12. Practical Programming for NLP

Patrick Jeuniaux[1], Andrew Olney[2], Sidney D'Mello[2]
[1] *Université Laval, Canada,* [2] *University of Memphis, USA*

ABSTRACT

This chapter is aimed at students and researchers who are eager to learn about practical programmatic solutions to natural language processing (NLP) problems. In addition to introducing the readers to programming basics, programming tools, and complete programs, we also hope to pique their interest to actively explore the broad and fascinating field of automatic natural language processing. Part I introduces programming basics and the Python programming language. Part II takes a step by step approach in illustrating the development of a program to solve a NLP problem. Part III provides some hints to help readers initiate their own NLP programming projects.

INTRODUCTION

Natural language processing (NLP) attempts to automatically analyze the languages spoken by humans (i.e., natural languages). For instance, you can program a computer to automatically identify the language of a text, extract the grammatical structure of sentences (see Chapter XXX of this book), categorize texts by genre (e.g., decide whether a text is a scientific or a narrative text; see Chapter XXX for classification applications), summarize a book (see Chapter XXX), etc. This chapter is aimed at teaching specialized, yet introductory, programming skills that are required to use available NLP tools. We hope that this chapter serves as a catalyst to launch NLP projects by motivating novice programmers to learn more about programming and encouraging more advanced programmers to develop NLP programs. The chapter is aimed at readers from the interdisciplinary arena that encompasses computer science, cognitive psychology, and linguistics. It is geared for individuals who have a practical NLP problem and for curious readers who are eager to learn about practical solutions for such problems.

Fortifying students with the requisite programming skills to tackle an NLP problem in a single chapter is a daunting task for two primary reasons. First, along with advanced statistics, programming is probably the most intimidating task that practitioners in disciplines like linguistics or cognitive psychology can undertake. The typical student or researcher in these fields has little formal training in mathematics, logic, and computer science, hence, their first foray into programming can be a bit challenging. Second, although computer scientists have considerable experience with programming and have mastered many computer technologies, they might not be privy to the libraries or packages that are readily and freely available for NLP projects. In other words, there is a lot to cover if we attempt to address both these audiences, and it seems like an impossible challenge to design a chapter extending from the basics of programming to the specifics of NLP. Fortunately, for the reader and us, the availability of state-of-the-art NLP technologies and the enhanced usability available through easy-to-use interfaces alleviates some of these challenges.

Because of space limitations, we could not achieve the coverage depth we had hoped for. We originally had planned to include programming projects in several languages such as Python, Perl, Java and PHP, along with numerous screen captures of captivating programming demonstrations. The chapter is now more focused on examples in Python. Fortunately, the materials that could not be included in the chapter (e.g., scripts, examples, screen captures), are available for your convenience on the companion website at `http://patrickjeuniaux.info/NLPchapter`. It also provides a series of links to NLP resources, as well as detailed instructions about how to execute the programs that are needed for the exercises. A great advantage of having a website is that it can be updated with current content, so do not hesitate to contact us if you wish to give us feedback.

This chapter has three parts. Part I offers an introduction to programming. Part II gives a concrete example of programming for a specific NLP project. Part III provides general hints about starting your own NLP programming project. Readers who do not have programming experience or who do not know Python should definitely start with Part I. Individuals who have a working knowledge of Python can skip most of Part I. Among these people, the ones who do not know about NLTK could limit their reading of Part I to the section on functions and onwards. Although Part I covers a lot of material, the topic coverage is far from exhaustive. When you are done with this chapter, we encourage you to read a more complete introduction. We particularly recommend Elkner, Downey, and Meyers (2009). The same can be said of Part II. We also recommend reading Bird, Klein and Loper (2009), who give a thorough treatment of NLP programming with Python's Natural Language Processing Toolkit (NLTK).

PART I. PROGRAMMING BASICS

Computers are controlled by sets of instructions called programs. Because they are somewhat simple machines, computers can only follow the most unambiguous instructions. To achieve this ideal of precision, a program is written in a restricted language. Programming languages use a specific vocabulary (i.e. a set of words), a syntax (i.e., a set of rules defining how to use these words), and semantics (i.e., the meaning of the words and rules from a programming point of view). Learning the basic rules of a language is the first step towards writing meaningful and useful programs. But prior to learning a language, it might be good to learn about the history of computer programming. Knowing the historical motivation behind programming will help you grasp what programming is all about.

Programming in Context

Today we usually think of a computer as a general purpose device. However, this was not always the case. Whether you consider Babbage's Difference Engine (Swade, 2002), which solved polynomial functions, or Colossus, which helped decipher the Enigma codes during World War II (Hodges, 2000) to be computers, the fact remains that a "computer" is simply something that performs calculations. In fact, before the 20th century people whose jobs were to perform complex calculations for various purposes were called 'computers' (Anderson, 2009).

As the science and technology of computing advanced, man-made computers became more complex and were able to perform more complex calculations. However, the process for doing so was extremely tedious and error prone. Computers were massive beasts of machines in those days, often taking up entire rooms. Programming sometimes meant re-patching cables on a switchboard (Petzold, 2000), a far cry from the text editors and visual interfaces that we are familiar with today.

At this time it became clear to the scientists involved with creating and using these machines that the difficulty of using computers was an obstruction to their widespread use and acceptance. Their solution to this problem was to create layers of abstraction between the user and the computer. This process of increasing abstraction has continued to the present day and shows no signs of stopping in the foreseeable future. For example, consider your computer's desktop. "Desktop" is just an abstraction and analogy for a physical desktop for pens and paper. Similarly, the folders on your desktop are analogous to physical file cabinets used to store paper documents.

Programming languages are just another kind of abstraction over the underlying computer instructions ("machine code"). The machine code is simply a very detailed and hard to use programming language. For most applications, programmers do not use machine code but use instead modern programming languages which are designed to simplify the programmer's life (by reducing the size of the program, reducing the likelihood of programming error, etc.). Programming languages have evolved in such a way that programming is no longer restricted to the purview of professional computer scientists. So-called high level languages allow practitioners of other fields (like psychology, and linguistics) to enjoy the power and flexibility of programming. One of the goals of this chapter is to show how this is feasible. Like in all fields, it is not possible to immediately benefit from practical applications without knowing the fundamental principles underlying them. Hence, the next section is aimed at bringing you up to speed with such principles.

Fundamental Concepts of Computer Programming

Fundamental programming concepts include (a) values and types, (b) syntax and semantics, (c) operations, (d) variables, constants, and assignments, (e) data structures, (f) conditionals, (g) iterations, and (h) functions. We start by presenting these concepts with step-by-step examples of programs written in the Python language – a language whose simplicity seduces the most unwilling learners. While reviewing these basic ideas we also present some programming constructs that are especially relevant for NLP; these include strings, corpora, text files, input-output (I/O), etc. Before we begin, it is important to consider the two steps involved in writing a program: *pseudo-code* (planning) and *implementation* (executing). Finally, we will describe one important aspect of efficient code implementation: *incremental programming*.

Pseudo-code

As you will see in the subsequent examples, Python has a quite intuitive syntax. In some respects, Python syntax looks like pseudo-code. Pseudo-code is a high-level description of a program that makes no reference to a particular language. For instance, Table 1 presents pseudo-code for a program that translates sentences in a document. Each line is an instruction. The first line opens an input file and the last line closes it. The lines in between translate each line in the file. Pseudo-code is important because it provides a conceptual representation of what you intend to program, before you do any real programming. Planning by using some kind of pseudo-code (whether purely textual or even graphical) is an important part of conducting a successful programming project, because it very frequently unveils obstacles that you did not originally foresee when you started to think about the problem. Once you have figured out what to program (i.e., with a pseudo-code description), most of the remaining difficulties are mainly technical in nature (i.e., how to program).

Table 1. Pseudo-code for translating a document.

```
Open the file.

For each line in the file,

         Translate the line.

Close the file.
```

Implementation

Learning to program is analogous to learning a new skill. It would be impossible to learn to ride a bicycle by reading about bicycles. Reading programming books will not suffice. You need to get your hands dirty and actually program things in conjunction with reading about them. Learning a programming language is also similar to learning a real language. You learn a real language by placing yourself in new situations where you have to communicate. Likewise you learn programming by putting yourself in new situations where you have to solve new problems. This involves downloading resources, installing the required software, setting up your working environment (e.g., a text editor or an Integrated Development Environment), writing code, and executing it. The greater the variety of the problems you solve, the more experienced of a programmer you will become.

Therefore, we cannot encourage you enough to download Python and reproduce all the examples which are explained below. These examples all assume that you are using Python IDLE – the interactive console and development environment provided with Python. Complete instructions are available on the companion website. They tell you where to find Python, how to install it, and how to use the software that can run your code. Get ready to see how user friendly Python is!

During this journey, if you are serious about implementing code, you will likely accumulate a substantial amount of files containing either code or data (i.e., material generated or used by your programs). Similar to how it is easier to find a document on a well ordered desktop than in a pile of random papers, it will be useful for you to organize the files on your computer in an orderly manner. If you are not familiar with the file system on your machine (i.e., how files and folders are organized, how to create them, move them, etc.), you should consult the links available on the companion website. When you are ready, create one folder on your machine. Let's call it the `test` folder. This is where you are going to put all your files. Because you wish to provide a structure to your files, within the test folder, create two other folders: the `data` and `scripts` folders, where you will place your data and your code, respectively.

Incremental programming

Most problems will require programmers to implement their solutions step-by-step (i.e., increment by increment), leaving some parts of the program untouched until they feel it is time to tackle them. Making errors is a natural outcome of any programming activity, and it is best to work on a program one piece at a time, so that one knows where the errors come from, rather than programming all aspects simultaneously. In general, such a divide-and-conquer approach to programming is very efficient. Table 2 exemplifies the order in which a programmer could have implemented the Pseudo-code exhibited in Table 1.

Table 2. The incremental approach to programming.

