2. Programming Fundamentals and Python

2.1 Introduction

This chapter provides a non-technical overview of Python and will give you the know-how to survive the next few chapters. It contains many working snippets of code which you should try yourself — in fact, we insist! There is no better way to learn about programming than to dive in and try the examples. Hopefully, you will then feel brave enough to modify and combine them for your own purposes, and so before you know it you are programming!

For a more detailed overview, we recommend that you consult one of the introductions listed in the further reading section below. More advanced programming material is contained in a later tutorial.

2.2 Python the Calculator

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

```
Python 2.4.1 (#65, Mar 30 2005, 09:13:57) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Depending on your environment, the version number and the blurb at the top might be different. As long as you are running Python 2.4 or greater everything will be fine. The Python prompt `>>>` indicates it 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, like the brown prompt above. We have colorized the notes in the same way, so that you can tell if you have typed the code in correctly.

So, let's use the Python prompt as a calculator:

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

There are several things to notice here, the first being that once the interpreter has finished calculating the answer and displaying it, the prompt reappears. This means the Python interpreter is waiting for another instruction. The second thing to notice is that Python deals with the order of operations correctly (unlike some older calculators), so the `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 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.

You probably already have a calculator, so why is it so important that you can use Python like one? Well, there are two main reasons:

- this demonstrates that you *can* work interactively with the interpreter, which means you can experiment and explore;
- your intuitions about numerical expressions will be very useful for manipulating all kinds of data in Python.

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

Here we have produced a **syntax error** because it doesn't make sense to end an instruction with a plus sign. The Python interpreter indicates on what line and where on the line the particular problem occurred. Be careful about this though because sometimes the interpreter can get confused and point to something after the actual error. If you are in IDLE, you will only see the last line and the previous line will be highlighted where the problem is.

2.3 Understanding the Basics

2.3.1 Representing text

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

This is an example of the Python interpreter getting confused about where the syntax error is. It should actually be pointing to the beginning of the `Hello`.

Python represents text using a **type** called 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.

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

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

This operation is called **concatenation** and it returns a new string which is a copy of the two original strings pasted together end to end. Notice that this doesn't do anything clever like insert a space between the words — because the Python interpreter has no way of knowing that you want a space, it only does *exactly* what it is told. Based on this it is also possible to guess what multiplication might do (we have given you a hint too!):

```
>>> '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!

2.3.2 Storing and reusing values

After a while it gets quite tiresome to be retyping strings and other expressions over and over again. Your programs won't get very interesting or helpful either. What we need is a place for storing results of calculations and other values, so we can access them more conveniently, say with a name for instance. The named place is called a **variable**.

In Python we create variables by **assignment**, which involves putting a value into the variable:

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

Here we have created a variable called `msg` (short for message) and stored in it the string value `'Hello World'`. Notice the Python interpreter returns immediately because assignment returns no value. On the next line we inspect the contents of the variable by naming it on the command line `msg`. The interpreter prints out the contents on the next line.

We can use variables in any place where we used values previously:

```
>>> msg + msg
'Hello WorldHello World'
>>> msg * 3
'Hello WorldHello WorldHello World'
>>>
```

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

```
>>> msg = msg*2
>>> msg
'Hello WorldHello World'
>>>
```

Here we have taken the value of `msg`, multiplied it by `2` and then stored that new string (`Hello WorldHello World`) back into the variable `msg`. You need to be careful not to treat the `=` as equality as it would be in mathematics, but instead treat it as taking the result of the expression calculated on the right, and assign it to the variable on the left.

2.3.3 Printing and inspecting

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 it into the interpreter. For example, we can look at the contents of `msg` by going:

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

However, this trick only works in the interactive interpreter. When we want users of our programs to be able to see the output, we must use the `print` statement:

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

Hmm, that seems to do the same thing! However, on closer inspection you will see that the quotation marks which indicate that Hello World is a string are missing in the second case. That is because inspecting a variable (that only works in the 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.

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.

2.3.4 Exercises

1. Start up the Python interpreter (e.g. by running IDLE). Try the examples in the last section, before experimenting with the use of Python as a calculator.

