Learning to Code with Python

Xavier Rubio-Campillo (xavier.rubio@ed.ac.uk)

18th May 2017

Summary

This tutorial will teach you to:

- understand the basic concepts of a programming language
- repeat tasks with loops
- structure your code with functions
- add some logic to your code
- open text files
- plot stuff

Why code?

Software is a vital part of any field of knowledge. We use databases to store information, analytical tools to explore it, word processors to write our results and browsers to look for relevant research. All this software is created in a similar manner: first, a set of complex instructions is written in a specific programming language such as C++ or Java. The next step is to *compile* this code: to generate a binary version that can actually be understood by any computer. The final step is to distribute this binary so you can download, install and use the package.

This is useful as long as you find a specific tool that can meet the requirements of the task you have in mind, but what if this is not the case? Maybe you read a new method in some paper and you want to apply it to your dataset, or maybe you would like to tweak the behavior of the program because it was not designed for your needs. Some of this software is distributed under open source licenses so you can check, edit and extend its functionality. If you have some basic understanding of coding then you will be able to modify or even create new software adapted to your needs.

Disciplines such as Biology or Physics are aware of the importance of programming and it is part of their teaching, but this is not common in the Social Sciences and Humanities. This tutorial will explain the basics of programming for social scientists and humanists by using one of the currently most popular languages: Python.

Why Python? There are several reasons behind this choice:

1. it's easy to learn because its syntax makes sense
2. it's free and open
3. its community is diverse and friendly
4. it has everything you will ever need for research: statistics, visualization, parallel processing...

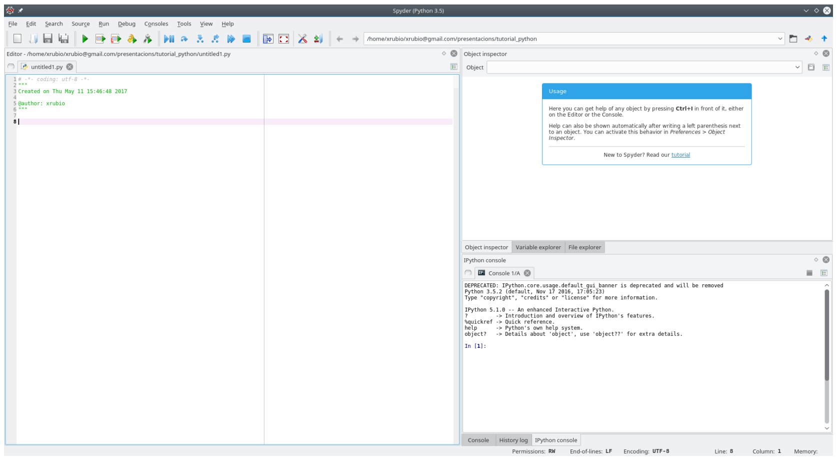
The task

We (humans) usually learn things when it is useful for us. This is particularly true when you need to invest a decent amount of time on the process, or the learning curve is steep such as coding. In this tutorial we will create the code to solve a specific task so you can explore the potential of coding to your own interests. This is what we are going to do: we will identify the most frequent words used in a book. This task is rather common to a large amount of disciplines in the Social Sciences and Humanities, and it is rather similar to other tasks that you could be interested on solving. Overall it will be a good example of what you can achieve by learning Python.

We need to learn lots of things before we can achieve this task, but don't panic yet! We will go step by step so let's begin with the basics concepts of any programming language; you will need them once you focus on the task.

Basic Concepts

The first thing you need as a coder is some place to write and run your program. We will be using *spyder*, one of the best IDEs (Integrated Development Environment) for Python programmers. Open *spyder* in your computer to see an interface like the one in Figure 1


Spyder (Python 3.5)
File Edit Search Source Run Debug Consoles Tools View Help
/home/xrubio/xrubio@gmail.com/presentacions/tutorial_python
Editor - /home/xrubio/xrubio@gmail.com/presentacions/tutorial_python/untitled1.py
untitled1.py
Object inspector
Object
Object inspector Variable explorer File explorer
IPython console
Console 1/A
Console History log IPython console
Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 8 Column: 1 Memory:

```
# -*- coding: utf-8 -*-
"""
Created on Thu May 11 15:46:48 2017

@author: xrubio
"""

```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector*.