(a)	(b)	(c)	(d)
`Open the file.`	`Open the file.`	`Open the file.`	`Open the file.`
		`For each line in the file,`	`For each line in the file,`
		`    print the line.`	`    translate the line.`
	`Close the file.`	`Close the file.`	`Close the file.`

Panel (a) in Table 2 shows that the programmer first attempts to open the file. To implement this idea, several lines might have to be written. The programmer works on that part until the result is satisfactory. In Panel (b), one more component is added: closing the file. It is generally important to free a resource that was mobilized when "opening" a file. Once this is working, it is typically a good idea to inspect the content of the file, line by line, by printing each line on the screen, as shown on panel (c). If one can realize such a simple step, then one can address the more involved operation of translating the line, as shown on panel (d). Whereas panel (a) focused on the input to the program, panel (c) added a way of producing some output. In general, it is useful to implement simple operations for introducing some information from the external environment into your program (i.e., the input), and saving information from your program in the external environment (i.e., the output), before addressing more sophisticated parts of the program, like the translation step of panel (d). The next sections illustrate programming basics with simple Python commands. Try them out by typing them or copying them from the website.

Values and Data Types

Programs contain instructions that tell the computer how to use values. Values can be of different types (or "data types"). Table 3 provides examples. The prompt of Python IDLE (i.e., the signal that Python expects an instruction) is represented by >>>. The commands follow that sign. To execute a command, type it and then press the `Enter` key. The output resulting from the execution of the command is displayed in bold characters. The first command in Panel (a) of Table 3 requires the computer to print a value of the *string* type, and whose content is `"Hello World!"`. A string is a sequence of characters (letters, numbers, punctuation marks, and other special characters), usually surrounded by single quotes (`''`) or double quotes (`""`). To check that `"Hello World!"` is a string, use the second command in

Panel (a). Python IDLE prints a message indicating the type (i.e., `str` for *string*). Another type is the *integer*, which corresponds to natural numbers like 1, 2, 3, etc. The first command in Panel (b) shows how to print the integer `123`. To check that it is indeed an integer, use the `type` command on the number, as illustrated in Panel (b). If you surround the number by quotes it will be interpreted as a string.

Table 3. Examples of commands and outputs to understand values and types in Python.

(a)	(b)
`>>> print "Hello World!"`	`>>> print 123`
`Hello World!`	**`123`**
`>>> type("Hello World!")`	`>>> type(123)`
`<type 'str'>`	**`<type 'int'>`**
	`>>> type('123')`
	`<type 'str'>`

Syntax and Semantics

As shown in Table 3, quotes are important to recognize strings. This simple example illustrates that a programmer must respect the syntax of a language (i.e., rules) in order to express ideas in a program (also called “code”). These ideas correspond to the semantics of the program (i.e., what it is supposed to do). In other words, when we need to use natural numbers (i.e., a semantic constraint) we do not use quotes (i.e., a syntactical constraint). The syntactical aspect of a language is where beginners first stumble. Luckily for them, a language like Python has a very gentle syntax, and this allows a beginner to readily focus on the more interesting aspect of the language, namely its semantics and what it can do for the programmer.

Operations

Why are there several data types like strings and integers? It is because these values have different properties, and allow the computer to perform different kinds of operations. Operations in a programming language can modify values. For instance, strings can have an arbitrary content (e.g., on the screen you could have printed `"Hello World!"` or the entire works of Shakespeare) but they do not support conventional numerical operations (e.g., adding Othello to Hamlet does not result in a number). Contrary to strings, integers must be natural numbers (e.g., 1, 2, 3, 4.5, 2.78) and you can perform conventional numerical operations with them (e.g., addition, multiplication). Table 4 presents examples of operations in Python.

Table 4. Examples of operations in Python.

(a)

```
>>> print 1 + 2

3

>>> 3 * 4

12
```

(b)

```
>>> "pine" + "apple"

'pineapple'

>>> 3 * "pine"

'pinepinepine'

>>> 1 + "pine"

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    1 + "pine"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Panel (a) requests Python to add 1 and 2 to obtain 3. The second command in Panel (a) shows how to multiply 3 by 4 to obtain 12. Despite the fact that we did not use the `print` command, `12` is printed because the Python interactive software prints values by default. In the context of numbers, the signs `+` and `*` behave like arithmetical operators.

When a string is involved, their meaning is different. For instance, Panel (b) shows how to *concatenate* the strings `"pine"` and `"apple"`. This operation pastes the two strings to create a new string – the string `"pineapple"`. We then create a new string – the string `"pinepinepine"` – by repeating `"pine"` three times. Finally we attempted to perform an invalid operation based on an erroneous use of syntax, since it is not possible to add a string to an integer or concatenate a string with an integer (i.e., `1 + "pine"` is meaningless). Read the error message which is generated: it is very instructive, because it mentions the key notions of operators, types, integer and string.

Variables, Constants, and Assignment

What happened to all the values we have created? As soon as `"Hello Word!"` is printed, or that `1 + 2` is computed, these values disappear, in the sense that if you want to reproduce these events again, you need to repeat the commands used to create them. In order to maintain access to values in a program, you need to give them a name. Names, like all kinds of words, are symbols which stand for (or represent) values. For instance, the word *car* represents a four-wheel vehicle used by humans. It is mainly a matter of convention. It could have been called something else, like *platypus* or *bora-bora.* The only thing which matters is that the users of the word agree upon the convention. In programming terms, these names are called *variables.*

Similarly to what happens in real life, the computer program will agree to identify variables by respecting the labels you have adopted (i.e., that is by following your convention). To assign a label to values, you use the equal sign (=). Table 5 gives some examples. Panel (a) in Table 5 shows how to assign the value `3` to the variable `X`. If you then type `X`, the value `3` is displayed (i.e., the content referred to by `X` is displayed).

Table 5. Examples of assignments in Python.

`>>> X = 3`	`>>> X = 1 + 2`	`>>> X = 1`	`>>> Name1 = "Patrick"`
`>>> X`	`>>> Y = 4`	`>>> X`	`>>> Name2 = "Andrew"`
`3`	`>>> X * Y`	**`1`**	`>>> Name3 = "Sidney"`
	`12`	`>>> X = "Hello World!"`	`>>> S = ", "`
		`>>> X`	`>>> Authors = Name1 + S + Name2 + S + Name3`
		`'Hello World!'`	`>>> Authors`
			`'Patrick, Andrew, Sidney'`
(a)	(b)	(c)	(d)

What are variables useful for? Like we said, they represent the real value. That very simple fact helps us to save lots of time. In the same way that in reality we do not need to have a cat, a mammoth, or a unicorn in front of us to speak about them, in programming we simply use variables to realize operations on the values the names refer to. For instance, in Panel (b), we multiply `3` by `4` by using the variables `X` and `Y` instead of directly using the integers `3` and `4`.

In Python, names whose values *will change* during the course of the program execution are called *variables*; otherwise they are said to be *constant*. For instance, the value of `X` in Panels (a) and (b) never changes, and so `X` is a constant. However, it changes in panel (c), and therefore, in that example it is a variable. In other programming languages, you must specify from the start if a label is a variable or a constant by using a particular syntax; there is no such rule in Python.

Python is also *dynamically typed*, in the sense that you decide at any time, what the data type of a value is (i.e., you do not have to decide it from the start like in C or Java). For instance, Panel (c) shows that X is assigned an integer, and then assigned a string. When this occurs, `X` represents the string `"Hello World!"` and not the integer `1`. This is because the string assignment statement followed the integer assignment statement.

Finally, although the choice of labels is mostly a question of personal convention, there are few rules you need to respect. First, you cannot use empty spaces in your variable names, you cannot start the variable name with a number, and you cannot use certain keywords like `if`, `and`, `def`, `return`, which are part of the Python syntax as variable names (see a Python introduction for details). Violating these rules will result in a *syntax error*. Second, because programming is about saving time, you want your labels to be meaningful. Panel (d) shows such labels for referring to our names (Name1, Name2, Name3, Authors). For the program itself it does not matter (a computer is after all just a symbol crunching machine, which does not understand symbols in the human sense), but it would save lots of trouble to any programmer who tries to understand your program (including yourself at a later date).

Data Structures

The assignment of values to labels is only one step towards the power of abstraction behind a programming language. Data structures are one step further. A data structures is basically a collection of data that is organized for efficient and quick access. Instead of having a name referring to a simple value (like a digit) we now refer to a collection of values. Just as some data types are better for some operations than others, data structures have their own specializations. Table 6 provides examples of basic data structures that are part of Python: strings, tuples, lists and dictionary.

Table 6. Examples of data structures in Python: string, tuple, list and dictionary.

```
>>> X = "abc"
>>> X
'abc'

>>> X * 2
'abcabc'

>>> X[0]
'a'

>>> X[1]
'b'

>>> X[0] = "z"
Traceback (most
recent call last):
  File
"<pyshell#34>", line
1, in <module>
    X[0] = "z"
TypeError: 'str'
object does not
support item
assignment

>>> X = "zbc"
```

(a)

```
>>> X = (1,2,"abc")
>>> X
(1, 2, 'abc')

>>> X * 2
(1, 2, 'abc', 1, 2, 'abc')

>>> X[0]
1

>>> X[0] = "z"
Traceback (most recent
call last):
  File "<pyshell#41>",
line 1, in <module>
    X[0] = "z"
TypeError: 'tuple' object
does not support item
assignment

>>> X = X + (7,8,9)
>>> X

(1, 2, 'abc', 7, 8, 9)
```

(b)

```
>>> X = [1,2,"abc"]
>>> X
[1, 2, 'abc']

>>> X * 2
[1, 2, 'abc', 1, 2, 'abc']

>>> X[0]
1

>>> X[0] = "z"

>>> X[3]
Traceback (most recent
call last):
  File "<pyshell#50>",
line 1, in <module>
    X[3]
IndexError: list index out
of range

>>> X = X + [7,8,9]
>>> X

[1, 2, 'abc', 7, 8, 9]
```

(c)

```
>>> X =
{'dog':'chien',
'mouse':'souris'}

>>> X['cat'] =
'chat'
>>> X
{'mouse':
'souris', 'dog':
'chien', 'cat':
'chat'}

>>> X['dog']
'chien'

