Chapter 2

Programming Fundamentals and Python

This chapter provides a non-technical overview of Python and will cover the basic programming knowledge needed for the rest of the chapters in Part 1. It contains many examples and exercises; there is no better way to learn to program than to dive in and try these yourself. You should then feel confident in adapting the example for your own purposes. Before you know it you will be programming!

2.1 Python the Calculator

One of the friendly things about Python is that it allows you to type directly into the interactive **interpreter** — the program that will be running your Python programs. We want you to be completely comfortable with this before we begin, so let's start it up:

```
Python 2.4.3 (#1, Mar 30 2006, 11:02:16)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This blurb depends on your installation; the main thing to check is that you are running Python 2.4 or greater (here it is 2.4.3). The `>>>` prompt indicates that the Python interpreter is now waiting for input. If you are using the Python interpreter through the Interactive DeveLopment Environment (IDLE) then you should see a colorized version. We have colorized our examples in the same way, so that you can tell if you have typed the code correctly. Let's begin by using the Python prompt as a calculator:

```
>>> 3 + 2 * 5 - 1
12
>>>
```

There are several things to notice here. First, once the interpreter has finished calculating the answer and displaying it, the prompt reappears. This means the Python interpreter is waiting for another instruction. Second, notice that Python deals with the order of operations correctly (unlike some older calculators), so the multiplication `2 * 5` is calculated before it is added to `3`.

Try a few more expressions of your own. You can use asterisk (`*`) for multiplication and slash (`/`) for division, and parentheses for bracketing expressions. One strange thing you might come across is that division doesn't always behave how you expect:

```
>>> 3/3
1
>>> 1/3
0
>>>
```

The second case is surprising because we would expect the answer to be `0.333333`. We will come back to why that is the case later on in this chapter. For now, let's simply observe that these examples demonstrate how you can work interactively with the interpreter, allowing you to experiment and explore. Also, as you will see later, your intuitions about numerical expressions will be useful for manipulating other kinds of data in Python.

You should also try nonsensical expressions to see how the interpreter handles it:

```
>>> 1 +
Traceback (most recent call last):
  File "<stdin>", line 1
    1 +
      ^
SyntaxError: invalid syntax
>>>
```

Here we have produced a **syntax error**. It doesn't make sense to end an instruction with a plus sign. The Python interpreter indicates the line where the problem occurred.

2.2 Understanding the Basics: Strings and Variables

2.2.1 Representing text

We can't simply type text directly into the interpreter because it would try to interpret the text as part of the Python language:

```
>>> Hello World
Traceback (most recent call last):
  File "<stdin>", line 1
    Hello World
              ^
SyntaxError: invalid syntax
>>>
```

Here we see an error message. Note that the interpreter is confused about the position of the error, and points to the end of the string rather than the start.

Python represents a piece of text using a **string**. Strings are **delimited** — or separated from the rest of the program — by quotation marks:

```
>>> 'Hello World'
'Hello World'
>>> "Hello World"
'Hello World'
>>>
```

We can use either single or double quotation marks, as long as we use the same ones on either end of the string.

Now we can perform calculator-like operations on strings. For example, adding two strings together seems intuitive enough that you could guess the result:

```
>>> 'Hello' + 'World'
'HelloWorld'
>>>
```

When applied to strings, the + operation is called **concatenation**. It produces a new string which is a copy of the two original strings pasted together end-to-end. Notice that concatenation doesn't do anything clever like insert a space between the words. The Python interpreter has no way of knowing that you want a space; it does *exactly* what it is told. Given the example of +, you might be able guess what multiplication will do:

```
>>> 'Hi' + 'Hi' + 'Hi'
'HiHiHi'
>>> 'Hi' * 3
'HiHiHi'
>>>
```

The point to take from this (apart from learning about strings) is that in Python, intuition about what should work gets you a long way, so it is worth just trying things to see what happens. You are very unlikely to break something, so just give it a go.

2.2.2 Storing and reusing values

After a while, it can get quite tiresome to keep retyping Python statements over and over again. It would be nice to be able to store the **value** of an expression like `'Hi' + 'Hi' + 'Hi'` so that we can use it again. We do this by saving results to a location in the computer's memory, and giving the location a name. Such a named place is called a **variable**. In Python we create variables by **assignment**, which involves putting a value into the variable:

```
>>> msg = 'Hello World'          ①
>>> msg                          ②
'Hello World'                    ③
>>>
```

At ① we have created a variable called `msg` (short for 'message') and set it to have the string value `'Hello World'`. We used the = operation, which **assigns** the value of the expression on the right to the variable on the left. Notice the Python interpreter does not print any output; it only prints output when the statement returns a value, and an assignment statement returns no value. At ② we inspect the contents of the variable by naming it on the command line: that is, we use the name `msg`. The interpreter prints out the contents of the variable at ③.

Variables stand in for values, so instead of writing `'Hi' * 3` we could write:

```
>>> msg = 'Hi'
>>> num = 3
>>> msg * num
'HiHiHi'
>>>
```

We can also assign a new value to a variable just by using assignment again:

```
>>> msg = msg * num
>>> msg
'HiHiHi'
>>>
```

Here we have taken the value of `msg`, multiplied it by `3` and then stored that new string (`HiHiHi`) back into the variable `msg`.

2.2.3 Printing and inspecting strings

So far, when we have wanted to look at the contents of a variable or see the result of a calculation, we have just typed the variable name into the interpreter. For example, we can look at the contents of `msg` using:

```
>>> msg
'Hello World'
>>>
```

However, there are some situations where this isn't going to do what we want. To see this, open a text editor, and create a file called `test.py`, containing the single line

```
msg = 'Hello World'
```

Now, open this file in IDLE, then go to the `Run` menu, and select the command `Run Module`. The result in the main IDLE window should look like this:

```
>>> ================================ RESTART ================================
>>>
>>>
```

But where is the output showing the value of `msg`? The answer is that the program in `test.py` will only show a value if you explicitly tell it to, using the `print` command. So add another line to `test.py` so that it looks as follows:

```
msg = 'Hello World'
print msg
```

Select `Run Module` again, and this time you should get output which looks like this:

```
>>> ================================ RESTART ================================
>>>
Hello World
>>>
```

On close inspection, you will see that the quotation marks which indicate that `Hello World` is a string are missing in this case. That is because inspecting a variable (only possible within the interactive interpreter) prints out the Python **representation** of a value, whereas the `print` statement only prints out the value itself, which in this case is just the text in the string.

You will see that you get the same results if you use the `print` command in the interactive interpreter:

```
>>> print msg
Hello World
>>>
```

In fact, you can use a sequence of comma-separated expressions in a `print` statement.