New to Spyder? Read our tutorial

```
DEPRECATED: IPython.core.usage.default_gui_banner is deprecated and will be removed
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Figure 1: The Spyder IDE

There are 3 windows:

1. In the left you got an editor to write your code
2. The window at the top right can be used to inspect what is going on in your program
3. The bottom right is the Python console

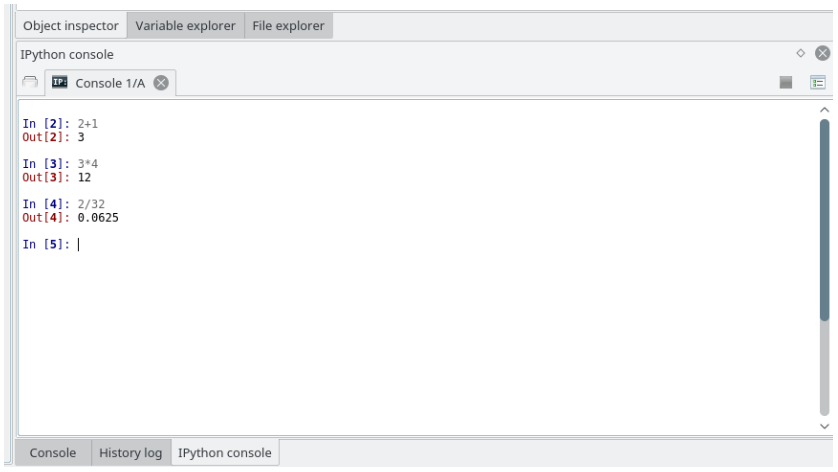
Performing arithmetics

The thing you need to remember is that a computer is essentially a calculator; yes, it is a super-mega complex calculator but it is *still* a calculator. Let's start with simple arithmetics; type in the console (bottom right):

Attention!! Don't worry if you get errors when you run your code: it is normal while you are developing a program. It can be caused by typing the wrong symbol or asking for the wrong operator. You should read carefully the error in order to fix it. Also, you cannot break the computer from spyder so experiment with the console as much as you want!

```
2+1
3*4
2/32
```

As you press the Enter key after each operation you will see the results as shown in Figure 2.



```
In [2]: 2+1
Out[2]: 3

In [3]: 3*4
Out[3]: 12

In [4]: 2/32
Out[4]: 0.0625

In [5]: |
```

Figure 2: The Python console will run any command you type inmediately

Storing results in variables

This seems interesting, right? Well, not really but we will get there! One basic idea of any programming language is the concept of a variable. It is essentially a symbol that you will use to access results. For example if you type:

A *variable* is a symbol that represents a quantity. You create variables to store values and combine them into new calculations

```
myFirstVariable = 2
theNameOfThisVariableIsTooLong = 5
myFirstVariable + theNameOfThisVariableIsTooLong
```

You should get a seven as the output. You can also store the result into a new variable and print it on the console:

```
myResult = myFirstVariable + theNameOfThisVariableIsTooLong
myResult
```

All these variables are integers. In Python you can create variables of several different types, such as *floats* and *strings*:

An *integer* is a whole number: 2, 31, -12361...

A *float* is a real number (one with decimals): 2.31, -6.1213232...

a *string* is a sequence of characters: "hello", "how are you?"

```
thisIsAFloat = 2.75652
andThisIsAString = "this is a string"

thisIsAFloat + andThisIsAString
```

Wow what happened here? It gave you an error telling you that you tried to sum a float and a string (or *str*), which as you can imagine does not make a lot of sense. It's ok; it is difficult to get the perfect code so these errors will help you identify and solve problems in your program.

You can also create variables as an aggregation of other variables. One type that is really useful is the list. Let's create one list of integers:

A *list* is a collection of variables

```
aList = [3,1,4,1,5]
aList
```

Lists are really powerful because you can access anytime its elements by index (the first element, the second element, and so on...). Indexes always start at 0. In this example the number 3 is in position 0 and the number 5 is in position 4. Try it!

I know, it's weird that programmers start counting at 0 instead of 1 but we have our reasons! The index of a list points towards a memory position in your computer; the index works as the offset (so the first position has offset 0

```
aList[0]
aList[4]
```

Now try to access an element that does not exist such as *aList[5]*. What do you get? Yep, another error...if you continue coding you will see lots of them, trust me!

You can also modify the contents of a list after you create it:

```
aList[2] = "thisIsNotANumber"
aList[4] = 3.14159
aList
```

But what happens if you want to modify all the contents of a list? For example, imagine that you have a list of integers and you want to add 1 to each of the values in the list; you could do this one by one but...what if you got 63145 elements? You could a) spend a nice weekend with it or b) iterate through the elements with a loop.

working with the editor

Before we loop we need an additional tool. The code for a loop requires more than a single line so we will move from spyder's console to the editor (the left window). Anything you type here will be executed if you click on the *Run* button or you go to the menu bar and click on Run →Run. There are some lines already but they are comments that won't be evaluated by the system

A single line comment starts with a # symbol

A multiline comment starts and ends with three double quotes ”””.

One important difference between working in the console and running a script is that the output won't be printed by default. You can do it by using the function *print*:

A *script* is a text file with a small program ready to be executed. Python is an scripting programming language because you don't need to compile the code before running it; there is an installed interpreter that takes care of translating your code to something that your computer can understand.

```
# this is a comment
print("Hello World!")
```

If you Run now the code you should see in the console an output such as:

A *function* is a piece of code that receives some inputs (between parenthesis), does some calculations and (usually) returns an output

```
In [1]: runfile('/home/foo/.spyder2-py3/temp.py'...
Hello World!
```

So as you can see the comment is ignored while the second line is executed (and some stuff printed).

It is always good to document your code with comments explaining its purpose and functionality

looping through a list

Now that we covered the basics of the editor we can go back to our original question: how can we modify the contents of a list as a whole? We need to iterate through the contents of the list. This can be done in this way:

Attention! Indentation is very important in Python as it defines the limits of the loop. Make sure that the lines printing and modifying the values of the list are one *tab* or 4 spaces to the right

```
# create a list of 10 numbers
aList = [3,1,4,1,5,9,2,6,5,3]
# store its length on a new variable
aListLength = len(aList)
print("list:",aList,"has length:",aListLength)

# loop through a range of 0 to aListLength
for index in range(0, aListLength):
    # contents of the loop are indented!
    print("value",aList[index],"at position:",index)
    # add one to the number at position index of the list
    aList[index] = aList[index]+1
print("modified list:",aList)
```

Let's take a look at the code line by line:

- *aList = [3,1,4,1,5,9,2,6,5,3]* - creates a list with 10 integers inside (you can choose others if you want)

- *aListLength = len(aList)* - here we use the function *len* to get the length of our list. The output is stored in a variable called *aListLength*.
- *print("list:",aList,"has length:",aListLength)* - not much here; we print the contents of the list and also its length
- *for index in range(0, aListLength):* - this is the loop; we iterate through all the values between 0 and aListLength-1 (so 0 and 9). During each iteration the contents of the loop will see the variable *index* with a different value inside the range, so *index* is automatically incremented between iterations.

The function *range* gives you a range of values between its first parameter up to the second one, so in this case it will give: 0,1,2,3,4,5,6,7,8,9.

- *print("value",aList[index],"at position:",index)* - during each iteration we print the index (so it will be 0,1,2,3...until 9) and its value.
- *aList[index] = aList[index]+1* - here we modify the value by adding one to the previous one.
- *print("modified list:",aList)* - a final print to show the modified contents

If you run this code you should see this in the console:

```
list: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3] has length: 10
changing value 3 at position: 0
changing value 1 at position: 1
changing value 4 at position: 2
changing value 1 at position: 3
changing value 5 at position: 4
changing value 9 at position: 5
changing value 2 at position: 6
changing value 6 at position: 7
changing value 5 at position: 8
changing value 3 at position: 9
modified list: [4, 2, 5, 2, 6, 10, 3, 7, 6, 4]
```

This worked! You almost have all the basics of any programming language. You only need 2 more ideas: how to create functions and how to run code based on logic conditions.

conditions and functions

In the previous code we used a couple of standard functions in Python: *len* and *range*. This code receives some input variables as parameters and return an output. In the first case it receives a container of some kind such as a list and returns its length; in the second case it receives two values (minimum and maximum) and returns the sequence of integers between them. It also has an abbreviated version where it receives just the maximum and defines the minimum at 0.

so *range(0,10)* and *range(10)* returns exactly the same sequence

How do you define a function? Imagine that you want to create a function that returns the maximum value of a list. You will need: a) the list as input b) iterate through it and get the maximum value and c) return the maximum value you found. Let's write this to make our algorithm clear:

Attention! Don't type this! It is a piece of *pseudocode*; it is useful to clarify your ideas without the hassle of the specific syntax

```
function getMaximumValue( aList ):
    maxValue initialized at -1
    for item in aList then:
        if item is larger then maxValue then:
            maxValue equals item

    return maxValue
```

See what we did here? We loop through the list and then we apply a condition: if the current value of item is larger than a minimum value then we enter the next sub-block of code. In this sub-block we update *maxValue*. If the condition is evaluated as *False* then we will not run the sub-block; the lines in it will be ignored and the loop will continue with the next iteration. Once we finish we return the maximum value we found. This can be translatedinto Python as:

You can save the editor's contents going to File →Save as... and specifying a file name. A new file can be created by going to File →New File...

