we want[3]. `10/3` should be `3.3333`, not just `3`. casting `float()`

QUESTION: What are the types of the following objects:
1) `42`
2) `'jabberwocky'`
3) `42.0`
4) `["Gawain", 'find the Holy Grail', 3, 5, "blue"]`[2]

4.2 Loops

If we want to look at every element of a list, we could simply write some code for each element. This would work for a short list like this where we know all the elements, but it would still take up a lot of space. However, lists are often much longer and generated on-the-fly, so there this approach would break down completely. We'd much rather say something general like "take each element of the list and do the following things to it:..." than "take the first element and do X, then do X to the second element as well, then repeat X for the third element, then...". The general statement is exactly what we do in line 29:

[3]You can turn this off, and later versions of Python are smarter about this, but it is better to be explicit and safe.

```
for current_sentence in mini_text:
```

It tells Python to take each element from `mini_text`, call it `current_sentence`, and then do the things that follow to it. Note that the variable name `current_sentence` never occurred before. We assign it here and use it right away.

This structure is called a **`for`-loop**. Its general **syntax** is simple:

`for`-loop

syntax

```
for <element> in <list>:
    ...
```

This is just a template, `<element>` and `<list>` are placeholder here. You will put in the variable names you want in the code (as in line 29). Note that the words `for` and `in` are reserved words: you can not use them to name any of your variables! The colon at the end of the line is important: it tells Python that everything that comes afterwards is what you want to do with each element.

There is another type of loop, the **`while`-loop**. It just repeats something until a condition is true, and can be used to build a for-loop. In practice, you will rarely need it, so we won't go into any more detail here.

`while`-loop

In the first round (or **iteration**), we take `Monty Python is full of memorable quotes.` and assign it to `current_sentence`. Whenever we use `current_sentence` in the following lines, it will thus automatically refer to the sentence we are currently looking at. The beauty of a for-loop is that we do not have to write out that assignment every iteration! The for-loop changes the value of `current_sentence` for us. Again, this is why variables are useful: the object changes, but the name you use to refer to that new object stays the same. If you are a TA, you could have a system for picking whom to quiz each lessen. It might go something like "for each class in this semester, ask the second student from the left in the first row a question". This is essentially a for loop. There might be a different student in that position every class, but that does not change your system of whom to ask!

iteration

QUESTION: What is the list of the for-loop in the TA example?[3]

QUESTION: What is the value of `current_sentence` in the second iteration?[4]

Everything that is indented after the for-loop (in lines 30–51) is part of each iteration of the for-loop, i.e., these are all things that we do with each element of the list. Python uses indentation to mark these **blocks** (other languages use some form of brackets). It makes things more readable, but it also means that you have to be careful what you indent and how far! Once we write a line that has the same indentation as the `for`, it means that the loop is over. This line is only executed once we have finished going through all elements in our for-loop. There is nothing after this for-loop in this program. Once we have reached the end of the loop, we simply quit. You can tell by looking at the indentation: there is no line that starts at the same level as the for-loop.

blocks

In line 33–35, we have another for-loop.

```
    for character in current_sentence:
        # increase the character count by 1
        number_of_characters = number_of_characters + 1
```

Note that this loop is further indented than the previous loop, so it is part of that previous loop. Here, we iterate over the string that was assigned to `current_sentence` in this iteration. Remember:

Python treats strings as list of characters. Here, we take each character of the sentence in turn, assign it to the variable `character`, and then increment the value of `number_of_characters`, which we instantiated in line 31. Note that the right side of the equation (`number_of_characters + 1`) is executed first, and then assigned to the name `number_of_characters` on the left. Note also that line 31 is inside the first for-loop (it is further indented). That means that at the beginning of each sentence, we set the character count to 0.
Once we have reached the end of the sentence, `number_of_characters` is equal to the length of the sentence in characters.

The loop in lines 39–41 is similar, but counts the words in the current sentence (again, it is part of the outer for-loop that iterates over the sentences).

```
for word in current_sentence.split():
    # increase the word count by one (alternative syntax)
    number_of_words += 1
```

It is different in two aspects, though. First, it uses the command **`split()`**. This takes a string and splits it along the spaces into a list. The first sentence `Monty Python is full of memorable quotes.` is thus turned into the list `["Monty", "Python", "is", "full", "of", "memorable", "quotes."]`. The resulting list is the list the for-loop in line 39 iterates over. `split()`

QUESTION: What is the value of `word` in the second iteration over the first sentence?[5]

Secondly, we use a shorthand to increment the count of `number_of_words`. Instead of `number_of_words = number_of_words + 1`, as we have done in line 35, we simply write `number_of_words += 1`. It does exactly the same (increment the value of `number_of_words` by one), but is shorter to write. You can use either way, but it is good to know that there are variants.