>>> X['cat'] = 'z'
```

(d)

The operations illustrated in these examples offer only a glimpse of the myriad of possibilities. You can consult a programming reference to learn more about them. Moreover, external packages, or libraries (often available on the internet) provide other specialized data structures. Finally you can build your own data structures (for instance, a data structure to represent syntactical trees).

Let's return to the string type. Unlike the integer type, the string type is made up of other types – *characters* – which you can selectively access. Strings can be thought of as sequences of characters. For instance, Panel (a) in Table 6 shows a string of three characters (`"abc"`). You can assign such a string to a variable `X`. Type `X` and press the `Enter` key to make sure the string has been assigned to `X`. If you multiply the string by `2`, it will create a new string in which the initial string is repeated twice.

You can access the individual characters of a string by using an *index* surrounded by brackets. The first character has the index `0`, the second character the index `1`, etc. Pay close attention to the fact that the first element is indexed by the digit `0`, and not by the digit `1`! Indexing in Python starts at zero. This is a common source of confusion for novice programmers. The last character corresponds to `X[2]`. Beyond index `2`, there is nothing, so you would get an error message by typing, say, `X[3]`. Moreover, strings are *immutable*, which means that you cannot decide to selectively change one of its characters. For instance, if you want to set the first element (i.e., the element whose index is `0`) to `"z"`, you get an error message. To have a `"z"` instead of `"a"` you would need to reconstruct a string from scratch (i.e., `"zbc"`) and reassign it to the variable which pointed to your initial string (i.e., `"abc"`).

Panel (b) shows a *tuple* made of three elements. A tuple represents a sequence of values (i.e., values following each other in a strict order). Contrary to strings which only contain characters, tuples can contain any type of values. In Panel (b), the first two elements of `X` are integers (`1` and `2`) and the last element is a string (`"abc"`). Besides this exception, tuples are very much like strings. Similar to strings, you can access the element of tuples with indices starting at `0`. You will also get an error message if you go beyond the range of available elements or attempt to change an element (i.e., tuples are immutable too). As with strings you can repeat tuples (i.e., by using the multiplication operator) and use the concatenation operator (i.e., "+") to make bigger tuples.

Panel (c) illustrates the use of a *list* also made of two integers and a string. Note the change in syntax (i.e., use of square brackets instead of parentheses). Lists are like tuples (e.g., you can multiply the list by a number and obtain another list) except that lists are *mutable*. This means that you can selectively change elements of a list. For instance, you can set the first element to `"z"`. Lists are therefore more flexible than tuples.

Dictionaries are a very powerful and frequently used data type. Panel (d) shows how to use the dictionary data structure to map English words onto French words. You access the French words by using the corresponding English words. X is first initialized to a dictionary whose *key* is `dog` and *value* `chien`, etc (note the use of curly brackets and simple quotes). Note that the command is extended to more than one line because it is too large for the column in Table 6. A third element is then added (with `cat` as a key and `chat` as a value). Contrary to strings, tuples and lists which are ordered sets of elements, dictionaries are not inherently ordered. Instead of using indices to access elements of a dictionary, you access elements with the keys of any type (e.g., integers, strings, tuples). Similar to lists, you can reassign new values to the elements of a dictionary (e.g., you can assign `'z'` to the entry *cat*). In other words, like lists, dictionaries are mutable.

The essential characteristics of the string, tuple, list and dictionary are summarized in Table 7.

Table 7. Comparing four data types in Python: string, tuple, list and dictionary.

Data type	Is mutable	Is a sequence	Restrictions on values
String	No	Yes	Characters only
Tuple	No	Yes	None
List	Yes	Yes	None
Dictionary	Yes	No	None

In essence, (1) dictionaries and lists are mutable, but tuples and strings are not; (2) lists, tuples and strings are sequences, but dictionaries are not; (3) dictionaries, lists, and tuples can contain different types of elements, whereas strings only contain characters.

Conditionals

The preceding examples have portrayed programming as the execution of lines of code, one after the other in a linear fashion. Although some simple programs might be organized this way, most programs contain instructions to be executed in some circumstances, and avoided in others, depending on a test. If the test is passed, then the corresponding lines of code can be executed. For instance, you could design rules to determine the gender of Spanish names: *if* the name ends with the letter *o* (like in *Mario*), *then* the person is a male, but *if* the name ends with the letter *a* (like in *Maria*), *then* the person is a female, etc. Such *if-then* sequences are called *conditionals* (see the examples in Table 8).

Panel (a) shows a sequence of tests for the value of `X`. First `X` is assigned the value `2`. The first conditional assigns the value `0` to `X` if `X` is equal to `1` (note the use of the equality operator `= =`, different from the assignment operator). The two other conditionals test whether `X` is smaller, or greater than `1`. Panel (b) shows the same program but where `X` is initialized to `1`, instead of `2`. Panel (c) uses the *logical operators* (and, or, not). The first conditional checks whether `X` is equal to `1 and Y` is greater than `3`. The second conditional checks whether `X` is equal to `1` or `Y` is greater than `3`. The third conditional checks whether it is true that `Y` is not greater than `3`.

Table 8. Examples of conditionals in Python.

`>>> X = 2`	`>>> X = 1`	`>>> X = 1`
		`>>> Y = 2`
`>>> if X == 1: X = 0`	`>>> if X == 1: X = 0`	
`>>> X`	`>>> X`	`>>> if X == 1 and Y > 3: print 'a'`
`2`	**`0`**	
		`>>> if X == 1 or Y > 3: print 'b'`
`>>> if X < 1: X = 3`	`>>> if X < 1: X = 3`	
`>>> X`	`>>> X`	**`'b'`**
`2`	**`3`**	
		`>>> if not Y > 3: print 'c'`
`>>> if X > 1: X = 4`	`>>> if X > 1: X = 4`	
`>>> X`	`>>> X`	**`'c'`**
`4`	**`4`**	
(a)	(b)	(c)

Iterations

Programs are especially helpful when they repeat operations a great number of times. For instance, imagine that you have to count the number of persons who have a disyllabic name in the telephone directory. One could easily write a program that reads the directory one entry at a time, checks whether the name is disyllabic and increments a counter if that is the case. *Iterations* are one way to repeat lines of code in a program. Recall that Table 1 involved pseudo-code that iterated through the lines of a document and translated each line. Panel (a) in Table 9 first shows how to iterate through the characters of a string. Note the use of a `for ... in ... :` syntax.

We can also iterate through the elements of a list, using the same syntax. Panel (b) shows how to iterate through a dictionary. The first two `for ... in ... :` statements return the keys of the dictionary (i.e., *dog* and *cat*), whereas the third statement returns the values (i.e., *chien* and *chat*). The last statement returns tuples with both keys and values.

Panel (c) uses a `while` loop to iterate through the elements of a list. The index `i` is initialized to `0`. Then, while `i` is less than `3`, we keep printing the i^{th} element of `X` and then increment `i` by one. The `while` statement actually contains two commands, separated by a semicolon (`;`). Semicolons are used when several commands are on the same line. The `while` statement has been repeated in Panel (c), this time with one command per line. In that case, it is not necessary to use the semicolon. However, it is now necessary to use some indentation in order to tell Python that these commands are to be executed "inside that while loop". This is called a nested structure. You can indent your code with the `Tab` key on your keyboard.

Table 9. Examples of iterations in Python.

```
>>> X = "abc"

>>> for z in X: print z
a
b
c


>>> X = [1,2,3]

>>> for z in X: print z
1
2
3
```

(a)

```
>>> X = {}
>>> X['dog'] = 'chien'
>>> X['cat'] = 'chat'

>>> for z in X: print z
dog
cat

>>> for z in X.keys(): print z
dog
cat

>>> for z in X.values(): print z
chien
chat

>>> for z in X.items(): print z
('dog', 'chien')
('cat', 'chat')
```

(b)

```
>>> X = [1,2,3]

>>> i = 0

>>> while(i < 3): print X[i]; i=i+1
1
2
3

>>> i = 0

>>> while(i < 3):
        print X[i]
        i = i + 1

1
2
3
```

(c)

Functions

Functions are key components of programming abstraction. A function is like a mini-program which you can invoke at any time in your program. For example, you might need to compute the average daily balance of a million customers. This can be easily achieved by using a function for calculating the average for one customer. This function can then be used a million times; once for each customer. In another example, the rule for determining the gender of Spanish name could be placed in a function called `gender()`. When you needed to apply the rules to a name, you would call this function and retrieve the gender of a person. All the details associated with the rules to determine a person's gender would be conveniently hidden from you. The organization of your code would be improved, thereby allowing you to navigate through it with ease. Functions makes it easier to engage in the incremental programming approach described earlier (see Table 2). For instance, one could implement the example in Table 2, by writing all the code except for a `translate()` function which would be fleshed out last. This approach is illustrated in the implementation section of the NLP project in Part II.

Another advantage of functions is that you reuse functions programmed by other people or write your own if you cannot find one to fit your needs. For instance, the `type()` command used Table 3 is a function already defined in Python (i.e., you do not have to program it yourself). In further section, we will explain how you can build your own functions.

Functions have *parameters* and *return values*. Return values are the values that functions generates and can which can be reused inside the program. For instance, `type(123)` returns the string `<type 'int'>`. You could capture that string by assigning it to a variable: `x = type(123)`. Parameters specify how you can use a function. The `type()` function has an "object" parameter. One could formulate this by writing `type(object)`. In our previous example, we decided to inquiry about the type of the number `123`, by calling the function `type()` with `123` as an *argument*: `type(123)`. We call `123` an argument because it is the specific values that we decide to assign to the "object" parameter of the function.

Table 10 illustrates the notion of functions that are available in an external library (i.e., outside of your code). Defining your own functions is sometimes tricky, and it is often a significant time-saver to reuse functions developed by other programmers who gave them careful thought. The library we are using in this example is NLTK. The example was inspired by the second chapter of Bird et al (2009). It illustrates how to load a sample of the Gutenberg corpus (`http://www.gutenberg.org`). The Gutenberg project makes available over 30,000 free electronic books (eBooks). It was started in the 1970s by Michael Hart and includes books like Alice in Wonderland (by Lewis Carroll), Moby Dick (by Herman Melville), Roget's Thesaurus, the Devil's dictionary (by Ambrose Bierce), and the Bible. The example has also been built step by step – from Panel (a) to Panel (h) – in order to illustrate the incremental approach to programming, hereby stimulating you to adopt a similar attitude when you repeat the examples, play with the code, or work on a programming project of your own. Panel (i) shows the final result. In Panel (i), comments (in italics) have been added to the program to help you understand what the code is doing. Comments must start with a `#` sign (called a hash, pound, or number sign). This sign tells Python not to execute the statement that follows. It is a good practice to put a reasonable amount of comments in your code in order to facilitate its readability and maintenance.

NLTK is designed as a Python module – a resource which is not inherently provided with Python but can be imported at will. To import the NLTK module, we use the `import` statement at the beginning of the code. But before being able to import the module you need to download and install it on your machine (see the companion website for instructions). In Panel (a), there is an `import` statement to load the real division function (pay attention to the use of double underscores). Depending on the version of Python you use you may not need that statement. In some versions of Python, the division sign (/) does not behave like the conventional division operator, and that statement is therefore necessary. A second `import` statement loads NLTK. Finally a third `import` statement refers to the NLTK sample of Gutenberg corpus.

Table 10. Use of external functions present in the NLTK module.

(a)
```
from __future__ import division