```
def getMaximumValue( aList ):
    maxValue = -1
    for item in aList:
        print("current:",maxValue,"test:",item)
        if item > maxValue:
            maxValue = item
            print("updated next value!", maxValue)
    return maxValue
```

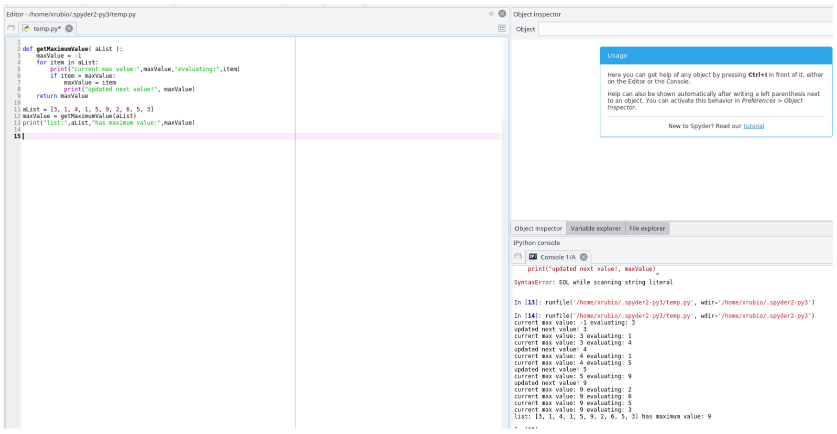
As you probably noticed this loop is slightly different than the previous one. Here we only care about the values in *aList* and not their index, so the loop is simpler as we don't need the length of the list of the range of indexes.

Create a new file and type the definition of the function *getMaximumValue*. If you run it nothing will happen (well, maybe some errors...). You need to *call* the function to evaluate it so type this after the definition of the *getMaximumValue* function:

```
aList = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
maxValue = getMaximumValue(aList)
```

```
print("list:",aList,"has maximum value:",maxValue)
```

You should get an output such as the one in Figure 3.



```
def getMaximumValue( aList ):
    maxValue = -1
    for item in aList:
        print("current max value:",maxValue,"evaluating:",item)
        if item > maxValue:
            maxValue = item
            print("updated next value!", maxValue)
    return maxValue

aList = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
maxValue = getMaximumValue(aList)
print("list:",aList,"has maximum value:",maxValue)
```

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector*.

```
    print("updated next value!, maxValue)
                                          ^
SyntaxError: EOL while scanning string literal


In [13]: runfile('/home/xrubio/.spyder2-py3/temp.py', wdir='/home/xrubio/.spyder2-py3')

In [14]: runfile('/home/xrubio/.spyder2-py3/temp.py', wdir='/home/xrubio/.spyder2-py3')
current max value: -1 evaluating: 3
updated next value! 3
current max value: 3 evaluating: 1
current max value: 3 evaluating: 4
updated next value! 4
current max value: 4 evaluating: 1
current max value: 4 evaluating: 5
updated next value! 5
current max value: 5 evaluating: 9
updated next value! 9
current max value: 9 evaluating: 2
current max value: 9 evaluating: 6
current max value: 9 evaluating: 5
current max value: 9 evaluating: 3
list: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3] has maximum value: 9
```

Figure 3: Calling a function

This is it! We *iterated*(!) through the list of basic concepts of programming using Python. It is now time to finally go for our task.

Counting words

As you remember what we want to do is to show the most frequent words in a book. Specifically we will need to:

- load the contents of a book
- identify the words in the text
- count the number of repetitions (i.e. frequency) of each word
- sort the words by their frequency
- get and print the list of top words and their frequencies

Let's start with the main code; once we know what it should look like it will be easier to implement the different functions we need.

The main block of code

The list of tasks that we need define our *main* code. It can be defined as:

Attention! This code will not work now because we need to define the functions that we are using: *getWords*, *countWords* and *getFrequent*

.