```
>>> msg2 = 'Goodbye'
>>> print msg, msg2
Hello World Goodbye
>>>
```

So, if you want the users of your program to be able to see something then you need to use `print`. If you just want to check the contents of the variable while you are developing your program in the interactive interpreter, then you can just type the variable name directly into the interpreter.

2.2.4 Exercises

1. ☼ Start up the Python interpreter (e.g. by running IDLE). Try the examples in section 2.1, then experiment with using Python as a calculator.

2. ☼ Try the examples in this section, then try the following.

   a) Create a variable called `msg` and put a message of your own in this variable. Remember that strings need to be quoted, so you will need to type something like:

   ```
   >>> msg = "I like NLP!"
   ```

   b) Now print the contents of this variable in two ways, first by simply typing the variable name and pressing enter, then by using the `print` command.

   c) Try various arithmetic expressions using this string, e.g. `msg + msg`, and `5 * msg`.

   d) Define a new string `hello`, and then try `hello + msg`. Change the `hello` string so that it ends with a space character, and then try `hello + msg` again.

2.3 Slicing and Dicing

Strings are so important (especially for NLP!) that we will spend some more time on them. Here we will learn how to access the individual **characters** that make up a string, how to pull out arbitrary **substrings**, and how to reverse strings.

2.3.1 Accessing individual characters

The positions within a string are numbered, starting from zero. To access a position within a string, we specify the position inside square brackets:

```
>>> msg = 'Hello World'
>>> msg[0]
'H'
>>> msg[3]
'l'
>>> msg[5]
' '
>>>
```

This is called **indexing** or **subscripting** the string. The position we specify inside the square brackets is called the **index**. We can retrieve not only letters but any character, such as the space at index `5`.

> **Note**
>
> Be careful to distinguish between the string `' '`, which is a single whitespace character, and `''`, which is the empty string.

The fact that strings are numbered from zero may seem counter-intuitive. However, it goes back to the way variables are stored in a computer's memory. As mentioned earlier, a variable is actually the

name of a location, or **address**, in memory. Strings are arbitrarily long, and their address is taken to be the position of their first character. Thus, if we assign `three = 3` and `msg = 'Hello World'` then the location of those values will be along the lines shown in Figure 2.1.

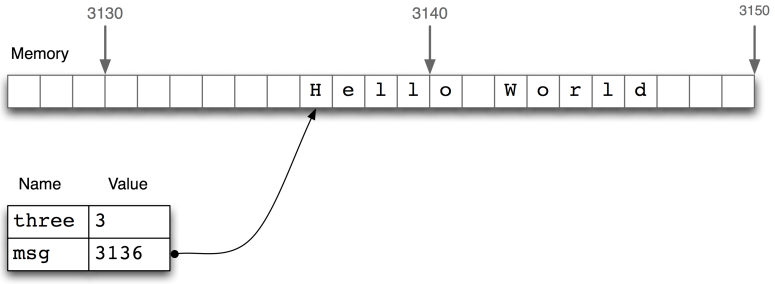

Figure 2.1: Variables and Computer Memory

When we index into a string, the computer adds the index to the string's address. Thus `msg[3]` is found at memory location `3136 + 3`. Accordingly, the first position in the string is found at `3136 + 0`, or `msg[0]`.

If you don't find Figure 2.1 helpful, you might just want to think of indexes as giving you the position in a string immediately *before* a character, as indicated in Figure 2.2.

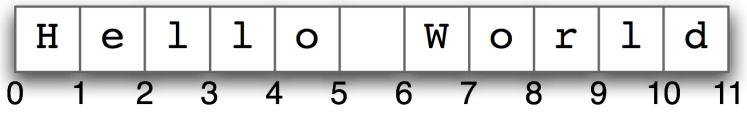

Figure 2.2: String Indexing

Now, what happens when we try to access an index that is outside of the string?

```
>>> msg[11]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>>
```

The index of `11` is outside of the range of valid indices (i.e., 0 to 10) for the string `'Hello World'`. This results in an error message. This time it is not a syntax error; the program fragment is syntactically correct. Instead, the error occurred while the program was running. The `Traceback` message indicates which line the error occurred on (line 1 of 'standard input'). It is followed by the name of the error, `IndexError`, and a brief explanation.

In general, how do we know what we can index up to? If we know the length of the string is n, the highest valid index will be $n - 1$. We can get access to the length of the string using the `len()` function.

```
>>> len(msg)
11
>>>
```

Informally, a **function** is a named snippet of code that provides a service to our program when we **call** or execute it by name. We call the `len()` function by putting parentheses after the name and giving it the string `msg` we want to know the length of. Because `len()` is built into the Python interpreter, IDLE colors it purple.

We have seen what happens when the index is too large. What about when it is too small? Let's see what happens when we use values less than zero:

```
>>> msg[-1]
'd'
>>>
```

This does not generate an error. Instead, negative indices work from the *end* of the string, so `-1` indexes the last character, which is `'d'`.

```
>>> msg[-3]
'r'
>>> msg[-6]
' '
>>>
```

Now the computer works out the location in memory relative to the string's address plus its length, e.g. `3136 + 11 - 1 = 3146`. We can also visualize negative indices as shown in Figure 2.3.

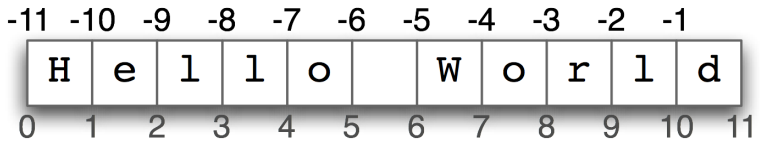

Figure 2.3: Negative Indices

Thus we have two ways to access the characters in a string, from the start or the end. For example, we can access the space in the middle of `Hello` and `World` with either `msg[5]` or `msg[-6]`; these refer to the same location, because `5 = len(msg) - 6`.

2.3.2 Accessing substrings

Next, we might want to access more than one character at a time. This is also pretty simple; we just need to specify a range of characters for indexing rather than just one. This process is called **slicing** and we indicate a slice using a colon in the square brackets to separate the beginning and end of the range:

```
>>> msg[1:4]
'ell'
>>>
```

Here we see the characters are `'e'`, `'l'` and `'l'` which correspond to `msg[1]`, `msg[2]` and `msg[3]`, but not `msg[4]`. This is because a slice *starts* at the first index but finishes *one before* the end index. This is consistent with indexing: indexing also starts from zero and goes up to *one before* the length of the string. We can see that by indexing with the value of `len()`:

```
>>> len(msg)
11
>>> msg[0:11]
'Hello World'
>>>
```

We can also slice with negative indices — the same basic rules of starting from the start index and stopping one before the end index applies; here we stop before the space character:

```
>>> msg[0:-6]
'Hello'
>>>
```

Python provides two shortcuts for commonly used slice values. If the start index is `0` then you can leave it out entirely, and if the end index is the length of the string then you can leave it out entirely:

```
>>> msg[:3]
'Hel'
>>> msg[6:]
'World'
>>>
```

The first example above selects the first three characters from the string, and the second example selects from the character with index 6, namely `'W'`, to the end of the string. These shortcuts lead to a couple of common Python idioms:

```
>>> msg[:-1]
'Hello Worl'
>>> msg[:]
'Hello World'
>>>
```

The first chomps off just the last character of the string, and the second makes a complete copy of the string (which is more important when we come to lists below).

2.3.3 Exercises

1. ☼ Define a string `s = 'colorless'`. Write a Python statement that changes this to "colourless" using only the slice and concatenation operations.

2. ☼ Try the slice examples from this section using the interactive interpreter. Then try some more of your own. Guess what the result will be before executing the command.

3. ☼ We can use the slice notation to remove morphological endings on words. For example, `'dogs'[:-1]` removes the last character of `dogs`, leaving `dog`. Use slice notation to remove the affixes ending from these words (we've inserted a hyphen to indicate the affix boundary, but omit this from your strings): `dish-es`, `run-ning`, `nation-ality`, `un-do`, `pre-heat`.

4. ☼ We saw how we can generate an `IndexError` by indexing beyond the end of a string. Is it possible to construct an index that goes too far to the left, before the start of the string?

5. ☼ We can also specify a step size for the slice. The following returns every second character within the slice, in a forwards or reverse direction:

```
>>> msg[6:11:2]
'Wrd'
>>> msg[10:5:-2]
'drW'
>>>
```

Experiment with different step values.

6. ☼ What happens if you ask the interpreter to evaluate `msg[::-1]`? Explain why this is a reasonable result.

2.4 Strings, Sequences, and Sentences

We have seen how words like *Hello* can be stored as a string `'Hello'`. Whole sentences can also be stored in strings, and manipulated as before, as we can see here for Chomsky's famous nonsense sentence:

```
>>> sent = 'colorless green ideas sleep furiously'
>>> sent[16:21]
'ideas'
>>> len(sent)
37
>>>
```

However, it turns out to be a bad idea to treat a sentence as a sequence of its characters, because this makes it too inconvenient to access the words or work out the length. Instead, we would prefer to represent a sentence as a sequence of its *words*; as a result, indexing a sentence accesses the words, rather than characters. We will see how to do this now.

2.4.1 Lists

A **list** is designed to store a sequence of values. A list is similar to a string in many ways except that individual items don't have to be just characters; they can be arbitrary strings, integers or even other lists.

A Python list is represented as a sequence of comma-separated items, delimited by square brackets. Let's create part of Chomsky's sentence as a list and put it in a variable `phrase1`:

```
>>> phrase1 = ['colorless', 'green', 'ideas']
>>> phrase1
['colorless', 'green', 'ideas']
>>>
```

Because lists and strings are both kinds of sequence, they can be processed in similar ways; just as strings support `len()`, indexing and slicing, so do lists. The following example applies these familiar operations to the list `phrase1`:

```
>>> len(phrase1)
3
>>> phrase1[0]
'colorless'
>>> phrase1[-1]
```

```
'ideas'
>>> phrase1[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Here, `phrase1[-5]` generates an error, because the fifth-last item in a three item list would occur before the list started, i.e., it is undefined. We can also slice lists in exactly the same way as strings:

```
>>> phrase1[1:3]
['green', 'ideas']
>>> phrase1[-2:]
['green', 'ideas']
>>>
```

Lists can be concatenated just like strings. Here we will put the resulting list into a new variable `phrase2`. The original variable `phrase1` is not changed in the process:

```
>>> phrase2 = phrase1 + ['sleep', 'furiously']
>>> phrase2
['colorless', 'green', 'ideas', 'sleep', 'furiously']
>>> phrase1
['colorless', 'green', 'ideas']
>>>
```

Now, lists and strings do not have exactly the same functionality. Lists have the added power that you can change their elements. Let's imagine that we want to change the 0th element of `phrase1` to `'colorful'`, we can do that by assigning to the index `phrase1[0]`:

```
>>> phrase1[0] = 'colorful'
>>> phrase1
['colorful', 'green', 'ideas']
>>>
```

On the other hand if we try to do that with a string (for example changing the 0th character in `msg` to `'J'` we get:

```
>>> msg[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

This is because strings are **immutable** — you can't change a string once you have created it. However, lists are **mutable**, and their contents can be modified at any time. As a result, lists support a number of operations, or **methods**, which modify the original value rather than returning a new value.

> **Note**
>
> Methods are functions, so they can be called in a similar manner. However, as we will see later on in this book, methods are tightly associated with objects of that belong to specific **classes** (for example, strings and lists). A method is called on a particular object using the object's name, then a period, then the name of the method, and finally the parentheses containing any arguments.

Two of these methods are **sorting** and **reversing**:

```
>>> phrase2.sort()
>>> phrase2
['colorless', 'furiously', 'green', 'ideas', 'sleep']
>>> phrase2.reverse()
>>> phrase2
['sleep', 'ideas', 'green', 'furiously', 'colorless']
>>>
```

As you will see, the prompt reappears immediately on the line after `phrase2.sort()` and `phrase2.reverse()`. That is because these methods do not return a new list, but instead modify the original list stored in the variable `phrase2`.

Lists also support an `append()` method for adding items to the end of the list and an `index` method for finding the index of particular items in the list:

```
>>> phrase2.append('said')
>>> phrase2.append('Chomsky')
>>> phrase2
['sleep', 'ideas', 'green', 'furiously', 'colorless', 'said', 'Chomsky']
>>> phrase2.index('green')
2
>>>
```

Finally, just as a reminder, you can create lists of any values you like. They don't even have to be the same type, although this is rarely a good idea:

```
>>> bat = ['bat', [[1, 'n', 'flying mammal'], [2, 'n', 'striking instrument']]]
>>>
```

2.4.2 Working on sequences one item at a time

We have shown you how to create lists, and how to index and manipulate them in various ways. Often it is useful to step through a list and process each item in some way. We do this using a `for` loop. This is our first example of a **control structure** in Python, a statement that *controls* how other statements are run:

```
>>> for word in phrase2:
...     print len(word), word
5 sleep
5 ideas
5 green
9 furiously
9 colorless
4 said
7 Chomsky
```

This program runs the statement `print len(word), word` for every item in the list of words. This process is called **iteration**. Each iteration of the `for` loop starts by assigning the next item of the list `phrase2` to the **loop variable** `word`. Then the indented **body** of the loop is run. Here the body consists of a single command, but in general the body can contain as many lines of code as you want, so long as they are all indented by the same amount.

> **Note**
>
> The interactive interpreter changes the prompt from `>>>` to the `...` prompt after encountering a colon (`:`). This indicates that the interpreter is expecting an indented block of code to appear next. However, it is up to you to do the indentation. To finish the indented block just enter a blank line.

We can run another `for` loop over the Chomsky nonsense sentence, and calculate the average word length. As you will see, this program uses the `len()` function in two ways: to count the number of characters in a word, and to count the number of words in a phrase. Note that `x += y` is shorthand for `x = x + y`; this idiom allows us to **increment** the `total` variable each time the loop is run.

```
>>> total = 0
>>> for word in phrase2:
...     total += len(word)
...
>>> total / len(phrase2)
6
>>>
```

Finally, note that we can write `for` loops to iterate over the characters in strings.

```
>>> sent = 'colorless green ideas sleep furiously'
>>> for char in sent:
...     print char,
c o l o r l e s s  g r e e n  i d e a s  s l e e p  f u r i o u s l y
```

2.4.3 Tuples

Python tuples are just like lists, except that there is one important difference: tuples cannot be changed in place, for example by `sort()` or `reverse()`. In other words, like strings they are immutable. Tuples are formed with enclosing parentheses rather than square brackets, and items are separated by commas. Like lists, tuples can be indexed and sliced.

```
>>> t = ('walk', 'fem', 3)
>>> t[0]
'walk'
>>> t[1:]
('fem', 3)
>>> t[0] = 'run'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

2.4.4 String Formatting

The output of a program is usually structured to make the information easily digestible by a reader. Instead of running some code and then manually inspecting the contents of a variable, we would like the code to tabulate some output. We already saw this above in the first `for` loop example, where each line of output was similar to `5 sleep`, consisting of a word length, followed by the word in question.

There are many ways we might want to format such output. For instance, we might want to place the length value in parentheses *after* the word, and print all the output an a single line:

```
>>> for word in phrase2:
...     print word, '(', len(word), '),',
sleep ( 5 ), ideas ( 5 ), green ( 5 ), furiously ( 9 ), colorless ( 9 ),
said ( 4 ), Chomsky ( 7 ),
```

Notice that this `print` statement ends with a trailing comma, which is how we tell Python not to print a newline at the end.