import nltk

from nltk.corpus import gutenberg
```

(b)
```
path = "C:/test/data/nltk"
nltk.data.path = [path] + nltk.data.path
```

(c)
```
for id in gutenberg.fileids(): print id
```

(d)
```
for id in gutenberg.fileids():
       print id
```

(e)
```
for filename in gutenberg.fileids():
       print filename
```

(f)
```
for id in gutenberg.fileids():
       NC = len(gutenberg.raw(id))
       print NC
```

(g)
```
for id in gutenberg.fileids():
       NC = len(gutenberg.raw(id))
       NW = len(gutenberg.words(id))
       print NC / NW
```

(h)
```
for id in gutenberg.fileids():
       NC = len(gutenberg.raw(id))
       NW = len(gutenberg.words(id))
       NS = len(gutenberg.sents(id))
       AWL = NC / NW
       ASL = NW / NS
       print AWL, ASL, id
```

(i)
```
# import
from __future__ import division
import nltk
from nltk.corpus import gutenberg

# path
path = "C:/test/data/nltk"
nltk.data.path = [path] + nltk.data.path

# for each file in the corpus, get its id
for id in gutenberg.fileids():

# get number of characters
NC = len(gutenberg.raw(id))

# get number of words
NW = len(gutenberg.words(id))

# get number of sentences
NS = len(gutenberg.sents(id))

# average word length (in characters)
AWL = NC / NW

# average sentence length (in words)
ASL = NW / NS

# print on the screen
print AWL, ASL, id
```

In Panel (b), to make sure that NLTK knows where to find the corpus, we set a path variable, and we add it to the list of paths called `nltk.data.path` which is provided by NLTK (Panel b). To add a new path to the list, we turn the path (a string data type) into a list, by surrounding it with brackets (`[ ]`). The `+` operator becomes in this context a concatenation operator and adds the two lists together to form a larger list. The specific path you will use will depend on where you decided to install the NLTK corpora. Please see the website for further instruction regarding the confusing topic of system path. It is not particularly complicated to understand, but it might be intimidating for the novice.

In Panel (c), we iterate through all files of the corpus, retrieving the name of each file (e.g., `carroll-alice.txt`, `Shakespeare-hamlet.txt`) in a list with the `fileids()` function. That code is interesting because, if it executes properly, not only can we see all the files in the corpus, but we also know that we have access to the corpus through the NLTK module! Once again, this illustrates the incremental approach to programming. When you develop your code for the first time, it is preferable to ensure that the basic elements of your code function properly before engaging in more sophisticated operations. Pay close attention to the fact that we did not have to define the `fileids()` function; it is provided by NLTK, and we are happy to use it! Finally, note that all of this is done on one line. This is not practical in more complex situations, so a more readable version of the code is presented in Panel (d).

As you may have noticed, the filename being printed is contained in a variable called `id`. We could have called it `filename`, like in Panel (e). The choice does not really matter so far as we are consistent with ourselves (see a Python introduction for more coverage on that topic). However, the name `fileids()` is fixed, and we must use it if we want to invoke that function.

Inside the loop presented in Panels (c)-(e), more interesting operations than printing can occur. In panel (f), we use the `raw()` function of the Gutenberg corpus to retrieve all the characters of the file in a string. We provide the result of that function to another function which, this time, is built into Python (i.e., is not part of NLTK): the `len()` function (short for *length*). In this context, `len()` counts the number of characters (i.e., the length of the string in characters). That count is then assigned to the `NC` variable, whose content is finally printed. The len() function is very useful and flexible function which will adapt its behavior to different data types. For instance, for a list `A`, len(A) will count the number of elements in A.

Panels (g) and (h) represent two more steps in our incremental programming journey. In (g) we retrieve the words through the `words()` function. After counting the number of words, and dividing the number of characters by the number of words, you get the average size of words in terms of characters. In Panel (h) we retrieve the number of sentences through the `sents()` function, and compute the average size of sentence in terms of words. Three values are then printed. Pay attention to the commas in the print statement. These commas serve to separate the three values within a tuple. What is therefore being printed is a tuple. Again, play around with the code (e.g., compute the average size of sentence in terms of characters) to gain further insight.

Imagine the time you are saving by using functions to access text files and information without having to deal with the difficult aspects of file manipulation, tokenization (i.e., character and word recognition), and sentence splitting. As is apparent in Chapter XXX of this book, such operations are time consuming. Panel (i) shows the final code.

What would you do if you had to repeat the same operations for other corpora? For instance, you may want to repeat the exercise for the book of Genesis, translated in several languages, also available through NLTK. Would it not be annoying to rewrite the same lines of code? The good news is that you can embed any part of your program in a function and make it generic enough so that you can reuse it in multiple contexts (i.e., Gutenberg, the book of Genesis, etc.). Panel (a) in Table 11 shows precisely how to do that by creating a `getCorpusInfo()` function.

A function definition starts with a `def` keyword, then the name of the function (`getCorpusInfo`), followed by parentheses and, possibly one or more *parameters*. In the case of `getCorpusInfo()` there is only one parameter (`corpus`) which stands for the name of the corpus (whatever it is) that is going to be used when the function is invoked. The `def` statement ends with a colon (`:`), and the rest of the program is similar to Table 10 with the exception that, first, there is another layer of indentation; second, we removed the comments; and, third, we replaced the name `gutenberg`, by the more generic name `corpus`. When the function has been defined, you simply need to import the corpora, and then use the `getCorpusInfo()` function with the name of your corpora passed as an argument to the function (e.g., `getCorpusInfo(Genesis)`).

Table 11. Defining your own functions.

(a)

```
# import
from __future__ import division
import nltk
from nltk.corpus import (gutenberg, genesis)

# path
path = "C:/test/data/nltk"

nltk.data.path = [path] + nltk.data.path

# my function
def getCorpusInfo(corpus):
        for id in corpus.fileids():
                NC = len(corpus.raw(id))
                NW = len(corpus.words(id))
                NS = len(corpus.sents(id))
                AWL = NC / NW
                ASL = NW / NS
                print AWL, ASL, id

# get information about each corpus
getCorpusInfo(gutenberg)
getCorpusInfo(genesis)
```

(b)

```
# import
from __future__ import division
import nltk
from nltk.corpus import (gutenberg, genesis)

# path
path = "C:/test/data/nltk"

nltk.data.path = [path] + nltk.data.path

# my functions
def getCorpusInfo(corpus):
        for id in corpus.fileids():
                getTextInfo(id, corpus)

def getTextInfo(id, corpus):
        NC = len(corpus.raw(id))
        NW = len(corpus.words(id))
        NS = len(corpus.sents(id))
        AWL = NC / NW
        ASL = NW / NS
        print AWL, ASL, id

# get information about each corpus
getCorpusInfo(gutenberg)
getCorpusInfo(genesis)

getTextInfo("austen-emma.txt", gutenberg)
getTextInfo("french.txt", genesis)
```

An excerpt of the result of applying the function is displayed in Figure 1. On Figure 1, you can see there that the average word length of the texts from the Gutenberg corpus does not vary very much (i.e., between 4 and 5 for every line), as explained in Bird et al (2009) because they all are English texts and the English language exhibits this property. When you explore the Genesis corpus, you realize that the variability increases, probably because the corpus contains texts written in different languages.

```
>>> getCorpusInfo(Gutenberg)
4.60990921232 18.3701193317 austen-emma.txt
4.74979372727 23.6841978287 austen-persuasion.txt
4.75378595242 21.8144838213 austen-sense.txt
4.28688156382 33.5843551657 bible-kjv.txt

(...)