You will notice that we used a dot between the variable name and the `split()` command. The dot means that we make use of a property of the thing before the dot. This is like a possessive marker. It's like saying `norbert.beard()` instead of "Norbert's beard". Note that not every object has the same properties (just as not everybody has a beard). You can't, for example, write `number_of_words.split()`, because `number_of_words` is a number, and you cannot split numbers along white space (since they don't have white space)!

4.3 Input/Output

We have iterated over all words and all characters in all five sentences and computed the values. Now we want to do something with them. The easiest option is to print them to the screen. We can always print variables to see what their current value is by using the **`print`** command: `print`

```
print current_sentence
```

This simply outputs the current value of that variable, whatever it is. As we loop through the different sentences, the value of `current_sentence` changes, and `print` will always output the value that is currently assigned to `current_sentence`.

Lines 48 and 49 first print the string we supply (i.e., `"number of words:"` and `"number of characters:"`, respectively), and then the current value of the respective variable after the comma (i.e., `number_of_words` or `number_of_characters`).

```
    print current_sentence "number of words:", number_of_words
    print "number of characters:", number_of_characters
```

The comma simply inserts a space between the string and the value of the variable (which is an integer).

If we want to format a sentence so that certain values occur in certain places, we can also use the placeholder format in line 50:

```
    print "this sentence has about %s characters per word" % (average_word_length)
```

For every `%s` in the string, Python takes a value from the list supplied after the `%`. The number of `%s`s thus has to match the number of values! Here, we simply replace the `%s` with the current value of `average_word_length`.

For the first sentence, the output to the screen looks like this:

```
Monty Python is full of memorable quotes.
number of words: 7
number of characters: 41
this sentence has about 5.85714285714 characters per word

```

Note the blank line at the end.

And that was it! In this program, we have seen how assignment of different variable types works (strings, lists, integers, and floats), how to loop over a list, and how to print something to the screen.

There are some more variable types we have not yet seen, as well as the very useful control structures. We will look at both in the next section. Before you move on, make sure that you have understood all concepts that were introduced here.

5 Program 2: Word Counts — Dictionaries and Control Structures

In the last program, we have learned about variable assignment, loops, and printing to the screen. There are three more variable types we have not yet covered, and a very useful construct that lets us set conditions. We will see them in our second program, as well as IO for reading from files.

The second problem is somewhat simpler: we want to know how often each word occurs in a file. Again, let's first think about what we have and what we want. We have a file, and we want the counts for the words in there. So there is a file, sentences, words, and counts. We need to read the file, get the sentences, for each sentence get the words, and somehow record their counts. In the end, we just print out the counts again. We can display this like in Figure 3.

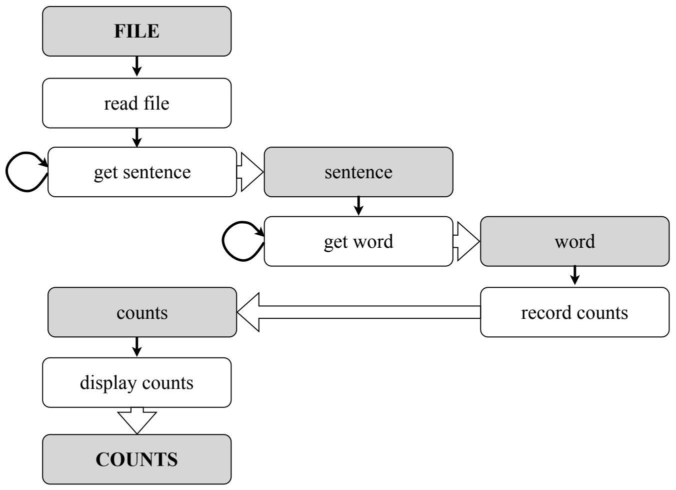

Figure 3: Flow diagram for our word count problem

Now let's look at the program: it takes a file, reads it in, keeps a running count for each word, and prints those counts at the end.

```
# if we want to give our script parameters, we need a special library
import sys

# sys has a list of all arguments given to the script
file_name = sys.argv[1]

# open the file for reading
text_file = open(file_name, 'rU')

word_count = {}

# go through the open file line by line
for line in text_file:
    # get rid of the line break at the end
```

```
15      line = line.strip()
16
17      # split sentence along spaces
18      sentence = line.split()
19
20      # go through words
21      for word in sentence:
22          # check whether word is already in dictionary
23          if word in word_count:
24              # if yes: increment
25              word_count[word] += 1
26          # if not, add an entry
27          else:
28              word_count[word] = 1
29
30  # close the file after reading
31  text_file.close()
32
33  # take each pair of word and frequency in the dictionary
34  for (word, frequency) in word_count.iteritems():
35      print word, frequency
```