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

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

   ```
   >>> 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.

2.4 Manipulating strings and storing multiple values

2.4.1 Accessing individual characters and substrings

Often we want to manipulate part of the text in a string, be it a single letter, digit, punctuation or space (a **character**) or a smaller part of the whole string (a **substring**). Accessing individual characters simply involves specifying which character you want to access inside square brackets:

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

This process is called **indexing** or **subscripting** the string, and the integer position we are interested in is called the **index**. Notice that the positions start from zero (`msg[0]` has the value `'H'`). This might seem strange to you now but it is much more convenient that way, and you will find it natural and consistent eventually. Also, we can retrieve not only letters but any character, such as the space at index `5`.

How do we know what we can index up to? We can use 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. Here we have called 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.

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 (0 to 10) for the string `'Hello World'`. As a result, we get an error message, but this one is a bit different to the syntax error we have seen before, there is this `Traceback` message this time. This is because an `IndexError` is an **exception** which is how the Python indicates an error has occurred in a running program. We will talk more about exceptions in a later chapter.

Alright, so let's try another (hopefully) out of range value:

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

Hmm, this time something unexpected happens, we have got a `'d'` back rather than the error we expected. This is because negative indices in Python work from the back of the string, so `-1` indexes the last character which is `'d'`:

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

We can access the space in the middle of Hello and World with either `msg[5]` or `msg[-6]`.

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 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 starting indexing from zero: indexing starts from zero and goes up to *one before* the length of the string. We can see that by indexing with `len` directly:

```
>>> 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:

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

so here that means we stop before the space character. 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 `'W'` to the end of the string. This shortcut leads 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).

Finally, we can also specify a step size for the slice using an extra colon:

```
>>> msg[0:6:2]
'Hlo'
>>>
```

This returns every second character within the slice. We can use negative step values as well, which if combined with the slice `[:]` does something quite tricky:

```
>>> msg[::-1]
'dlroW olleH'
>>>
```

Yes, that's right! It *reverses* the string.

There are lots of combinations you can try with indexing and slices. They are incredibly powerful yet the notation is quite simple. You should try some more slices using the interactive interpreter. Try to guess what the result will be and then run it to make sure you are correct.

2.4.2 Storing more than one value

Python strings only represent *characters* of text. What can you use to store multiple strings, integers or some other type of value? You could use a whole lot of variables, but that isn't really practical because you might not know how many values you need to store when you write the program.

To help with this Python provides the list data structure. A **list** is designed to store zero or more values in a given order. We call these data structures **sequences**, and yes strings are sequences too. In fact, you can think of a list as similar to a string in many ways except the individual items can be *any type*, including 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 famous nonsense sentence as a list and put it in a variable `a`:

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

Because lists and strings are both sequence types, they share lots of functionality. In particular, sequence types support indexing and slicing, which we can do with a list as well. The following example performs many of the operations you have seen above for the list `a`:

```
>>> len(a)
3
>>> a[0]
'colorless'
>>> a[-1]
'ideas'
>>> a[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Aha, this is something we didn't mention above! You can have a negative index that is out of range if it falls off the beginning of the list or string. Here, `a[-5]` generates an error, because the fifth-last item in a three item list is undefined. We can also slice lists in exactly the same way as strings:

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

Lists can be concatenated just like strings:

```
>>> b = a + ['sleep', 'furiously']
>>> print b
['colorless', 'green', 'ideas', 'sleep', 'furiously']
>>> a
['colorless', 'green', 'ideas']
>>>
```

Here we have put the resulting list into a new variable `b`. The original variable `a` has not been changed in the process.

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 list `a` to `'colorful'`, we can do that by assigning to the index `a[0]`:

```
>>> a[0] = 'colorful'
>>> a
['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. Lists on the other hand are **mutable** which means you can change their value 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. Two of these are *sorting* and *reversing*:

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

Notice that the prompt reappears on the line after `b.sort()` and `b.reverse()`. That is because these methods do not return a new list, but instead modify the original list stored in the variable `b`. On the other hand, we can use the slice trick from above `b[::-1]` to create a *new* reversed list without changing `b`:

```
>>> b[::-1]
['colorless', 'furiously', 'green', 'ideas', 'sleep']
>>> b
['sleep', 'ideas', 'green', 'furiously', 'colorless']
>>>
```

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:

```
>>> b.append('said')
>>> b.append('Chomsky')
>>> b
['sleep', 'ideas', 'green', 'furiously', 'colorless', 'said', 'Chomsky']
>>> b.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:

```
>>> z = [123, 'abc', [1, 2, 3]]
>>>
```

2.4.3 Working on sequences one item at a time

We are usually storing lists of values together for some reason. Often because we want to treat everything in the list the same way (this is why different types in the same list rarely make sense). This means we want to perform the same operation (or run the same Python code) on every item of the list one at a time. Python makes this very easy with the `for` loop:

```
>>> odds = [1, 3, 5, 7, 9]
>>> for i in odds:
...     print i*2
...
2
6
10
14
18
>>>
```

What this does is run the statement `print i*2` over every single element of the list `odds` one at a time. This process is called *iteration* or *looping*. Each *iteration* of the `for` loop starts by assigning the next element of the list `odds` to the loop variable `i`. Then the *body* of the loop, which is indented is run. Here the body consists only of the line `print i*2`, but in principle the body can contain as many lines of code as you want.

Notice also that the interactive interpreter changes the prompt from `>>>` to the `...` prompt. This indicates it is expecting an indented block of code to appear next. To finish the indented block just enter a blank line. We say the `for` loop is a *control structure* because it changes the number of times the statements it *controls* are run.

We can run another for loop over the Chomsky nonsense, and print out the last character as well:

```
>>> for w in b:
...     print w[-1]
...
p
s
n
```

```
y
s
d
y
>>>
```

Observe that we used the square bracket notation `w[-1]` to access characters in the string `w`, just like we would if `w` was a string variable we had created by assignment ourselves.

One final thing for you to try is that `for` loops work for any sequence type, so you can also iterate over the characters of a string one at a time.

2.4.4 Converting between strings and lists

Often we want to convert a string containing a space separated list of words and into a list of strings. Other times you want to do the reverse: take a list of strings and make it into a single space separated string. Python strings have some methods of their own to make this task simple:

```
>>> nonsense = ' '.join(b)
>>> nonsense
'sleep ideas green furiously colorless said Chomsky'
>>>
```

This notation for the `join` method may look very odd at first. However, it follows exactly the same convention as `sort` and `append` above. A method is called on a particular object using the object, followed by a period followed by the name of the method and the parentheses containing any arguments.

Here, the object is a string that consists of a single space `' '`, and the name of the method is `join`, and the single argument to the `join` method is the list of strings `b`. As you can see from the example above, this takes the space and creates a new string by inserting it between all of the items in the list `b`. We have stored that string in the variable `nonsense`.

We can do the same with any string:

```
>>> ' blah '.join(b)
'sleep blah ideas blah green blah furiously blah colorless blah said blah Chomsky'
>>>
```

Ok, now we want to do the reverse process, splitting a string (and let's use the new string `nonsense` up on the space character:

```
>>> nonsense.split(' ')
['sleep', 'ideas', 'green', 'furiously', 'colorless', 'said', 'Chomsky']
>>> nonsense.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. If you don't specify an argument to `split` it will split on one or more whitespace characters.

2.4.5 Exercises

1. Using the Python interpreter in interactive mode, 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. Define a string `s = 'colorless'`. Write a Python statement that changes this to 'colourless', using only the slice and concatenation operations.

3. Process the list `b` using a `for` loop, and print the lengths of each word, one number per line. Now do the same thing, but store the result in a new list `c`. Hint: begin by assigning the empty list to `c`, using `c = []`, each time through the loop, use `append()` to add another length value to the list.

4. 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 this string into a list of strings, one per word, using Python's `split()` operation.
   b) Extract the second letter of each word and join them into a string, to get `'eoldrnnnna'`.
   c) Generate a list consisting of all the words up to (but not including) `in`. Hint: use the `index()` function in combination with list slicing.
   d) Combine these words back into a single string, using `join()`. Make sure the words in the resulting string are separated with whitespace.
   e) Print the words of this sentence in alphabetical order, one per line.

2.5 Making Decisions

2.5.1 Making simple decisions

So far our programs haven't been all that interesting, because although they can now iterate through items in a list, they still aren't making any decisions on our behalf. This decision making capacity is provided by `if` statements in Python (and almost all imperative programming languages). The `if` statement executes a block of code when a particular conditional expression is true:

```
>>> x = 3
>>> if x < 5:
...    print 'x is less than 5'
...
x is less than 5
>>>
```

In this example, we have created a variable called `x` containing the integer value `3`. The `if` statement then checks whether the condition `x < 5` is true (which in this case it is because we just set `x` to `3`). Because the conditional expression is true, the *body* of the `if` statement is executed, so the `print` statement is executed.