>>> getCorpusInfo(genesis)
4.3676838531 30.6183310534 english-kjv.txt
4.28260770872 19.7374551971 english-web.txt
5.94474169742 15.0834879406 finnish.txt
3.44761394102 40.367965368 swedish.txt
```

Figure 1. Using functions and NLTK to analyze different corpora.

The main difference between Panel (a) and Panel (b) is that we made a subpart of the original `getCorpusInfo()` function, a function in its own right, and called it `getTextInfo()`. By doing so, we modularized our code and increased its flexibility. For instance, by separating a `getCorpusInfo()` function and a `getTextInfo()` function, we can call `getTextInfo()` and therefore obtain information on only one text (instead of information on all texts). Moreover, when we are working on the code, we may modify one function without having to alter the other one.

Text File Input/Output

After going through all the examples presented so far, you most likely have learned important concepts but all the output which has been produced on the screen is now gone, or waiting for you to be manually copied in a text file, etc. Would it not be simpler if you could save the output information directly to a file? Similarly, what would you do if you wanted to open a text file, read it, process its contents, and save it under another name? These steps are the landmarks of many NLP programs which primarily handle text files. To illustrate these operations, you may for instance use the text displayed in Table 12.

Table 12. A sample from the Devil's Dictionary.

BEG, v.	To ask for something with an earnestness proportioned to the belief that it will not be given.
DENTIST, n.	A prestidigitator who, putting metal into your mouth, pulls coins out of your pocket.
LANGUAGE, n.	The music with which we charm the serpents guarding another's treasure.
LIBERTY, n.	One of Imagination's most precious possessions.
MEDAL, n.	A small metal disk given as a reward for virtues, attainments, or services more or less authentic.
NEPOTISM, n.	Appointing your grandmother to office for the good of the party.
PAINTING, n.	The art of protecting flat surfaces from the weather and exposing them to the critic.
PLAN, v.t.	To bother about the best method of accomplishing an accidental result.
SCRIBBLER, n.	A professional writer whose views are antagonistic to one's own.
TRUTHFUL, adj.	Dumb and illiterate.

The text presented in Table 12 is composed of ten entries extracted from the Devil's Dictionary by Ambrose Bierce. Type each entry on a separate line in a text file and save the file as `dictionary.txt` in the data folder. Or simply, copy and paste these lines from the companion website into a text file.

Panel (a) in Table 13 shows how to print the objects in your data directory (i.e., files and folders). There is an `import` statement for the `os` (i.e., operating system) module. The `listdir()` function returns a list of objects found in the path (`p`) passed as an argument. After executing that code, you should at least see `dictionary.txt` in the list displayed on the screen. In panel (b) we then add the filename `dictionary.txt` to the path. We import the `fileinput` module (a module to read and save files). Then, for each line in the file, we print the length of the line (in number of characters). Finally the input file is closed with the `close()` function. In Panel (c), two paths are set, one for the input (`dictionary.txt`) and one for the output (`dictionary.out.txt`). The built-in `open()` function specifies that a file must get ready to receive information in appending mode (i.e., the file will add text incrementally). For each line, we compute the length (in characters), print the length of the line and the line itself to the output file. The length of the line and the line are separated by a tabulation (represented by a tab sign `\t`). Finally we close the input and the output.

This short example of file operations concludes Part I. Although Part I was about programming basics, we have already laid a foundation for solving NLP problems. Part II builds on this foundation by concretely showing how to conduct an NLP project.

Table 13. Processing text files in Python.

```python
# set path
p = "C:/test/data"

# import
import os

# for each object
for x in os.listdir(p):
       print x
```

(a)

```python
# set path
p = "C:/test/data"

p = p + '/dictionary.txt'

# import
import fileinput

# for each line
for li in fileinput.input(p):

       L = str(len(li))

       print L + " :: " + li

# close
fileinput.close()
```

(b)

```python
# set path
p = "C:/test/data"

p_in = p + '/dictionary.txt'
p_out = p + '/dictionary.out.txt'

# import
import fileinput

# open file for saving
out = open(p_out, 'a')

# for each line
for line in fileinput.input(p_in):

       L = str(len(line))
       print line

       info = L + "\t" + line
       out.write(info)

# close
fileinput.close()
out.close()
```

(c)

PART II. A STEP-BY-STEP EXAMPLE

The chapter so far has been focused on programming basics. The section on Functions in Part I included an example of reuse of resources (NLTK) within a Python program (see Tables 10 and 11). That application was however not motivated by any specific NLP requirements. In other words, we were not in a situation in which we had to solve a particular problem. In this section, we describe an actual NLP problem and a possible solution. This problem and its solution have been chosen to offer a reasonable degree of complexity to both the novice and the more advanced programmer. The companion website includes other problems and solutions with various degrees of complexity and implemented in different programming languages (e.g., Python, Perl, and Java).

The Problem of Predicting the Speed of Lexical Access

The problem we will tackle in this section is fairly complicated, and we will only propose one possible – and easy to understand – solution. Humans have the remarkable ability to learn and use language. When they read a text they need to access the meaning of words in order to construct the meaning of sentences. Contrary to programming languages, with unambiguous semantics, natural languages contain numerous semantic ambiguities. For instance, a word like *page* is ambiguous in that it can refer to the side of a sheet of paper, as well as to a young person employed to do errands. Humans are usually very good at determining which meaning of a word is intended in a given text by examining the context surrounding the words.

Although humans seamlessly perform semantic disambiguation activities, the underlying cognitive processes employed for this task are not fully understood. Actually, even a seemingly simple step such as recognizing that a word is known (i.e., to access it in the mental lexicon) is still the topic of much research. Psycholinguistic experiments have demonstrated that it takes some time (several hundred milliseconds) to decide whether written words are known or not, and that response time is influenced by the different meanings of the words. In such experiments, words are presented on a computer screen, one at a time. In typical experiments, the participant has to decide whether a stimulus word is an existing word (e.g., *page*) or if it is a non-word (e.g., *gepa*), by pressing one of two keys on the keyboard. The number of milliseconds elapsed between the time the word (or non-word) is displayed and the time the key is pressed is called the response time, and is our primary measure of interest.

Many useful insights have been gleaned with this rather simple experimental protocol. For instance, Rodd, Gaskell and Marslen-Wilson (2002) found out that it takes on average longer to recognize ambiguous words (i.e., to establish that they are existing words) like *page* than unambiguous words like *load*. In one of their experiments, they operationalized ambiguous words as ones that had more than one entry in the Wordsmyth Dictionary (Parks et al., 1998). For instance, the word *page* has two different entries whereas the word *load* has only one entry.

Moreover, Rodd et al (2002) demonstrated that the number of senses of the words also had an impact on the speed of lexical access. By senses they referred to all the meanings that a word could have, whether these meanings were semantically related or not. For instance, in the Wordsmyth Dictionary, the first entry of the word *page* (the "sheet of paper") corresponds to a variety of senses including the content of the paper, an instance from the past (like in "a page of history"), the act of turning pages, etc. Similarly, the second entry of the word *page* (the "young person") has several senses including a young person who attends to a king or the act of calling someone via electronic communication (i.e., with a pager). Most interestingly, they found out that the number of senses has a reverse effect on lexical access time: while more ambiguity slowed down response time, more senses sped it up.

Several psychological accounts attempt to explain the impact of ambiguity and number of senses on response times (see Rodd et al, 2002). What matters for our limited NLP exercise, however, is the information about ambiguity and number of senses can be used to automatically predict response time. Such an automated application could be used to design experimental material, develop more efficient advertising campaigns. It can even find use in applications aimed at assessing text difficulty (e.g., Graesser, McNamara, Louwerse, & Cai, 2004). Although many more factors could potentially influence the ease of lexical access (Rayner, 1998; Rodd et al, 2002), semantic ambiguity and number of senses seem to be two important factors to take into account. For the sake of this example we define our NLP problem as follows: given any word, estimate its ambiguity, number of senses, and response time. The next section offers a simple solution to this problem.

Designing a Solution

The first question of course is *what knowledge and what computations shall we use to do the job?* Our problem actually involves three sub problems. The first one pertains to determining the ambiguity of a word. The second one concerns counting the number of senses of the word. The third one deals with determining the response time, on the basis of ambiguity and number of senses. For the first sub-problem, we know from the previous description of Rodd et al's (2002) experiment that they have used the online Wordsmyth dictionary to retrieve the number of entries associated with a given word; this is their measure of ambiguity. So we need some way to extract information from the online Wordsmyth dictionary. Although one could imagine accessing the online Wordsmyth dictionary from within the program, it is not convenient (e.g., when an internet connection is not available). Moreover, accessing online resources through artificial means (i.e., web crawlers etc) is usually discouraged by developers of online services. Hence, we decided to download a freely accessible dictionary which had a similar structure to the Wordsmyth dictionary. The dictionary we chose to use is the GNU version of the collaborative international dictionary of English, presented in the Extensible Markup Language (GCIDE_XML; Dyck, 2002).

For the second sub-problem, among other things, Rodd et al (2002) relied on WordNet (Miller, 1995). WordNet is a widely used online resource and was designed to serve as a lexical resource that was organized according to the principles underlying human semantic memory (the organization of words in terms of meaning). Moreover, WordNet is organized in terms of senses – whether they are semantically related or not – and contains information that is not readily available in conventional dictionaries such as the Wordsmyth dictionary or the GCIDE. Finally, it is also accessible in Python through the NLTK module (Bird et al, 2009). These advantages make WordNet look like an ideal candidate for a solution written in Python. A broad overview of our problem and its solution is depicted in Figure 2.

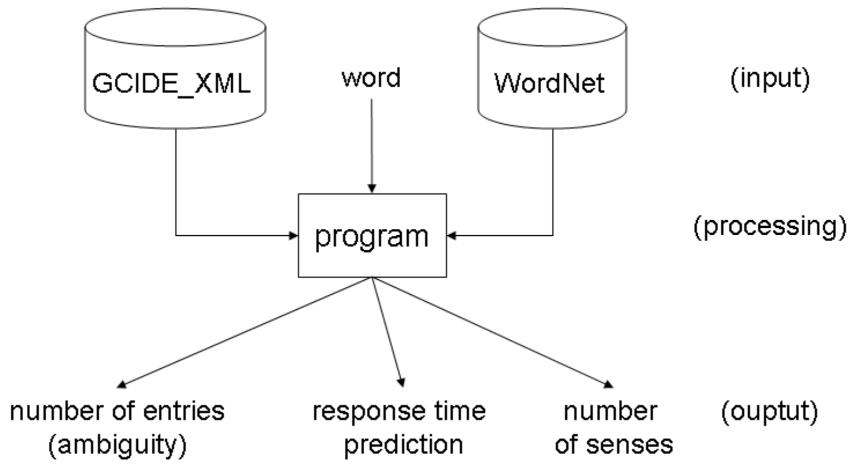

Figure 2: General representation of the problem and its solution

Pseudo-code for the general procedure used in the program to produce the number of entries, the number of senses, and the predicted response time is presented in Table 14.

Table 14. Pseudo-code describing the general procedure of the program.