argument

This program is run with an **argument**: we give it the name of a file for which we want the word counts. E.g., if you wanted to count the words in a file `example.txt`, you would run the program from the command line by typing

```
python word_count.py example.txt
```

If we want to access any arguments that we have given to the program, we need a special library. It contains commands to deal with files, and makes our lives a lot easier. In line 2, we tell the program to use that library:

```
import sys
```

This library has a variable called `argv`. It's a list that contains all the strings that were given as arguments to the program. The first element of the list is the program itself, the second is the first argument, and so forth. To get the first argument, we do this:

```
file_name = sys.argv[1]
```

It tells the program to go to `sys` and look up its variable `argv`. We access the `argv` list with the dot notation that we have seen in the last program, and then read in the element at position 1. The general syntax to access a the list element at a position (or **index**) is this:

index

```
<list>[<position>]
```

0-based

A curious thing about programming languages in general is that they are **0-based**, i.e., they start counting at 0. The first element in our list is thus at position 0^{th} element, the second element is at position 1, the third at position 2, and so forth. This is confusing, and you will make some mistakes with this. Don't worry, everybody does. And nobody thinks it makes sense in the beginning. It is just something that you get used to.

QUESTION: Which word from the list `hungarian_phrase_book = ['My', 'hovercraft', 'is', 'full', 'of', 'eels']` do you get when you type `hungarian_phrase_book[1]`?[6]

Since `argv` is a list and we want the second element, we can select the string at position 1 (because of the 0-based thing), and finally assign that to the variable `file_name` (line 5). The beauty of this is that we do not have to know in advance what the file we read is called! Python will find it out when we start the program and assign the value, whatever it is, to the variable. We can write our program and simply use the variable `file_name` in our code whenever we mean "the file that we run the program on". This is why variables are so practical as placeholders!

In line 8, we actually open the file (so far, we had only retrieved its name):

```
text_file = open(file_name, 'rU')
```

`open()`

`open()` is the function that reads in the file. It takes two arguments: the first is the name of the file we try to open, the second tells it what we want to do with it. Here, we want to read, so we use `r`. The `U` takes care of some pesky new line issues, so it is good to throw it in. We won't worry too much about it here. The result of running `open()` is a list, and we call that list `text_file`, so we can use it later on.

In line 10, we assign the name `word_count` to a dictionary:

```
word_count = {}
```

Dictionaries
key
value
maps

Dictionaries map from a **key** to a **value** (some other programming languages thus call them **maps**). Here, our keys are strings (the words), and the values we map them to are numbers (their respective frequencies). The curly braces tell Python that this is a dictionary. If we just use those braces, as we did here, we get an empty dictionary. There are no entries. If we wanted to start with a dictionary that already has some entries, we would use the following syntax:

```
<dictionary> = {<key1>:value1, <key2>:value2, ... <keyN>:valueN}
```

For example, we could already put the words "Python" and "linguist" in there and give them some counts:

```
word_count = {"Python":42, "linguist":12}
```

Note that the keys are enclosed by quotes (because they are strings), and the values are not (because they are integers). If you enclosed the values by quotes as well, they would also become strings, and you could not add anything to them later! Anything enclosed by quotes is a string, even digits.

In line 13, we start iterating through the file:

```
for line in text_file:
```

Since Python figures out that `text_file` is an open file, this gives us a list of all the lines. We can thus iterate over them with a for-loop. For each line in the file, we want to do a number of things. That is why lines 14–28 are all indented under the for-loop. First, we get rid of the line break and any white space at the beginning or end of the line. We could write code to do all these things, but there is an easier (and shorter) way:

```
    line = line.strip()
```

We use the **`strip()`** command (line 15) to remove all that whitespace from the line and assign it to the same name as before (`line`). So we changed the object, but not the name (remember Norbert's beard). Whenever we use `line` from now on, it is the "cleaned-up" version of the line. This is handy, since we use the `split()` command we have seen before (line 18).