However, this approach has a couple of problems. First, the `print` statement intermingles variables and punctuation, making it a little difficult to read. Second, the output has spaces around every item that was printed. A cleaner way to produce structured output uses Python's **string-formatting expressions**. Here's an example:

```
>>> for word in phrase2:
...     print "%s (%d)," % (word, len(word)),
sleep (5), ideas (5), green (5), furiously (9), colorless (9),
said (4), Chomsky (7),
```

Here, the `print` command is followed by a three-part object having the syntax: *`format`* `%` *`values`*. The *`format`* section is a string containing **format specifiers** such as `%s` and `%d` which Python will replace with the supplied values. The `%s` specifier tells Python that the corresponding variable is a string (or should be converted into a string), while the `%d` specifier indicates that the corresponding variable should be converted into a decimal representation. Finally, the *`values`* section of a formatting string is a tuple containing exactly as many items as there are format specifiers in the *`format`* section. (We will discuss Python's string-formatting expressions in more detail in Section 6.3.2).

In the above example, we used a trailing comma to suppress the printing of a newline. Suppose, on the other hand, that we want to introduce some additional newlines in our output. We can accomplish this by inserting the 'special' character `\n` into the `print` string:

```
>>> for word in phrase2:
...     print "Word = %s\nIndex = %s\n*****" % (word, phrase2.index(word))
...
Word = sleep
Index = 0
*****
Word = ideas
Index = 1
*****
Word = green
Index = 2
*****
Word = furiously
Index = 3
*****
Word = colorless
Index = 4
*****
Word = said
Index = 5
*****
Word = Chomsky
Index = 6
```

```
*****
>>>
```

2.4.5 Character encoding and Unicode

Our programs will often need to deal with different languages, and different character sets. The concept of "plain text" is a fiction. If you live in the English-speaking world you probably use ASCII. If you live in Europe you might use one of the extended Latin character sets, containing such characters as "ø" for Danish and Norwegian, "ő" for Hungarian, "ñ" for Spanish and Breton, and "ň" for Czech and Slovak.

[See `http://www.amk.ca/python/howto/unicode` for more information on Unicode and Python.]

2.4.6 Converting between strings and lists

Often we want to convert between a string containing a space-separated list of words and a list of strings. Let's first consider turning a list into a string. One way of doing this is as follows:

```
>>> str = ''
>>> for word in phrase2:
...     str += ' ' + word
...
>>> str
' sleep ideas green furiously colorless said Chomsky'
>>>
```

One drawback of this approach is that we have an unwanted space at the start of `str`. It is more convenient to use the `string.join()` method:

```
>>> import string
>>> phrase3 = string.join(phrase2)
>>> phrase3
'sleep ideas green furiously colorless said Chomsky'
>>>
```

Now let's try to reverse the process: that is, we want to convert a string into a list. Again, we could start off with an empty list `[]` and `append()` to it within a `for` loop. But as before, there is a more succinct way of achieving the same goal. This time, we will *split* the new string `phrase3` on the whitespace character:

```
>>> phrase3.split(' ')
['sleep', 'ideas', 'green', 'furiously', 'colorless', 'said', 'Chomsky']
>>> phrase3.split('s')
['', 'leep idea', ' green furiou', 'ly colorle', '', ' ', 'aid Chom', 'ky']
>>>
```

We can also split on any character, so we tried splitting on `'s'` as well.

2.4.7 Exercises

1. ☼ Using the Python interactive interpreter, experiment with the examples in this section. Think of a sentence and represent it as a list of strings, e.g. ['Hello', 'world']. Try the various operations for indexing, slicing and sorting the elements of your list. Extract individual items (strings), and perform some of the string operations on them.

2. ☼ We pointed out that when `phrase` is a list, `phrase.reverse()` returns a modified version of `phrase` rather than a new list. On the other hand, we can use the slice trick mentioned in the exercises for the previous section, `[::-1]` to create a *new* reversed list without changing `phrase`. Show how you can confirm this difference in behaviour.

3. ☼ We have seen how to represent a sentence as a list of words, where each word is a sequence of characters. What does `phrase1[2][2]` do? Why? Experiment with other index values.

4. ☼ Write a `for` loop to print out the characters of a string, one per line.

5. ☼ What happens if you call `split` on a string, with no argument, e.g. `phrase3.split()`? What happens when the string being split contains tab characters, consecutive space characters, or a sequence of tabs and spaces?

6. ☼ Create a variable `words` containing a list of words. Experiment with `words.sort()` and `sorted(words)`. What is the difference?

7. ◑ Process the list `phrase2` using a `for` loop, and store the result in a new list `lengths`. Hint: begin by assigning the empty list to `lengths`, using `lengths = []`. Then each time through the loop, use `append()` to add another length value to the list.

8. ◑ Define a variable `silly` to contain the string: `'newly formed bland ideas are unexpressible in an infuriating way'`. (This happens to be the legitimate interpretation that bilingual English-Spanish speakers can assign to Chomsky's famous phrase, according to Wikipedia). Now write code to perform the following tasks:

   a) Split `silly` into a list of strings, one per word, using Python's `split()` operation.

   b) Extract the second letter of each word in `silly` and join them into a string, to get `'eoldrnnnna'`.

   c) Combine the words in `phrase4` back into a single string, using `join()`. Make sure the words in the resulting string are separated with whitespace.

   d) Print the words of `silly` in alphabetical order, one per line.

9. ◑ The `index()` function can be used to look up items in sequences. For example, `'unexpressible'.index('e')` tells us the index of the first position of the letter `e`.

   a) What happens when you look up a substring, e.g. `'unexpressible'.index('re')`?

   b) Define a variable `words` containing a list of words. Now use `words.index()` to look up the position of an individual word.

c) Define a variable `silly` as in the exercise above. Use the `index()` function in combination with list slicing to build a list `phrase` consisting of all the words up to (but not including) `in` in `silly`.

2.5 Making Decisions

So far, our simple programs have been able to manipulate sequences of words, and perform some operation on each one. We applied this to lists consisting of a few words, but the approach works the same for lists of arbitrary size, containing thousands of items. Thus, such programs have some interesting qualities: (i) the ability to work with language, and (ii) the potential to save human effort through automation. Another useful feature of programs is their ability to *make decisions* on our behalf; this is our focus in this section.

2.5.1 Making simple decisions

Most programming languages permit us to execute a block of code when a **conditional expression**, or `if` statement, is satisfied. In the following program, we have created a variable called `word` containing the string value `'cat'`. The `if` statement then checks whether the condition `len(word) < 5` is true. Because the conditional expression is true, the body of the `if` statement is invoked and the `print` statement is executed.

```
>>> word = "cat"
>>> if len(word) < 5:
...     print 'word length is less than 5'
...
word length is less than 5
>>>
```

If we change the conditional expression to `len(word) >= 5` — the length of `word` is greater than or equal to `5` — then the conditional expression will no longer be true, and the body of the `if` statement will not be run:

```
>>> if len(word) >= 5:
...     print 'word length is greater than or equal to 5'
...
>>>
```

The `if` statement, just like the `for` statement above is a **control structure**. An `if` statement is a control structure because it controls whether the code in the body will be run. You will notice that both `if` and `for` have a colon at the end of the line, before the indentation begins. That's because all Python control structures end with a colon.

What if we want to do something when the conditional expression is not true? The answer is to add an `else` clause to the `if` statement:

```
>>> if len(word) >= 5:
...     print 'word length is greater than or equal to 5'
... else:
...     print 'word length is less than 5'
...
word length is less than 5
>>>
```

Finally, if we want to test multiple conditions in one go, we can use an `elif` clause which acts like an `else` and an `if` combined:

```
>>> if len(word) < 3:
...     print 'word length is less than three'
... elif len(word) == 3:
...     print 'word length is equal to three'
... else:
...     print 'word length is greater than three'
...
word length is equal to three
>>>
```

2.5.2 Conditional expressions

Python supports a wide range of operators like < and >= for testing the relationship between values. The full set of these **relational operators** are shown in Table inequalities.