```
Load all the words into a dictionary W.

For each word w in W [1],

         # Pseudo-code to obtain ambiguity

         Open the page of GCIDE where you can find the word w.
         Count the number of entries for the word w.

         If that number is greater or equal to an "ambiguity threshold",
                  the word w is ambiguous; otherwise it is unambiguous.

         Close the page of the GCIDE.

         # Pseudo-code to get number of senses
         Retrieve the number of senses in WordNet for the word w.

         If that number is greater or equal to a "sense threshold",
                  the word w has many senses; otherwise it has few senses.

         Predict the mean response time of the word w.
         Add the new information about the word w to the dictionary W.

Evaluate the quality of the classification.

Write the information stored in W to a file.
```

To test our program we plan to read a text file containing the words from Rodd et al's (2002) Experiment 2. For evaluation purposes, the text file will also indicate whether the words are ambiguous or not, and whether they have many senses or not, as indicated in Rodd et al's (2002) Appendix B. Table 15 displays the first five lines of this file. This information is loaded in a dictionary `W`. For each word `w` in `W`, the number of entries of `w` is first determined by consulting the GCIDE. Based on the number of entries in the

[1] Note that `w` (lower case) refers to a word in the dictionary `W` (upper case).

GCIDE and a preset "ambiguity threshold" we decided if `w` is ambiguous or not. The number of senses is then retrieved from WordNet. This information, coupled with a preset "sense threshold" is used to decide if `w` has few or many senses. These two pieces of information are used to predict the mean response time for `w`. These findings are saved in the main dictionary `W`. When all words have been processed, `W` is saved to a file, and the quality of the obtained classification (ambiguous vs. unambiguous; few senses vs. many senses) is evaluated by comparing it to results reported in Rodd et al (2002).

Table 15. Five first lines of input to the program; data is from Rodd et al's (2002) Experiment 2.

Word	**Ambiguous (1 = yes ; 0 = no)**	**Has many senses (1 = yes ; 0 = no)**
ash	1	0
duck	1	1
heap	0	0
roll	0	1

The pseudo-code of Table 14 left out some essential information such as the threshold values and how to compute the mean response time from ambiguity and number of senses. We turn to Rodd et al's (2002) experiment for this data. The authors defined ambiguous words as ones with two or more entries in the Wordsmyth dictionary. Unfortunately, the GCIDE has more entries than the Wordsmyth dictionary, which will likely lead to an overestimation of the number of ambiguous words when applying Rodd et al's (2002) rule. For this reason, the optimal value for our threshold will have to be estimated from the data. This task will be tackled in the *Implementing the Solution* section.

Similarly, Rodd et al (2002) used a procedure to determine whether a word has many senses or only a few. That rule however was not made fully explicit, and relied on both the Wordsmyth dictionary and WordNet. For the sake of simplicity we decided to use a simple threshold based on the number of senses extracted from WordNet, and estimated this threshold from the data similar to the ambiguity threshold.

The predicted response times correspond to the mean response times of Rodd et al's (2002) Experiment 2. The "prediction" is simply the assignment of one of five mean response times, depending on the nature of the word: 587 ms for ambiguous words with few senses, 578 ms for ambiguous words with many senses, 586 ms for unambiguous words with few senses, 567 ms for unambiguous words with many senses and 659 ms for non-words.

Implementing the Solution

The following description is a step-by-step implementation of the program. That description involves nine steps. You will find the corresponding nine versions of the script on the companion website. This step-by-step implementation follows the incremental approach to programming which has been advocated at the beginning of this chapter. You should be able to execute each step of the program in the Python interface and analyze its behavior.

Step 1 lays down the general structure of the program (see Table 16). Let's read it from bottom to top. The main program is located at the bottom of the code. It calls four functions (`load_words()`, `process_words()`, `evaluate_results()`, and `save_results()`) that we need to be defined. The `load_words()` function will load the input material in the format specified in Table 15. The `process_words()` function will generate the results we attempt to achieve. The `evaluate_results()` function will evaluate those results, and the `save_results()` function will save them to a file. Above the main program, a *functions definitions* section contains place holders for these definitions, using the `pass` keyword to tell Python to ignore these incomplete definitions. We will implement these functions one by one, following the natural progression from the input to the output. A *parameter* section will contain data which the user of the program will be able to change at will (e.g., specifying other values for the path to the input data file). The `words_input_path` variable contains the path to the input material, it is used as an argument of the `load_words()` function to load the desired material. The *import* section is still empty but will contain import statements as needed.

Table 16. The general structure of the program.

```
# import

# parameters
words_input_path = "C:/test/data/Rodd_et_al_2002_experiment_2.txt"

# functions definitions
def load_words(input_path): pass
def process_words(): pass
def evaluate_results(): pass
def save_results(output_path): pass

# main program
print "--- START"
words = load_words(words_input_path)
process_words()
evaluate_results()
save_results(output_path)
print "--- END"
```

Step 2 consists of implementing the `load_words()` function (see Table 17). Since it uses the `fileinput` Python module, we need to add an import statement accordingly. This function initializes a dictionary called `W`, and stores information about each word, using the word as a key in the dictionary `W`. More detailed comments about this function are available in the function itself (Table 17).

Table 17. The `load_words()` function.

```
# import
import fileinput

# parameters
(...)
def load_words(input_path):
    W = {} # initialization of the dictionary
    i = 0 # counter used to skip the first line of the file because it is a header
    for line in fileinput.input(input_path):
        i = i + 1
        if i == 1: continue # if this is the first line, skip it (i.e., continue to the next line)
        line = line.strip() # remove the white character at the beginning and end of the line
        information = line.split("\t") # split the file, using the tab as a separator
        (word, ambiguous, many_senses) = information # assign its elements to distinct variables
        W[word] = {} # initialize a dictionary in the dictionary with that word as a key
        W[word]["ambiguous"] = int(ambiguous) # we record the value, converting it to an integer
        W[word]["many_senses"] = int(many_senses)
    fileinput.close()
    return W # we return the dictionary
```

In step 3, we implement the process_words() function (see Table 18). It will apply five other functions (for which we create place holders), for each word in `W`. We will now implement each of them, one at a time.

In step 4, the `retrieve_number_of_entries()` function is implemented (see Table 19). The path to the folder containing the GCIDE is defined in the *parameters* section. The parentheses *(...)* are not part of the program but indicate that a portion of code has been skipped. The function will be looking for the word surrounded by two HTML tags (`<hw>`, `</hw>`), which is the way separate entries are distinguished in the GCIDE. Since the dictionary is distributed on separate files (one for each letter of the alphabet), the first letter of the word is extracted in order to identify the file in which the word must be searched. The file is then opened and read line by line, while counting the number of times the word is encountered.

Table 18. The `process_words()` function.

```
def process_words():
    for word in W:
        print word
        retrieve_number_of_entries(word)
        determine_word_ambiguity(word)
        retrieve_number_of_senses(word)
        determine_if_word_has_many_senses(word)
        predict_mean_response_time(word)

def retrieve_number_of_entries(word): pass
def determine_word_ambiguity(word): pass
def retrieve_number_of_senses(word): pass
def determine_if_word_has_many_senses(word): pass
def predict_mean_response_time(word): pass
```

Table 19. The `retrieve_number_of_entries()` function.

```
# parameters

GCIDE_xml_path = "C:/Resources/Various/P/Programming/NLP/chapter/data/gcide_xml"
(...)
def retrieve_number_of_entries(word):
    target = "<hw>"+word+"</hw>" # the target is the word with HTML tags
    target = target.upper() # we convert to upper case letters for normalization purposes
    first_letter = word[0] # we extract the first letter of the word
    path = GCIDE_xml_path + "/gcide_" + first_letter + ".xml" # define the file to the page
    entries = 0 # we initialize the counter for number of entries

    for line in fileinput.input(path):
        line = line.upper() # to upper case letter, again for normalization purposes
        line = line.replace('`', '') # ` characters are removed
        line = line.replace('"', '') # " characters are removed
        line = line.replace('*', '') # * characters are removed
        if target in line: entries = entries + 1 # if we find an entry, we count it

    fileinput.close()
    W[word]["number_of_entries"] = entries # record the number of entries
```

One will note that this is a time consuming operation, which will be unduly repeated for each word. For instance, a same dictionary page will be read twice for two words with the same first letter. Our function is more straightforward from an instructional point of view and could be replaced by a more computationally efficient one. A standard approach would be to read all words into a dictionary similar to `W` and store it for later reuse.

In step 5, the `determine_word_ambiguity()` function is developed (see Table 20). It stores a value of 1 for words whose number of entries are greater than or equal to a specified threshold.

Table 20. The `determine_word_ambiguity()` function.

```
# parameters

ambiguity_threshold = 4
(...)
def determine_word_ambiguity(word):
    if W[word]["number_of_entries"] >= ambiguity_threshold:
        W[word]["predicted_ambiguity"] = 1
    else:
        W[word]["predicted_ambiguity"] = 0
```

In step 6, the `retrieve_number_of_senses()` and `determine_if_word_has_many_senses()` functions are implemented. The *import* section is modified to import the NLTK module. The *parameters* section is modified to include the path to the NLTK resources. The `retrieve_number_of_senses()` function simply counts the number of senses retrieved from WordNet. The senses for any given word are accessed by NLTK through the `synsets()` function which returns a list of senses. The size of that list is then calculated by the `len()` function.

Table 21. The `retrieve_number_of_senses()` and `determine_if_word_has_many_senses()` functions.

```
# import

import fileinput, nltk

# parameters

NLTKp = "C:/test/data/nltk"
nltk.data.path = [NLTKp] + nltk.data.path

senses_threshold = 9
(...)
def retrieve_number_of_senses(word):
    number_of_senses = len(nltk.corpus.wordnet.synsets(word))
    W[word]["number_of_senses"] = number_of_senses

def determine_if_word_has_many_senses(word):
    if W[word]["number_of_senses"] >= senses_threshold:
        W[word]["predicted_many_senses"] = 1
    else:
        W[word]["predicted_many_senses"] = 0
```

In step 7, the `predict_mean_response_time()` function is implemented. It simply assigns the values (in milliseconds) which are defined in the `RT` dictionary in the *parameters* section. The values of `RT` come from Rodd et al's (2002) Table 8. A word which has a number of senses equal to zero is considered as a non-word and therefore assigned the response time associated to non-words.

Table 22. The `predict_mean_response_time()` function.