If we change the conditional expression to `x >= 5` (`x` 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 x >= 5:
...     print 'x 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 again 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 x >= 5:
...     print 'x is greater than or equal to 5'
... else:
...     print 'x is less than 5'
...
x 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 x < 3:
...     print 'x is less than three'
... elif x == 3:
...     print 'x is equal to three'
... else:
...     print 'x is greater than three'
...
x 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:

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

You can test these conditional operators directly at the prompt:

```
>>> 3 < 5
```

```
True
>>> 5 < 3
False
>>>
```

Python also supports conditional operators for lists and strings:

```
>>> msg = 'Hello World'
>>> 'H' in msg
True
>>> 'X' in msg
False
>>> 'ell' in msg
True
>>>
```

Try this for yourself using the Chomsky nonsense list as well.

Finally, Python strings have a couple of useful methods for testing the what appears at the beginning and the end of a string (as opposed to anywhere in the string:

```
>>> msg.startswith('Hell')
True
>>> msg.endswith('rld')
True
>>>
```

2.5.3 Putting it all together

Right, so everything we have seen so far has been in bits and pieces. 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'`. To do this we must:

a) split the string into a list of words

b) iterate over the list of words

c) for each word check if it ends with a `'ow'`;

d) if it does we must print it out.

This might seem a bit overwhelming at first, so let's start with simply splitting the string into a list which we can store in a variable called **`words`**:

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

This is where the Python interactive interpreter is useful. It allows us to inspect the contents of variables while we are writing larger pieces of software.

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

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

The next stage is to only print out the words if they end in the string `'ow'`. Now we could try to insert this directly into our loop, or we could test it out first to make sure we know what is going on:

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

Now we are ready to put the `if` statement inside the `for` loop. This time we will display the whole program in one go:

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

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

2.5.4 Exercises

1. Assign a new value to `sentence`, 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.

2.6 Other ways of storing data

2.6.1 Accessing data with data

Elements of a list are accessed using an index, by which we mean that there is a *mapping* between the index (an integer) and the value stored *at* that index (which can have any type).

Often however, we want to access one value using another arbitrary value which may not be an integer. For example, using a phonebook we lookup a phone number (an integer) based on a name (a string). Another example is using a dictionary to find the definition (a string) of a word (another string).

Python provides a *dictionary* data structure for just this kind of occasion. We access elements in a dictionary using the familiar square bracket notation. Here we define the variable `d` is an empty dictionary and then add three entries to it:

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

The empty dictionary is created using the pair of braces `{}`, which are often called **empty braces** because there is nothing between them. We can see what the contents of the dictionary look like now by inspecting the variable `d`:

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

Because the interpreter always displays the contents of variables in Python syntax we can see how to create a dictionary containing values. Rather than using the empty braces and adding entries we could have just typed that one line.

The data is occurs in pairs where the value we are using for lookups (called the **key**) appears first, followed by a colon, and then the value we want to retrieve (called the **value**). The entries in a dictionary are thus often called **key-value pairs** and are separated by commas in the example above.

The second thing to notice is that the order of the key-value pairs has changed. Although we entered the key `'colorless'` with the value `'adj'` first, it appears second in the list of keys. This is because unlike lists Python dictionaries do not maintain order because they are not *sequences*.

We can retrieve individual values using their corresponding key:

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

And just like with indexing on lists and strings we get an exception when we try to lookup the value of a key that does not exist:

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

However, this causes a problem here. Unlike lists and strings where we know which integers are legal indices (by looking at the length of the string) it isn't so easy to do with dictionaries. Instead we can check whether a key exists in a dictionary, just like we can check whether a value is in a list, using the `in` operator:

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

Be careful with the `in` operator for dictionaries, it only applies to the keys and not the values they correspond to.