Operator	Relationship
<	less than
<=	less than or equal to
==	equal to (note this is two not one = sign)
!=	not equal to
>	greater than
>=	greater than or equal to

Table 2.1:

Normally we use conditional expressions as part of an `if` statement. However, we can test these relational operators directly at the prompt:

```
>>> 3 < 5
True
>>> 5 < 3
False
>>> not 5 < 3
True
>>>
```

Here we see that these expressions have **Boolean** values, namely `True` or `False`. `not` is a Boolean operator, and flips the truth value of Boolean statement.

Strings and lists also support conditional operators:

```
>>> word = 'sovereignty'
>>> 'sovereign' in word
True
>>> 'gnt' in word
True
>>> 'pre' not in word
```

```
True
>>> 'Hello' in ['Hello', 'World']
True
>>> 'Hell' in ['Hello', 'World']
False
>>>
```

Strings also have methods for testing what appears at the beginning and the end of a string (as opposed to just anywhere in the string:

```
>>> word.startswith('sovereign')
True
>>> word.endswith('ty')
True
>>>
```

Note

Integers, strings and lists are all kinds of **data types** in Python. In fact, every value in Python has a type. The type determines what operations you can perform on the data value. So, for example, we have seen that we can index strings and lists, but we can't index integers:

```
>>> one = 'cat'
>>> one[0]
'c'
>>> two = [1, 2, 3]
>>> two[1]
2
>>> three = 3
>>> three[2]
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in -toplevel-
    three[2]
TypeError: 'int' object is unsubscriptable
>>>
```

You can use Python's `type()` function to check what the type of an object is:

```
>>> data = [one, two, three]
>>> for item in data:
...     print "item '%s' belongs to %s" % (item, type(item))
...
item 'cat' belongs to <type 'str'>
item '[1, 2, 3]' belongs to <type 'list'>
item '3' belongs to <type 'int'>
>>>
```

Because strings and lists (and tuples) have so much in common, they are grouped together in a higher level type called **sequences**.

2.5.3 Iteration, items, and `if`

Now it is time to put some of the pieces together. We are going to take the string `'how now brown cow'` and print out all of the words ending in `'ow'`. Let's build the program up in stages. The first step is to split the string into a list of words:

```
>>> sentence = 'how now brown cow'
>>> words = sentence.split()
>>> words
['how', 'now', 'brown', 'cow']
>>>
```

Next, we need to iterate over the words in the list. Just so we don't get ahead of ourselves, let's print each word, one per line:

```
>>> for word in words:
...     print word
...
how
now
brown
cow
```

The next stage is to only print out the words if they end in the string `'ow'`. Let's check that we know how to do this first:

```
>>> 'how'.endswith('ow')
True
>>> 'brown'.endswith('ow')
False
>>>
```

Now we are ready to put an `if` statement inside the `for` loop. Here is the complete program:

```
>>> sentence = 'how now brown cow'
>>> words = sentence.split()
>>> for word in words:
...     if word.endswith('ow'):
...         print word
...
how
now
cow
>>>
```

As you can see, even with this small amount of Python knowledge it is possible to develop useful programs. The key idea is to develop the program in pieces, testing that each one does what you expect, and then combining them to produce whole programs. This is why the Python interactive interpreter is so invaluable, and why you should get comfortable using it.

2.5.4 Exercises

1. ☼ Assign a new value to `sentence`, namely the string `'she sells sea shells by the sea shore'`, then write code to perform the following tasks:

a) Print all words beginning with `'sh'`:

b) Print all words longer than 4 characters.

c) Generate a new sentence that adds the popular hedge word `'like'` before every word beginning with `'se'`. Your result should be a single string.

2. ☼ Write code to abbreviate text by removing all the vowels. Define `sentence` to hold any string you like, then initialize a new string `result` to hold the empty string `''`. Now write a `for` loop to process the string, one character at a time, and append any non-vowel characters to the result string.

3. ◑ Write conditional expressions, such as `'H' in msg`, but applied to lists instead of strings. Check whether particular words are included in the Chomsky nonsense sentence.

4. ◑ Write code to convert text into *hAck3r*, where characters are mapped according to the following table:

Input:	e	i	o	l	s	.	ate
Output:	3	1	0	|	5	5w33t!	8

Table 2.2:

2.6 Getting organized

Strings and lists are a simple way to organize data. In particular, they **map** from integers to values. We can 'look up' a string using an integer to get one of its letters, and we can also look up a list of words using an integer to get one of its strings. These cases are shown in Figure 2.4.

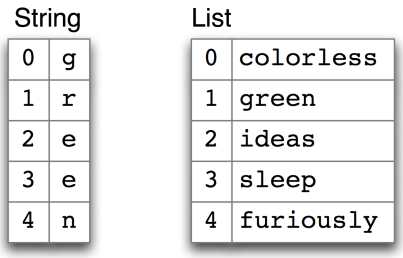

Figure 2.4: Sequence Look-up

However, we need a more flexible way to organize and access our data. Consider the examples in Figure 2.5.

In the case of a phone book, we look up an entry using a *name*, and get back a number. When we type a domain name in a web browser, the computer looks this up to get back an IP address. A word frequency table allows us to look up a word and find its frequency in a text collection. In all these cases, we are mapping from names to numbers, rather than the other way round as with indexing into sequences. In general, we would like to be able to map between arbitrary types of information. The following table lists a variety of linguistic objects, along with what they map.

Phone List

Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
pythonb.org	18.21.3.144
sourceforge.net	51.98.23.53

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Figure 2.5: Dictionary Look-up

Linguistic Object	**Maps**	
	from	**to**
Document Index	Word	List of pages (where word is found)
Thesaurus	Word sense	List of synonyms
Dictionary	Headword	Entry (part of speech, sense definitions, etymology)
Comparative Wordlist	Gloss term	Cognates (list of words, one per language)
Morph Analyzer	Surface form	Morphological analysis (list of component morphemes)

Table 2.3:

Most often, we are mapping from a string to some structured object. For example, a document index maps from a word (which we can represent as a string), to a list of pages (represented as a list of integers). In this section, we will see how to represent such mappings in Python.

2.6.1 Accessing data with data

Python provides a **dictionary** data type, which can be used for mapping between arbitrary types.

> **Note**
>
> A Python dictionary is somewhat like a linguistic dictionary — they both give you a systematic means of looking things up, and so there is some potential for confusion. However, we hope that it will usually be clear from the context which kind of dictionary we are tallking about.

Here we define `pos` to be an empty dictionary and then add three entries to it, specifying the part-of-speech of some words. We add entries to a dictionary using the familiar square bracket notation:

```
>>> pos = {}
>>> pos['colorless'] = 'adj'
>>> pos['furiously'] = 'adv'
>>> pos['ideas'] = 'n'
>>>
```

So, for example, `pos['colorless'] = 'adj'` says that the look-up value of `'colorless'` in `pos` is the string `'adj'`.