`strip()`

```
    sentence = line.split()
```

Remember, it splits a sentence along the white spaces into a list, so if we had extra white spaces, it would create empty entries in our list. The list of strings resulting from `split()` is assigned to `sentence`, and we then iterate over that list. We have seen this before, so I will skip to the next interesting part here: control structures.

5.1 Control Structures

So far, we have simply executed one command after the next. We never had to make a decision or choose among options. Now we have to. If a word is already listed in our dictionary, we want to increase its count by one (we know how to do that). If the word is not in our dictionary, however, we have to make an entry. Otherwise, we would try to increment a count that does not even exist (you cannot look up something that is not in the dictionary).

To make the decision what to do, we use the **if...then...else** structure or **conditional** (lines 23–28):

if...then...else
conditional

```
    if word in word_count:
        # if yes: increment
        word_count[word] += 1
    # if not, add an entry
    else:
        word_count[word] = 1
```

The structure of the conditional is simple:

```
if <condition is true>:
    <action1>
else:
    <action2>
```

Here `<condition>` is another type of variable, a so-called **boolean**. They are named after the mathematician Boole, and have only two values: `True` and `False` (note the capital spelling!). In our case, the value comes from the outcome of the condition `word in word_count` to check whether the dictionary `word_count` contains the key `word`. **`in`** is one of Python's reserved words. You can use it to check whether a variable is in a dictionary, a list, or other **collections**.

boolean

`in`
collections

Sometimes, there are more than just two cases (something being true or false) that we would like to account for. In that case, we can check for more conditions:

```
if <condition1 is true>:
    <action1>
elif <condition2 is true>:
    <action2>
else:
    <action3>
```

You can add as many `elif` cases as you want! We will see an example of this in the next section.

To access the value of a dictionary key, we use the bracketed index notation we have also seen for lists:

```
<dictionary>[<key>] = <value>
<dictionary>[<key>] += <value>
```

The index in dictionaries is the key, not a position, as in list. In our case, it's the word we are counting.

So if the word we look at is indeed in our dictionary, we increment its count by one (line 25). In line 28, we add a new item to our dictionary, simply by assigning a the word entry a value:

```
        word_count[word] = 1
```

This puts the current word in the dictionary and sets its counter to 1.

After we have read the whole file, we close it (line 31).

```
text_file.close()
```

We use the **`close()`** command on the file variable. The dot tells us that it is a property of files. Note that this line is no longer indented under the for-loop, but at the same level. This means that it is only executed once we have completed all our iterations of the for-loop, in this case, after we have read all lines in the file! `close()`

Finally, we want to print all the counts we have collected to the screen. We use another for-loop (lines 34–35).

```
for (word, frequency) in word_count.iteritems():
    print word, frequency
```

This time, it iterates over a list of **tuples**. Tuples are a lot like lists, with the big difference that they have a fixed size. They are less flexible than lists. In Python, we denote tuples by round brackets (instead of square ones as for lists). The function **`iteritems()`** of a dictionary returns a list of tuples of each key and its respective value. We print each word and its frequency separated by a space (that is why there is a comma, see above). tuples iteritems()

QUESTION: How would you format the last line to print `The word "Python" has frequency 42`[7]

In this program, we have learned how to read in a file, how to use control structures, and we have seen the new variable types dictionaries, booleans and tuples.

You have now seen all the basics. Really, that's it! All other programs can be written with these elements. Everything you learn from now on will only make your programs faster and more powerful. And there is always something more you can learn. Welcome to programming in Python!

6 Program 3: Affixes — More on Lists

Lists are very powerful and can be used for a lot of things. Also, remember that strings are lists, too. In this section, we will make use of this, and of a number of other list properties, to count how often a specific affix occurs in a text.[4]

QUESTION: What are the input and output of this problem?[8]

QUESTION: Describe the steps we need to take to solve the problem.[9]

QUESTION: What are the elements we deal with in this problem?[10]

The program takes a text and the affix we are interested in as input.

```
python affixes.py <text file> <affix>
```

Affixes come in (at least) two flavors: prefixes and suffixes. You can tell the program to search for prefixes by adding a dash after the affix, and for suffixes by adding a dash in front of the affix. So this:

```
python affixes.py some_file.txt -ery
```

would search for all occurrences of the suffix "-ery" in the file `some_file.txt`. If you want to search for the prefix "anti-" you would use this:

```
python affixes.py some_file.txt anti-
```