```
# parameters
bookFile = "frankenstein.txt"
# top 10 frequent words
```

```
numFrequentWords = 10
print("counting top",numFrequentWords,"from:",bookFile)

# split the book in words and count them
wordList = getWords(bookFile)
freqs = countWords(wordList)

# select and print the most frequent ones
topWords,topFreqs = getFrequent(freqs,numFrequentWords)
print(numFrequentWords,"top frequent words are:",
topWords,"with freqs:",topFreqs)
```

If you run this then you will get the output of the first print BUT it will give you an error as we did not define our functions. Let's start with *getWords*.

parsing the text file

If you go to the folder where this tutorial is placed you will find a subfolder called *books* with some files. Each of the files is an entire book downloaded from the Guttenberg project; you can open and read any of them using a text or word editor such as Microsoft Word or Notepad.

This is cool if you want to read the book, but we are interested on automatically reading and parsing the books *from* Python. The idea here is to open the file and then loop through its lines; for each line we will divide its contents based on space characters to finally get the words. This is the basic code:

functions need to be defined *before* they are called so type this before the main text

```
def getWords(bookFile):
    book = open(bookFile)

    for line in book:
        print(line)
```

Bear in mind that the variable *bookFile* should define the path towards to text file so please, check that you are using the correct one (otherwise you will get a *FileNotFoundError*). If everything works you will see the entire content of the book printed in the console. Now we need to *split* each line into words (i.e. blocks of characters separated by spaces) and return a list of them. Replace the function definition with the complete one:

```
def getWords(bookFile):
    # create empty list
    listOfWords = []
    book = open(bookFile)

    for line in book:
        splitLine = line.split()
        for word in splitLine:
            listOfWords.append(word)
    print("number of words:",len(listOfWords))
    return listOfWords
```

If you run again the script it will print the number of words before throwing an error about the next function that has not been implemented (*countWords*). Some remarks:

- *open* opens the file and gives its contents as output
- *split* is a functionality included into any variable of type *string*. It splits the string based on a separator (space if none provided) and returns the list of strings generated by the split.

confused by *split*? Type this in the console: "this is a sentence".split()

- *append* is a functionality included into any variable of type *list*. It allows the list to include a new item (in this case the new word).
- We nested a loop within another one; the first one iterates through the lines of the book while the second one iterates through the words of each line.

  Done, now for the next function: *countWords*.

counting words

We will need now to create a set of unique words so there are no repetitions inside the list of words. At the same time we need to also store the number of repetitions for each of these unique words. This can be done using a *dictionary* variable:

A *dictionary* is similar to a list, but each value inside must be unique and works as a key for a given value; in our case the keys are the words and the values are their frequencies

```
def countWords(listOfWords):
    # create an empty dictionary
    uniqueWords = {}
    for word in listOfWords:
        if word in uniqueWords:
            uniqueWords[word] = uniqueWords[word]+1
        else:
            uniqueWords[word] = 1
    print("number of unique words:",len(uniqueWords))
    return uniqueWords
```

The behavior of the function can be summarized as follows; we iterate through the list of words. For each word we evaluate if it is already stored in our dictionary; if it is then we add one to its count; otherwise we register it and assign to the word a count of 1. The final step is to return the dictionary.

selecting top words

Now we got our list of unique words with frequencies so the next step is to select the most common ones. We will sort the list of words based on their values and select only the top ones. We can do the first step by using the function *sorted*. It receives three parameters: 1) the dictionary to sort, 2) the key to sort it, in our case the value, 3) an optional parameter to choose between ascending and descending (i.e. reversed) order. The function *sorted* returns the complete list of words so we can subset the most frequent ones from it.

Our new function uses this output to generate and return 2 lists: a) the list of top words and b) the frequency of these words. In python a function can return multiple variables by simply listing them with commas:

```
def getFrequent(freqs, numFrequentWords):
    sortedWords = sorted(freqs, key=freqs.get, reverse=True)
    mostFrequent = sortedWords[0:numFrequentWords]
    listFreqs = []
    for word in mostFrequent:
        listFreqs.append(freqs[word])
    return mostFrequent,listFreqs
```

As you can see once we compile the list of sorted words then we can subset from the list from position 0 (the most frequent word) to position *numFrequentWords*. Done!

Plotting frequencies

We completed our task but some work could be done in its output. For example, what is the distribution of frequencies amongst the words? Are these frequencies balanced? Is a majority of words appearing only once?

Let's see how can we create a basic plot visualizing the frequency distribution of words. We will use a python library called *matplotlib* that has lots of cool functionality on...yep, plots..

See lots of examples of *matplotlib* here: `https://matplotlib.org/`

We will load the library and then plot a histogram of frequencies of the top words (at the end of our *main* code):

By default spyder places the plot into the console. If you want a popup window go to the main menu Tools →Preferences →IPython Console →Graphics and select Qt as Backend. You may also need to restart spyder.