To look up a value in `pos`, we again use indexing notation, except now the thing inside the square brackets is the item whose value we want to recover:

```
>>> pos['ideas']
'n'
>>> pos['colorless']
'adj'
>>>
```

The item used for look-up is called the **key**, and the data that is returned is known as the **value**. As with indexing a list or string, we get an exception when we try to access the value of a key that does not exist:

```
>>> pos['missing']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'missing'
>>>
```

This raises an important question. Unlike lists and strings, where we can use `len()` to work out which integers will be legal indices, how do we work out the legal keys for a dictionary? Fortunately, we can check whether a key exists in a dictionary using the `in` operator:

```
>>> 'colorless' in pos
True
>>> 'missing' in pos
False
>>> 'missing' not in pos
True
>>>
```

Notice that we can use `not in` to check if a key is *missing*. Be careful with the `in` operator for dictionaries: it only applies to the keys and not their values. If we check for a value, e.g. `'adj' in pos`, the result is `False`, since `'adj'` is not a key. We can loop over all the entries in a dictionary using a `for` loop.

```
>>> for word in pos:
...     print "%s (%s)" % (word, pos[word])
...
colorless (adj)
furiously (adv)
ideas (n)
>>>
```

We can see what the contents of the dictionary look like by inspecting the variable `pos`:

```
>>> pos
{'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}
>>>
```

Here, the contents of the dictionary are shown as **key-value pairs**. As you can see, the order of the key-value pairs is different from the order in which they were originally entered. This is because dictionaries are not sequences but mappings. The keys in a mapping are not inherently ordered, and any ordering that we might want to impose on the keys exists independently of the mapping. As we shall see later, this gives us a lot of flexibility.

We can use the same key-value pair format to create a dictionary:

```
>>> pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}
>>>
```

Using the dictionary methods `keys()`, `values()` and `items()`, we can access the keys and values as separate lists, and also the key-value pairs:

```
>>> pos.keys()
['colorless', 'furiously', 'ideas']
>>> pos.values()
['adj', 'adv', 'n']
>>> pos.items()
[('colorless', 'adj'), ('furiously', 'adv'), ('ideas', 'n')]
>>>
```

2.6.2 Counting with dictionaries

The values stored in a dictionary can be any kind of object, not just a string — the values can even be dictionaries. The most common kind is actually an integer. It turns out that we can use a dictionary to store **counters** for many kinds of data. For instance, we can have a counter for all the letters of the alphabet; each time we get a certain letter we increment its corresponding counter:

```
>>> phrase = 'colorless green ideas sleep furiously'
>>> count = {}
>>> for letter in phrase:
...     if letter not in count:
...         count[letter] = 0
...     count[letter] += 1
>>> count
{'a': 1, ' ': 4, 'c': 1, 'e': 6, 'd': 1, 'g': 1, 'f': 1, 'i': 2,
 'l': 4, 'o': 3, 'n': 1, 'p': 1, 's': 5, 'r': 3, 'u': 2, 'y': 1}
```

Observe that `in` is used here in two different ways: `for letter in phrase` iterates over every letter, running the body of the `for` loop. Inside this loop, the conditional expression `if letter not in count` checks whether the letter is missing from the dictionary. If it is missing, we create a new entry and set its value to zero: `count[letter] = 0`. Now we are sure that the entry exists, and it may have a zero or non-zero value. We finish the body of the `for` loop by incrementing this particular counter using the += assignment operator. Finally, we print the dictionary, to see the letters and their counts. This method of maintaining many counters will find many uses, and you will become very familiar with it.

There are other useful ways to display the result, such as sorting alphabetically by the letter:

```
>>> sorted(count.items())
[(' ', 4), ('a', 1), ('c', 1), ('d', 1), ('e', 6), ('f', 1), ...,
...('y', 1)]
```

> **Note**
>
> The function `sorted()` is similar to the `sort()` method on sequences, but rather than sorting in-place, it produces a new sorted copy of its argument. Moreover, as we will see very soon, `sorted()` will work on a wider variety of data types, including dictionaries.

2.6.3 Getting unique entries

Sometimes, we don't want to count at all, but just want to make a record of the items that we have seen, regardless of repeats. For example, we might want to compile a vocabulary from a document. This is a sorted list of the words that appeared, regardless of frequency. At this stage we have two ways to do this. The first uses lists.

```
>>> sentence = "she sells sea shells by the sea shore".split()
>>> words = []
>>> for word in sentence:
...     if word not in words:
...         words.append(word)
...
>>> sorted(words)
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
```

We can write this using a dictionary as well. Each word we find is entered into the dictionary as a key. We use a value of `1`, but it could be anything we like. We extract the keys from the dictionary simply by converting the dictionary to a list:

```
>>> found = {}
>>> for word in sentence:
...     found[word] = 1
...
>>> sorted(found)
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
```

There is a third way to do this, which is best of all: using Python's **set** data type. We can convert `sentence` into a set, using `set(sentence)`:

```
>>> set(sentence)
set(['shells', 'sells', 'shore', 'she', 'sea', 'the', 'by'])
```

The order of items in a set is not significant, and they will usually appear in a different order to the one they were entered in. The main point here is that converting a list to a set removes any duplicates. We convert it back into a list, sort it, and print. Here is the complete program:

```
>>> sentence = "she sells sea shells by the sea shore".split()
>>> sorted(set(sentence))
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
```

Here we have seen that there is sometimes more than one way to solve a problem with a program. In this case, we used three different built-in data types, a list, a dictionary, and a set. The set data type mostly closely modelled our task, so it required the least amount of work.

2.6.4 Scaling it up

We can use dictionaries to count word occurrences. For example, the following code reads *Macbeth* and counts the frequency of each word:

```
>>> from nltk_lite.corpora import gutenberg
>>> count = {}                                          # initialize a dictionary
>>> for word in gutenberg.raw('shakespeare-macbeth'): # tokenize Macbeth
...     word = word.lower()                             # normalize to lowercase
...     if word not in count:                           # seen this word before?
...         count[word] = 0                             # if not, set count to zero
...     count[word] += 1                                # increment the counter
...
>>>
```

This example demonstrates some of the convenience of NLTK in accessing corpora. We will see much more of this later. For now, all you need to know is that `gutenberg.raw()` returns a list of words, in this case from Shakespeare's play *Macbeth*, which we are iterating over using a `for` loop. We convert each word to lowercase using the string method `word.lower()`, and use a dictionary to maintain a set of counters, one per word. Now we can inspect the contents of the dictionary to get counts for particular words:

```
>>> count['scotland']
12
>>> count['the']
692
>>>
```

2.6.5 Exercises

1. ☼ Using the Python interpreter in interactive mode, experiment with the examples in this section. Create a dictionary `d`, and add some entries. What happens if you try to access a non-existent entry, e.g. `d['xyz']`?

2. ☼ Try deleting an element from a dictionary, using the syntax `del d['abc']`. Check that the item was deleted.

3. ☼ Create a dictionary `e`, to represent a single lexical entry for some word of your choice. Define keys like `headword`, `part-of-speech`, `sense`, and `example`, and assign them suitable values.

4. ☼ Create two dictionaries, `d1` and `d2`, and add some entries to each. Now issue the command `d1.update(d2)`. What did this do? What might it be useful for?

5. ◑ Write a program that takes a sentence expressed as a single string, splits it and counts up the words. Get it to print out each word and the word's frequency, one per line, in alphabetical order.

2.7 Defining Functions

It often happens that part of a program needs to be used several times over. For example, suppose we were writing a program that needed to be able to form the plural of a singular noun, and that this needed to be done at various places during the program. Rather than repeating the same code several times over, it is more efficient (and reliable) to localize this work inside a **function**. A function is a programming construct which can be called with one or more inputs, and which returns an output. We define a function using the keyword `def` followed by the function name and any input parameters, followed by a colon; this in turn is followed by the body of the function. We use the keyword `return` to indicate the value that is produced as output by the function. The best way to convey this is with an example. Our function `plural()` in Listing 2.1 takes a singular noun as input, and generates a plural form as output.

(There is much more to be said about ways of defining in functions, but we will defer this until Section 6.4.)