For example, if we check for adjective `'adj'` rather than `'colorless'` we get the result `False`:

```
>>> 'adj' in d
False
>>>
```

The example above also demonstrates the use of `not` in a `in` expression to check if a key is missing rather than to check if it is present. We can also get access to the keys and values as separate lists, and also key-value pairs:

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

If we wanted to loop over all of the keys we could use a `for` loop with `d.keys()` but Python provides a shortcut:

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

Remember again that this is over the *keys* and not the *values*. However, as you can see above, we can get access to the value quite simply using `d[w]`.

Dictionaries provide a number of useful methods as well. The `get` method checks to see if a key exists and if not, returns a default value:

```
>>> d.get('colorless', 'unk')
'adj'
>>> d.get('missing', 'unk')
'unk'
>>>
```

The first argument passed to the `get` method is the key to lookup, and the second parameter is the default value to use if the key does not appear in the dictionary.

2.6.2 Getting unique entries

Often we want to create a set of unique entries, that is, remove all of the duplicates. Typically we might want to remove extra copies of a string from a list of strings. One simple way to do this would be to use the keys of a dictionary, because it is only possible to store one copy of a key in a dictionary:

```
>>> d = {}
>>> d['dog'] = 'n'
>>> d['dog'] = 'v'
>>> d
{'dog': 'v'}
>>>
```

Now what we might have expected here was for two entries for `'dog'` to be in the dictionary, one for the noun sense and another for the verb sense. Unfortunately, dictionaries can only map from a key to one value, so we end up replacing the noun case `'n'` with the verb case `'v'`.

However, this has its uses for duplication removal:

```
>>> d = {}
>>> for w in ['dog', 'cat', 'dog', 'dog', 'cat', 'mouse']:
...     d[w] = 1
...
>>> d
{'mouse': 1, 'dog': 1, 'cat': 1}
>>> d.keys()
['mouse', 'dog', 'cat']
>>>
```

Notice here it doesn't matter what we use for the values in this dictionary, since we are only interested in the keys, so we have just used the value `1`. Once we have iterated over the list and set the value for every key we have seen to `1`, we can just get the list of unique words back using `d.keys()`.

This became such a common trick in Python that the **set** data structure was introduced to store *sets* of values. Sets have the property that values can only appear once. This means we can convert the example above into a one line expression:

```
>>> s = set(['dog', 'cat', 'dog', 'dog', 'cat', 'mouse'])
>>> s
set(['mouse', 'dog', 'cat'])
>>>
```

Remember that the order of items in a set is not significant. When you display a set, you will see its elements in some arbitrary order, and probably not in the order you gave them originally.

2.6.3 Putting it together (with some NLTK)

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
...
>>>
```

This example demonstrates some of the power of NLTK that we will show you later in the book. For now, all you need to know is that `gutenberg.raw()` returns a list of words, in this case from Shakespeare's Macbeth, which we are iterating over using a `for` loop. For each of these we convert it to lowercase using the string method `word.lower()`. We then check to see if the lowercase word has already been seen in the dictionary `count`. If not, we create a new entry with an initial count of zero. Then we add one to the count (whether or not it has been seen before). This is the standard idiom in Python for counting things — particularly strings.

Now we can inspect the contents of the dictionary:

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

We can even go further and produce a list of words sorted in descending order by frequency:

```
>>> frequencies = [(freq, word) for (word, freq) in count.items()]
>>> frequencies.sort()
>>> frequencies.reverse()
>>> print frequencies[:20]
[(1986, ','), (1245, '.'), (692, 'the'), (654, "'"), (567, 'and'),
 (482, ':'), (399, 'to'), (365, 'of'), (360, 'i'), (255, 'a'),
 (246, 'that'), (242, '?'), (224, 'd'), (218, 'you'), (213, 'in'),
 (207, 'my'), (198, 'is'), (170, 'not'), (165, 'it'), (156, 'with')]
>>>
```

Don't worry if you don't understand how all of this works. Basically the first line just creates a list of pairs of frequencies and words. We need to do this so we can sort on frequency rather than by alphabetical order. The rest of the lines you should recognise. We sort and reverse the list using the `sort` and `reverse` methods of the list, and then print out the first 20 elements using a Python slice.

2.6.4 Exercises

1. Create a dictionary `d`, and add some entries. What happens if you try to access a non-existent entry, e.g. `d['xyz']`?

2. Create a dictionary `e`, to represent a single lexical entry. Define keys like `headword`, `part-of-speech`, `sense`, and `example`.

2.7 Miscellaneous trickery

2.7.1 Regular Expressions

Python has a **regular expression** module `re` — this supports powerful search and substitution inside strings.