```
import matplotlib.pyplot as plt
plt.hist(topFreqs)
```

You should get something like Figure 4. As you see a handful of words have very high frequency but the count rapidly decreases.

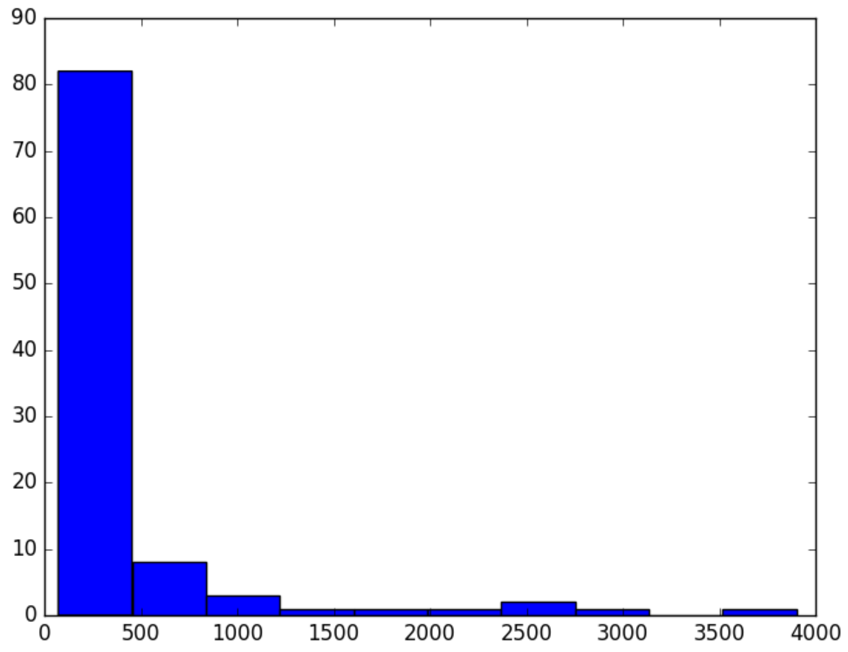


Figure 4: Histogram of frequencies for 100 top words

However, the words are lost...if we are interested on them then we could create a scatterplot with the list of words as the X axis:

```
rangeWords = range(len(topWords))
plt.plot(rangeWords, topFreqs, "o")
plt.xticks(rangeWords, topWords, rotation=45)
plt.show()
```

Here at Figure 5 we plotted the frequency of the top 20 words.

You managed to go through the basics of Python, parse the book, count frequencies and even plot its results! Not bad for a short tutorial...hopefully it allowed you to understand the possibilities of Python in terms of your own interests and requirements. If you want to improve this code take a look at the following section.

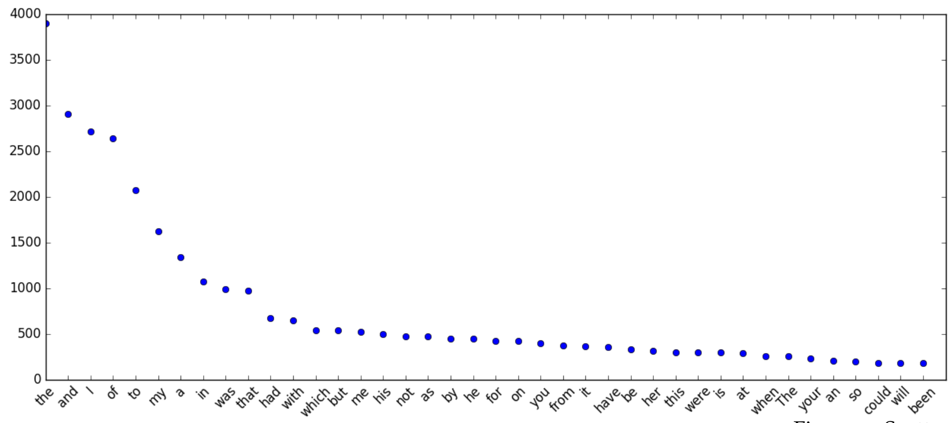


Figure 5: Scatter plot of words and frequencies for 40 top words

Improvements

There are plenty of things you could try to improve the functionality you implemented today. Here is a list of challenges that will guide your learning of Python.

Cleaning the book

You probably noticed that some of the words are not correct (they have commas, dots, some of them are upper or lowercase...). You could use some of the Python functionality to clean them before you add them to the *listOfWords* within the *getWords* function. For example the *isalpha* method:

```
def cleanWord(word):
    cleanedWord = []
    for character in word:
        if character.isalpha():
            cleanedWord.append(character)
    return "".join(cleanedWord)
```

Stop words

The most frequent words of any text are boring because they will probably be common articles, verbs and prepositions. This is not good if you want to see differences between texts because *the* or *being* will be present in all of them at a really high frequency. You could apply a filter to the list of words before counting them to avoid this.

The trick is to receive as input the blacklist of words to ignore and do not append them to listOfWords if they are present in this blacklist.

You can get a popular list of stop words in English from here: `https://gist.github.com/sebleier/554280`

Differences between books

We could analyse the words used by the different authors to identify patterns between works; are fiction works more similar between them than with essays? What are the words used in non-fiction works? What words are used across any type of work?

Identify the similarities and differences between the list of top words provided by each of the 4 books (or download new ones!).

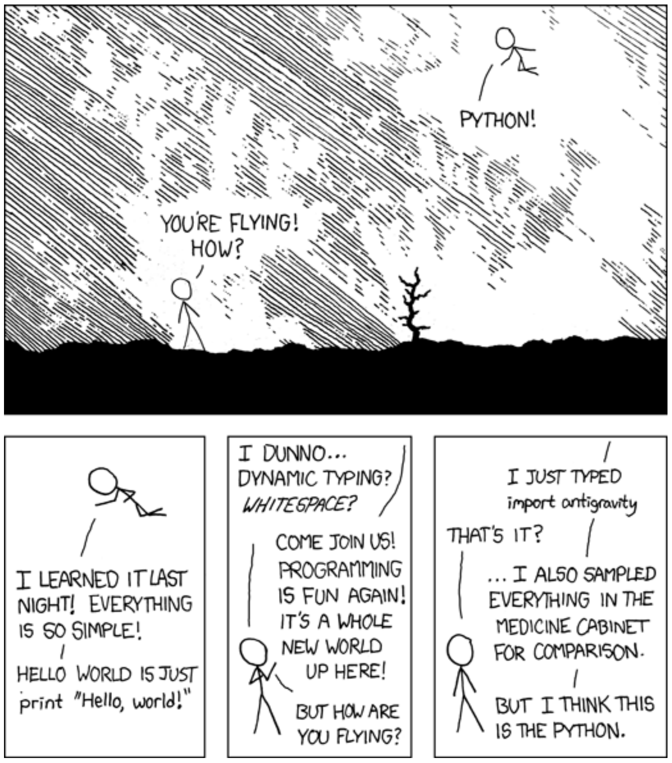
Appendix - Resources

All books were downloaded from the Project Gutenberg: `https://en.wikipedia.org/wiki/Project_Gutenberg`. You will find here the links to the original works:

- Pride and Prejudice by Jane Austen `https://www.gutenberg.org/ebooks/1342`
- The Communist Manifesto by Karl Marx `https://www.gutenberg.org/ebooks/61`
- On the Origin of Species by Charles Darwin `https://www.gutenberg.org/ebooks/1228`
- Frankenstein; or the Modern Prometheus by Mary Shelley `https://www.gutenberg.org/ebooks/84`

You will find more information on Python and coding on these sites:

- The task of this tutorial was inspired by the awesome tutorial *Python Programming for the Humanities* by Folgert Karsdorp - `http://www.karsdorp.io/python-course`
- There is a large list of Tutorials for Python in its official webpage - `https://wiki.python.org/moin/BeginnersGuide/Programmers`
- Are you a researcher? This tutorial on Python for Social Scientists focus on the tools you need for analysis - `https://realpython.com/blog/python/python-for-social-scientists/`


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

`https://xkcd.com/353/`