Here is the program that does it:

```python
import sys
# we will use a special dictionary that makes counting easier
from collections import defaultdict

# this program has two parameters: the text to search and the affix
text = sys.argv[1]
affix_searched = sys.argv[2]

# check whether affix starts or ends with a dash
# if the first character is a dash, it's a suffix: -ery
if affix_searched[0] == '-':
    affix_type = "suffix"
    # get the affix without the dash
    affix = affix_searched[1:]
# if the last character is a dash, it's a prefix: un-
elif affix_searched[-1] == '-':
    affix_type = "prefix"
```

[4]I had to do this manually for my first term paper, and it has haunted me ever since. With this program, I could have done the data collection in an afternoon and spent more time writing. Which would have been a Good Thing. In many ways...

```
    affix = affix_searched[:-1]
# otherwise, there is a problem
else:
    # print an error message
    print "Affix type not recognized. Must start or end with '-'"
    # ... and quit
    sys.exit()

# the length
affix_length = len(affix)

# every key starts with count 0
affix_counts = defaultdict(int)

# go through the file
for line in open(text, 'rU'):
    line = line.strip()

    for word in line.split():
        # check whether the word starts with the prefix
        if affix_type == "prefix" and word[ :affix_length] == affix:
            affix_counts[affix] += 1
        # or ends with the suffix
        elif affix_type == "suffix" and word[-affix_length: ] == affix:
            affix_counts[affix] += 1

print "The %s '%s' occurs %s times" % (affix_type, affix, affix_counts[affix])
```

In the previous program, we have seen the index notation for dictionaries to access a specific element.

```
word_count[word] = 1
```

Let's look at our program. We first import the special library `sys` in order to deal with the arguments. We have seen that before. Then we import something else: it is a special dictionary that is part of another library:

```
from collections import defaultdict
```

This tells Python to go to the library `collections` and search for an object called `defaultdict`. Once it has found it, we want to use it in this program. There are many libraries in Python, and they all have wonderful things that make your life easier. You just have to know where they are. That is part of learning the language...

Then we assign our arguments to two variables, `text` and `affix_searched`. They are both strings stored in `sys.argv`. We have seen this in the last program.

So far, we don't know whether the program is supposed to look for a prefix or suffix. In order to find out, we look at the first character of the affix argument we got, and check whether it is a dash:

```
if affix_searched[0] == '-':
```

Here, we make use of the fact that strings are lists. `affix_searched[0]` gives us the first character of the argument (remember that lists are 0-based). Remember the `if` control structure? It tests whether a condition is true. Here, our condition is that the first character is a dash. If yes, then we have a suffix like "-ery". In order to test whether an object is the same as another object, we use the double equals sign (==) in Python. It's not a typo! This ensures that it is different from an assignment, where we only use one equals sign! Do not use one equals sign if you want to test for equality of two things! This is a common mistake, everybody has done it, and you will probably end up doing it at least once. Don't beat yourself up if it happens: now you know what to look for.[5] For example, `favorite_color = "blue"` is an assignment. It sets `favorite_color` to the string value `blue`. This statement is by default always true.[6] However, `favorite_color == "blue"`, with two equal signs, is an **equality check**. It checks whether `favorite_color` indeed has the value `blue`. This is of course not always true. So if you type one equals sign instead of two, you make and assignment, and that leads to your if-conditional always being true. You might not want that...

equality check

QUESTION: Which of the following expressions are assignments, and which ones are equality checks?
1) `favorite_color = "yellow"`
2) `favorite_color == blue`
3) `favorite_color == "blue"`
4) `favorite_color == 42`
5) `favorite_color = 42` [11]

So, if we have a suffix, we set the type and get the suffix morpheme without the dash:

```
    affix_type = "suffix"
    # get the affix without the dash
    affix = affix_searched[1:]
```

Note the notation of the last line here. We know that strings are list, and how to get an element from a list. But now, we want more than one element: we want everything after the dash. The colon operator (`:`) tells Python that we want not a single element, but several. This is called a **slice** (because we slice something out of the list). The syntax of the slice operator is this:

slice

```
<list>[<start> : <end>]
```

If we do not specify a value for the end, then the slice spans the whole list.

QUESTION: Which words from the string `hungarian_phrase_book = 'My hovercraft is full of eels'` do you get when you type `hungarian_phrase_book[3:]`?[12]

If we don't have a suffix, we might be dealing with a prefix, like "anti-". To check, we have to look at the last character and check whether it is a dash. Luckily, there is a special index that let's us access the last element:

[5]And yes, it was a stupid idea to make two different things so similar!
[6]This is a case of a speech act where "saying makes it so"...

```
elif affix_searched[-1] == '-':
```

The index syntax to access the character is the same as before, but there is a special trick to looking at the last character: we use `-1` as index. This tells Python to look at the last character. In fact, every index that starts with a `-` is counted from the back.[7]

QUESTION: Which word from the list `hungarian_phrase_book = ['My', 'hovercraft', 'is', 'full', 'of', 'eels']` do you get when you type `hungarian_phrase_book[-3]`?[13]

Note that we use the `elif` keyword. This means there is more than one alternative.