```
>>> import re
>>> from nltk_lite.utilities import re_show
>>> sent = "colorless green ideas sleep furiously"
>>>
```

We use a utility function `re_show` to show how regular expressions match against substrings. First we search for all instances of a particular character or character sequence:

```
>>> re_show('l', sent)
co{l}or{l}ess green ideas s{l}eep furious{l}y
>>> re_show('green', sent)
colorless {green} ideas sleep furiously
>>>
```

Now we can perform substitutions. 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`.

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

So far we have only seen simple patterns, consisting of individual characters or sequences of characters. However, regular expressions can also contain special syntax, such as `|` for disjunction, e.g.:

```
>>> re_show('(green|sleep)', sent)
colorless {green} ideas {sleep} furiously
>>> re.findall('(green|sleep)', sent)
['green', 'sleep']
>>>
```

We can also disjoin individual characters. 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
>>> re.findall('[^aeiou][aeiou]', sent)
['co', 'lo', 'le', 're', ' i', 'de', 'le', 'fu', 'ri']
>>>
```

We can put parentheses around parts of an expression in order to generate structured results. For example, here we see all those non-vowel characters which appear before a vowel:

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

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')]
>>>
```

For an extended discussion of regular expressions, please see the regular expression tutorial.

2.7.2 Accessing Files and the Web

It is easy to access local files in Python. Here are some examples. (You will need to create a file called `corpus.txt` before you can open it for reading.)

```
>>> f = open('corpus.txt', 'rU')
>>> f.read()
'Hello world.\nThis is a test file.\n'
>>>
```

The first stage is to **open** a file using the builtin function `open`, which takes two arguments, the name of the file, here corpus.txt, and the mode to open the file with — for now trust us and always use `'rU'` for files you are opening for reading.

To read the contents of the file we can use lots of different methods. The one above uses the file read method `f.read()`, which reads the entire contents of a file into a string.

You might be wondering what is the funny `'\n'` character on the end of the string. This is a **newline** character, which is equivalent to pressing *Enter* and starting a new line. There is also a `'\t'` character for representing tab.

We can also read a file one line at a time using the `for` loop construct:

```
>>> f = open('corpus.txt')
>>> for line in f:
...     print line[:-1]
Hello world.
This is a test file.
>>>
```

Here we use the slice trick `line[:-1]` to chomp of the newline we are reading in from the file since the `print` statement already adds an extra newline.

Finally, it isn't too much more difficult in Python to read in a webpage:

```
>>> from urllib import urlopen
>>> page = urlopen("http://news.bbc.co.uk/").read()
>>> page = re.sub('<[^>]*>', '', page)    # strip HTML markup
>>> page = re.sub('\s+', ' ', page)       # strip whitespace
>>> print page[:60]
BBC NEWS | News Front Page News Sport Weather World Service
>>>
```

2.7.3 Exercises

1. Describe the class of strings matched by the following regular expressions:

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

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

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

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

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

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`.

1. Write code to convert text into *hAck3r*, where `e` → `3`, `i` → `1`, `o` → `0`, `l` → `|`, `s` → `5`, `.` → “ 5w33t¡‘, `ate` → `8`. Convert 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`.

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

3. 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.

2.8 Accessing NLTK

NLTK consists of a set of Python *modules*, each of which defines classes and functions related to a single data structure or task. Before you can use a module, you must `import` its contents. The simplest way to import the contents of a module is to use the `from module import *` command. For example, to import the contents of the `nltk_lite.util` module, which is discussed in this tutorial, type:

```
>>> from nltk_lite.utilities import *
>>>
```

A disadvantage of this style of import statement is that it does not specify what objects are imported; and it is possible that some of the import objects will unintentionally cause conflicts. To avoid this disadvantage, you can explicitly list the objects you wish to import. For example, as we saw earlier, we can import the `re_show` function from the `nltk_lite.util` module as follows:

```
>>> from nltk_lite.utilities import re_show
>>>
```

Another option is to import the module itself, rather than its contents. Now its contents can then be accessed using *fully qualified* dotted names:

```
>>> from nltk_lite import utilities
>>> utilities.re_show('green', sent)
colorless {green} ideas sleep furiously
>>>
```

For more information about importing, see any Python textbook.

NLTK is distributed with several corpora, listed in the introduction. Many of these corpora are supported by the NLTK `corpora` module. First we import the Gutenberg corpus (a selection of texts from the Project Gutenberg electronic text archive).