2.8 Regular Expressions

For a moment, imagine that you are editing a large text, and you have strong dislike of repeated occurrences of the word *very*. How could you find all such cases in the text? To be concrete, let's suppose that the variable `str` is bound to the text shown below:

```
>>> str = """Google Analytics is very very very nice (now)
... By Jason Hoffman 18 August 06
... Google Analytics, the result of Google's acquisition of the San
... Diego-based Urchin Software Corporation, really really opened it's
... doors to the world a couple of days ago, and it allows you to
... track up to 10 sites within a single google account.
... """
>>>
```

Listing 1

```
def plural(word):
    if word[-1] == 'y':
        return word[:-1] + 'ies'
    elif word[-1] in 'sx':
        return word + 'es'
    elif word[-2:] in ['sh', 'ch']:
        return word + 'es'
    elif word[-2:] == 'an':
        return word[:-2] + 'en'
    return word + 's'

>>> plural('fairy')
'fairies'
>>> plural('woman')
'women'
```

The triple quotes """ are useful here, since they allow us to break a string across lines.

One approach to our task would be to convert the string into a list, and look for adjacent items which are both equal to the string 'very'. We use the range(n) function in this example to create a list of consecutive integers from 0 up to, but not including, n:

```
>>> text = str.split(' ')
>>> for n in range(len(text)):
...     if text[n] == 'very' and text[n+1] == 'very':
...                     print n, n+1
...
3 4
4 5
>>>
```

However, such an approach is not very flexible or convenient. In this section, we will present Python's **regular expression** module re, which supports powerful search and substitution inside strings. As a gentle introduction, we will start out using a utility function re_show() to illustrate how regular expressions match against substrings. re_show() takes two arguments, a pattern that it is looking for, and a string in which the pattern might occur.

```
>>> import re
>>> from nltk_lite.utilities import re_show
>>> re_show('very very', str)
Google Analytics is {very very} very nice (now)
...
>>>
```

(We have only displayed first part of str that is returned, since the rest is irrelevant for the moment.) As you can see, re_show places curly braces around the first occurrence it has found of the string 'very very'. So an important part of what re_show is doing is searching for any substring of str which **matches** the pattern in its first argument.

Now we might want to modify the example so that re_show highlights cases where there are two *or more* adjacent sequences of 'very'. To do this, we need to use a **regular expression operator**,

namely '+'. If *s* is a string, then *s*+ means: 'match one or more occurrences of *s*'. Let's first look at the case where *s* is a single character, namely the letter 'o':

```
>>> re_show('o+', str)
G{oo}gle Analytics is very very very nice (n{o}w)
...
>>>
```

'o+' is our first proper regular expression. You can think of it as matching an *infinite set* of strings, namely the set {'o', 'oo', 'ooo', ...}. But we would really like to match against the set which contains strings of least two 'o's; for this, we need the regular expression 'oo+', which matches any string consisting of 'o' followed by one or more occurrences of o.

```
>>> re_show('oo+', str)
G{oo}gle Analytics is very very very nice (now)
...
>>>
```

Let's return to the task of identifying multiple occurrences of 'very'. Some initially plausible candidates won't do what we want. For example, 'very+' would match 'veryyy' (but not 'very very'), since the + scopes over the immediately preceding expression, in this case 'y'. To widen the scope of +, we need to use parentheses, as in '(very)+'. Will this match 'very very'? No, because we've forgotten about the whitespace between the two words; instead, it will match strings like 'veryvery'. However, the following *does* work:

```
>>> re_show('(very\s)+', str)
Google Analytics is {very very very }nice (now)
>>>
```

Characters which are preceded by a \, such as '\s', have a special interpretation inside regular expressions; thus, '\s' matches a whitespace character. We could have used ' ' in our pattern, but '\s' is better practice in general. One reason is that the sense of 'whitespace' we are using is more general than you might have imagined; it includes not just inter-word spaces, but also tabs and newlines. If you try to inspect the variable str, you might initially get a shock:

```
>>> str
"Google Analytics is very very very nice (now)\nBy Jason Hoffman
18 August 06\n\nGoogle
...
>>>
```

You might recall that '\n' is a special character that corresponds to a newline in a string. The following example shows how newline is matched by '\s'.

```
>>> str2 = "I'm very very\nvery happy"
>>> re_show('very\s', str2)
I'm {very }{very
}{very }happy
>>>
```

Python's re.findall(*patt*, *str*) is a useful function which returns a list of all the substrings in *str* that are matched by *patt*. Before illustrating, let's introduce two further special characters, '\d' and '\w': the first will match any digit, and the second will match any alphanumeric character.

```
>>> re.findall('\d\d', str)
['18', '06', '10']
>>> re.findall('\s\w\w\w\s', str)
[' the ', ' the ', ' the ', ' and ', ' you ']
>>>
```

As you will see, the second example matches three-letter words. However, this regular expression is not quite what we want. First, the leading and trailing spaces are extraneous. Second, it will fail to match against strings such as `'the San'`, where two three-letter words are adjacent. To solve this problem, we can use another special character, namely `'\b'`. This is sometimes called a 'zero-width' character; it matches against the empty string, but only at the begining and ends of words:

```
>>> re.findall(r'\b\w\w\w\b', str)
['now', 'the', 'the', 'San', 'the', 'ago', 'and', 'you']
```

> **Note**
>
> This example uses a Python **raw string**: `r'\b\w\w\w\b'`. The specific justification here is that in an ordinary string, `\b` is interpreted as a backspace character. Python will convert it to a backspace in a regular expression unless you use the `r` prefix to create a raw string as shown above. Another use for raw strings is to match strings which include backslashes. Suppose we want to match 'either\or'. In order to create a regular expression, the backslash needs to be escaped, since it is a special character; so we want to pass the pattern `\\` to the regular expression interpreter. But to express this as a Python string literal, each backslash must be escaped again, yielding the string `'\\\\'`. However, with a raw string, this reduces down to `r'\\'`.

Returning to the case of repeated words, we might want to look for cases involving `'very'` or `'really'`, and for this we use the disjunction operator `|`.

```
>>> re_show('((very|really)\s)+', str)
Google Analytics is {very very very }nice (now)
By Jason Hoffman 18 August 06
Google Analytics, the result of Google's acquisition of the San
Diego-based Urchin Software Corporation, {really really }opened it's
doors to the world a couple of days ago, and it allows you to
track up to 10 sites within a single google account.
>>>
```

In addition to the matches just illustrated, the regular expression `'((very|really)\s)+'` will also match cases where the two disjuncts occur with each other, such as the string `'really very really '`.

Let's now look at how to perform substitutions, using the `re.sub()` function. In the first instance we replace all instances of `l` with `s`. Note that this generates a string as output, and doesn't modify the original string. Then we replace any instances of `green` with `red`.

```
>>> sent = "colorless green ideas sleep furiously"
>>> re.sub('l', 's', sent)
'cosorsess green ideas sseep furioussy'
>>> re.sub('green', 'red', sent)
'colorless red ideas sleep furiously'
>>>
```

We can also disjoin individual characters using a square bracket notation. For example, `[aeiou]` matches any of `a`, `e`, `i`, `o`, or `u`, that is, any vowel. The expression `[^aeiou]` matches anything that is *not* a vowel. In the following example, we match sequences consisting of non-vowels followed by vowels.

```
>>> re_show('[^aeiou][aeiou]', sent)
{co}{lo}r{le}ss g{re}en{ i}{de}as s{le}ep {fu}{ri}ously
>>>
```

Using the same regular expression, the function `re.findall()` returns a list of all the substrings in `sent` that are matched:

```
>>> re.findall('[^aeiou][aeiou]', sent)
['co', 'lo', 'le', 're', ' i', 'de', 'le', 'fu', 'ri']
>>>
```