So we have checked for prefixes and affixes. But what if the user did not use a dash? Maybe he or she forgot, or mistyped. In these cases we don't know what to search for, so we might as well stop. It is common to print some error message that helps the user avoid the mistake the next time. We do this in lines 21–24.

```
else:
    # print an error message
    print "Affix type not recognized. Must start or end with '-'"
    # ... and quit
    sys.exit()
```

Here, we use another command from the `sys` library, **`exit()`**. It simply quits the program at that point. `exit()`

We then get the length of the affix (without the dash), using

```
length = len(affix)
```

The **`len()`** command can be used on both strings and lists, and returns the length as an integer. `len()`

Next, we assign our special dictionary to the variable `affix_counts`:

```
affix_counts = defaultdict(int)
```

Remember how we had to check whether a certain key was in our dictionary in the last program? `defaultdict` frees us of that check. If a key is not in the dictionary, it returns some default value, depending on the type we use for the dictionary values. Here, we use integers (`int`), and the default value is simply `0`.

You can also use `float` (which we have seen before as a cast), `list` or `str` (for strings). The default value for non-existent keys in `defaultdict(float)` is `0.0` , for `defaultdict(list)` it's an empty list (`[]`), and for `defaultdict(str)` it's an empty string (”).

Now we are ready to go through our file line by line, strip all white space, and look at each word in turn. We have seen all of that before. We change two little things that save us a line each. In line 33, we open the file and start searching it right away, instead of first opening it and then iterating. Instead of this:

[7]Only this time, we do not start at 0.

```
file = open(text, 'rU')
for line in file:
```

we now do this:

```
for line in open(text, 'rU'):
```

Likewise, instead of first splitting the line into a list and then using that list in the for-loop, as we have done previously, we do both things at once in line 36. So instead of this:

```
    sentence = line.split()
    for word in sentence:
```

we now do this:

```
    for word in line.split():
```

The first variant in both cases is more comprehensive, but both versions are perfectly fine. Again, it is good to know that you can do both.

In lines 38–42, we have another if-conditional to see if the word contains our affix.

```
        if affix_type == "prefix" and word[ :affix_length] == affix:
            affix_counts[affix] += 1
            # or ends with the suffix
        elif affix_type == "suffix" and word[-affix_length: ] == affix:
            affix_counts[affix] += 1
```

We have two cases we want to check for: if we are looking for a prefix, we need to check the beginning of each word, if we are looking for a suffix, we need to check the end. Note that there are two conditions in both cases: the affix type has to be one thing, and the according end of the word needs to match our affix. We do not care if the word started with our affix, but we are searching for a suffix. Both conditions have to be true for us to count the word as an occurrence of the affix. To express this joint condition, we use the reserved word **`and`**. If one of them was sufficient, we could have used **`or`**. `and` `or`

As before, we use the double equals sign to check for equality! If we forgot and did this:

```
        if affix_type = "prefix" and word[ :affix_length] == affix:
```

we would have set `affix_type` to `"prefix"`, which is now automatically true, and would then proceed to check whether the first few characters match our affix. This is pretty bad if we are actually looking for a suffix, because the program will never bother to check for them (after all, we just set the type to prefixes). In these cases, you have not done something illegal, so Python will not throw an error, but it is clearly pretty catastrophal in terms of our result. So be careful with the equal signs...

Here we also see another advantage of `defaultdict`. If we add something to a non-existent key, `defaultdict` automatically creates that key, sets it to `0`, and then adds `1`.

We have not assigned the open file a special variable, so we don't have to close it specifically, as we did in the last program. After we are done with the file, we simply output our result using the formatted print statement we have seen before:

```
print "The %s '%s' occurs %s times" % (affix_type, affix, affix_counts[affix])
```

QUESTION: What do you think the line

```
print "The %s '%s' occurs %s times" % (affix_type, affix,
affix_counts[affix])
```

produces if we found "-ery" 28 times in our text?[14]

The only difference to the way we used the formatted printing before is that this time, we have several placeholders instead of just one.

In this section, you have seen the `len()` operator, `defaultdict`s, the slice operator for lists, how to access elements from the back of a list, how to check for equality, creating joint conditions, and how to use implicit assignments in your for-loops.