```
>>> from nltk_lite.corpora import gutenberg
>>> gutenberg.items
['austen-emma', 'austen-persuasion', 'austen-sense', 'bible-kjv',
 'blake-poems', 'blake-songs', 'chesterton-ball', 'chesterton-brown',
 'chesterton-thursday', 'milton-paradise', 'shakespeare-caesar',
 'shakespeare-hamlet', 'shakespeare-macbeth', 'whitman-leaves']
>>>
```

We access the text content using a special Python construct called an *iterator*. It produces a stream of words, which we can access using the syntax `for item in iterator`, as shown below:

```
>>> count = 0
>>> for word in gutenberg.raw('whitman-leaves'):
...     count += 1
>>> print count
154873
>>>
```

NLTK-Lite also includes the Brown Corpus, the first million word, part-of-speech tagged electronic corpus of English, created in 1961 at Brown University. Each of the sections **a** through **r** represents a different genre.

```
>>> from nltk_lite.corpora import brown
>>> brown.items
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r']
>>>
```

We can extract individual sentences from the corpus iterator using the `extract()` function:

```
>>> from nltk_lite.corpora import extract
>>> print extract(0, brown.raw())
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an',
 'investigation', 'of', "Atlanta's", 'recent', 'primary', 'election',
 'produced', '``', 'no', 'evidence', "''", 'that', 'any', 'irregularities',
 'took', 'place', '.']
>>>
```

We can also access the tagged text using the `brown.tagged()` method:

```
>>> print extract(0, brown.tagged())
[('The', 'at'), ('Fulton', 'np-tl'), ('County', 'nn-tl'), ('Grand', 'jj-tl'),
 ('Jury', 'nn-tl'), ('said', 'vbd'), ('Friday', 'nr'), ('an', 'at'),
 ('investigation', 'nn'), ('of', 'in'), ("Atlanta's", 'np$'), ('recent', 'jj'),
 ('primary', 'nn'), ('election', 'nn'), ('produced', 'vbd'), ('``', '``'),
 ('no', 'at'), ('evidence', 'nn'), ("''", "''"), ('that', 'cs'),
 ('any', 'dti'), ('irregularities', 'nns'), ('took', 'vbd'), ('place', 'nn'),
 ('.', '.')]
>>>
```

NLTK-Lite includes a 10% fragment of the Wall Street Journal section of the Penn Treebank. This can be accessed using `treebank.raw()` for the raw text, and `treebank.tagged()` for the tagged text.

```
>>> from nltk_lite.corpora import treebank
>>> print extract(0, treebank.parsed())
(S:
  (NP-SBJ:
    (NP: (NNP: 'Pierre') (NNP: 'Vinken'))
    (,: ',')
    (ADJP: (NP: (CD: '61') (NNS: 'years')) (JJ: 'old'))
    (,: ','))
  (VP:
    (MD: 'will')
    (VP:
      (VB: 'join')
      (NP: (DT: 'the') (NN: 'board'))
      (PP-CLR:
        (IN: 'as')
        (NP: (DT: 'a') (JJ: 'nonexecutive') (NN: 'director')))
      (NP-TMP: (NNP: 'Nov.') (CD: '29'))))
  (.: '.'))
>>>
```

NLTK contains some simple chatbots, which will try to talk intelligently with you. You can access the famous Eliza chatbot using `from nltk_lite.chat import eliza`, then run `eliza.demo()`. The other chatbots are called `iesha` (teen anime talk), `rude` (insulting talk), and `zen` (gems of Zen wisdom), and were contributed by other students who have used NLTK.

2.8.1 Exercises

1. Use the corpus module to read `austin-persuasion.txt`. How many words does this book have?

2. Use the Brown corpus reader `brown.raw()` to access some sample text in two different genres.

3. Try running the various chatbots. How *intelligent* are these programs? Take a look at the program code and see if you can discover how it works. You can find the code online at: `http://nltk.sourceforge.net/lite/nltk_lite/chat/`.

2.9 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/