2.8.1 Groupings

Returning briefly to our earlier problem with unwanted whitespace around three-letter words, we note that `re.findall()` behaves slightly differently if we create **groups** in the regular expression using parentheses; it only returns strings which occur within the groups:

```
>>> re.findall('\s(\w\w\w)\s', str)
['the', 'the', 'the', 'and', 'you']
>>>
```

The same device allows us to select only the non-vowel characters which appear before a vowel:

```
>>> re.findall('([^aeiou])[aeiou]', sent)
['c', 'l', 'l', 'r', ' ', 'd', 'l', 'f', 'r']
>>>
```

By delimiting a second group in the regular expression, we can even generate pairs (or **tuples**), which we may then go on and tabulate.

```
>>> re.findall('([^aeiou])([aeiou])', sent)
[('c', 'o'), ('l', 'o'), ('l', 'e'), ('r', 'e'), (' ', 'i'),
    ('d', 'e'), ('l', 'e'), ('f', 'u'), ('r', 'i')]
>>>
```

Our next example also makes use of groups. One further special character is the so-called wildcard element, `'.'`; this has the distinction of matching any single character (except `'\n'`). Given the string `str3`, our task is to pick out login names and email domains:

```
>>> str3 = """
... <hart@vmd.cso.uiuc.edu>
... Final editing was done by Martin Ward <Martin.Ward@uk.ac.durham>
... Michael S. Hart <hart@pobox.com>
... Prepared by David Price, email <ccx074@coventry.ac.uk>"""
```

The task is made much easier by the fact that all the email addresses in the example are delimited by angle brackets, and we can exploit this feature in our regular expression:

```
>>> re.findall(r'<(.+)@(.+)>', str3)
[('hart', 'vmd.cso.uiuc.edu'), ('Martin.Ward', 'uk.ac.durham'),
('hart', 'pobox.com'), ('ccx074', 'coventry.ac.uk')]
>>>
```

Since `'.'` matches any single character, `'.+'` will match any non-empty *string* of characters, including punctuation symbols such as the period.

One question which might occur to you is how do we specify a match against a period? The answer is that we have to place a `'\'` immediately before the `'.'` in order to escape its special interpretation.

```
>>> re.findall(r'(\w+\.)', str3)
['vmd.', 'cso.', 'uiuc.', 'Martin.', 'uk.', 'ac.', 'S.',
'pobox.', 'coventry.', 'ac.']
>>>
```

Now, let's suppose that we wanted to match occurrences of both `'Google'` and `'google'` in our sample text. If you have been following up till now, you would reasonably expect that this regular expression with a disjunction would do the trick: `'(G|g)oogle'`. But look what happens when we try this with `re.findall()`:

```
>>> re.findall('(G|g)oogle', str)
['G', 'G', 'G', 'g']
>>>
```

What is going wrong? We innocently used the parentheses to indicate the scope of the operator `'|'`, but `re.findall()` has interpreted them as marking a group. In order to tell `re.findall()` "don't try to do anything special with these parentheses", we need an extra piece of notation:

```
>>> re.findall('(?:G|g)oogle', str)
['Google', 'Google', 'Google', 'google']
>>>
```

Placing `'?:'` immediately after the opening parenthesis makes it explicit that the parentheses are just being used for scoping.

2.8.2 Practice Makes Perfect

Regular expressions are very flexible and very powerful. However, they often don't do what you expect. For this reason, you are strongly encouraged to try out a variety of tasks using `re_show()` and `re.findall()` in order to develop your intuitions further; the exercises below should help get you started. One tip is to build up a regular expression in small pieces, rather than trying to get it completely right first time.

As you will see, we will be using regular expressions quite frequently in the following chapters, and we will describe further features as we go along.

2.8.3 Exercises

1. ☼ Describe the class of strings matched by the following regular expressions. Note that `'*'` means: match zero or more occurrences of the preceding regular expression.

   a) `[a-zA-Z]+`

   b) `[A-Z][a-z]*`

c) `\d+(\.\d+)?`

d) `([bcdfghjklmnpqrstvwxyz][aeiou][bcdfghjklmnpqrstvwxyz])*`

e) `\w+|[^\w\s]+`

Test your answers using `re_show()`.

2. ☼ Write regular expressions to match the following classes of strings:

   a) A single determiner (assume that *a*, *an*, and *the* are the only determiners).

   b) An arithmetic expression using integers, addition, and multiplication, such as `2*3+8`.

3. ◑ Using `re.findall()`, write a regular expression which will extract pairs of values of the form *`login name`*, *`email domain`* from the following string:

```
>>> str = """
... austen-emma.txt:hart@vmd.cso.uiuc.edu  (internet)  hart@uiucvmd (bitnet)
... austen-emma.txt:Internet (72600.2026@compuserve.com); TEL: (212-254-5093)
... austen-persuasion.txt:Editing by Martin Ward (Martin.Ward@uk.ac.durham)
... blake-songs.txt:Prepared by David Price, email ccx074@coventry.ac.uk
... """
```

4. ◑ Write code to convert text into *hAck3r* again, this time using regular expressions and substitution, where `e` → `3`, `i` → `1`, `o` → `0`, `l` → `|`, `s` → `5`, `.` → `5w33t!`, `ate` → `8`. Normalise the text to lowercase before converting it. Add more substitutions of your own. Now try to map `s` to two different values: `$` for word-initial `s`, and `5` for word-internal `s`.

5. ◑ Write code to read a file and print it in reverse, so that the last line is listed first.

6. ◑ Write code to access a favorite webpage and extract some text from it. For example, access a weather site and extract the forecast top temperature for your town or city today.

7. ★ Read the Wikipedia entry on the *Soundex Algorithm*. Implement this algorithm in Python.

2.9 Summary

- Text is represented in Python using strings, and we type these with single or double quotes: `'Hello'`, `"World"`.
- The characters of a string are accessed using indexes, counting from zero: `'Hello World'[1]` gives the value `e`. The length of a string is found using `len()`.
- Substrings are accessed using slice notation: `'Hello World'[1:5]` gives the value `ello`. If the start index is omitted, the substring begins at the start of the string, similarly for the end index.
- Sequences of words are represented in Python using lists of strings: `['colorless', 'green', 'ideas']`. We can use indexing, slicing and the `len()` function on lists.

- Strings can be split into lists: `'Hello World'.split()` gives `['Hello', 'World']`. Lists can be joined into strings: `string.join(['Hello', 'World'], '/')` gives `'Hello/World'`.
- Lists can be sorted in-place: `words.sort()`. To produce a separate, sorted copy, use: `sorted(words)`.
- We process each item in a string or list using a `for` statement: `for word in phrase`. This must be followed by the colon character and an indented block of code, to be executed each time through the loop.
- We test a condition using an `if` statement: `if len(word) < 5`. This must be followed by the colon character and an indented block of code, to be executed only if the condition is true.
- A dictionary is used to map between arbitrary types of information, such as a string and a number: `freq['cat'] = 12`. We create dictionaries using the brace notation: `pos = {}`, `pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}`.
- [More: regular expressions]

2.10 Further Reading

Guido Van Rossum (2003). *An Introduction to Python*, Network Theory Ltd.

Guido Van Rossum (2003). *The Python Language Reference*, Network Theory Ltd.

Guido van Rossum (2005). *Python Tutorial* http://docs.python.org/tut/tut.html

A.M. Kuchling. *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/

Python Documentation http://docs.python.org/

Allen B. Downey, Jeffrey Elkner and Chris Meyers () *How to Think Like a Computer Scientist: Learning with Python* http://www.ibiblio.org/obp/thinkCSpy/