7 Program 4: Split — Functions and Syntactic Sugar

We have seen that we can do pretty much anything we want with just the four basic structures of programming. However, it might be inconvenient to type out something that you use frequently. For these actions, there are often abbreviations. I.e., somebody wrote a command that does all the things you want, but you only have to type one word. This is called **syntactic sugar**. You have already encountered a number of these abbreviations, for example `strip()` or `split()`. The latter takes a sentence and returns a list of the words. It is convenient to just use that one word, but it really is a **function** that uses all the things we have learned so far.

syntactic sugar

function

A function is like a mini-program: it gets and input and produces an output, and uses the simple principles we have learned to get from one to the other. Here, the input is a string, and the output is a list of strings. We deal with words and characters. So underneath, `split()` looks something like this:[8]

```python
def my_split(sentence):
    # the result of this function is a list
    result = []

    # we need to keep track of each word
    current_word = ''

    # iterate over all characters in the input
    for character in sentence:
        # if we find a space
        if character == ' ':
            # add the current word to the result list
            result.append(current_word)
            # ... and reset the current word
            current_word = ''
        # for all other characters
        else:
            # add it to the current word
            current_word += character

    # check whether we have a word at the end
    if current_word is not '':
        result.append(current_word)

    # return the list we created
    return result
```

`def` stands for definition. We tell Python that we create our own command here, and that we name it `my_split`. It takes an input, which we call `sentence`. This tells the function what to split: it is called an **argument** of the function. To call our function, we would use `my_split(sentence)`.[9]

argument

[8]The actual function is somewhat more complicated.

The syntax for defining your own function is

```
def <function name>(<argument1>, <argument2>, ..., <argumentN>):
```

You can define as many arguments as necessary (or none, if your function needs no arguments). Everything the function does is indented under it.
You have seen all of the steps in here before, but let's step through them quickly. In the end, we want a list, so we first create an empty one and assign it to a variable `result`. We then iterate over the input sentence, and every time we see a space, we add the current word to our results list and start a new word. There is probably no space at the end of the sentence, so once we have reached the end of the for-loop (line 22), we check whether there is a current word, and if so, add it as last item to our list. If we didn't, then `"This is an ex-parrot"` would only be split into `["This", "is", "an"]` instead of `["This", "is", "an", "ex-parrot"]`.

Note that we use the reserved words **`is`** and **`not`** to form our condition. You have not seen them before, but they are pretty straightforward: you would use them just as you would in English. `is` `not`

The only new command we have used here is **`append()`**. This is another property of lists. It simply takes whatever argument you give it and puts it at the end of the list. This is extremely practical: you do not have to specify your list in the code, but can build it up as you go along! The syntax is simple: `append()`

```
<list>.append(<element>)
```

You can probably guess that this command can be re-written with our simple basics as well...

If we call **`split()`**, we expect to get a list. That is why we have to use the **`return`** command. It lets us do something like this `split()` `return`

```
words = my_split(sentence)
```

Here, we take the list that we get from calling our function on `sentence` and assign it to `words`.
As you can see, the simple commands we have covered (assignment, loops, conditionals, and IO) can be combined in ever new ways. All the fancy one-word commands we have used are really just combinations of those simple elements. You now know how to write functions, so you can create your own shorthand commands. It will make your code a lot easier and more readable.

[9] Note that the regular `split()` is a property of strings, so we don't need an argument, but can simply use the dot notation `sentence.split()`.

8 Debugging

So far, we have pretended that everything works right away. But, let's face it: programs rarely work the first time you try to run them. Not because you didn't wanted them to work... But it is easy to forget to put one assignment in (that you of course had planned to put there), or maybe you got distracted and typed one equals instead of two when you wanted to check for equality, and now Python complains.