```
# parameters

RT = {}
RT["ambiguous_few_senses"] = 587
RT["ambiguous_many_senses"] = 578
RT["unambiguous_few_senses"] = 586
RT["unambiguous_many_senses"] = 567
RT["nonwords"] = 659
(...)
def predict_mean_response_time(word):

    if W[word]["number_of_senses"] == 0:
        W[word]["predicted_response_time"] = RT["nonwords"]
    else:
        ambiguity = "unambiguous"
        senses = "few_senses"

        if W[word]["predicted_ambiguity"] == 1:
            ambiguity = "ambiguous"

        if W[word]["predicted_many_senses"] == 1:
            senses = "many_senses"

        W[word]["predicted_response_time"] = RT[ambiguity + "_" + senses]
```

In step 8, we implement the `evaluate_results()` function (see Table 23). It starts by building six lists, `A`, `B`, `A1`, `A2`, `B1`, and `B2` with the codes corresponding to our classification as well as to the data of Rodd et al (2002). The lists `A` and `B` are made of tuples – one tuple per word – where each tuple consists of two integers (`0` or `1`). Each tuple is therefore either `(0,0)`, `(1,0)`, `(0,1)` or `(1,1)`. The first integer specifies whether the word was classified as ambiguous (1) or not (0), whereas the second integer represents whether the word was classified has having many (1) or few (0) senses. The procedure we have decided to use to build these lists of tuples employs the *list comprehension* feature of Python. It is essentially a device to build lists in a more concise way. Basically, we iterate through `W` for each word `w`, and build a tuple with the ambiguity value and the sense value for `w`. All these tuples are then put in a list.

The list `A` corresponds to the classification made by our program whereas the list `B` corresponds to the classification of Rodd et al (2002). We then compare these two classifications through a `confusion_matrix()` function. That function builds a confusion matrix through a function provided by NLTK (similar to the one displayed on Table 25). The percentage of correct responses is then computed. Finally, the matrix and the percentage are displayed on the screen. Note that at the time of the writing the operator division truncated the result of the division to an integer in the presence of integer operands (e.g., `7/2` would give `3` instead of the expected `2.5`). In order to change that behavior it was necessary to add the `from __future__ import division` statement at the top of your file.

In order to evaluate the ambiguity classification separately from the sense classification we build four more lists: `A1` and `B1` for ambiguity classification and `A2` and `B2` for sense classification. These new lists are obtained by transferring the corresponding members of each tuple to separate lists, through the `zip()` function. The asterisk preceding the argument in that function (e.g., `zip(*A)`), is a way of telling Python to "unzip" the members of the tuples to separate lists. Confusions matrices are finally constructed for these two groups.

Table 23. The `evaluate_results()` and `confusion_matrix()` functions.

```
# import

from __future__ import division
import fileinput, nltk
(...)
def evaluate_results():

    A = [(W[w]["predicted_ambiguity"],W[w]["predicted_many_senses"]) for w in W]
    B = [(W[w]["ambiguous"],W[w]["many_senses"]) for w in W]
    confusion_matrix(A, B) # analyze our joint classification in terms of ambiguity and senses

    A1, A2 = zip(*A)
    B1, B2 = zip(*B)

    confusion_matrix(A1, B1) # analyze our ambiguity classification
    confusion_matrix(A2, B2) # analyze our senses classification

def confusion_matrix(reference, test):
    CM = nltk.ConfusionMatrix(reference, test) # build the confusion matrix
    percent_correct = CM._correct / len(reference)
    print CM
    print "correct = " + str(round(percent_correct, 2)) + " % "
```

The two thresholds used in our program have been estimated by running the program with different values, and keeping the values corresponding to the best performance. Table 24 displays the proportion of correct classifications for ambiguity and senses for different values of the thresholds.

The best results were obtained with an ambiguity threshold of 4 and a sense threshold of 9. The combined classification in terms of ambiguity and sense resulting from using these two threshold values has a 69% accuracy. The confusion matrix for this classification is displayed in Table 25. The rows contain the reference classifications (i.e., the ground truth) (Rodd et al, 2002). The columns contain classifications from our program. The cells in the diagonal correspond to correct classifications (n = 88), whereas the cells off the diagonal correspond to incorrect classification (n = 40).

Table 24. Proportions of correct classifications for ambiguity and senses depending on the threshold.

Ambiguity		Senses	
Threshold	Proportion of correct classifications	Threshold	Proportion of correct classifications
2	0.55	6	0.77
3	0.61	7	0.86
4	**0.77**	8	0.88
5	0.68	**9**	**0.89**
6	0.6	10	0.84
7	0.54	11	0.75

It is important to note that this validation process is only a first step towards ensuring the quality of our program. If we were confident that the principles behind our program were worth pursuing, we could cross-validate it or validate it using another data set (see Manning & Schütze, 1999). However, an analysis of the classification errors made by our program might motivate us to try improving the resources or mechanisms that we use in order to reproduce Rodd et al's (2002) classification. Indeed, 75 % of the errors were cases in which our program overestimated the number of entries or the number of senses compared to the reference. More precisely, about half of the errors were unambiguous words (as per the reference) classified as ambiguous words by our program.

Table 25. Confusion matrix with ambiguity threshold = 4 and sense threshold = 9.

			Our classification			
			Unambiguous		Ambiguous	
			Few senses	Many senses	Few senses	Many senses
Reference (Rodd et al, 2002)	Unambiguous	Few senses	**29**	4	13	2
		Many senses	1	**23**	1	6
	Ambiguous	Few senses	2	0	**16**	4
		Many senses	0	5	2	**20**

Reusing a local copy of the Wordsmyth dictionary (instead of the GCIDE) would improve the results. Another method would be to filter out the senses which are too strongly related (and which might be counted as separate entries in the GCIDE). Finally, about a quarter of the errors were words with few senses (as per the reference) classified as having many senses by our program. Here again, it would be helpful to detect the senses which are too strongly related, in order to decrease the number of senses considered as distinct.

At step 9, the classification is saved to a file (see Table 26). The `os` module is imported. The path to the output file is specified. A list with column names is specified. It contains the name of key variables in the `W` dictionary. In the `save_results()` function, the presence of the output is first detected. The file is removed if it already exists. The column names are then added, separated by tabs (`\t`). Finally, the data in `W` is written to the file, one word per line.

Table 26. The `save_results()` function.

```
# import

from __future__ import division
import fileinput, nltk, os
(...)
# parameters

output_path = "C:/test/data/Rodd_et_al_2002_experiment_2_output.txt"