Usually, it will tell you where the error occurred, and often a little bit about why it did. This will look for example like this:

```
Traceback (most recent call last):
  File "error.py", line 14, in <module>
    half = affix/2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

You can ignore the first line, it does not tell you anything interesting. The second line tells you where exactly the error occurred (in the file `error.py`, in line 14), and the the third line shows you the line of code that produced it (`half = affix/2`). Sometimes, you can guess what went wrong just by looking at this. Maybe you misspelled a variable, or you forgot to delete a line that you actually wanted to take out. In this case, there is nothing obviously wrong.

Luckily, the last line tells you what kind of error it was. Here, I made a type error. What I wanted to do, was to compute half the length of the variable `affix` and assign that value to `half`. While the idea is clear, I failed to remember the bishop! `affix` has the type string (it is probably a word), not a number. You can do a lot of things with strings, but you can not divide them by a number (what is "spam" divided by 5?).

What I really wanted was to divide the *length* of the string in `affix` by two. This is easy enough to fix, I just have to get the length first.

QUESTION: How do we get the length of a string? How would you rewrite this line?[15]

If you cannot figure out what the error is despite the information, you can use several steps:

1. Google the problem, especially the last line. There are many **error codes**, and you are not the first person to get this one. Somebody probably has figured out what it means and written about how to solve it.

   error codes

2. Remember the bishop! Make sure everything is what you think it is. The easiest way to check this is via `print`. In the example, I could have added `print affix` before line 14, so I would have seen that `affix` is in fact a string, not a number.

3. Add checks an balances. Sometimes, you don't know everything that might happen, but you want to make sure that whatever it is, it fulfills certain conditions. I might want to make sure that my affix has at least two characters before I divide it. To do that, we can use **`assert`**. It tells Python to stop if something is not as expected. The syntax is simple:

   `assert`

   ```
   assert <condition that must be true>, <error message>
   ```

   The error message is optional, but it makes sense to put it in there. So if I added this to my code:

```
assert len(affix) > 1, "affix needs to have at least two characters"
```

and I encounter an affix that has only length 1, I would see this error:

```
Traceback (most recent call last):
  File "error.py", line 14, in <module>
    assert len(affix) > 2, "affix needs to have at least two characters"
AssertionError: affix needs to have at least two characters
```

Where the error occurred is not so interesting now (after all, we put the statement there to create an error message), but we now know that we encountered an affix that was not what we expected.
It would be even more helpful to get the value of the affix, something like:

```
Traceback (most recent call last):
  File "error.py", line 14, in <module>
    assert len(affix) > 2, "affix needs to have at least two characters"
AssertionError: affix needs to have at least two characters, encountered ’s’
```

QUESTION: How would you do that? Think about how we formatted strings earlier...[16]

9 Further Reading

There are tons of books on Python out there. In general, all books on programming published by O'Reilly are a good read, and usually very beginner-friendly.

There is a website by Ron Zacharski, called Python for Linguists (`http://www.zacharski.org/books/python-for-linguists/`), which covers similar ground as this tutorial. It is much more in-depth, and makes a great complimentary read. The tutorial is still a draft, but already lets you download the chapters and programs.

After you feel more comfortable with the basic concepts, you should check out `http://www.diveintopython.net/`, which gives a general introduction. It is more technical, but very thorough. It uses the same approach we have seen here, where concepts are introduced as parts of larger programs. The problems might be a little more abstract, but the coverage is great. One of the best books out there. You can access the PDF and the programs from the website.

If you want to dive even deeper into Python and also explore a useful area of programming that has an enormous impact on all kinds of subjects, check out "Machine Learning—An Algorithmic Perspective" by Stephen Marsland. It is not only the most readable book on the subject, it also gives you Python implementations of all major techniques. This requires some math, but will give you a great understanding of this exciting area.

Notes

[1]ANSWER: 1,2, and 4 are valid names. 3 is not, because it starts with a number. Note that 2 and 4 are different variable names, because of the uppercase `G` in 4!

[2]ANSWER: 1) integer, 2) string, 3) float (you can tell by the decimal point), 4) list (note that the list contains both strings and numbers)

[3]ANSWER: The list of classes in that semester.

[4]ANSWER: `My hovercraft is full of eels.`

[5]ANSWER: `Python`

[6]ANSWER: `hovercraft`

[7]ANSWER: `print "The word %s has frequency %s" % (word, frequency)`

[8]ANSWER: We have two inputs: a file and an affix. Our output is simply the frequency of that affix.

[9]ANSWER: We have to check for each word in the file whether it contains the affix, and if so, record the count. In the end, we print the resulting number.

[10]ANSWER: The text, its words (maybe you want to include the sentences as well), the affix, and the count.

[11]ANSWER: 1 and 5 are assignments. 2, 3, and 4 are equality checks. The only thing that matters is the number of equal signs, not, whether the expression makes sense, or what type the elements have!

[12]ANSWER: `hovercraft is full of eels`

[13]ANSWER: `full`

[14]ANSWER: `The suffix 'ery' occurs 28 times`

[15]ANSWER: `half = len(affix)/2`

[16]ANSWER: `assert len(affix) > 1, "affix needs to have at least two characters, encountered '%s'" % (affix)`