column_names = ["number_of_entries", "number_of_senses"]
column_names.append("predicted_ambiguity")
column_names.append("predicted_many_senses")
column_names.append("predicted_response_time")
(...)
def save_results(output_path):
    if os.path.exists(output_path): os.remove(output_path)
    out = open(output_path, 'a')
    out.write("word" + "\t" +  "\t".join(column_names))
    for word in W:
        data = "\n" + word
        for variable in column_names:
            data += "\t" + str(W[word][variable])
        out.write(data)
    out.close()
```

Discussing the Example

What general lessons can we learn from this example? What general programming principles can we distinguish? We have tried to synthesize some of the lessons and principles below:

1. Our program required us to transfer information from one structure to another (e.g., integer, string, list, tuple). We used variables to refer to these structures.
2. Programming is about saving some of your time, notably by reusing code written by other programmers. Hence our example was reusing features of the NLTK module (which was itself reusing the WordNet dictionary), and other modules related to the operating system (os) and the file system (fileinput).
3. There was a certain degree of organization in our program. Our examples isolated parts of the code in functions. Organization helps program maintenance, particularly for longer programs. The better your organization, the easiest the maintenance.
4. If you are novice at programming, the program might appear overwhelming to you. Although our example was developed step by step, and its development has been made apparent to facilitate its understanding, it will only make sense if you also examine it, step by step, actually try to reproduce the program, and attempt to see the connection between the code and its output. Understanding this connection will be greatly facilitated if you devote some of your time to learning the underlying programming language.

The program developed in this example was only one example of a possible solution to this problem. For the sake of achieving certain pedagogical goals, we chose quite a naive approach to a quite complex problem. One could have imagined other solutions (different algorithms), possibly using other packages, and with other languages. A good NLP project will be based on a clear understanding of the problem and knowledge of what tools can be employed for what purposes. It is therefore important to invest in understanding the theoretical aspects of programming and NLP, and their practical aspects (i.e., libraries, and other resources). These topics are briefly discussed in Part III.

PART III. HOW TO CONDUCT AN NLP PROGRAMMING PROJECT

Part I informed the novice programmers and programmers unfamiliar with Python to basic programming in Python. Part II illustrated how to solve a practical NLP problem with a Python program. Now you might be motivated to undertake a project on your own. This section aims at delivering helpful hints to start such an adventure.

So Which Language is Right for Me?

The choice of a programming language depends on the nature of the problem as well as on available resources like the programmer's experience, the time devoted to solve the problem and the access to software, documentation, corpora, etc. There are a multitude of programming languages to choose from. Programming languages vary in terms of vocabulary, syntax, availability of libraries, ease of learning, vitality of the supporting communities, etc. Choosing a language involves striking a delicate balance between its advantages and disadvantages. For instance, computationally efficient (i.e., fast) languages like C++ might be particularly appropriate to solve certain NLP problems which require speed, but they might be more difficult to learn. Alternatively, you might not have much time to implement a solution so you use an available library, but that library does not include the most efficient program for your problem. Should you use the slower but ready-made existing program or write your own faster program from scratch? In other words, a programmer faces a problem of limited computational and human resources which must be allocated wisely.

The Right Tool for the Job at Hand

The programming languages option space might appear to be quite large. However, the option space is somewhat restricted by the fact that some languages were designed for specific kinds of problems (e.g., Matlab for numerical computing; R for statistical analysis; PHP for web applications; SQL for interfacing with databases, LaTeX as a document markup). In other words, one language might be more appropriate for a particular problem than another language, thereby reducing the option space. Thus perhaps the best and most useful analogy is to consider each language itself to be a kind of tool. A computing professional will know a dozen languages or more, and like a master craftsman will select the right tool for the job at hand.

In general, however, when considering a language to learn and use there are many issues to consider (see Table 27). These questions, in one way or another, all tie back into the question of time and efficiency. A language that is hard to learn and useful for only one project may not be the best choice, especially for a first language. But a professional who already knows several languages might make a different choice.

Table 27. Questions to ask oneself before choosing a language for a particular task.

- What kind of user community exists for this language (books/forums/friends)?
- What libraries exist for this language that I can use in my work?
- What kinds of development tools are available for this language?
- How well does this language address the problems I am working on now or in the future?
- How easy is it to learn the language and the various other resources I need?

When learning anything, it is important to think about the resources available. Typically, more popular languages have better resources, but not always. Very specific languages might have virtually no user base, no recent books, and essentially no future. A major issue to consider, especially for first time language learners, is whether the language you are learning has useful libraries for the problems you are working on. As a first time language learner, it will take some time before you can create sophisticated libraries of your own. However, even a novice can use a library written by someone else (which you have already started to do by using NLTK in Parts I and II). Fortunately, there exist a broad set of tools/packages which can be used and re-used for a variety of NLP objectives. See the companion website for a table with pointers to these resources.

Finally, a point that is often overlooked by even professional programmers is the development tools available for a language. Many programmers start out using a text editor (e.g., notepad) and a compiler, and many stay with those tools their whole careers. However, as anyone who has ever used a well constructed IDE (integrated development environment) can tell you, an IDE with an integrated editor, compiler, debugger, support for refactoring, and built-in documentation can save tremendous amounts of time. The Python IDLE that you might have used while working out the examples of Parts I and II is just an example. Have you noticed that it kindly used a special color coding scheme to highlight the structure of your programs? A simple text editor would not do that. On the other hand, it should be noted that an IDE by itself can have an initial learning curve.

A Word of Advice

If you are new at programming, you are likely to be new to NLP as well, and you will benefit from a friendly language like Python. You can learn both programming and NLP techniques simultaneously through NLTK, which was explicitly designed for educational purposes (Bird et al, 2008). But even if you are not new to programming or to NLP, NLTK offers the advantage of an integrated set of routines, lexicons, and corpora which allows you to easily prototype your ideas. If you are familiar with Java or C++, you will likely prefer using toolboxes like LingPipe or GATE, amongst others (see the companion website for the relevant links).

Of course, you might not need an entire NLP toolbox to carry the operations you wish to accomplish. If, for instance, you are merely concerned with parsing a group of texts, you may want to use a tool tailored for that goal, like the Stanford or Charniak parsers. In general, although at first glance it might appear tempting to conduct a whole project within the same programming environment, you might consider picking and choosing from a variety of languages and packages for different parts of your project – hereby considering the set of available toolboxes as one giant toolbox. Actually, many NLP pipelines operate on text and return text (which is why Part I contained a section about text input-output). Therefore you can easily use different programming languages or tools.

For example, let's say that you want to parse a text (as illustrated in Chapter XXX of this book). You might first use a tagger. This would output a list of tagged words. Then you can run a parser over it using these tagged words as input, and outputting a parse in a Penn Treebank style. You could then use that parse and add semantic information to it (see Chapter XXX and Chapter XXX in this book), and use that information to answer some questions about the text (e.g., Chapter XXX). This demonstrates the basic idea of an *NLP pipeline* where each operation produces a new file of text, which the succeeding state operates upon. At every step you could use a different tool or programming language. Overall, the nature of your project will dictate much of the technology you have to use.

For complete beginners, it is important to first gain grasp of simple programming operations and learn as many fundamentals of programming as possible. If you do not do that, you run the risk of wasting your time reinventing the wheel, creating inefficient solutions to very well known problems, or making certain mistakes which will vitiate your project at its core. Implementing the program is likely to be the most difficult part for a beginner. Towards this end, readers are encouraged to consult some of the many excellent tutorials and books on programming (e.g., Sande & Sande, 2009).

For more advanced users, we advise starting with the how-to tutorials that come with the various NLP tools listed on the companion website. Both beginners and advanced users will gain from reusing small pieces of code and incrementally building from there, similar to what we have done in Parts I and II.

CONCLUSION

Engaging in the practical task of programming NLP applications is not beyond the reach of novice programmers with an eye for solving NLP problems. A multitude of resources have been developed by a very vivid community of researchers, computer scientists, and coders. The reader should be now familiar with basic notions of programming (in particular, data is acquired, transformed, and released), and the notion that a variety of tools and languages exist in order to program NLP operations. Before beginning a project it is important to choose wisely because the choice of a tool/language depends on a variety of factors (nature of the task, power of the tool, learning curve, documentation, etc.). It is also important to reflect on the problem before starting programming and to proceed incrementally. Overall, programming can be useful and even fun. In some case, it can even produce a flow-like experience (Csikszentmihalyi, 1990) when the programmer is so absorbed in the task that time and fatigue disappear. We hope that this will be your experience with programming.

ACKNOWLEDGMENTS

We want to thank Laurence Dumont, Katherine Guérard, Bernard Jeuniaux, Michaël Jeuniaux, William Jeuniaux, and Bernard Lebel for commenting on various versions of this chapter. Their help was crucial in improving the readability of the chapter. We also thank two reviewers, for their extremely helpful comments and corrections. The first author is also grateful to the Linguistic Society of America for providing funds to attend the 2007 Linguistic Institute, where Steven Bird, Ewan Klein, and Edward Loper gave a fantastic course involving Python and NLTK. This research was also supported by the National Science Foundation (REC 0106965, ITR 0325428, HCC 0834847) and Institute of Education Sciences, U.S. Department of Education (R305A080594). Any opinions, findings and conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF and DoE.

REFERENCES

Anderson, D. P. (2009). An Interview with Maurice Wilkes. *Communications of the ACM*, *52*(9), 39-42.

Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python. O'Reilly Media.

Bird, S., Klein, E., Loper, E., & Baldridge, J. (2008). Multidisciplinary instruction with the Natural Language Toolkit. In Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics, Columbus, Ohio.

Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience*. New York: Harper and Row.

Dyck, M. (2002). The GNU version of The Collaborative International Dictionary of English, presented in the Extensible Markup Language. Retrieve May 20, 2010, from `http://www.ibiblio.org/webster/`

Elkner, J., Downey, A. B., & Meyers, C. (2009). How to Think Like a Computer Scientist. Learning with Python (2nd Edition). Retrieved December 27, 2009, from `http://www.openbookproject.net/thinkCSpy`

Graesser, A.C., McNamara, D. S., Louwerse, M. M., & Cai, Z. (2004). Coh-Metrix: Analysis of text on cohesion and language. *Behavior Research Methods, Instruments, & Computers*, *36*(2), 193-202.

Hodges, A. (2000). Alan Turing: The enigma. Walker & Company.

Manning, C. D., & Schütze, H. (1999). Foundations of statistical natural language processing. The MIT Press.

Miller, G. A. (1995). WordNet: A Lexical Database for English. *Communications of the ACM*, *38* (11), 39-41.

Parks, R., Ray, J., & Bland, S. (1998).Wordsmyth English dictionary – Thesaurus. [ONLINE]. Available: `http://www.wordsmyth.net/` [1999, February 1], University of Chicago.

Petzold, C. (2000). Code: The Hidden Language of Computer Hardware and Software. Microsoft Press.

Sande, W., & Sande, C. (2009). *Hello World!: Computer Programming for Kids and Other Beginners*. Manning Publications.

Rodd, J., Gaskell, G., & Marslen-Wilson, W. (2002). Making sense of semantic ambiguity: semantic competition in lexical access. *Journal of Memory and Language*, *46*, 245-266.

Swade, D. (2002). *The Difference Engine: Charles Babbage and the Quest to Build the First Computer*. Penguin.

ADDITIONAL READING SECTION

Blackburn, P. & Bos, J. (2005). Representation and Inference for Natural Language: A First Course in Computational Semantics. CSLI Publications, Stanford, CA.

Feldman, R., & Sanger, J. (2006). The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data. Cambridge University Press.

Gries, S. T. (2009). Quantitative Corpus Linguistics with R: A Practical introduction. Routledge.

Guzdial, M. (2005). Introduction to Computing and Programming in Python: A Multimedia Approach. Prentice Hall.

Jurafsky, D., & Martin, J. H. (2009). Speech and language processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition (2d Ed.).

Konchady, M. (2006). Text Mining Application Programming. Charles River Media.

LingPipe's Competition (2009). Retrieved December 27, 2009, from `http://alias-i.com/lingpipe/web/competition.html`

Marcus, M. P., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: the Penn Treebank. *Computational linguistics*, *19*(2), 313-330.

McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction. Microsoft Press.

Mertz, D. (2003). Text Processing in Python. Addison-Wesley, Boston.

Meyer, C. (2002). English Corpus Linguistics: An Introduction. Cambridge University Press.

Mitkov, R. (2005). The Oxford Handbook of Computational Linguistics. Oxford University Press, USA.

Nugues, P. M. (2006). An introduction to language processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German (Cognitive Technologies). Springer.

OpenNLP (2009). Retrieved December 27, 2009, from `http://opennlp.sourceforge.net`

KEY TERMS & DEFINITIONS

Artificial intelligence: study of how to design intelligent creatures

Computational linguistics: study of the computational properties of language

Computer science: study of how to make computers and programs

Linguistics: study of human languages

Natural language processing: using computers to automatically deal with human languages

Programming: giving instructions to computer to perform certain tasks

Corpus: organized set of documents